On Matching in CLF

Iliano Cervesato* **Frank Pfenning***
Jorge Luis Sacchini* **Carsten Schürmann**
Robert J. Simmons*

July 2012
CMU-CS-12-114, CMU-CS-QTR-114

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

†Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
*Carnegie Mellon University, Qatar campus
‡IT University of Copenhagen — Copenhagen, Denmark

The authors can be reached at iliano@cmu.edu, fp@cs.cmu.edu, sacchini@qatar.cmu.edu, carsten@itu.dk, and rjsimmon@cs.cmu.edu.

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.

Keywords: CLF, concurrent traces, matching, unification, reasoning about concurrency

Abstract

Matching is an important component of a logical framework. It is at the heart of many reasoning tasks and is sufficient to support the operational semantic of well-moded logic programs. Matching is poorly understood for logical frameworks such as CLF, designed to effectively capture the specifications of parallel, concurrent and distributed systems. The witnesses of their computations, and therefore their term language, are concurrent traces. A concurrent trace is a sequence of computations where independent steps can be permuted. We study the problems of matching concurrent traces on large fragments of CLF. Specifically, we give a sound and complete algorithm for matching traces with a single variable standing for an unknown subtrace. We also examine the unification problem for some simple fragments of CLF and give an algorithm for solving unification problems with with one variable standing for an unknown subtrace on each side of the equation.

Contents

List of Figures

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 foundations that underlie systems such as Coq [14], Isabelle [33, 22], Agda [23], and Twelf [25], 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 [20] or Petri nets [24], that exhibit a parallel, concurrent or distributed semantics, does not fit this pattern, however. 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 in which independent computation steps are executed while still capturing the causal dependencies between inputs and outputs [26].

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. [13] did precisely this when developing a significant portion of the meta-theory of the π-calculus in Coq. 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 [31, 7], an extension of the LF type theory [12] 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 [29], 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 report presents an initial step in this direction.

A key functionality for 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, matching is sufficient to define the operational semantics of well-moded programs in a logical framework. This is advantageous because matching algorithms are often more efficient and better behaved than unification procedures. This appears to be the case for CLF as well.

In this report, we examine the matching problem for a succession languages of computational traces of increasing expressive power, leading to a large sublanguage of CLF. We show that, for the fragments considered, matching is decidable although highly non-deterministic. We define a series of matching algorithms for the case where there is at most one logic variable standing for an unknown trace, and prove their soundness and completeness.

	$\text{CLF}_{\multimap}$	$\text{CLF}_{@!}$	CLF_{Π}	CLF_{x}
1-var matching	Yes	Yes	Yes	Yes
n-var matching	Decidable	Decidable	Decidable	Decidable
1-var unification	Yes	Yes	No	No
n-var unification	No	No	No	No

Table 1: Status of Matching and Unification in Fragments of CLF

We also survey the much harder unification problem for two simply-typed fragments of CLF. For them, we adapt the corresponding matching algorithms to handle unification in the case of equations with one logic variable standing for an unknown trace on each side.

1.1 Overview

To make our exposition easier to follow, we consider a series of subsystems of CLF of increasing expressiveness. We analyze the matching problem for each for them. This way, the complexities and design choices of the matching algorithm are presented incrementally.

Specifically, we consider the following four subsystems of CLF of increasing expressive power:

1. $\text{CLF}_{\multimap}$ (Section 2): This is the fragment of CLF featuring purely linear traces and simple types. Its expressive power is equivalent to that of place/transition Petri nets [24] or propositional multiset rewriting [8]. Its traces are bipartite directed acyclic graphs [35].

2. $\text{CLF}_{@!}$ (Section 3): This language extends $\text{CLF}_{\multimap}$ with affine and persistent hypotheses. It remains simply-typed.

3. CLF_{Π} (Section 4): This fragment of CLF extends $\text{CLF}_{@!}$ with dependent types. It can represent many real-world specifications and is indeed as expressive as languages such as GAMMA [2, 16] and colored Petri nets [15].

4. CLF_{x} (Section 5): This language extends CLF_{Π} with embedded clauses. This enables it to simulate directly constructs found in the π-calculus [20, 27] and other process algebras.

CLF extends this last system with standard constructs found in the type theories of LF [12, 25] and its linear variant, LLF [5]. Unification, and therefore matching, are well-understood in LF and LLF [30, 4, 19, 21], neither of which supports concurrent traces.

For each of the four systems examined in this report, we define a matching algorithm where there is at most one logic variable standing for an unknown trace. We show how the matching algorithm is extended to account for the new features provided by each subsystem. We also analyze the general matching problem where there any number of logic variables. We show that this problem is decidable, although highly non-deterministic: the number of solutions of a problem can be exponential on the size of the trace. In the case of $\text{CLF}_{\multimap}$ and $\text{CLF}_{@!}$, we also consider the unification problem, where there is at most one logic variable standing for an unknown trace on each side of the equation (Sections 2.3 and 3.3, respectively).

Table 1 shows a summary of what problems have been solved for each system: 1-var matching (resp. n-var matching) refers to the matching problem where there is at most one (resp. any number of) logic variables standing for unknown trace on one side of an equation — the other side being

ground. This report describes correct algorithms for 1-var matching but only notes that the problem is decidable for n-var. Similarly, 1-var unification (resp. n-var unification) refers to the unification problem where there is at most one (resp. any number of) logic variable standing for unknown trace on each side of the equation. This report describes an algorithmic procedure to solve 1-var unification in $\text{CLF}_{\multimap}$ and $\text{CLF}_{@!}$, but not in the other systems. We have not examined the general unification problem with an arbitrary number of trace variables.

