Trace Matching in a Concurrent Logical Framework [*]

Iliano Cervesato
Carnegie Mellon University
Doha, Qatar
iliano@cmu.edu

Frank Pfenning
Carnegie Mellon University
Pittsburgh, PA, USA
fp@cs.cmu.edu

Jorge Luis Sacchini
Carnegie Mellon University
Doha, Qatar
sacchini@qatar.cmu.edu

Carsten Schürmann
IT University of Copenhagen — Copenhagen, Denmark
carsten@itu.dk

Robert J. Simmons
Carnegie Mellon University — Pittsburgh, PA, USA
rjsimmon@cs.cmu.edu

Abstract

Matching and unification play an important role in implementations of proof assistants, logical frameworks, and logic programming languages. In particular, matching is at the heart of many reasoning tasks and underlies the operational semantic for well-moded logic programs. In this paper, we study the problem of matching on concurrent traces in the CLF logical framework, an extension of LF that supports the specification of concurrent and distributed systems. A concurrent trace is a sequence of computations where independent steps can be permuted. We give a sound and complete algorithm for matching traces with one variable standing for an unknown subtrace. Extending the result to general traces and to unification is left to future work.

Categories and Subject Descriptors F.4.4 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—lambda calculus and related systems

Keywords Concurrent traces, matching, logical frameworks

1. Introduction

Meta-logical frameworks are specialized formalisms designed to capture the meta-theory of formal systems such as programming languages and logics. They allow expressing properties such as type preservation, semantics-preserving compilation and cut-elimination, as well as their proofs. Meta-logical frameworks form the very foundation that underlies systems such as Coq [10], Isabelle [26], Agda [16], and Twelf [18], which can automate the verification that a proof is correct. The form of reasoning that the current generation of meta-logical frameworks handles well operates on inductively-defined derivation trees that obey a simple equational theory (often just equality modulo α-equivalence). Typing and evaluation derivations for sequential programming languages have this form; so do the derivations of many logics.

Reasoning about languages, such as the π-calculus [14] or Petri nets [17], that exhibit a parallel, concurrent or distributed semantics, does not fit this pattern, however. Some steps on parallel threads can occur in any order without affecting the result of a computation, but communication and other forms of concurrency introduce dependencies that force the order of some steps. This selective permutability of steps poses a new challenge for the design of meta-logical frameworks for concurrent systems. The resulting equational theory is more complex and algorithmically not as well understood. Indeed, computation traces in these systems are often depicted as directed acyclic graphs, because graphs are agnostic to the particular order of the computation steps that are executed while still capturing the causal dependencies between inputs and outputs [19].

Reasoning about such languages can be automated in two ways. One way is to encode the equational theory of concurrent computations in a traditional logical framework. Honsell et al. [9] did precisely this when developing a significant portion of the meta-theory of the π-calculus in Coq [10]. The main drawbacks of this approach are that it is extremely labor intensive and that applying it to a new language often amounts to starting from scratch. The other way is to develop a logical framework that internalizes the equational theory of concurrent computations. This is the approach taken in CLF [5, 24], an extension of the LF type theory with monads and constructs borrowed from linear logic. CLF has supported the syntax and semantics of every concurrent language we have attempted to encode in it: we could simulate the execution of concurrent programs written in these languages in the accompanying Celf tool [22], obtaining proof-terms that, thanks to CLF's equational theory, express the corresponding computations in their full generality. We are now in the process of extending Celf with support for reasoning about these concurrent computations. This paper presents an initial step in this direction.

The key to any reasoning task on computations is to isolate steps and name subcomputations: the steps are immediately examined and the subcomputations are analyzed recursively or co-recursively. Operationally, naming computations is realized through unification when subcomputations of different origin are required to be equal, or *matching* when reasoning about one given trace. Furthermore,

[*] This work was partially supported by the Qatar National Research Fund under grant NPRP 09-1107-1-168, the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under Grant NGN-44, and the Danish Council for Strategic Research, Programme Commission on Strategic Growth Technologies under grant 10-092309.

matching underlies the operational semantics of well-moded programs in a logical framework.

In this paper, we examine the matching problem for a large fragment of CLF's language of computational traces. We show that, for the fragment considered, matching is decidable although highly non-deterministic. We propose a sound and complete algorithm for the case where there is at most one logic variable standing for an unknown trace.

We are not aware of any comprehensive study of matching, let alone unification, for computational traces, even as they are at the heart of automated reasoning on parallel and concurrent computations. Closest is the work of Messner [12] that studies a specific form of matching for Mazurkiewicz traces [19], a simpler notion than ours. Those problems have however been studied extensively for related equational theories. Examples include unification in a commutative monoid (ACU unification) which is the basis of any multiset rewriting framework and string unification (associative with unit) [1, 6], as well as graph matching and isomorphism [28].

The fragment of CLF examined in this paper captures a general form of concurrent computation based on state transition rules. Languages that can be directly expressed in this fragment include place/transition Petri nets [17], colored Petri nets [11] and various forms of multiset rewriting [2, 3]. The associated computational traces are a more concrete, more general form of Mazurkiewicz traces [19] as our notion of independence is given by a binding structure, while in Mazurkiewicz traces, independence is given by a fixed relation between the elements of a trace. For space reasons, we do not consider the type of traces produced by computations in shallow encodings of process algebra [14], although CLF supports them. CLF also allows arbitrary combinations of forward chaining, typical of concurrent computations, and backward chaining found in top-down logic programming. We focus on matching for forward chaining traces, as unification in the backward chaining sublanguage is well understood [15].

The rest of the paper is organized as follows: in Section 2 we define the fragment of CLF we use, its equational theory, and the infrastructure to explore the matching problem. We present a matching algorithm for it and prove its correctness in Section 3. We outline areas of future development in Section 4.

2. Setting

In this section, we define the syntax, typing, and equality for our language and some of the infrastructure needed for equations.

2.1 Language

The language examined in this paper is a large fragment of the CLF type theory [5, 24] (see Section 2.2 for a comparison). For simplicity, we rely on a somewhat non-standard presentation organized around contexts, terms, traces, and types [20].

Contexts A *context* is a sequence of variable declarations of the form $\square x{:}A$, where A is a type (defined below) and $\square$ is one of three *modalities*: ! (persistent), @ (affine), or $\downarrow$ (linear).

$$\begin{array}{rrcl} \textit{Modalities:} & \square & ::= & ! \mid @ \mid \downarrow \\ \textit{Contexts:} & \Delta & ::= & \cdot \mid \Delta, \square x{:}A \\ \textit{Signatures:} & \Sigma & ::= & \cdot \mid \Sigma, c{:}A \end{array}$$

We assume that each variable name x is declared at most once in a context. The typing semantics below allows persistent variables ($!x$) to occur free in a type, but not linear ($\downarrow x$) or affine ($@x$) variables. Said differently, we allow variable dependencies on persistent, but not affine or linear declarations.

We write $!\Delta$ for the largest subcontext of Δ declaring only persistent variables. Similarly, $@\Delta$ is the largest subcontext of Δ containing only affine or persistent declarations (i.e., $@\Delta$ is obtained from Δ by removing all linear variables). For uniformity, we write $\downarrow\Delta$ as a synonym for Δ. At times, $!\Delta$ will denote a context consisting of persistent declarations only.

Given a context Δ and a variable x declared in Δ, we denote with $\Delta \setminus\!\!\setminus x$ the context obtained by removing the declaration for x if it is affine or linear, formally: $(\Delta_0, \square x{:}A, \Delta_1) \setminus\!\!\setminus x = (\Delta_0, \Delta_1)$ if $\square \in \{@, \downarrow\}$ and $(\Delta \setminus\!\!\setminus x) = \Delta$ if $!x{:}A \in \Delta$ for some A.

As usual, a signature declares the constants in use. We will implicitly assume a global signature Σ.

Terms The language of terms of our language combines aspects of the spine calculus of [4] and of the pattern-based presentation of CLF in [21]. It is given by the following grammar:

$$\begin{array}{rrcl} \textit{Terms:} & N & ::= & \widehat{\lambda}\Delta.\, H{\cdot}S \\ \textit{Heads:} & H & ::= & x \mid c \\ \textit{Spines:} & S & ::= & \cdot \mid \square N, S \end{array}$$

A *term* $\widehat{\lambda}\Delta.\, H{\cdot}S$ consists of an abstraction pattern Δ applied to an *atomic term* $H{\cdot}S$. Its head H can be either a constant from the global signature or a variable. Its *spine* S is a sequence of moded terms, and has therefore a structure similar to a context. A spine can be viewed as the uncurrying of iterated applications in a λ-calculus. An atomic term is closely related to the monadic terms of [21, 24]. The abstraction pattern Δ is a binder for the term and its scope is $H{\cdot}S$. It can be understood as the uncurrying of iterated abstractions over individual variables. In examples, we omit empty abstraction contexts, writing $H{\cdot}S$ for $\widehat{\lambda}(\cdot).\, H{\cdot}S$. We also omit empty spines in an atomic term.

Given a spine S and a context Δ of the same length (and type, see below), we write S/Δ for the simultaneous substitution of each variable in Δ with the corresponding term in S. Given a term N, we write $N[S/\Delta]$ for the term obtained after applying S/Δ *hereditarily* to N. Hereditary substitution applies S/Δ and at once reduces the result to canonical form [21, 24]. We similarly write $S'[S/\Delta]$ and $\Delta'[S/\Delta]$ for the simultaneous substitution hereditarily applied to a spine and a context, respectively.

Traces and expressions The trace of a concurrent computation is a record of all the steps performed together with any dependency among them. Each step *uses* certain resources modeled here as context variables, possibly embedded within terms, and *produces* other resources, modeled as a context with fresh variables. This notion of step is found in all concurrent languages based on state transitions, e.g., Petri nets [11, 17] and multiset rewriting [2, 3].

Traces and the related notion of expressions are defined as follows in our language:

$$\begin{array}{rrcl} \textit{Traces:} & \epsilon & ::= & \diamond \mid \epsilon_1; \epsilon_2 \mid \{\Delta\}{\leftarrow}c{\cdot}S \\ \textit{Expressions:} & E & ::= & \{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\} \end{array}$$

A *trace* is either empty ($\diamond$), a composition of two traces ($\epsilon_1; \epsilon_2$), or an individual computation step of the form $\{\Delta\}{\leftarrow}c{\cdot}S$, where c is a constant in the global signature Σ, and S is a spine. We write δ for a generic computational step $\{\Delta\}{\leftarrow}c{\cdot}S$. We call c the *head* of δ. A step of the form $\{\Delta\}{\leftarrow}c{\cdot}S$ represents an atomic computation c that uses the variables in S and produces the variables in Δ. Linear and affine variables in S are consumed and cannot be used again. Persistent variables can be used more than once.

In a step $\delta = \{\Delta\}{\leftarrow}c{\cdot}S$, the context Δ acts as a binder for the variables it declares. Its scope is any trace that may follow δ. As with any binder, variables bound in this way are subject to automatic α-renaming as long as no variable capture arises. These variables will need to be managed with care in the matching algorithm in Section 3.