1.2 Organization

Sections 2, 3, 4 and 5 examine the matching problem for the sublanguages $\text{CLF}_{\multimap}$, $\text{CLF}_{@!}$, CLF_{Π} and CLF_{x}, respectively. The first two also discuss unification. We review related work in the literature in Section 6. We conclude and outline areas of future development in Section 7. Two case studies that exhibit the kind of matching problems examined in this report are presented in Appendix A.

2 $\text{CLF}_{\multimap}$: Linear Traces

In this section, we consider the matching problem for the purely linear sublanguage of CLF, denoted $\text{CLF}_{\multimap}$. It includes only traces and linear functions (no affine nor intuitionistic functions) and simple types. $\text{CLF}_{\multimap}$ is expressive enough to represent place/transition Petri nets [7] and traditional multiset rewriting [2, 16].

2.1 Language

Contexts A context is an *ordered* sequence of variable declarations of the form $x{:}a$, where a is a base type:

$$\textit{Contexts:}\quad \Delta \quad ::= \quad \cdot \mid \Delta, x{:}a$$

Contexts support a non-deterministic splitting operation: a context Δ *splits* into Δ_1 and Δ_2, written judgmentally as $\Delta = \Delta_1 \bowtie \Delta_2$, if every declaration in Δ appears in exactly one of the contexts Δ_1 and Δ_2. Formally, splitting is defined by the following rules:

$$\frac{}{\cdot = \cdot \bowtie \cdot} \qquad \frac{\Delta_0 = \Delta_1', \Delta_1'' \bowtie \Delta_2}{\Delta_0, x{:}a = \Delta_1', x{:}a, \Delta_1'' \bowtie \Delta_2} \qquad \frac{\Delta_0 = \Delta_1 \bowtie \Delta_2', \Delta_2''}{\Delta_0, x{:}a = \Delta_1 \bowtie \Delta_2', x{:}a, \Delta_2''}$$

We will generally use it as a (non-deterministic) operation, writing $\Delta_1 \bowtie \Delta_2$ in a context position in a rule. We will also use the same syntax to indicate the context obtained by merging Δ_1 and Δ_2. In this case, we assume that Δ_1 and Δ_2 declare distinct variables.

We write $\Delta_1 \approx \Delta_2$ if Δ_1 and Δ_2 only differ in the order of their declarations. I.e., $\Delta_1 \approx \Delta_2$ iff $\Delta_1 = \Delta_2 \bowtie \cdot$. This relations are used for the purposes of typing.

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 forms of concurrency based on state transitions, e.g., Petri nets [24, 15] and multiset rewriting [2, 3].

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

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

A *trace* is either empty ($\diamond$), a composition of two traces ($\epsilon_1; \epsilon_2$), or an individual computation step of the form $\{\Delta\}{\leftarrow}c{\cdot}\Delta'$, where c is a constant in a global signature Σ (defined below). We write δ for a generic computational step $\{\Delta\}{\leftarrow}c{\cdot}\Delta'$. We call c the *head* of δ, written $\mathsf{head}(\delta)$. A step of the form $\{\Delta\}{\leftarrow}c{\cdot}\Delta'$ represents an atomic computation c that uses the variables in Δ' and produces the variables in Δ. In examples, we will often omit the type of the declarations in Δ and Δ' for readability.

In a step $\delta = \{\Delta'\}{\leftarrow}c{\cdot}\Delta$, 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.

An *expression* $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$ is essentially a trace with delimited scope: no variable produced by ϵ is visible outside it. These variables are all collected in the context Δ.

The global *signature* collects the constant declarations. Formally signatures are defined by the following grammar:

$$\textit{Signatures:}\quad \Sigma \quad ::= \quad \cdot \mid c{:}\Delta \multimap \{\Delta'\}$$

Each constant is declared with a type of the form $\Delta \multimap \{\Delta'\}$, where Δ' corresponds to the positive types of CLF [28]. The scope of the declarations in Δ and $\{\Delta'\}$ is limited to the context itself. In other words, the names of the declared variables are meaningless; the situation will be different for the systems with dependent types presented in Sections 4 and 5.

Since we are assuming a fixed global signature, we will not mention it implicitly in the judgments used in this report.

Typing The typing rules of CLF$_{\multimap}$ are defined by the following judgments:

$$\begin{array}{ll} \textit{Traces:} & \Delta \vdash \epsilon : \Delta' \\ \textit{Expressions:} & \Delta \vdash E \Leftarrow \{\Delta'\} \end{array}$$

In the second judgment, the form $\{\Delta'\}$ is viewed as the type of the expression E. We call it a *monadic type*.

The typing rules are by given in Figure 1. Viewing contexts as the states of a concurrent computation, the typing rules for traces allow us to see a trace as the witness of a state transformation. The empty trace does not change the state (rule $\mathsf{tp}_{\multimap}$-empty). A step transforms a part of the state (rule $\mathsf{tp}_{\multimap}$-step); the step uses Δ_1 and produces the Δ_2 leaving the rest of the state, represented by Δ_0, intact. Since variables declared in the type of constant c are meaningless, we can α-rename the type to match the variables used in the step.

The typing rule for trace composition effectively composes the transformations given by each trace (rule $\mathsf{tp}_{\multimap}$-comp). Note that the monadic type of an expression does not leak out the variables produced by the trace it embeds.

Expressions:

$$\frac{\Delta_1 \vdash \epsilon : \Delta_2 \qquad \Delta_2 \approx \Delta'}{\Delta_1 \vdash \{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta'\} \Leftarrow \{\Delta'\}}\ \mathsf{tp}_{\multimap_\circ}\text{-expr}$$

Traces:

$$\frac{}{\Delta \vdash \diamond : \Delta}\ \mathsf{tp}_{\multimap_\circ}\text{-empty} \qquad \frac{c{:}\Delta_1 \multimap \{\Delta_2\} \in \Sigma}{\Delta_0 \bowtie \Delta_1 \vdash \{\Delta_2\} \leftarrow c \cdot \Delta_1 : \Delta_0 \bowtie \Delta_2}\ \mathsf{tp}_{\multimap_\circ}\text{-step}$$

$$\frac{\Delta_0 \vdash \epsilon_1 : \Delta_1 \qquad \Delta_1 \vdash \epsilon_2 : \Delta_2}{\Delta_0 \vdash \epsilon_1; \epsilon_2 : \Delta_2}\ \mathsf{tp}_{\multimap_\circ}\text{-comp}$$

Figure 1: Typing Rules of $\mathrm{CLF}_{\multimap_\circ}$

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:

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

where $\mathrm{dom}(\Delta)$ denotes the set of variables declared in Δ. Variables in a trace ϵ that do not belong to either $\bullet\epsilon$ or $\epsilon\bullet$ are *internal*. Internal variables are subject to implicit α-renaming (with the usual proviso that doing so does not identify distinct variables).

In $\mathrm{CLF}_{\multimap_\circ}$, for any typable expression $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$, we have that $\epsilon\bullet = \mathrm{dom}(\Delta)$. In the extensions of this language examined in later sections of this report, only the right-to-left inclusion will hold, in general.

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 $\bullet\epsilon_2 \cap \epsilon_1\bullet = \emptyset$ [26]. Permuting a typable composition of independent traces is always typable (Lemma 2.1).

Trace equality Two traces are *equal* if there is an α-renaming of their internal variables so that they both contain the same steps (although possibly in a different, dependency-preserving, order). For example, trace $(\{x_2\}\leftarrow c\cdot x_1; \{y_2\}\leftarrow c\cdot y_1)$ is equal to trace $(\{y_2\}\leftarrow c\cdot y_1; \{x_2\}\leftarrow c\cdot x_1)$. Formally, trace equality, written $\equiv$, is defined as the closure of the relation defined by the following rules:

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

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

The first two rules state that the empty trace is a unit element, while the third rule states that composition is associative. This effectively endows traces with a monoidal structure. The fourth rule states that independent traces can be permuted. Finally, the last two rules define the compatible closure of the equality relation: it is a congruence.

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,

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

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 this implicit α-renaming of bound variables in expressions, we can define expression equality simply as

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

For example, the following relation hold between traces

$$\begin{pmatrix}\{x\}\leftarrow c;\\ \{y\}\leftarrow c'\cdot x\end{pmatrix} \equiv \begin{pmatrix}\{z\}\leftarrow c;\\ \{y\}\leftarrow c'\cdot z\end{pmatrix} \not\equiv \begin{pmatrix}\{x\}\leftarrow c;\\ \{z\}\leftarrow c'\cdot x\end{pmatrix}$$

The last trace is not equal to the first two, since the output interface contains z, while in the first two traces the output interface contains y. On the other hand, the following expressions are all α-equivalent

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

2.1.1 Metatheory

We present some metatheoretical results that are needed in the rest of the paper. As a subsystem of CLF, $\mathrm{CLF}_{-\circ}$ enjoys desirable properties such as decidability of type checking. However, we only focus on properties that are related to our presentation using traces, or needed later in the proofs of soundness and completeness of the matching algorithm.

Since we only consider linear functions, weakening and strengthening of the typing relation are not valid in $\mathrm{CLF}_{-\circ}$ (although traces satisfy a form of frame rule). Typing is invariant under permutations of independent subtraces, as stated in the following lemma.

Lemma 2.1 *If* $\Delta \vdash \epsilon_1 : \Delta'$ *and* $\epsilon_1 \equiv \epsilon_2$, *then* $\Delta \vdash \epsilon_2 : \Delta'$.

Proof: By induction on the type derivation. □

In particular, whenever $\epsilon_1 \parallel \epsilon_2$, we have that $\Delta \vdash \epsilon_1 : \Delta'$ iff $\Delta \vdash \epsilon_2 : \Delta'$. Typing is also invariant under reordering of the context.

Lemma 2.2

1. *If* $\Delta_1 \vdash \epsilon : \Delta_2$ *and* $\Delta_1 \approx \Delta_1'$ *and* $\Delta_2 \approx \Delta_2'$, *then* $\Delta_1' \vdash \epsilon : \Delta_2'$.
2. *If* $\Delta_1 \vdash E : \{\Delta_2\}$ *and* $\Delta_1 \approx \Delta_1'$, *then* $\Delta_1' \vdash E : \{\Delta_2\}$.