An *expression* $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$ is essentially a trace with delimited scope: no variable produced by ϵ is visible outside it. The context Δ collects all unused linear variables and some affine and persis-

Contexts:

$$\frac{}{!\Delta \vdash \cdot} \qquad \frac{!\Delta \vdash A : \mathsf{type} \qquad !(\Delta, \square x{:}A) \vdash \Delta'}{!\Delta \vdash \square x{:}A, \Delta'}$$

Base types:

$$\frac{a{:}\Pi !\Delta'.\mathsf{type} \in \Sigma \qquad !\Delta \vdash S : !\Delta'}{!\Delta \vdash a{\cdot}S : \mathsf{type}} \qquad \frac{!\Delta \vdash \Delta'}{!\Delta \vdash \{\Delta'\} : \mathsf{type}}$$

Types:

$$\frac{!\Delta \vdash \Delta' \qquad !(\Delta, \Delta') \vdash P : \mathsf{type}}{!\Delta \vdash \Pi\Delta'.P : \mathsf{type}}$$

Kinds:

$$\frac{!\Delta \vdash !\Delta'}{!\Delta \vdash \Pi !\Delta'.\mathsf{type} : \mathsf{kind}}$$

Figure 1. Typing rules for types and kinds

tent variables. In CLF [21, 24], this context is essentially a spine. Furthermore, CLF expressions can appear in terms. We disallow this here. Finally, CLF terms of the form $\widehat{\lambda}\Delta.\,E$ are not considered since they play no part in our matching algorithm, although we will occasionally use them in the examples.

Types and kinds Terms and expressions are classified by types, themselves classified by kinds. They are defined by the following grammar:

Base types:	P	$::=$	$a{\cdot}S \mid \{\Delta\}$
Types:	A	$::=$	$\Pi\Delta.P$
Kinds:	K	$::=$	$\Pi!\Delta.\mathsf{type}$

Base types are either *atomic* or *monadic*. Atomic types have the form $a{\cdot}S$, where a is a constant defined in the signature Σ applied to a spine S containing only persistent terms (i.e., terms of the form $!N$). Monadic types have the form $\{\Delta\}$, which are equivalent to the positive types of CLF [22]. The scope of the declarations in $\{\Delta\}$ is limited to Δ itself.

For convenience, we write $A \to B$ for $\Pi(!x{:}A).B$ when x does not occur in B. Similarly, we write $A \multimap B$ for $\Pi(\downarrow x{:}A).B$ and $A \mathbin{-\!@} B$ for $\Pi(@x{:}A).B$ — recall that only persistent variables can appear free in a type. These are the types of persistent, linear and affine functions, respectively. Although the syntax prevents iterated functions, the general form $\Pi\Delta.P$ captures them in uncurried form: for example, the function $A \to B \multimap C \mathbin{-\!@} D$ is expressed as $\Pi(!x{:}A, \downarrow y{:}B, @z{:}C).\,D$. We will often curry such types for clarity. As for terms, we usually omit empty contexts in types and kinds, writing type for $\Pi(\cdot).\mathsf{type}$ and P for $\Pi(\cdot).P$.

2.2 Typing and Equality

As in most type theories, the typing semantics of our language is based on a notion of equality over its various syntactic constructs. We present the typing rules and then the underlying equational theory.

Typing The typing semantics of our language is expressed by the following judgments [20]:

Terms:

$$\frac{!\Delta \vdash \Delta' \qquad \Delta, \Delta' \vdash H{\cdot}S \Leftarrow P}{\Delta \vdash \widehat{\lambda}\Delta'.\, H{\cdot}S \Leftarrow \Pi\Delta'.P}$$

$$\frac{\begin{array}{c}\square x{:}\Pi\Delta'.a{\cdot}S' \in \Delta \\ \Delta \setminus\!\!\setminus x \vdash S \Leftarrow \Delta' \qquad P \equiv a{\cdot}S'[S/\Delta']\end{array}}{\Delta \vdash x{\cdot}S \Leftarrow P}$$

$$\frac{c{:}\Pi\Delta'.a{\cdot}S' \in \Sigma \qquad \Delta \vdash S \Leftarrow \Delta' \qquad P \equiv a{\cdot}S'[S/\Delta']}{\Delta \vdash c{\cdot}S \Leftarrow P}$$

Spines:

$$\frac{}{@\Delta \vdash \cdot : \cdot}$$

$$\frac{!\Delta_1 \vdash N \Leftarrow A \qquad \Delta_0 \vdash S : \Delta_2[!N/!x]}{\Delta_0 \bowtie @\Delta_1 \vdash !N, S \Leftarrow !x{:}A, \Delta_2}$$

$$\frac{@\Delta_1 \vdash N \Leftarrow A \qquad \Delta_0 \vdash S : \Delta_2}{\Delta_0 \bowtie @\Delta_1 \vdash @N, S \Leftarrow @x{:}A, \Delta_2}$$

$$\frac{\downarrow\Delta_1 \vdash N \Leftarrow A \qquad \Delta_0 \vdash S : \Delta_2}{\Delta_0 \bowtie \downarrow\Delta_1 \vdash \downarrow N, S \Leftarrow \downarrow x{:}A, \Delta_2}$$

Traces:

$$\frac{}{\Delta \vdash \diamond : \Delta} \qquad \frac{\Delta \vdash \epsilon_1 : \Delta_1 \qquad \Delta_1 \vdash \epsilon_2 : \Delta_2}{\Delta \vdash \epsilon_1 ; \epsilon_2 : \Delta_2}$$

$$\frac{c{:}\Pi\Delta'.\{\Delta''\} \in \Sigma \qquad \Delta_1 \vdash S \Leftarrow \Delta' \qquad \Delta_2 \equiv \Delta''[S/\Delta']}{\Delta_0 \bowtie \Delta_1 \vdash \{\Delta_2\}{\leftarrow}c{\cdot}S : \Delta_0, \Delta_2}$$

Expressions:

$$\frac{\Delta_0 \vdash \epsilon : \Delta_1 \qquad \Delta_1 \preccurlyeq \Delta'}{\Delta_0 \vdash \{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta'\} \Leftarrow \{\Delta'\}}$$

Figure 2. Typing rules for terms and traces

Kinds:	$!\Delta \vdash K : \mathsf{kind}$
Base types:	$!\Delta \vdash P : \mathsf{type}$
Types:	$!\Delta \vdash A : \mathsf{type}$
Contexts:	$!\Delta \vdash \Delta'$
Expressions:	$\Delta \vdash E \Leftarrow \{\Delta\}$
Traces:	$\Delta \vdash \epsilon : \Delta'$
Spines:	$\Delta \vdash S \Leftarrow \Delta'$
Terms:	$\Delta \vdash N \Leftarrow A$

Their definition, in Figures 1 and 2, relies on equality $\equiv$ (defined below) and two additional relations. First, context Δ *splits* into Δ_1 and Δ_2, written $\Delta = \Delta_1 \bowtie \Delta_2$, iff $!\Delta = !\Delta_1 = !\Delta_2$ and each affine or linear declaration in Δ appear in exactly one of Δ_1 and Δ_2. Second, we say that Δ is *weaker* than Δ', denoted $\Delta \preccurlyeq \Delta'$ (or equivalently $\Delta' \succcurlyeq \Delta$), iff $\Delta = \Delta' \bowtie @\Delta_0$ for some context Δ_0, i.e., if Δ' is included in Δ and Δ does not contain any linear hypotheses not present in Δ'. Furthermore, we will be able to rely on implicit α-renaming to ensure that a context extension or concatenation does not declare duplicate variable names.

The rules for types and kinds in Figure 1 are standard. Note that only persistent hypotheses are used for type-checking them as linear and affine hypotheses cannot appear free in them — $!\Delta$ in these rules is required to mention only of persistent variables while $!$ applied to constructed contexts (e.g., $!(\Delta, \Delta')$) filters out non-persistent declarations. Context typing works similarly.

The typing rules for terms and spines are a minor variant of the semantics of [4, 21] for these entities. The typing rules for traces reflect that a trace can be seen as a transformation on states, modeled as contexts. The empty trace does not change the state. A step transforms a part of the state. The spine S uses Δ_1 and produces Δ_2 leaving the rest of the state, represented by Δ_0, intact. The typing rule for composition of traces effectively composes the transformations given by each trace. Note that the monadic type of an expression does not leak out the variables produced by the trace it embeds.

By relying on hereditary substitution, these judgment ensure that well-typed objects are canonical, i.e., do not contain redexes [21, 24]. They also satisfy the following properties.

Lemma 1 (Frame rule). *If $\Delta_1 \vdash \epsilon : \Delta_2$, then $\Delta_0 \bowtie \Delta_1 \vdash \epsilon : \Delta_0 \bowtie \Delta_2$.*

Lemma 2 (Inversion for trace typing). *If $\Delta_0 \vdash \epsilon_1; \epsilon_2 : \Delta_2$, then there exists Δ_1 such that $\Delta_0 \vdash \epsilon_1 : \Delta_1$ and $\Delta_1 \vdash \epsilon_2 : \Delta_2$.*

Equality Equality for terms, spines, and types, denoted $\equiv$, is defined in the usual way, up to α-equivalence. Context equality is also defined in the usual way: two contexts are equal if they declare the same variables with equal types. We omit their definition [20]. Equality for traces and expressions is more complex as it allows permuting independent computations.

Independence The *input interface* of a trace ϵ, denoted $\bullet\epsilon$, is the set of variables available for use in ϵ, the free variables of ϵ. The *output interface* of ϵ, denoted $\epsilon\bullet$, is the set of variables available for use to any computation that may follow ϵ. Together they form the *interface* of ϵ. They are formally defined as follows:

$$\begin{aligned}
\bullet(\diamond) &= \emptyset \\
\bullet(\{\Delta\}{\leftarrow}c{\cdot}S) &= \mathrm{FV}(S) \\
\bullet(\epsilon_1; \epsilon_2) &= \bullet\epsilon_1 \cup (\bullet\epsilon_2 \setminus \epsilon_1\bullet)
\end{aligned}$$

$$\begin{aligned}
(\diamond)\bullet &= \emptyset \\
(\{\Delta\}{\leftarrow}c{\cdot}S)\bullet &= \mathrm{dom}(\Delta) \\
(\epsilon_1; \epsilon_2)\bullet &= \epsilon_2\bullet \cup (\epsilon_1\bullet \setminus \bullet\epsilon_2) \cup !(\epsilon_1\bullet)
\end{aligned}$$

where $\mathrm{FV}(S)$ denotes the free variables of S, defined in the usual way, and $\mathrm{dom}(\Delta)$ is the set of variables declared in Δ. Observe that $(\epsilon_1; \epsilon_2)\bullet$ includes the persistent variables introduced by ϵ_1 since these variables are available after executing ϵ_2 even if ϵ_2 uses them. The input and output interfaces of a typable ϵ are the variables declared in smallest contexts Δ and Δ' respectively such that $\Delta \vdash \epsilon : \Delta'$.

Two traces ϵ_1 and ϵ_2 are *independent*, denoted $\epsilon_1 \parallel \epsilon_2$, iff $\bullet\epsilon_1 \cap \epsilon_2\bullet = \emptyset$ and $\bullet\epsilon_2 \cap \epsilon_1\bullet = \emptyset$ [19]. Permuting a typable composition of independent traces is always typable [20].

Trace equality Trace composition is associative with the empty trace as it unit, thereby endowing traces with a monoidal structure [20]. Furthermore, independent subtraces can be permuted. Formally, trace equality, also written $\equiv$, is defined by the following rules:

$$\frac{}{\epsilon; \diamond \equiv \epsilon} \qquad \frac{}{\epsilon \equiv \epsilon; \diamond} \qquad \frac{}{\epsilon_1; (\epsilon_2; \epsilon_3) \equiv (\epsilon_1; \epsilon_2); \epsilon_3}$$

$$\frac{\epsilon_1 \parallel \epsilon_2}{\epsilon_1; \epsilon_2 \equiv \epsilon_2; \epsilon_1} \qquad \frac{\epsilon_1 \equiv \epsilon_1'}{\epsilon_1; \epsilon_2 \equiv \epsilon_1'; \epsilon_2} \qquad \frac{\epsilon_2 \equiv \epsilon_2'}{\epsilon_1; \epsilon_2 \equiv \epsilon_1; \epsilon_2'}$$

For example, the following equality among traces holds:

$$\begin{pmatrix} \{\downarrow x_2\}{\leftarrow}c{\cdot}{\downarrow}x_1; \\ \{\downarrow y_2\}{\leftarrow}c{\cdot}{\downarrow}y_1 \end{pmatrix} \equiv \begin{pmatrix} \{\downarrow y_2\}{\leftarrow}c{\cdot}{\downarrow}y_1; \\ \{\downarrow x_2\}{\leftarrow}c{\cdot}{\downarrow}x_1 \end{pmatrix}$$

Expression equality In an expression $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$, the scope of the variables produced by ϵ extends to the context Δ (and indeed stops there). It is therefore natural to allow the output variables to α-vary in unison with the variables declared in Δ when defining α-equivalence over expressions. Specifically, two expressions $\{\mathsf{let}\ \epsilon_1\ \mathsf{in}\ \Delta_1\}$ and $\{\mathsf{let}\ \epsilon_2\ \mathsf{in}\ \Delta_2\}$ are α-equivalent if the internal and output variables of the two traces ϵ_1 and ϵ_2 can be renamed (without distinct variables within each trace being identified) as to become syntactically equal and the same output renaming makes the two contexts syntactically equal as well. For example,

$$\{\mathsf{let} \begin{pmatrix} \{z\}{\leftarrow}c; \\ \{y\}{\leftarrow}c'{\cdot}z \end{pmatrix} \mathsf{in}\ y\} \qquad \text{and} \qquad \{\mathsf{let} \begin{pmatrix} \{x\}{\leftarrow}c; \\ \{z\}{\leftarrow}c'{\cdot}x \end{pmatrix} \mathsf{in}\ z\}$$

are α-equivalent, although the traces in them are not (since their output interfaces differ). As usual, α-equivalence yields the derived notion of α-renaming, which we will exploit to implicitly rewrite an expression $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$ by altering synchronously the output interface $\epsilon\bullet$ of ϵ and the variables $\mathrm{dom}(\Delta)$ declared by Δ whenever convenient.

Exploiting the implicit α-renaming of bound variables, we can define expression equality simply as:

$$\frac{\epsilon_1 \equiv \epsilon_2 \qquad \Delta_1 \equiv \Delta_2}{\{\mathsf{let}\ \epsilon_1\ \mathsf{in}\ \Delta_1\} \equiv \{\mathsf{let}\ \epsilon_2\ \mathsf{in}\ \Delta_2\}}$$

Comparison with CLF The language presented in this paper differs from CLF [21, 24] in a several ways. We exclude CLF's additive conjunction on types as it does not complicate the matching problem. Furthermore, we replaced CLF's untyped patterns with contexts in binding positions. This has the effect of simplifying the typing rules in two ways: first contexts are flat while patterns can be nested arbitrarily, and second the availability of the typing information avoids hunting it down. The most substantial difference concerns expressions: using our syntax, CLF expressions have the general form $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ S\}$ rather than $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$. We make this restriction for two reasons: first, most CLF specifications we have developed fit in the fragment presented in this paper; second, matching trailing spines S (monadic terms in CLF parlance) is a much more difficult problem than matching trailing contexts (patterns in CLF) as discussed in the next section. Finally, the head of a CLF step can be a variable, not just a constant. Although this feature allows directly representing process algebras rather than just state transition systems, we omitted it for simplicity — see [20] for a full treatment.

The language presented here allows us to develop a simpler matching algorithm that is described in the next section. We will address the matching problem in full CLF in forthcoming work.

2.3 Equations

An *equation* is a postulated equality between entities that may contain logic variables. A *logic variable* X stands for an unknown term or trace. Logic variables are distinct from term variables x. We now develop the machinery to solve equations in Section 3.

Logic variables A *contextual modal context* [15] is a sequence of logic variable declarations, formally defined as follows:

$$\Psi ::= \cdot \mid \Psi, X :: \Delta \vdash A$$

Each declaration of the form $X :: \Delta_X \vdash A_X$ determines a distinct logic variable X with its own context Δ_X and type A_X (under Δ_X). We will assume a global contextual modal context Ψ.

Within a term defined in a context Δ_0, a logic variable X is accompanied by a *substitution* θ that maps the variables in Δ_X to well-typed terms over Δ_0. The pair is denoted $X[\theta]$. Substitutions are defined by the following grammar:

$$\theta ::= \cdot \mid \theta, \square N / \square' x$$

In a substitution item $\Box N/\Box' x$, the modalities $\Box$ and $\Box'$ may be different: indeed, a linear variable may be replaced by a persistent term without violating typing (so that $!N/{\downarrow}x$ is allowed), while the opposite may yield an ill-typed term (e.g., ${\downarrow}N/!x$ may lead to multiple copies of ${\downarrow}N$). The valid values for $(\Box, \Box')$ are given by the reflexive-transitive closure of $\{(!, @), (@, \downarrow)\}$. Substitutions of this form are called *linear changing*.

We extend the syntax of our language by allowing logic variables as heads and in steps:

$$\begin{array}{rccl} \textit{Heads:} & H & ::= & x \mid c \mid X[\theta] \\ \textit{Steps} & \delta & ::= & \{\Delta\}{\leftarrow}c{\cdot}S \mid \{\Delta\}{\leftarrow}X[\theta] \end{array}$$

A logic variable is *atomic* (resp. *monadic*) if its type is atomic (resp. monadic). Atomic logic variables can only occur as heads of terms, while monadic logic variables can only occur as heads of steps.

Substitutions are type-checked using the following judgment:

$$\Delta_1 \vdash \theta : \Delta_2$$

The typing rules related to logic variables and substitutions are given in Figure 3.

In this paper, logic variables have base types. Indeed, logic variables with functional types can always be *lowered* [15], i.e., replaced with a new logic variable of base type. Given a logic variable $X :: \Delta_X \vdash \Pi\Delta.P$, we introduce a new logic variable Y declared by $Y :: \Delta_X, \Delta \vdash P$ and replace every occurrence of X by $\widehat{\lambda}\Delta.\, Y[\mathrm{id}]$, where id is the identity substitution.

An *assignment* σ is a sequence of bindings of the form $X \leftarrow N$ where X is a logic variable. An assignment σ is well typed if for every $X \leftarrow N$ with $X :: \Delta_X \vdash A_X$, we have $\Delta_X \vdash N : A_X$. Applying an assignment $[X \leftarrow N]$ to a term M (resp. expression, type, etc.), denoted $[X \leftarrow N]M$, means replacing every occurrence of $X[\theta]$ in M with θN and reducing the resulting expression to canonical form. For traces, applying an assignment is defined by the following rule:

$$[X \leftarrow \{\mathsf{let}\ \epsilon_0\ \mathsf{in}\ \Delta_0\}](\epsilon_1; \{\Delta\}{\leftarrow}X[\theta]; \epsilon_2) = (\epsilon_1; \theta\epsilon_0; [\Delta_0/\Delta]\epsilon_2)$$

In the examples below, we usually do not specify the declaration of logic variables and leave local variables implicit. E.g., given a logic variable $X :: x_1{:}a_1, x_2{:}a_2 \vdash \{\Delta\}$ we write $X[y_1, y_2]$ for $X[y_1/x_1, y_2/x_2]$.

Pattern substitutions A *pattern substitution* [13] is a substitution whose codomain consists of distinct variables: it has the form $\Box'_1 y_1/\Box_1 x_1, \ldots, \Box'_n y_n/\Box_n x_n$ where $y_1, \ldots, y_n$ are pairwise distinct. Pattern substitutions are bijections. Because they are injective, an equation of the form $X[\theta] = N$ has at most one solution if θ is a (linear-changing) pattern substitution, namely $\theta^{-1}N$. However, it may have no solution when N contains variables not present in θ.

For linear-changing pattern substitutions the existence of the inverse applied to a term is subject to some conditions on the occurrences of variables. We say that a variable x occurs in a *persistent position* (resp. *affine position*, *linear position*) in a term N if it occurs in N inside a term of the form $!N'$ (resp. $@N'$, ${\downarrow}N'$).

Lemma 3 (Inversion [23]). *Let T be either an expression, a trace, a spine or a term, and θ a linear-changing pattern substitution. There exists T' such that $T \equiv \theta T'$ iff the following conditions hold:*

- *for every $!y/{\downarrow}x \in \theta$, $!y$ occurs exactly once in T in a linear position;*
- *for every $!y/@x \in \theta$, $!y$ occurs at most once in T in a linear or affine position;*
- *for every $@y/{\downarrow}x \in \theta$, $@y$ occurs exactly once in T in a linear position.*

Logic variables:

$$\frac{X :: \Delta_X \vdash \Pi\Delta'.a{\cdot}!S' \in \Psi \qquad \Delta_1 \vdash \theta : \Delta_X \qquad \Delta_2 \vdash S : \theta\Delta' \qquad P \equiv a{\cdot}!\theta S'[S/\Delta']}{\Delta_1 \bowtie \Delta_2 \vdash X[\theta]{\cdot}S : P}$$

$$\frac{X :: \Delta_X \vdash \Pi\Delta'.\{\Delta''\} \in \Psi \qquad \Delta_1 \vdash \theta : \Delta_X \qquad \Delta_2 \vdash S \Leftarrow \theta\Delta' \qquad \Delta_3 \equiv \Delta''[S/\Delta']}{\Delta_0 \bowtie \Delta_1 \bowtie \Delta_2 \vdash \{\Delta_3\}{\leftarrow}X[\theta]{\cdot}S : \Delta_0, \Delta_3}$$

Substitutions:

$$\frac{}{@\Delta \vdash \cdot : \cdot} \qquad \frac{\Delta_0 \vdash \theta : \Delta_2 \qquad \Box_1\Delta_1 \vdash N \Leftarrow A}{\Delta_0 \bowtie \Box_1\Delta_1 \vdash \theta, \Box_1 N/\Box_2 x : \Delta_2, \Box_2 x{:}A}$$

Figure 3. Typing rules for substitutions and logic variables

2.4 Example

As a running example, we consider a formalization of the asynchronous π-calculus with correspondence assertions [27], taken from [25]. We use a deep embedding to avoid the use of embedded clauses (which would require using steps headed by variables). The syntax of processes is given by:

$$\begin{array}{rl} Q ::= \mathbf{0} \mid (Q_1|Q_2) \mid\ !Q \mid (\nu x).Q \mid Q_1 + Q \\ \mid x(y).Q \mid x\langle y\rangle \mid \mathsf{begin}\, L; Q \mid \mathsf{end}\, L; Q \end{array}$$

where correspondence assertions are given by the constructors begin and end.

CLF allows representations using higher-order abstract syntax. In the case of the π-calculus, we use types pr (process), nm (name), and label, and the following signature constants to represent processes. As anticipated, we curry declarations and terms for readability. We also omit unused variables.

$$\begin{array}{rl} \mathsf{stop} : & \mathsf{pr} \\ \mathsf{choose} : & \mathsf{pr} \to \mathsf{pr} \to \mathsf{pr} \\ \mathsf{par} : & \mathsf{pr} \to \mathsf{pr} \to \mathsf{pr} \\ \mathsf{out} : & \mathsf{nm} \to \mathsf{nm} \to \mathsf{pr} \\ \mathsf{repeat} : & \mathsf{pr} \to \mathsf{pr} \\ \mathsf{inp} : & \mathsf{nm} \to (\mathsf{nm} \to \mathsf{pr}) \to \mathsf{pr} \\ \mathsf{new} : & (\mathsf{nm} \to \mathsf{pr}) \to \mathsf{pr} \\ \mathsf{begin} : & \mathsf{label} \to \mathsf{pr} \to \mathsf{pr} \\ \mathsf{end} : & \mathsf{label} \to \mathsf{pr} \to \mathsf{pr} \end{array}$$

The process stop corresponds to $(\mathbf{0})$ and represents a finished computation; $\mathsf{par}\, Q_1\, Q_2$ corresponds to $(Q_1|Q_2)$ and represents a concurrent execution of Q_1 and Q_2; $\mathsf{repeat}\, Q$ corresponds to $!Q$ and represents a process that can create copies of itself; $\mathsf{new}\,(\lambda x.Q)$ corresponds to $\nu x.Q$ and represents a process that creates a new name x that can be used in Q (note the use of higher-order abstract syntax to represent names); $\mathsf{choose}\, Q_1\, Q_2$ encodes $Q_1 + Q_2$ and represents a non-deterministic choice between Q_1 and Q_2; $\mathsf{inp}\, x\, (\lambda y.Q)$ and $\mathsf{out}\, x\, y$ correspond to $x(y).Q$ and $x\langle y\rangle$, respectively, and represent communication between processes; finally, $\mathsf{begin}\, L\, Q$ and $\mathsf{end}\, L\, Q$ represent correspondence assertions that have to be matched in well-formed processes [27].