Proof: By induction on the type derivation. □

Typing of traces satisfy the following frame rule.

Lemma 2.3 (Frame rule) *If* $\Delta_1 \vdash \epsilon : \Delta_2$, *then* $\Delta_0 \bowtie \Delta_1 \vdash \epsilon : \Delta_0 \bowtie \Delta_2$.

Proof: By induction on the type derivation. □

The next lemma states the inverse of the frame rule.

Lemma 2.4 *If* $\Delta_1 \vdash \epsilon : \Delta_2$, *then there exists* $\Delta_0, \Delta_1', \Delta_2'$ *such that* $\Delta_1 = \Delta_0 \bowtie \Delta_1'$, $\Delta_2 = \Delta_0 \bowtie \Delta_2'$, $\bullet\epsilon = \mathrm{dom}(\Delta_1')$, $\epsilon\bullet = \mathrm{dom}(\Delta_2')$, *and* $\Delta_1' \vdash \epsilon : \Delta_2'$.

The following lemma states a form of inversion of the typing derivation for the particular case of traces.

Lemma 2.5 *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* $\Delta_1 \vdash \epsilon_2 : \Delta_2$.

Proof: By induction on the type derivation. □

2.1.2 Equations

An *equation* is a postulated equality between entities that may contain logic variables. A *logic variable* X stands for an unknown trace. Logic variables are distinct from term variables x.

Logic variables They are declared in a *contextual modal context* [21]. Formally a contextual modal context is a sequence of logic variable declarations, defined as follows:

$$\Psi ::= \cdot \mid \Psi, X :: \Delta \vdash \{\Delta'\}$$

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

Within a trace defined in a context Δ_0, a logic variable X is accompanied by a *substitution* θ that maps the variables in Δ_X to variables in Δ_0, denoted $X[\theta]$. Substitutions are defined by the following grammar:

$$\theta ::= \cdot \mid \theta, y/x$$

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

$$\textit{Steps} \quad \delta \quad ::= \quad \{\Delta\}{\leftarrow}c{\cdot}\Delta' \mid \{\Delta\}{\leftarrow}X[\theta]$$

Substitutions are type-checked using the following judgment:

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

Logic variables:

$$\frac{X :: \Delta_X \vdash \{\Delta'_X\} \in \Psi \qquad \Delta_1 \vdash \theta : \Delta_X}{\Delta_0 \bowtie \Delta_1 \vdash \{\Delta'_X\}{\leftarrow}X[\theta] : \Delta_0, \Delta'_X}\,\mathsf{tp}_{\multimap}\text{-lvar}$$

Substitutions:

$$\frac{}{\cdot \vdash \cdot : \cdot}\,\mathsf{tp}_{\multimap}\text{-sub-empty} \qquad \frac{\Delta_0 \vdash \theta : \Delta_2}{\Delta_0 \bowtie y{:}A \vdash \theta, y/x : \Delta_2, x{:}A}\,\mathsf{tp}_{\multimap}\text{-sub-cons}$$

Figure 2: Typing Rules for Substitutions and Logic Variables of CLF$_{\multimap}$

The typing rules related to logic variables and substitutions are given in Figure 2.

In the case of CLF$_{\multimap}$, a substitution replaces variables with variables, since there is no notion of terms. The definition of substitutions is revised in subsequent systems that include more complex notions of terms.

An *assignment* σ is a sequence of bindings of the form $X \leftarrow E$ where X is a logic variable. An assignment σ is well typed if for every $X \leftarrow E \in \sigma$ with $X :: \Delta_X \vdash \{\Delta'_X\}$, we have $\Delta_X \vdash E : \{\Delta'_X\}$. 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:

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

In CLF$_{\multimap}$, any well-typed substitution is a bijection between variables. Because they are injective, an equation of the form $X[\theta] = \epsilon$ has at most one solution namely $\theta^{-1}\epsilon$. However, it may have no solution if ϵ contains variables not present in θ.

Lemma 2.6 (Inversion for CLF$_{\multimap}$) *Let E be a well-typed expression. There exists E' such that $E \equiv \theta E'$ if $\mathrm{FV}(E) = \mathrm{rng}(\theta)$.*

2.2 Matching

Given two objects T_1 and T_2 of the same syntactic class (expression or traces) 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 $T_1 \stackrel{?}{=} T_2$ to denote a matching problem.

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

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

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

Matching is decidable: in $\epsilon_1 \stackrel{?}{=} \epsilon_2$, 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 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 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 logic variable. This restriction suffices for many reasoning tasks of interest, for example monitoring a computation and some program transformation. Furthermore, restricting ourselves to only one logic variable reduces the explosion on the size of the search space.

A trace matching problem constrained in this way can then be expressed by the equation

$$\begin{pmatrix} \delta_1; \ldots; \delta_k; \\ \{\Delta\}{\leftarrow}X[\theta]; \\ \delta_{k+1}; \ldots; \delta_n \end{pmatrix} \stackrel{?}{=} \left(\delta'_1; \ldots; \delta'_m\right) \tag{1}$$

where δ_i, δ'_i have the form $\{\Delta'\}{\leftarrow}c{\cdot}\Delta$. The algorithm proceeds by matching individual steps anywhere from both traces. 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 with the same head, until a problem of the form

$$\left(\{\Delta\}{\leftarrow}X[\theta]\right) \stackrel{?}{=} \epsilon$$

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 where matched in the wrong order (meaning that 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. We will examine an example below.

Design of the matching algorithm Matching traces involves picking an appropriate permutations of one of the traces and finding a renaming that identifies the variables introduced by the trace. The matching algorithm for traces is given by a judgment of the form

$$\epsilon_1 \stackrel{?}{=} \epsilon_2 \mapsto \sigma$$

meaning that $\sigma\epsilon_1 \equiv \epsilon_2$. We assume that ϵ_1 and ϵ_2 have the same interface, i.e., there exists Δ_1 and Δ_2 such that $\Delta_1 \vdash \epsilon_i : \Delta_2$ for $i = 1, 2$. Furthermore, we assume that ϵ_2 is ground and ϵ_1 contains at most one logic variable.

The algorithm proceeds by matching individual steps on each side of the equation. Each step of the form $\{\Delta'\}{\leftarrow}c{\cdot}\Delta$ in ϵ_1 must be matched with a similar step in ϵ_2. We rely on α-conversion to match internal variables. Once a step is matched, it is removed from both traces and the process is repeated until either an empty trace is found on both sides (meaning that ϵ_1 is ground), or we are left with an equation of the form

$$\{\Delta'\}{\leftarrow}X[\theta] \stackrel{?}{=} \epsilon$$

Since both sides have the same interface (an invariant of the algorithm), the term $\theta^{-1}\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta'\}$ is well typed and is the solution for X.

The basic structure of the matching algorithm is kept in later systems (matching individual steps until we are left with only the logic variable). However, as we introduce new features in the language we are forced to adapt the algorithm to handle them. In $\mathrm{CLF}_{@!}$ (Section 3) the

introduction of affine and persistent hypotheses means that we cannot rely anymore on implicit α-conversion, as variables may be used more than once (persistent), or not at all (affine). We deal with this problem by introducing explicit renamings (Section 3.2). In CLF_{Π} and CLF_{x}, we introduce full terms, meaning that the matching algorithm needs to combine trace matching with traditional higher-order matching (Section 4.2). These languages also allow dependent types, which will force steps to be matched at either ends of a trace, but not in the middle.

The algorithm The matching algorithm is defined by the judgments:

$$\text{Expressions: } E_1 \stackrel{?}{=} E_2 \mapsto \sigma$$

$$\text{Traces: } \epsilon_1 \stackrel{?}{=} \epsilon_2 \mapsto \sigma$$

Matching on expression and traces return an assignment. The invariant for both judgment is that both sides have the same type.

The rules of the matching algorithm are given in Figure 3. This algorithm is non-deterministic algorithm and it fails when no rule is applicable. The non-determinism comes from rule $\mathsf{dec}_{\_\circ}$-$\mathsf{tr\text{-}step}$, where any matching steps on both sides can be chosen.

Some remarks about the matching rules are in order. In rule $\mathsf{dec}_{\_\circ}$-expr we rely on implicit α-conversion to ensure the context on both sides is the same. Typing invariants ensure that this is possible. Matching on traces is defined by rules $\mathsf{dec}_{\_\circ}$-$\mathsf{tr\text{-}empty}$, $\mathsf{dec}_{\_\circ}$-$\mathsf{tr\text{-}step}$, and $\mathsf{dec}_{\_\circ}$-$\mathsf{tr\text{-}inst}$. If rule $\mathsf{dec}_{\_\circ}$-$\mathsf{tr\text{-}empty}$ applies, it means that the left side in the original equation does not have any logic variable. In this case the matching algorithm amounts to checking trace equivalence.

Rule $\mathsf{dec}_{\_\circ}$-$\mathsf{tr\text{-}step}$ performs a match between two individual steps in both traces that have the same head. Note that doing so may change the interface of the traces on each side of the equation. As in rule $\mathsf{dec}_{\_\circ}$-expr, we use implicit α-renaming to ensure that the interface of the matched steps is the same. Matching proceeds with the rest of the trace, after removing the matched steps. After repeatedly applying rule $\mathsf{dec}_{\_\circ}$-$\mathsf{tr\text{-}step}$ we reach either an empty trace on both sides which is matched using rule $\mathsf{dec}_{\_\circ}$-$\mathsf{tr\text{-}empty}$, or the left side has a logic variable in which case we apply rule $\mathsf{dec}_{\_\circ}$-$\mathsf{tr\text{-}inst}$. Rule $\mathsf{dec}_{\_\circ}$-$\mathsf{tr\text{-}inst}$ is applicable if the output of ϵ coincides with the output of Δ (the algorithm invariants ensure that this condition is satisfied); the term $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$ is thus well typed.

Examples We illustrate the matching algorithm with an example. In this and the following examples, we do not specify the declaration of logic variables and omit local variables (unless necessary). 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 $X[y_1/x_1, y_2/x_2]$. We also omit the type of context variables.