The operational semantics is modeled by the type constructor $\mathsf{run} : \mathsf{pr} \to \mathsf{type}$. The process state is represented in the context as a sequence of running processes of the form $\mathsf{run}\, P$ and actions that transform the state.

Examples of the actions representing the operational semantics are the following:

$$\begin{aligned}
\mathsf{ev\_stop} &: \mathsf{run}\,\mathsf{stop} \multimap \{\cdot\}\\
\mathsf{ev\_par} &: \mathsf{run}\,(\mathsf{par}\,Q_1\,Q_2) \multimap \{@\mathsf{run}\,Q_1, @\mathsf{run}\,Q_2\}\\
\mathsf{ev\_new} &: \mathsf{run}\,(\mathsf{new}(\lambda !x.P\,x)) \multimap \{!x, @\mathsf{run}(P\,x)\}\\
\mathsf{ev\_repeat} &: \mathsf{run}\,(\mathsf{repeat}\,Q) \multimap \{!\mathsf{run}\,Q\}\\
\mathsf{ev\_sync} &: \mathsf{run}\,(\mathsf{out}\,X\,Y) \multimap \mathsf{run}\,(\mathsf{inp}\,X\,(\lambda y.Q\,y))\\
&\multimap \{@\mathsf{run}\,(Q\,Y)\}\\
\mathsf{ev\_begin} &: \mathsf{run}\,(\mathsf{begin}\,L\,Q) \multimap \{@\mathsf{run}\,Q\}
\end{aligned}$$

At run-time, the implicitly Π-quantified variables denoted with capital letters are used as atomic logic variables.

A process state is modeled by declarations of type $@\mathsf{run}\,Q$ and $!\mathsf{run}\,Q$; the latter represents a process that can be executed repeatedly. The objects of type $\mathsf{run}\,Q \multimap \{\cdot\}$ represent computations starting from a process Q. Computations that only differ in the order of independent steps are represented by the same object in CLF. We illustrate execution with an example: consider the process $\nu x.x(y).Q(y)|x\langle x\rangle$, represented by $P \equiv \mathsf{new}(\lambda x.\mathsf{par}(\mathsf{inp}\,x\,(\lambda y.Q(y)))(\mathsf{out}\,x\,x))$. Assuming $p : \mathsf{run}\,P$, an execution of this process is the following:

$$E_P \equiv \{\mathsf{let}\begin{pmatrix}\{!x, p'\}\leftarrow\mathsf{ev\_new}\,p;\\ \{p_1, p_2\}\leftarrow\mathsf{ev\_par}\,p';\\ \{p_1, P_2\}\leftarrow\mathsf{ev\_sync}\,p_1\,p_2;\end{pmatrix}\mathsf{in}\ \cdot\}$$

We will demonstrate our matching algorithm on computations drawn from this language below.

3. Matching

Given two objects T_1 and T_2 of the same syntactic class (traces, terms, expressions, or spines) such that T_2 is ground (i.e., does not contain logic variables), the matching problem tries to find an assignment σ for the logic variables in T_1 such that $\sigma T_1 \equiv T_2$. We write $T_1 \overset{?}{=} T_2$ to denote a matching problem. Matching is a well-understood problem for terms, spines and types, as they have a simple equational theory [15], but not so much in the case of traces with permutations. This is what we will be focusing on.

Matching on traces is inherently non-deterministic. For example, the equation

$$\{\mathsf{let}\begin{pmatrix}\{\cdot\}\leftarrow X;\\ \{\cdot\}\leftarrow Y\end{pmatrix}\mathsf{in}\ \cdot\} \overset{?}{=} \{\mathsf{let}\begin{pmatrix}\{\cdot\}\leftarrow c_1;\\ \dots;\\ \{\cdot\}\leftarrow c_n\end{pmatrix}\mathsf{in}\ \cdot\}$$

has 2^n solutions: it encodes the problem of partitioning the multiset $\{c_1, \dots, c_n\}$ into the (disjoint) union of multisets X and Y.

Assuming that all logic variables in a trace matching problem are applied to (linear-changing) pattern substitutions, matching is decidable: in $\epsilon_1 \overset{?}{=} \epsilon_2$, any solution instantiates the monadic logic variables in ϵ_1 to subtraces of ϵ_2. Since there are only finitely many subtraces, one can try all possible dependency-preserving partitions of ϵ_2 among these monadic logic variables; a solution is found if the interface of each subtrace matches the interface of the logic variable and the inverse of the pattern substitution can be applied. Clearly, this approach is extremely inefficient.

We present an algorithm for solving the matching problem in the presence of (at most) one monadic logic variable, where every logic variable is applied to a (linear-changing) pattern substitution. This restriction suffices for many reasoning tasks of interest, for example monitoring a computation and some program transformations. Lifting this restriction is the topic of on-going work [20].

A matching problem on traces can then be expressed by an equation of the form

$$\begin{pmatrix}\delta_1; \dots; \delta_k;\\ \{\Delta\}\leftarrow X[\theta];\\ \delta_{k+1}; \dots; \delta_n\end{pmatrix} \overset{?}{=} (\delta'_1; \dots; \delta'_m) \qquad (*)$$

where δ_i, δ'_i have the form $\{\Delta\}\leftarrow c{\cdot}S$, where S might contain atomic logic variables. Without loss of generality, the algorithm fixes the order of the left-hand side of this equation and permutes independent steps on the right-hand side as needed. The algorithm proceeds by matching individual steps from each end of the trace. To match, two steps must have the same head, use the same resources, and produce resources that are used in the same way. Matching pairs of steps are removed from the traces. The process is repeated, matching two steps at the beginning or at the end of the trace, until a problem of the form

$$(\{\Delta\}\leftarrow X[\theta]) \overset{?}{=} \epsilon$$

is obtained. The solution is then $X \leftarrow \theta^{-1}(\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\})$, if this term is well typed and the inverse substitution θ^{-1} can be applied. If this is not the case, then either the steps were matched in the wrong order (meaning that it is necessary to backtrack and try a different permutation of the right-hand side), or if all possible orders have been tried, the problem has no solution.

Example We continue with the π-calculus example from the previous section. We consider the operational semantics from Gordon and Jeffrey [8] based on event sequences. An event is either $\mathsf{begin}\,L$ (where L is a label), $\mathsf{end}\,L$, tint for internal actions, or $\mathsf{gen}\,N$, where N is a name (corresponding to new names). A computation is represented by $P \xrightarrow{s} P'$, where s is an event sequence.

Event sequences are represented in CLF by a type ev and the following signature constants:

$$\begin{aligned}
\mathsf{snil} &: \mathsf{ev}\\
\mathsf{sint} &: \mathsf{ev} \to \mathsf{ev}\\
\mathsf{sgen} &: (\mathsf{nm} \to \mathsf{ev}) \to \mathsf{ev}\\
\mathsf{sbegin} &: \mathsf{label} \to \mathsf{ev} \to \mathsf{ev}\\
\mathsf{send} &: \mathsf{label} \to \mathsf{ev} \to \mathsf{ev}
\end{aligned}$$

The event sequence snil represents the empty sequence, sint represents an internal event (synchronization or choosing a process in a non-deterministic choice), sgen represents the event of creating a name, and finally sbegin and send represent the correspondence assertions.

The operational semantics is given by an abstraction predicate that relates an execution of a process with an event sequence. It is represented in CLF by a type family $\mathsf{abst} : \{\cdot\} \to \mathsf{ev} \to \mathsf{type}$. Some of the constructors of this type family are the following:

$\mathsf{abst\_sync}$:

$$\begin{aligned}
&\mathsf{abst}\,\{\mathsf{let}\begin{pmatrix}\{@r\}\leftarrow\mathsf{ev\_sync}\,@R_1\,@R_2;\\ \{\cdot\}\leftarrow E\ r\end{pmatrix}\mathsf{in}\ \cdot\}\ (\mathsf{sint}\,!s)\\
&\leftarrow (\Pi r.\mathsf{abst}\,(E\,@r)\,!s)
\end{aligned}$$

$\mathsf{abst\_begin}$:

$$\begin{aligned}
&\mathsf{abst}\,\{\mathsf{let}\begin{pmatrix}\{@r\}\leftarrow\mathsf{ev\_begin}\,!L\,@R;\\ \{\cdot\}\leftarrow E\ r\end{pmatrix}\mathsf{in}\ \cdot\}\ (\mathsf{sbegin}\,!L\,!s)\\
&\leftarrow (\Pi r.\mathsf{abst}\,(E\,@r)\,!s)
\end{aligned}$$

$\mathsf{abst\_new}$:

$$\begin{aligned}
&\mathsf{abst}\,\{\mathsf{let}\begin{pmatrix}\{!x, @r_1\}\leftarrow\mathsf{ev\_new}\,@R;\\ \{\cdot\}\leftarrow E\ x, r_1\end{pmatrix}\mathsf{in}\ \cdot\}\ (\mathsf{sgen}\,(\lambda !x.s\,x)\\
&\leftarrow (\Pi x, r_1.\mathsf{abst}\,(E\ x, r_1 c)\,(s\,x))
\end{aligned}$$

where $A \leftarrow B$ is a synonym of $A \rightarrow B$. The variable E is used, at run-time, as a monadic logic variable, while L, R, R_1, and R_2 are used as atomic logic variables.

The constructor $\mathsf{abst\_sync}$ relates an execution that starts with a synchronization step with the event sequence $\mathsf{sint}\, s$ if the rest of the execution is abstracted by s. Similarly, $\mathsf{abst\_begin}$ (resp. $\mathsf{abst\_end}$) treat the case where the execution starts with a begin (resp. end) step. Since independent steps in an execution can be reordered, an execution can be related to several event sequences.

We can view this definition as a Prolog-style backward chaining program where the first argument of abst is an input and the second is an output. Running this program with a query of the form $\mathsf{abst}\, e\, X$, where e is a ground computation and X is a logic variable, reduces to solving matching problems of the form (*). The variable E in each clause represents an unknown computation (a trace in CLF). For example, consider the process P and execution E_P given in Section 2.4. Querying $\mathsf{abst}\, E_P\, X$ for X would mean trying all the constructors of the type family abst. Trying $\mathsf{abst\_new}$ would trigger solving the following matching problem:

$$E_P \overset{?}{=} \{\mathsf{let}\ \{!x, @r_1\}{\leftarrow}\mathsf{ev\_new}\, @R; \{\cdot\}{\leftarrow}E\ x, r_1\ \mathsf{in}\ \cdot\}$$

where R and E are logic variables. Matching succeeds yielding a value for R and E, reducing the problem to solve a query of the form $\mathsf{abst}\, E\, Y$ for Y, where E is the value returned by matching.