Consider the following matching equation where f, g, and h are constants:

$$\{\mathsf{let}\ \begin{pmatrix}\{x_1, x_2\}{\leftarrow}g;\\ \{x_3\}{\leftarrow}f{\cdot}x_1;\\ \{\cdot\}{\leftarrow}X[x_3]\\ \{\cdot\}{\leftarrow}h{\cdot}x_2\end{pmatrix}\ \mathsf{in}\ \cdot\} \stackrel{?}{=} \{\mathsf{let}\ \begin{pmatrix}\{y_1, y_2\}{\leftarrow}g;\\ \{y_3\}{\leftarrow}f{\cdot}y_1;\\ \{\cdot\}{\leftarrow}h{\cdot}y_2\\ \{\cdot\}{\leftarrow}h{\cdot}y_3\end{pmatrix}\ \mathsf{in}\ \cdot\}$$

This problem has only one solution: $X \leftarrow \{\mathsf{let}\ \{\cdot\}{\leftarrow}h{\cdot}y_3\ \mathsf{in}\ \cdot\}$. Note that the algorithm may backtrack in order to find this solution. Let us consider what happens if the algorithm chooses a wrong pair of steps to match. For example, if the algorithm chooses initially to match $\{\cdot\}{\leftarrow}h{\cdot}x_2$ against

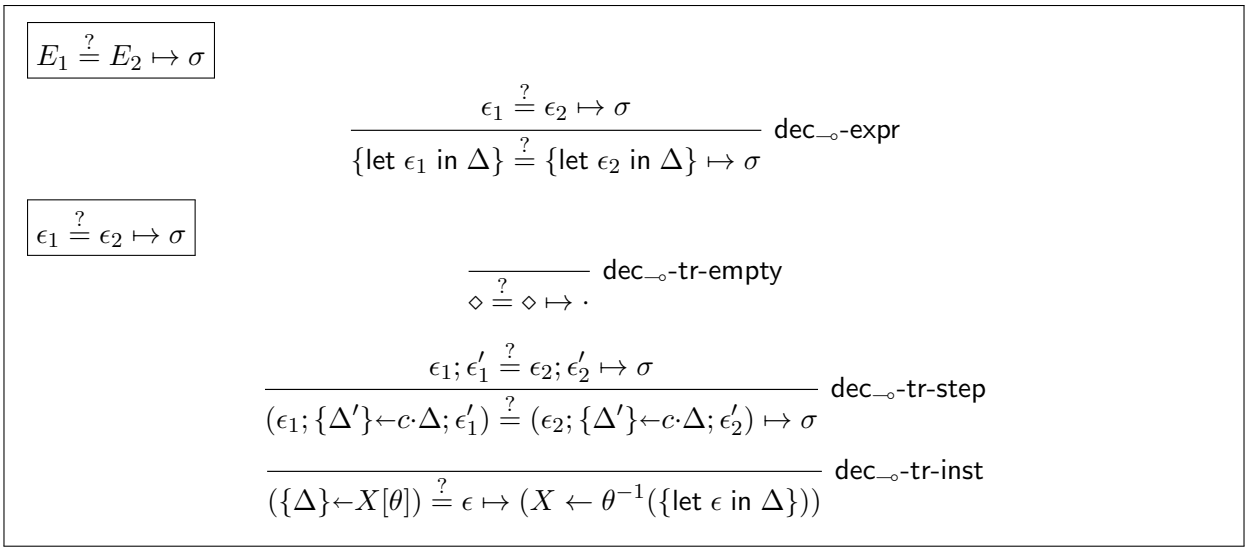


Figure 3: Matching Algorithm for CLF$_{\multimap}$

$\{\cdot\}{\leftarrow}h{\cdot}y_3$, this will force to rename the internal variables to make them match, e.g., y_3/x_2. The matching problem is reduced to

$$\begin{pmatrix} \{x_1, y_3\}{\leftarrow}g; \\ \{x_3\}{\leftarrow}f{\cdot}x_1; \\ \{\cdot\}{\leftarrow}X[x_3] \end{pmatrix} \stackrel{?}{=} \begin{pmatrix} \{y_1, y_2\}{\leftarrow}g; \\ \{y_3\}{\leftarrow}f{\cdot}y_1; \\ \{\cdot\}{\leftarrow}h{\cdot}y_2 \end{pmatrix}$$

At this point, matching $\{x_1, y_3\}{\leftarrow}g$ against $\{y_1, y_2\}{\leftarrow}g$ fails, since it implies matching y_2 against y_3; y_2 is an internal variable, while y_3 is in the output interface.

The algorithm then will backtrack and try to match $\{\cdot\}{\leftarrow}h{\cdot}x_2$ against $\{\cdot\}{\leftarrow}h{\cdot}y_2$ leading to the solution for X.

Correctness of the algorithm The matching algorithm of Figure 3 is sound and complete as stated in the following lemmas.

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 $i = 1, \ldots, n$.

Lemma 2.7 (Soundness of matching for CLF$_{\multimap}$)

- *If* $\Delta \vdash E_1, E_2 \Leftarrow \{\Delta'\}$ *and* $E_1 \stackrel{?}{=} E_2 \mapsto \sigma$, *then* $\sigma E_1 \equiv E_2$.
- *If* $\Delta \vdash \epsilon_1 : \Delta_1$ *and* $\Delta \vdash \epsilon_2 : \Delta_2$ *and* $\Delta_1 \approx \Delta_2$ *and* $\epsilon_1 \stackrel{?}{=} \epsilon_2 \mapsto \sigma$, *then* $\sigma\epsilon_1 \equiv \epsilon_2$.

Proof: By induction on the matching derivation. We only consider the most interesting cases.

Rule dec$_{\multimap}$-expr**.** By inversion on the typing derivation, there exists Δ_1 and Δ_2 such that $\Delta_1 \approx \Delta_2 \approx \Delta'$, $\Delta \vdash \epsilon_1 : \Delta_1$, and $\Delta \vdash \epsilon_1 : \Delta_2$. The result follows by IH.

Rule dec$_{\multimap}$-tr-step. Let $\delta = \{\Delta'\}{\leftarrow}c{\cdot}\Delta$. By Lemma 2.5 on the judgment $\Delta \vdash \epsilon_1; \delta_1; \epsilon_1' : \Delta_1$, there exists Δ_1^* such that the following diagram holds:

$$\Delta \xrightarrow{\epsilon_1} \Delta_1^* \bowtie \Delta \xrightarrow{\delta} \Delta_1^* \bowtie \Delta' \xrightarrow{\epsilon_1'} \Delta_1$$

By the frame rule on ϵ_1 and ϵ_1' and composing the results we have: $\Delta \bowtie \Delta' \vdash \epsilon_1; \epsilon_1' : \Delta_1 \bowtie \Delta$.

By a similar reasoning on the right-hand side, there exists Δ_2^* such that the following diagram holds:

$$\Delta \xrightarrow{\epsilon_2} \Delta_2^* \bowtie \Delta \xrightarrow{\delta_2} \Delta_2^* \bowtie \Delta' \xrightarrow{\epsilon_2'} \Delta_2$$

Similarly, by the frame rule, we have $\Delta \bowtie \Delta' \vdash \epsilon_2; \epsilon_2' : \Delta \bowtie \Delta_2$.

Since $\Delta_1 \bowtie \Delta' \approx \Delta_2 \bowtie \Delta'$, we can apply the IH; we have $\sigma(\epsilon_1; \epsilon_1') \equiv (\epsilon_2; \epsilon_2')$. Then $\sigma(\epsilon_1; \delta_1; \epsilon_1') \equiv (\epsilon_2; \delta_2; \epsilon_2')$.

Rule dec$_{\multimap}$-tr-inst. The expression $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$ is well typed, since both traces have the same interface. The conditions of Lemma 2.6 are satisfied, so the inverse substitution can be applied.

□

The matching algorithm of Figure 3 is complete as stated in the following lemmas.

Lemma 2.8 (Completeness of trace matching for CLF$_{\multimap}$) *Let* $\Delta \vdash \epsilon_1 : \Delta_1$ *and* $\Delta \vdash \epsilon_2 : \Delta_2$, *where* $\Delta_1 \approx \Delta_2$, *and* ϵ_1 *contains at most one logic variable. If* $\sigma\epsilon_1 \equiv \epsilon_2$, *then there exists a derivation of the judgment* $\epsilon_1 \stackrel{?}{=} \epsilon_2 \mapsto \sigma$.

Proof: Let ϵ_1 be of the form $\delta_1; \ldots; \delta_n; \{\Delta\}{\leftarrow}X[\theta]; \delta_{n+1}; \ldots; \delta_m$, and $\sigma = (X \leftarrow \{\mathsf{let}\ \epsilon_0\ \mathsf{in}\ \Delta_0\})$. 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 $\delta_i = \{\Delta_{1i}'\}{\leftarrow}c_i{\cdot}\Delta_{1i}$ and $\delta_i' = \{\Delta_{2i}'\}{\leftarrow}c_i'{\cdot}\Delta_{2i}$.

Matching succeeds for δ_1 and δ_1' since we must have $\Delta_{11} = \Delta_{21}$, and we can assume that $\Delta_{11}' = \Delta_{21}'$. Rule dec$_{\multimap}$-tr-step can be applied n times to match δ_i with δ_i' for $i = 1, \ldots, n$.

We can proceed the same way from the other end of the trace, since the interfaces are the same (modulo permutation). Rule dec$_{\multimap}$-tr-step 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 dec$_{\multimap}$-tr-inst for the last step containing the logic variable. □

Lemma 2.9 (Completeness of expression matching for CLF$_{\multimap}$) *Let* $\Delta \vdash E_1, E_2 : \{\Delta'\}$, *where* E_1 *contains at most one logic variable* X. *If* $\sigma E_1 \equiv E_2$, *then there exists a derivation of the judgment* $E_1 \stackrel{?}{=} E_2 \mapsto \sigma$.

Proof: Rule dec$_{\multimap}$-dec-expr can be applied; the result follows from the previous lemma. □

2.3 Unification

We can extend the matching algorithm given above to handle a special case of unification. The unification problem is the following: given an equation of the form $T_1 \stackrel{?}{=} T_2$, find a substitution σ for the logic variables in T_1 and T_2 such that $\sigma T_1 \equiv \sigma T_2$. As in the case of matching, we restrict to the case where T_1 and T_2 can contain at most one logic variable.

We make the further assumption that the logic variable on each side is distinct. The case where the same logic variable is used on both sides is unexpectedly more complex and is left for future work. For example, in the λ-calculus, a matching problem of the form $X[x,y] \stackrel{?}{=} X[y,x]$ does not have solution. However, in CLF, a matching problem of the form $\{\cdot\}{\leftarrow}X[x,y] \stackrel{?}{=} \{\cdot\}{\leftarrow}X[y,x]$ has solutions since x and y can be reordered in X if they are independent; for example, a solution for X is $\{\mathsf{let}\ \{\cdot\}{\leftarrow}Z[x]; \{\cdot\}{\leftarrow}Z[y]\ \mathsf{in}\ \cdot\}$.