Renamings The main issues in designing a matching algorithm for traces is how to deal with variables introduced in the trace. Recall that in an expression $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$ the context Δ does not necessarily list all the persistent and affine variables in the output interface $\epsilon\bullet$ of ϵ. When matching two expressions, which such unmentioned variables correspond to which is initially unknown but revealed incrementally as the two embedded traces are examined. Rather than guessing this correspondence a priori, we rely on the notion of *renaming* to delay it until matching step pairs force it, incrementally. A renaming is a modality-preserving substitution of variables by fresh variables:

$$\textit{Renamings:} \quad \varphi, \rho \quad ::= \quad \cdot \ \mid\ \varphi, \square x/\square y$$

We write $\epsilon\{\varphi\}$ and $\Delta\{\varphi\}$ for the application of the renaming φ to the free variables in a trace ϵ and a context Δ, respectively. We omit the straightforward definition. Because variables are always renamed to fresh variables, it will be convenient to extend this definition to bound variables as well: in this way we can maintain the correspondence between the variable names in an expression $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$ even after ϵ and Δ have been separated. For example, we will have $(\{\downarrow y\}{\leftarrow}c{\cdot}\downarrow x)\{\downarrow z/\downarrow y\} = (\{\downarrow z\}{\leftarrow}c{\cdot}\downarrow x)$.

Design of the matching algorithm Matching traces involves picking an appropriate permutation of one of the traces and finding a renaming that identifies the variables introduced by the trace. We will now illustrate some of the design choices we made through examples.

Intuitively, the algorithm is based on the judgment of the form $\Delta \vdash \epsilon_1 \overset{?}{=} \epsilon_2 \mapsto \sigma$ which attempts to match ϵ_1 against ϵ_2 (both well typed under context Δ) and results in the assignment σ (we will update this judgment shortly). Since both traces are well typed in the same context, input variables are considered *fixed* and cannot be renamed. On the other hand, variables introduced in the trace need to be matched. Consider the matching problem

$$\begin{pmatrix} \{\downarrow y_1\}{\leftarrow}c{\cdot}\downarrow x; \\ \{\downarrow z_1\}{\leftarrow}c{\cdot}\downarrow x'; \\ \{\downarrow p\}{\leftarrow}X[\downarrow y_1, \downarrow z_1] \end{pmatrix} \overset{?}{=} \begin{pmatrix} \{\downarrow y_2\}{\leftarrow}c{\cdot}\downarrow x; \\ \{\downarrow z_2\}{\leftarrow}c{\cdot}\downarrow x'; \\ \{\downarrow p\}{\leftarrow}c{\cdot}\downarrow y_2\, \downarrow z_2 \end{pmatrix}$$

where both terms are well typed in a context that contains x and x'. Because variables $\downarrow x$ and $\downarrow x'$ are defined in the outside context, their name is fixed, while variables $\downarrow y_1$, $\downarrow y_2$, $\downarrow z_1$, and $\downarrow z_2$ are internal to the trace and can thus be (α-)renamed. Here, $\{\downarrow y_1\}{\leftarrow}c{\cdot}\downarrow x$ must correspond to $\{\downarrow y_2\}{\leftarrow}c{\cdot}\downarrow x$, while we cannot match $\{\downarrow y_1\}{\leftarrow}c{\cdot}\downarrow x$ against $\{\downarrow z_2\}{\leftarrow}c{\cdot}\downarrow x'$ as $\downarrow x$ and $\downarrow x'$ are different variables and cannot be renamed.

Similarly, $\{\downarrow z_1\}{\leftarrow}c{\cdot}\downarrow x'$ must correspond to $\{\downarrow z_2\}{\leftarrow}c{\cdot}\downarrow x'$. Matching these two pairs of steps renames $\downarrow y_1$ and $\downarrow y_2$ to some fresh variable $\downarrow y$, and $\downarrow z_1$ and $\downarrow z_2$ to $\downarrow z$, reducing the problem to

$$\left(\{\downarrow p\}{\leftarrow}X[\downarrow y, \downarrow z]\right) \overset{?}{=} \left(\{\downarrow p\}{\leftarrow}c{\cdot}\downarrow y\, \downarrow z\right)$$

The only solution for $X[\downarrow y, \downarrow z]$ is $\{\mathsf{let}\ \{\downarrow p\}{\leftarrow}c{\cdot}\downarrow y\, \downarrow z\ \mathsf{in}\ \downarrow p\}$.

The above example shows how the algorithm proceeds when matching two steps at the beginning of a trace. Repeatedly matching steps at the beginning of the trace reduces the problem to

$$\left(\{\Delta\}{\leftarrow}X[\theta]; \epsilon_1\right) \overset{?}{=} \epsilon_2$$

where no step in ϵ_1 contains monadic logic variables. The variable X corresponds to a subtrace of ϵ_2. Instead of guessing the right value for X, we can match each step in ϵ_1 against a step in ϵ_2.

Let us consider the following problem involving matching steps at the end of a trace (for simplicity deprived of logic variables):

$$\{\mathsf{let} \begin{pmatrix} \{!x_1\}{\leftarrow}c; \\ \{!x_2\}{\leftarrow}c \end{pmatrix} \mathsf{in}\ \cdot\} \overset{?}{=} \{\mathsf{let} \begin{pmatrix} \{!y_1\}{\leftarrow}c; \\ \{!y_2\}{\leftarrow}c \end{pmatrix} \mathsf{in}\ \cdot\}$$

Both expressions are α-equivalent, so the matching algorithm should succeed. The problem is reduced to matching the inner traces, but note that their output interfaces are different. Matching steps at the end of two traces involves matching the output of steps, incrementally building a renaming between both traces. However, we should be careful not to rename variables twice. For example, in a problem of the form

$$\begin{pmatrix} \epsilon_1; \\ \{\cdot\}{\leftarrow}c{\cdot}!x_1; \\ \{\cdot\}{\leftarrow}c{\cdot}!x_1 \end{pmatrix} \overset{?}{=} \begin{pmatrix} \epsilon_2; \\ \{\cdot\}{\leftarrow}c{\cdot}!y_1; \\ \{\cdot\}{\leftarrow}c{\cdot}!y_2 \end{pmatrix}$$

matching $\{\cdot\}{\leftarrow}c{\cdot}!x_1$ against $\{\cdot\}{\leftarrow}c{\cdot}!y_2$ renames $!x_1$ and $!y_2$ to a fresh $!z$ (assuming both are introduced in the trace), reducing the problem to

$$\begin{pmatrix} \epsilon_1\{!z/!x_1\}; \\ \{\cdot\}{\leftarrow}c{\cdot}!z \end{pmatrix} \overset{?}{=} \begin{pmatrix} \epsilon_2\{!z/!y_2\}; \\ \{\cdot\}{\leftarrow}c{\cdot}!y_1 \end{pmatrix}$$

Now matching $\{\cdot\}{\leftarrow}c{\cdot}!z$ against $\{\cdot\}{\leftarrow}c{\cdot}!y_1$ would identify $!z$ and $!y_1$, but this is wrong since $!z$ is already defined in $\epsilon_2\{!z/!y_2\}$. To avoid this issue we mark variables introduced in the trace and remove the mark once they have been renamed (a marked variable is denoted with $\underline{x}$). For example, the equation over expressions

$$\{\mathsf{let} \begin{pmatrix} \{!x_1\}{\leftarrow}c_1; \\ \{!x_2\}{\leftarrow}c_2{\cdot}!x_1 \end{pmatrix} \mathsf{in}\ \cdot\} \overset{?}{=} \{\mathsf{let} \begin{pmatrix} \{!y_1\}{\leftarrow}c_1; \\ \{!y_2\}{\leftarrow}c_2{\cdot}!y_1 \end{pmatrix} \mathsf{in}\ \cdot\}$$

reduces to the following trace equation, where all introduced variables have been marked:

$$\begin{pmatrix} \{!\underline{x_1}\}{\leftarrow}c_1; \\ \{!\underline{x_2}\}{\leftarrow}c_2{\cdot}!\underline{x_1} \end{pmatrix} \overset{?}{=} \begin{pmatrix} \{!\underline{y_1}\}{\leftarrow}c_1; \\ \{!\underline{y_2}\}{\leftarrow}c_2{\cdot}!\underline{x_2} \end{pmatrix}$$

Matching $\{!\underline{x_1}\}{\leftarrow}c_1$ against $\{!\underline{y_1}\}{\leftarrow}c_1$ identifies $!\underline{x_1}$ with $!\underline{y_1}$ renaming both to a new unmarked variable, say $!z$, and reducing the problem to

$$\left(\{!\underline{x_2}\}{\leftarrow}c_2{\cdot}!z\right) \overset{?}{=} \left(\{!\underline{y_2}\}{\leftarrow}c_2{\cdot}!z\right)$$

In general, matching a step $\{\Delta_1\}{\leftarrow}c{\cdot}S_1$ against $\{\Delta_2\}{\leftarrow}c{\cdot}S_2$ implies matching S_1 against S_2 and Δ_1 against Δ_2. The latter problem is defined by the judgment

$$\Delta_1 \overset{?}{=} \Delta_2 \mapsto \varphi_1; \varphi_2$$

meaning that $\Delta_1\{\varphi_1\} \equiv \Delta_2\{\varphi_2\}$. The invariant of this judgment ensures that the domains of φ_1 and φ_2 contain only marked variables and the codomains contain only unmarked variables.

The matching judgment for traces is modified to return the renamings that match the variables introduced in the trace:

$$\Delta \vdash \epsilon_1 \overset{?}{=} \epsilon_2 \mapsto \sigma; \varphi_1; \varphi_2$$

meaning that $(\sigma\epsilon_1)\{\varphi_1\} \equiv \epsilon_2\{\varphi_2\}$.

An invariant of the algorithm is that ϵ_1 and ϵ_2 are well typed under Δ. In the presence of affine and persistent hypotheses, it is necessary to keep track of the type of each variable, as unused pattern steps cannot be matched. Let us illustrate the problem with the following matching equation:

$$\big(\{!\underline{x_1}\}{\leftarrow}X[!x]\big) \overset{?}{=} \begin{pmatrix}\{!\underline{y_1}\}{\leftarrow}c_1{\cdot}!x;\\ \{!\underline{y_2}\}{\leftarrow}c_2{\cdot}!x\end{pmatrix}$$

As the output of the trace on the right ($!\underline{y_1}$ and $!\underline{y_2}$) is not used, they are still marked and have no relation to the output of the left trace ($!\underline{x_1}$). The solution to this matching problem would be to assign to X the trace on the right hand side: $X \leftarrow \{\mathsf{let}\ \{!\underline{y_1}\}{\leftarrow}c_1{\cdot}!x; \{!\underline{y_2}\}{\leftarrow}c_2{\cdot}!x\ \mathsf{in}\ \bigcirc\}$. However, the output $\bigcirc$ is missing. This problem has a solution only if $!x_1$ has the same type as either $!y_1$ or $!y_2$.

The example above is a particular case of the matching problem

$$\big(\{\Delta\}{\leftarrow}X[\theta]\big) \overset{?}{=} \epsilon$$

This problem has a solution if we can match the context Δ with the output context of ϵ. Assume that both traces are well typed in a context Δ_0 and $\Delta_0 \vdash \epsilon : \Delta_2$. There is a solution for X if Δ is a valid output interface for ϵ, up to renaming. In other words, if there exists renamings φ_1 and φ_2 such that $\Delta_2\{\varphi_2\} \preccurlyeq \Delta\{\varphi_1\}$.

Matching in full CLF, where expressions have the form $\{\mathsf{let}\ \epsilon_0\ \mathsf{in}\ S\}$, is a much more difficult problem than the one considered here. This is because substituting a trace can trigger reductions [20]. The CLF substitution rule

$$[X \leftarrow \{\mathsf{let}\ \epsilon_0\ \mathsf{in}\ S\}](\epsilon_1; \{\Delta\}{\leftarrow}X[\theta]; \epsilon_2) = (\epsilon_1; \theta\epsilon_0; [S/\Delta]\epsilon_2)$$

does not necessarily produce a canonical term, as $[S/\Delta]\epsilon_2$ might have redexes. These reductions can have the effect of removing steps in ϵ_2. Indeed, $[S/\Delta]\epsilon_2$ could even reduce to the empty trace.

Some of the issues described above are related to the use of names to represent variables. A natural question is what happens if we use de Bruijn indices [7] for representing variables. However, de Bruijn indices are not a good representation for concurrent traces. First, any permutation of independent traces involves shifts. Second, a monadic logic variable stands for a shift of unknown size. Put together, this means that, when matching two steps at the end of a trace, the indices on both sides have no obvious relation between them as they depend on the order of the previous steps.

The algorithm Our matching algorithm relies on the following judgments [20]:

Contexts:	$\Delta_1 \overset{?}{=} \Delta_2 \mapsto \varphi_1; \varphi_2$
	$\Delta_1 \succcurlyeq \Delta_2 \mapsto \varphi_1; \varphi_2$
Spines:	$(\Delta_1 \vdash S_1) \overset{?}{=} (\Delta_2 \vdash S_2) \mapsto \sigma; \varphi_1; \varphi_2$
Terms:	$(\Delta_1 \vdash N_1) \overset{?}{=} (\Delta_2 \vdash N_2) \mapsto \sigma; \varphi_1; \varphi_2$
	$(\Delta_1 \vdash H_1{\cdot}S_1) \overset{?}{=} (\Delta_2 \vdash H_2{\cdot}S_2) \mapsto \sigma; \varphi_1; \varphi_2$
Expressions:	$\Delta \vdash E_1 \overset{?}{=} E_2 \mapsto \sigma$
Traces:	$\Delta \vdash \epsilon_1 \overset{?}{=} \epsilon_2 \mapsto \sigma; \varphi_1; \varphi_2$

The first matching context judgment is used for contexts introduced by steps, while the second judgment is used for the output contexts in equations of the form $(\{\Delta\}{\leftarrow}X[\theta]) \overset{?}{=} \epsilon$. They are defined in Figure 4. The output renamings have the property that the domain contains marked variables while the codomain contains unmarked variables (we call a renaming with this property *unmarked*). This is an invariant of the matching algorithm (easily checked by induction on the rules). Rules ctx-eq-* deal with context equality. A judgment $\Delta_1 \overset{?}{=} \Delta_2 \mapsto \varphi_1; \varphi_2$ is derivable if Δ_1 and Δ_2 have the same length, and variables in corresponding positions are either both marked (rule ctx-eq-mark) or both unmarked (rule ctx-eq-unmark).

The judgment $\Delta_1 \succcurlyeq \Delta_2 \mapsto \varphi_1; \varphi_2$, defined by the rules ctx-weak-*, is derivable when every declaration in Δ_1 has a corresponding declaration in Δ_2 (with a matching mark). It is defined by induction on the structure of Δ_1. If Δ_1 is empty (rule ctx-weak-empty), then Δ_2 can only contain affine and persistent variables. This means that all linear declarations in Δ_1 must be matched against a linear declaration in Δ_2. A marked variable in Δ_1 must be matched against a marked variable in Δ_2 (rule ctx-weak-mark) and similarly for unmarked variables (rule ctx-weak-unmark).

When matching spines (and terms) each is well typed in its own context. This is necessary for matching steps at the end of a trace: in the matching equation $\epsilon_1; \{\Delta_1\}{\leftarrow}c{\cdot}S_1 \overset{?}{=} \epsilon_2; \{\Delta_2\}{\leftarrow}c{\cdot}S_2$, spines S_1 and S_2 are well typed in different contexts that contain the variables introduced by ϵ_1 and ϵ_2, respectively. The algorithm also returns renamings between the marked variables in S_1 and S_2 that are propagated to ϵ_1 and ϵ_2, respectively. Matching on spines and terms is defined in Figure 5.

The judgment $(\Delta_1 \vdash S_1) \overset{?}{=} (\Delta_2 \vdash S_2) \mapsto \sigma; \varphi_1; \varphi_2$ is given by rules dec-sp-* and defined by induction on the structure of the spines. In rule dec-sp-nil we require both spines to be empty (to ensure that spines have the same length) and both context to have no linear hypotheses (to maintain the typing invariant). In the case of a non-empty spine, the first term of both spines is matched and the results are propagated to the rest of the two spines.

The judgments $(\Delta_1 \vdash N_1) \overset{?}{=} (\Delta_2 \vdash N_2) \mapsto \sigma; \varphi_1; \varphi_2$ and $(\Delta_1 \vdash H_1{\cdot}S_1) \overset{?}{=} (\Delta_2 \vdash H_2{\cdot}S_2) \mapsto \sigma; \varphi_1; \varphi_2$ are given by the rule dec-term-lam and the rules dec-head-*, respectively. Note in rule dec-term-lam that the context introduced by the abstraction is added without marking variables. Marked variables are only used for variables introduced by a trace.

In rule dec-head-const both terms must have the same constant at the head, so the problem reduces to matching both spines. Rule dec-head-var-unmark is similar for the case when both terms have the same unmarked variable at the head. In rule dec-head-var-mark both terms contain marked heads; the heads are equated and the spines are matched.

Finally, rule dec-head-lvar considers the case where the left term has a logic variable at the head. Recall that we assume that logic variables have base types, so that the right side must be of the form $H{\cdot}S$ (actually $\widehat{\lambda}(\cdot).\, H{\cdot}S$). The solution is basically $X \leftarrow \theta^{-1}(H{\cdot}S)$ (if defined), but since both terms are typed in different contexts, we first need to match the contexts. Note that Δ_1 may contain affine and persistent hypotheses (occurring in θ) that are not matched in Δ_2. However, every linear hypotheses in Δ_1 and Δ_2 must be matched. We assume that the rule is applicable only if the conditions of Lemma 3 are satisfied.

Matching on expressions and traces is defined in Figure 6. We write $\underline{\epsilon}$ for the trace obtained by marking every variable introduced in ϵ. We write $\overline{\Delta}$ for the context obtained by removing all marked variables from Δ.

Matching on expression is given by rule dec-expr. The problem is reduced to matching traces by renaming the part of the output interfaces that is collected in the expression. This does not imply

$$\frac{}{\cdot \overset{?}{=} \cdot \mapsto \cdot;\cdot}\ \textsf{ctx-eq-empty} \qquad \frac{\Delta_1 \overset{?}{=} \Delta_2 \mapsto \varphi_1;\varphi_2}{\square x{:}A, \Delta_1 \overset{?}{=} \square x{:}A, \Delta_2 \mapsto \varphi_1;\varphi_2}\ \textsf{ctx-eq-unmark}$$

$$\frac{[\square x/\square \underline{x_1}]\Delta_1 \overset{?}{=} [\square x/\square \underline{x_2}]\Delta_2 \mapsto \varphi_1;\varphi_2}{\square \underline{x_1}{:}A, \Delta_1 \overset{?}{=} \square \underline{x_2}{:}A, \Delta_2 \mapsto (\varphi_1, \square x/\square \underline{x_1});(\varphi_2, \square x/\square \underline{x_2})}\ \textsf{ctx-eq-mark}$$

$$\frac{}{\cdot \succcurlyeq @\Delta \mapsto \cdot;\cdot}\ \textsf{ctx-weak-empty} \qquad \frac{\square x{:}A \in \Delta_2 \qquad \Delta_1 \succcurlyeq (\Delta_2 \mathbin{\backslash\!\!\backslash} x) \mapsto \varphi_1;\varphi_2}{\square x{:}A, \Delta_1 \succcurlyeq \Delta_2 \mapsto \varphi_1;\varphi_2}\ \textsf{ctx-weak-unmark}$$

$$\frac{\square \underline{x_2}{:}A \in \Delta_2 \qquad \Delta_1\{\square x/\square \underline{x_1}\} \succcurlyeq (\Delta_2 \mathbin{\backslash\!\!\backslash} x_2)\{\square x/\square \underline{x_2}\} \mapsto \varphi_1;\varphi_2}{\square \underline{x_1}{:}A, \Delta_1 \succcurlyeq \Delta_2 \mapsto (\varphi_1, \square x/\square \underline{x_1});(\varphi_2, \square x/\square \underline{x_2})}\ \textsf{ctx-weak-mark}$$

Figure 4. Matching on contexts

$$\frac{}{(\cdot \vdash \cdot) \overset{?}{=} (\cdot \vdash \cdot) \mapsto \cdot;\cdot;\cdot}\ \textsf{dec-sp-nil}$$

$$\frac{\begin{array}{c}(\square_1, \square_2) \in \{(@, !), (@, @), (\downarrow, \downarrow)\} \\ (\Delta_1'' \vdash N_1) \overset{?}{=} (\Delta_2'' \vdash N_2) \mapsto \sigma;\varphi_1;\varphi_2 \qquad (\sigma\Delta_1'\{\varphi_1\} \vdash \sigma\varphi_1 S_1) \overset{?}{=} (\Delta_2'\{\varphi_2\} \vdash \varphi_2 S_2) \mapsto \sigma';\varphi_1';\varphi_2'\end{array}}{(\Delta_1' \bowtie \square_2\Delta_1'' \vdash \square_1 N_1, S_1) \overset{?}{=} (\Delta_2' \bowtie \square_2\Delta_2'' \vdash \square_1 N_2, S_2) \mapsto \sigma'\sigma;\varphi_1'\varphi_1;\varphi_2'\varphi_2}\ \textsf{dec-sp-cons}$$

$$\frac{(\Delta_1, \Delta' \vdash H_1{\cdot}S_1) \overset{?}{=} (\Delta_2, \Delta' \vdash H_2{\cdot}S_2) \mapsto \sigma;\varphi_1;\varphi_2}{(\Delta_1 \vdash \widehat{\lambda}\Delta'.\, H_1{\cdot}S_1) \overset{?}{=} (\Delta_2 \vdash \widehat{\lambda}\Delta'.\, H_2{\cdot}S_2) \mapsto \sigma;\varphi_1;\varphi_2}\ \textsf{dec-term-lam}$$

$$\frac{(\Delta_1 \vdash S_1) \overset{?}{=} (\Delta_2 \vdash S_2) \mapsto \sigma;\varphi_1;\varphi_2}{(\Delta_1 \vdash c{\cdot}S_1) \overset{?}{=} (\Delta_2 \vdash c{\cdot}S_2) \mapsto \sigma;\varphi_1;\varphi_2}\ \textsf{dec-head-const}$$

$$\frac{(\Delta_1 \mathbin{\backslash\!\!\backslash} x \vdash S_1) \overset{?}{=} (\Delta_2 \mathbin{\backslash\!\!\backslash} x \vdash S_2) \mapsto \sigma;\varphi_1;\varphi_2}{(\Delta_1 \vdash x{\cdot}S_1) \overset{?}{=} (\Delta_2 \vdash x{\cdot}S_2) \mapsto \sigma;\varphi_1;\varphi_2}\ \textsf{dec-head-var-unmark}$$

$$\frac{((\Delta_1 \mathbin{\backslash\!\!\backslash} \underline{x_1})\{x/\underline{x_1}\} \vdash S_1) \overset{?}{=} ((\Delta_2 \mathbin{\backslash\!\!\backslash} \underline{x_2})\{x/\underline{x_2}\} \vdash S_2) \mapsto \sigma;\varphi_1;\varphi_2}{(\Delta_1 \vdash \underline{x_1}{\cdot}S_1) \overset{?}{=} (\Delta_2 \vdash \underline{x_2}{\cdot}S_2) \mapsto \sigma;(\varphi_1, x/\underline{x_1});(\varphi_2, x/\underline{x_2})}\ \textsf{dec-head-var-mark}$$

$$\frac{\Delta_2 \succcurlyeq \Delta_1 \mapsto \varphi_1;\varphi_2}{(\Delta_1 \vdash X[\theta]) \overset{?}{=} (\Delta_2 \vdash H{\cdot}S) \mapsto (X \leftarrow (\varphi_1\theta)^{-1}\varphi_2(H{\cdot}S));\varphi_1;\varphi_2}\ \textsf{dec-head-lvar}$$

Figure 5. Matching on terms

that both traces have the same interface, as they might contain persistent and affine hypotheses that are not present in the context.

Matching on traces is given by the rules tr-*. In a matching equation of the form $\epsilon_1 \overset{?}{=} \epsilon_2$ we leave the order of ϵ_1 fixed, while we implicitly reorder ϵ_2 to match the order of ϵ_1. Rule tr-empty matches the empty trace on both sides. This rule is applicable only if the left trace does not contain monadic logic variables.

Rule tr-step-hd matches a step at the beginning of the traces. As explained above, we reorder the trace on the right side as needed (preserving dependencies). Both steps use the same constant at the head. The rule proceeds by matching the spines using the necessary part of the context $\overline{\Delta}$ (denoted $\overline{\Delta}|_{S_i}$). Note that S_1 and S_2 do not contain marked variables so matching returns the empty renaming. Then, the output context of the steps are matched. Matching proceeds with the rest of the trace. We add the output of the first step to the context to ensure that the rest of the trace is well typed.

Rule tr-step-tl matches a step at the end of the traces. The context necessary to type S_1 and S_2 is part of the output context of ϵ_1 and ϵ_2, respectively. Similar to tr-step-hd, the spines and output contexts of the step are matched, and the results are propagated to the rest of the trace. In this case, the context used to type the rest of the trace does not change.

Finally, rule tr-inst handles the case of a monadic logic variable. The input context on both sides is the same, but the output contexts might differ. Hence, we need to ensure that all variables in the output context on the left (Δ') are contained in the output on the right (Δ_2). The rule is applicable only if the inverse substitution θ^{-1} can be applied to $\{\text{let } \epsilon_2\{\varphi_2\} \text{ in } \Delta'\{\varphi_1\}\}$.

Correctness of the algorithm We outline a proof that the algorithm in Figures 4, 5, and 6 is sound and complete [20].

$$\frac{\overline{\Delta} \vdash \underline{\epsilon_1}\{\Delta'/\underline{\Delta_1}\} \overset{?}{=} \underline{\epsilon_2}\{\Delta'/\underline{\Delta_2}\} \mapsto \sigma; \varphi_1; \varphi_2}{\overline{\Delta} \vdash \{\mathsf{let}\ \epsilon_1\ \mathsf{in}\ \Delta_1\} \overset{?}{=} \{\mathsf{let}\ \epsilon_2\ \mathsf{in}\ \Delta_2\} \mapsto \sigma}\ \textsf{dec-expr}$$

$$\frac{}{\overline{\Delta} \vdash \diamond \overset{?}{=} \diamond \mapsto \cdot; \cdot; \cdot}\ \textsf{tr-empty}$$

$$\frac{\begin{array}{c}(\overline{\Delta}|_{S_1} \vdash S_1) \overset{?}{=} (\overline{\Delta}|_{S_2} \vdash S_2) \mapsto \sigma; \cdot; \cdot \qquad \sigma\Delta_1 \overset{?}{=} \Delta_2 \mapsto \varphi_1; \varphi_2 \\ \overline{\Delta} \bowtie \sigma\Delta_1\{\varphi_1\} \vdash \sigma\varphi_1\epsilon_1 \overset{?}{=} \varphi_2\epsilon_2 \mapsto \sigma'; \varphi_1'; \varphi_2'\end{array}}{\overline{\Delta} \vdash (\{\Delta_1\}\leftarrow c\cdot S_1; \epsilon_1) \overset{?}{=} (\{\Delta_2\}\leftarrow c\cdot S_2; \epsilon_2) \mapsto \sigma'\sigma; \varphi_1'\varphi_1; \varphi_2'\varphi_2}\ \textsf{tr-step-hd}$$

$$\frac{\begin{array}{c}\overline{\Delta} \vdash \epsilon_1 : \Delta_1' \qquad \overline{\Delta} \vdash \epsilon_2 : \Delta_2' \\ (\Delta_1'|_{S_1} \vdash S_1) \overset{?}{=} (\Delta_2'|_{S_2} \vdash S_2) \mapsto \sigma; \varphi_1; \varphi_2 \qquad \sigma\Delta_1 \overset{?}{=} \Delta_2 \mapsto \varphi_1'; \varphi_2' \\ \overline{\Delta} \vdash \sigma\epsilon_1\{\varphi_1\} \overset{?}{=} \epsilon_2\{\varphi_2\} \mapsto \sigma'; \varphi_1''; \varphi_2''\end{array}}{\overline{\Delta} \vdash (\epsilon_1; \{\Delta_1\}\leftarrow c\cdot S_1) \overset{?}{=} (\epsilon_2; \{\Delta_2\}\leftarrow c\cdot S_2) \mapsto \sigma'\sigma; \varphi_1''\varphi_1'\varphi_1; \varphi_2''\varphi_2'\varphi_2}\ \textsf{tr-step-tl}$$

$$\frac{\overline{\Delta_0} \vdash \epsilon : \Delta_2 \qquad \Delta' \succcurlyeq \Delta_2 \mapsto \varphi_1; \varphi_2}{\overline{\Delta_0} \vdash (\{\Delta'\}\leftarrow X[\theta]) \overset{?}{=} \epsilon \mapsto (X \leftarrow \theta^{-1}\{\mathsf{let}\ \epsilon\{\varphi_2\}\ \mathsf{in}\ \Delta'\{\varphi_1\}\}); \varphi_1; \varphi_2}\ \textsf{tr-inst}$$

Figure 6. Matching on traces

Before stating the results, we need some definitions. We say that a term (resp. assignment, context, expression, spine, trace) is *unmarked* if no free variable is marked. Recall that a renaming is unmarked if the domain contains only marked variables and its codomain contains only unmarked variables. It is easy to check that all renamings and assignments returned by the matching judgments are unmarked. In the proofs, we occasionally write $\Delta_0 \xrightarrow{\epsilon_1} \Delta_1 \ldots \xrightarrow{\epsilon_n} \Delta_n$ to mean that $\Delta_{i-1} \vdash \epsilon_i : \Delta_i$ for all $i = 1, \ldots, n$.

The following lemmas state the soundness of the various matching judgments [20].

Lemma 4 (Soundness of matching for contexts)**.**

- *If* $\Delta_1 \overset{?}{=} \Delta_2 \mapsto \varphi_1; \varphi_2$*, then* $\Delta_1\{\varphi_1\} \equiv \Delta_2\{\varphi_2\}$.
- *If* $\Delta_1 \succcurlyeq \Delta_2 \mapsto \varphi_1; \varphi_2$*, then* $\Delta_1\{\varphi_1\} \succcurlyeq \Delta_2\{\varphi_2\}$.

Proof. By induction on the given derivation. □

Lemma 5 (Soundness of matching for terms)**.**

- *Let* N_1 *and* N_2 *be well typed terms under contexts* Δ_1 *and* Δ_2*, respectively. If* $(\Delta_1 \vdash N_1) \overset{?}{=} (\Delta_2 \vdash N_2) \mapsto \sigma; \varphi_1; \varphi_2$*, then* $\Delta_1\{\varphi_1\} \preccurlyeq \Delta_2\{\varphi_2\}$*, and* $\sigma\varphi_1 N_1 = \varphi_2 N_2$.
- *Let* S_1 *and* S_2 *be well typed spines under contexts* Δ_1 *and* Δ_2*, respectively. If* $(\Delta_1 \vdash S_1) \overset{?}{=} (\Delta_2 \vdash S_2) \mapsto \sigma; \varphi_1; \varphi_2$*, then* $\Delta_1\{\varphi_1\} \preccurlyeq \Delta_2\{\varphi_2\}$*, and* $\sigma\varphi_1 S_1 = \varphi_2 S_2$.

Proof. By simultaneous induction on the given derivation. □

Lemma 6 (Soundness of matching for traces)**.**

- *If* $\overline{\Delta} \vdash E_1, E_2 \Leftarrow \{\Delta\}$ *and* $\Delta \vdash E_1 \overset{?}{=} E_2 \mapsto \sigma$*, then* $\sigma E_1 \equiv E_2$
- *If* $\overline{\Delta} \vdash \epsilon_1 : \Delta_1$*,* $\overline{\Delta} \vdash \epsilon_2 : \Delta_2$*, and* $\overline{\Delta} \vdash \epsilon_1 \overset{?}{=} \epsilon_2 \mapsto \sigma; \varphi_1; \varphi_2$*, then* $\Delta_2\{\varphi_2\} \preccurlyeq \Delta_1\{\varphi_1\}$*, and* $\sigma\epsilon_1\{\varphi_1\} \equiv \epsilon_2\{\varphi_2\}$.

Proof. By induction on the matching derivation. We expand the most relevant cases.

Rule dec-expr**.** By inversion on the typing derivation, there exists Δ_1' and Δ_2' such that $\overline{\Delta} \vdash \epsilon_1 : \Delta_1'$ with $\Delta_1' \preccurlyeq \Delta_1$ and $\overline{\Delta} \vdash \epsilon_2 : \Delta_2'$ with $\Delta_2' \preccurlyeq \Delta_2$. Applying renamings $\rho_1 = \overline{\Delta_2}/\underline{\Delta_1}$ and $\rho_2 = \overline{\Delta_2}/\underline{\Delta_2}$ respectively, we obtain $\overline{\Delta} \vdash \epsilon_1\{\rho_1\} : \Delta_1'\{\rho_1\}$ and $\overline{\Delta} \vdash \epsilon_2\{\rho_2\} : \Delta_2'\{\rho_2\}$. By IH, $\sigma\epsilon_1\{\rho_1\}\{\varphi_1\} \equiv \epsilon_2\{\rho_2\}\{\varphi_2\}$. We have then $\sigma\{\mathsf{let}\ \epsilon_1\{\rho_1\}\{\varphi_1\}\ \mathsf{in}\ \overline{\Delta_2}\} \equiv \{\mathsf{let}\ \epsilon_2\{\rho_2\}\{\varphi_2\}\ \mathsf{in}\ \overline{\Delta_2}\}$. The result follows since $\{\mathsf{let}\ \epsilon_1\{\rho_1\}\{\varphi_1\}\ \mathsf{in}\ \overline{\Delta_2}\} \equiv \{\mathsf{let}\ \epsilon_1\ \mathsf{in}\ \Delta_1\}$ and $\{\mathsf{let}\ \epsilon_2\{\rho_2\}\{\varphi_2\}\ \mathsf{in}\ \overline{\Delta_2}\} \equiv \{\mathsf{let}\ \epsilon_2\ \mathsf{in}\ \Delta_2\}$.

Rule tr-step-hd**.** By IH, $\sigma S_1 \equiv S_2$, and $\sigma\Delta_1\{\varphi_1\} \equiv \Delta_2\{\varphi_2\}$. Note that ϵ_1 and ϵ_2 are well typed under $\overline{\Delta}, \Delta_1$ and $\overline{\Delta}, \Delta_2$, respectively. Then, $\sigma\epsilon_1\{\varphi_1\}$ and $\epsilon_2\{\varphi_2\}$ are well typed under $\overline{\Delta} \bowtie \sigma\Delta_1\{\varphi_1\}$. By IH, $\sigma'\sigma\epsilon_1\{\varphi_1\}\{\varphi_1'\} \equiv \epsilon_2\{\varphi_2\}\{\varphi_2'\}$. Then, $(\{\Delta_2\}\leftarrow c\cdot S_2); \sigma'\sigma\epsilon_1\{\varphi_1\varphi_1'\} \equiv (\{\Delta_2\}\leftarrow c\cdot S_2); \epsilon_2\{\varphi_2\varphi_2'\}$. The result follows, since $\{\Delta_2\}\leftarrow c\cdot S_2 \equiv \sigma(\{\Delta_1\}\leftarrow c\cdot S_1)\{\varphi_1\varphi_1'\}$.

Rule tr-step-tl**.** By inversion on the typing derivation, there exists contexts Δ_1^0 and Δ_2^0 such that the following diagrams hold:

$$\overline{\Delta} \xrightarrow{\epsilon_1} \Delta_1^0 \bowtie \Delta_1' \xrightarrow{\{\Delta_1\}\leftarrow c\cdot S_1} \Delta_1^0 \bowtie \Delta_1$$
$$\overline{\Delta} \xrightarrow{\epsilon_2} \Delta_2^0 \bowtie \Delta_2' \xrightarrow{\{\Delta_2\}\leftarrow c\cdot S_2} \Delta_2^0 \bowtie \Delta_2$$

By renaming, $\overline{\Delta} \vdash \epsilon_1\{\varphi_1\} : (\Delta_1^0 \bowtie \Delta_1')\{\varphi_1\}$ and $\overline{\Delta} \vdash \epsilon_2\{\varphi_2\} : (\Delta_2^0 \bowtie \Delta_2')\{\varphi_2\}$. By IH, $\sigma'\sigma\epsilon_1\{\varphi_1\varphi_1''\} \equiv \epsilon_2\{\varphi_2\varphi_2''\}$. By IH on the last step of the trace, $\sigma\varphi_1 S_1 \equiv \varphi_2 S_2$, and $\sigma\Delta_1\{\varphi_1'\} \equiv \Delta_2\{\varphi_2'\}$. (Note that φ_i' does not affect ϵ_i nor S_i (for $i = 1, 2$).) Then, $\sigma(\{\Delta_1\}\leftarrow c\cdot S_1)\{\varphi_1\varphi_1'\}) \equiv (\{\Delta_2\}\leftarrow c\cdot S_2)\{\varphi_2\varphi_2'\})$. The result follows by combining with the IH from the rest of the trace.

Rule tr-inst**.** By IH $\Delta_2\{\varphi_2\} \preccurlyeq \Delta'\{\varphi_1\}$. Then $\{\mathsf{let}\ \epsilon\{\varphi_2\}\ \mathsf{in}\ \Delta'\{\varphi_2\}\}$ is well typed in context $\overline{\Delta_0}$. Let $X :: \Delta_X \vdash \{\Delta_X'\}$ and $\overline{\Delta_0'} \vdash \theta : \Delta_X$, where $\overline{\Delta_0'}$ is a subcontext of $\overline{\Delta_0}$. The result follows since we assume that applying θ^{-1} to $\{\mathsf{let}\ \epsilon\{\varphi_2\}\ \mathsf{in}\ \Delta'\{\varphi_2\}\}$ is well defined.

□

We give completeness statements for the different syntactic classes. The next lemma states completeness of the matching judgment for contexts.

Lemma 7 (Completeness of context matching)**.**

- *Let φ_1 and φ_2 be matching renamings such that $\Delta_1\{\varphi_1\} \equiv \Delta_2\{\varphi_2\}$. Then there exists $\varphi_1' \subseteq \varphi_1$ and $\varphi_2' \subseteq \varphi_2$ such that $\Delta_1 \overset{?}{=} \Delta_2 \mapsto \varphi_1'; \varphi_2'$.*
- *Let φ_1 and φ_2 be matching renamings such that $\Delta_1\{\varphi_1\} \succcurlyeq \Delta_2\{\varphi_2\}$. Then there exists $\varphi_1' \subseteq \varphi_1$ and $\varphi_2' \subseteq \varphi_2$ such that $\Delta_1 \succcurlyeq \Delta_2 \mapsto \varphi_1'; \varphi_2'$.*

Proof. By induction on the structure of Δ_1. □

The next lemma states completeness of the matching judgment for spines.

Lemma 8 (Completeness of term matching)**.** *Let F_1 and F_2 be elements of the same syntactic category (either terms, spines, or head applied to spine, i.e., $H{\cdot}S$), well typed under contexts Δ_1 and Δ_2, respectively. Assume there exists matching renamings φ_1 and φ_2, and an assignment σ such that $\sigma\varphi_1 F_1 = \varphi_2 F_2$ and $\Delta_1\{\varphi_1\} \preccurlyeq \Delta_2\{\varphi_2\}$. Then, there exists φ_1' and φ_2' such that $(\Delta_1 \vdash F_1) \overset{?}{=} (\Delta_2 \vdash F_2) \mapsto \sigma; \varphi_1'; \varphi_2'$ is derivable, $\varphi_1' \subseteq \varphi_1$, and $\varphi_2' = \varphi_2$.*

Proof. By induction on the structure of the structure of the elements F_1 and F_2. □

Next, we give completeness statements for traces and expressions.

Lemma 9 (Completeness of matching for traces)**.** *Let ϵ_1 and ϵ_2 be well-typed expressions under context Δ such that ϵ_2 is ground. Assume there exist renamings φ_1 and φ_2, where $\mathrm{dom}(\varphi_1) \subseteq @\epsilon_1\bullet$ and $\mathrm{dom}(\varphi_2) \subseteq @\epsilon_2\bullet$, and an assignment σ such that $\sigma\epsilon_1\{\varphi_1\} \equiv \epsilon_2\{\varphi_2\}$, then there exist φ_1' and φ_2' such that $\Delta \vdash \epsilon_1 \overset{?}{=} \epsilon_2 \mapsto \sigma; \varphi_1'; \varphi_2'$ is derivable.*

Proof. We can assume that ϵ_1 and ϵ_2 have all their variables marked, i.e., they are of the form $\underline{\epsilon}$. If there are renamings φ_1 and φ_2 between the output interfaces of ϵ_1 and ϵ_2, then there exists renamings φ_1^* and φ_2^* between the output interfaces of $\underline{\epsilon_1}$ and $\underline{\epsilon_2}$.

Let ϵ_1 be $\delta_1; \ldots; \delta_n; \{\Delta\}{\leftarrow}X[\theta]; \delta_{n+1}; \ldots; \delta_m$, and $\sigma = (X \leftarrow \{\mathsf{let}\ \epsilon_0\ \mathsf{in}\ \Delta_0\}), \sigma'$. Then ϵ_2 can be written as

$$\delta_1'; \ldots; \delta_n'; \epsilon_0'; \delta_{n+1}'; \ldots; \delta_m',$$

where each δ_i' corresponds to δ_i and ϵ_0 corresponds to X. Let $\delta_i = \{\Delta_i\}{\leftarrow}c_i{\cdot}S_i$ and $\delta_i' = \{\Delta_i'\}{\leftarrow}c_i'{\cdot}S_i'$. Matching succeeds for δ_1 and δ_1' since matching is complete for the terms in S_1 and S_1'. This introduces a renaming between Δ_1 and Δ_1' that is propagated to the rest of the trace. Rule $\mathsf{tr\text{-}step\text{-}hd}$ can be applied n times to match δ_i with δ_i' for $i = 1, \ldots, n$.

Since there is a renaming between the output interfaces of ϵ_1 and ϵ_2, matching δ_m and δ_m' succeeds; in particular, matching the output contexts return a renaming that is a subset of the renaming between ϵ_1 and ϵ_2. Rule $\mathsf{tr\text{-}step\text{-}tl}$ can be applied m times to match the steps δ_i and δ_i' for $i = n+1, \ldots, m$. $\sigma'\delta_1; \ldots; \delta_n\sigma'; \theta\epsilon_0; \sigma'\theta_0\delta_{n+1}; \ldots; \sigma'\theta_0\delta_m$ Finally rule $\mathsf{tr\text{-}inst}$ for the last step containing the logic variable. □

Lemma 10 (Completeness of matching for expressions)**.** *Let E_1 and E_2 be well-typed expressions under context Δ such that E_2 is ground. If there exists σ such that $\sigma E_1 \equiv E_2$, then $\Delta \vdash E_1 \overset{?}{=} E_2 \mapsto \sigma$ is derivable.*

Proof. Follows directly from the previous lemma. □

4. Conclusion

We have presented a sound and complete algorithm to perform matching on a large fragment of CLF, with at most one variable standing for an unknown concurrent trace. We showed that the matching problem is decidable for the fragment of CLF studied in this paper. We are in the process of extending this algorithm to larger fragments of CLF [20], and expect to be able to handle the whole language shortly. In the short term, this algorithm will be used as the basis for a run-time environment for well-moded CLF programs. Our long-term objective is to extend the Celf implementation of CLF with algorithms and methods for reasoning about concurrent and distributed computations. This work represents a small step in that direction.

We have recently started looking at the unification problem for CLF traces, which, not surprisingly, appears to be much more complicated than matching. We have some partial results for small fragments of CLF [20].

References

[1] Franz Baader and Wayne Snyder. *Handbook of Automated Reasoning*, chapter Unification Theory, pages 441–526. Elsevier, 2001.

[2] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[3] Iliano Cervesato, Nancy Durgin, Patrick D. Lincoln, John C. Mitchell, and Andre Scedrov. A Meta-Notation for Protocol Analysis. In *12th Computer Security Foundations Workshop — CSFW-12*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press.

[4] Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

[5] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A Concurrent Logical Framework II: Examples and Applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 2003.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, pages 76–87. Springer-Verlag LNCS 2706, June 2003.

[7] N.G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.

[8] Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300(1–3):379–409, 2003.

[9] Furio Honsell, Marino Miculan, and Ivan Scagnetto. Pi-calculus in (Co)Inductive Type Theories. *Theoretical Computer Science*, 253(2):239–285, 2001.

[10] INRIA. *The Coq Proof Assistant Reference Manual — Version 8.3*, 2010. Available at `http://coq.inria.fr/refman/`.

[11] Kurt Jensen. Coloured Petri nets. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 254 of *LNCS*, pages 248–299. Springer, 1986.

[12] Jochen Meßner. Pattern matching in trace monoids (extended abstract). In Rüdiger Reischuk and Michel Morvan, editors, *STACS*, volume 1200 of *LNCS*, pages 571–582. Springer, 1997.

[13] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[14] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[15] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3):1–49, June 2008.

[16] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[17] Carl Adam Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. IFIP*, pages 386–390, Amsterdam, 1963. North Holland Publ. Comp.

[18] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *CADE*, volume 1632 of *LNCS*, pages 202–206. Springer, 1999.

[19] Grzegorz Rozenberg and Volker Diekert, editors. *The Book of Traces*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995.

[20] Jorge Luis Sacchini, Iliano Cervesato, Frank Pfenning, Carsten Schürmann, and Robert J. Simmons. On trace matching and unification in CLF. Technical Report CMU-CS-12-114, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 2012.

[21] Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, January 2011.

[22] Anders Schack-Nielsen and Carsten Schürmann. Celf — A logical framework for deductive and concurrent systems (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *IJCAR*, volume 5195 of *LNCS*, pages 320–326. PUB-SP, 2008.

[23] Anders Schack-Nielsen and Carsten Schürmann. Pattern unification for the lambda calculus with linear and affine types. In Karl Crary and Marino Miculan, editors, *LFMTP*, volume 34 of *EPTCS*, pages 101–116, 2010.

[24] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A Concurrent Logical Framework I: Judgments and Properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 2003.

[25] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. Specifying properties of concurrent computations in CLF. *ENTCS*, 199:67–87, February 2008.

[26] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOL*, volume 5170 of *LNCS*, pages 33–38. Springer, 2008.

[27] Thomas Y. C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy*, SP ’93, pages 178–194, Washington, DC, USA, 1993. IEEE Computer Society.

[28] V. N. Zemlyachenko, N. M. Korneenko, and R. I. Tyshkevich. Graph isomorphism problem. *Journal of Mathematical Sciences*, 29(4):1426–1481, 1985.