The unification algorithm follows the same pattern as the matching algorithm, by matching individual steps on both sides of the equation. The base case is then of the form

$$\epsilon_1; (\{\Delta_1\}{\leftarrow}X_1[\theta_1]) \stackrel{?}{=} (\{\Delta_2\}{\leftarrow}X_2[\theta_2]); \epsilon_2 \tag{2}$$

or the symmetric equation. If the interface of ϵ_1 coincide with the interface of $\{\Delta_2\}{\leftarrow}X_2[\theta_2]$ and the interface of ϵ_2 coincide with the interface of $\{\Delta_1\}{\leftarrow}X_1[\theta_1]$, then a solution to this problem is $X_1 \leftarrow \theta_1^{-1}\{\mathsf{let}\ \epsilon_2\ \mathsf{in}\ \Delta_1\}$ and $X_2 \leftarrow \theta_2^{-1}\{\mathsf{let}\ \epsilon_1\ \mathsf{in}\ \Delta_2\}$. If the interfaces do not match, then certain conditions must satisfy in order to have a solution.

A more general solution to this problem looks like

$$X_1 \leftarrow \theta_1^{-1}(\{\mathsf{let}\ \{\Delta'\}{\leftarrow}Z[\theta']; \epsilon_2\ \mathsf{in}\ \Delta_1\})$$
$$X_2 \leftarrow \theta_2^{-1}(\{\mathsf{let}\ \epsilon_1; \{\Delta'\}{\leftarrow}Z[\theta']\ \mathsf{in}\ \Delta_2\})$$

where θ' and Δ' must be chosen so that the above expressions are well typed. The expression assigned to X_1 is well typed if the following holds:

$$\mathrm{FV}(\theta_1) = \bullet(\{\Delta'\}{\leftarrow}Z[\theta']; \epsilon_2)$$
$$\mathrm{dom}(\Delta_1) = (\{\Delta'\}{\leftarrow}Z[\theta']; \epsilon_2)\bullet$$

where $\mathrm{FV}(\theta_1)$ means $\bigcup_{a \in \mathrm{rng}(\theta_1)} \mathrm{FV}(a)$ — in $\mathrm{CLF}_{-\circ}$, $\mathrm{FV}(\theta_1)$ coincides with $\mathrm{rng}(\theta_1)$. (A similar condition holds for the expression assigned to X_2.) Expanding the definitions of input and output interface, we obtain the following equations:

$$\mathrm{FV}(\theta_1) = \mathrm{FV}(\theta') \cup (\bullet\epsilon_2 \setminus \mathrm{dom}(\Delta')) \qquad \mathrm{FV}(\theta_2) = \bullet\epsilon_1 \cup (\mathrm{FV}(\theta') \setminus \epsilon_1\bullet)$$
$$\mathrm{dom}(\Delta_1) = \epsilon_2\bullet \cup (\mathrm{dom}(\Delta') \setminus \bullet\epsilon_2) \qquad \mathrm{dom}(\Delta_2) = \mathrm{dom}(\Delta') \cup (\epsilon_1\bullet \setminus \mathrm{FV}(\theta'))$$

A necessary conditions for these equations to have a solution is that $\epsilon_2\bullet \subseteq \mathrm{dom}(\Delta_1)$ and $\bullet\epsilon_1 \subseteq \mathrm{FV}(\theta_2)$. Note that the number of solutions is finite as $\mathrm{FV}(\theta') \subseteq \mathrm{FV}(\theta_1)$, and $\mathrm{dom}(\Delta') \subseteq \mathrm{dom}(\Delta_2)$. Any solution to these equations gives a solution for the unification problem.

Thus, the algorithm of Figure 3 can be adapted to handle a unification problem by changing rule $\mathsf{dec}_{-\circ}$-tr-inst to the following rule ($\mathsf{dec}_{-\circ}$-tr-inst-$\mathsf{unif1}$):

$$\frac{\sigma = (X_1 \leftarrow \theta_1^{-1}(\{\mathsf{let}\ \{\Delta'\}{\leftarrow}Z[\theta']; \epsilon_2\ \mathsf{in}\ \Delta_1\})), (X_2 \leftarrow \theta_2^{-1}(\{\mathsf{let}\ \epsilon_1; \{\Delta'\}{\leftarrow}Z[\theta']\ \mathsf{in}\ \Delta_2\}))}{\epsilon_1; (\{\Delta_1\}{\leftarrow}X_1[\theta_1]) \stackrel{?}{=} (\{\Delta_2\}{\leftarrow}X_2[\theta_2]); \epsilon_2 \mapsto \sigma}$$

where Δ', and θ' satisfy the equations given above. We also need a symmetric rule, called $\mathsf{dec}_{\multimap}$-tr-inst-unif2, that handles the problem $(\{\Delta_1\}\leftarrow X_1[\theta_1]); \epsilon_1 \stackrel{?}{=} \epsilon_2; (\{\Delta_2\}\leftarrow X_2[\theta_2])$.

Note that there is no most-general solution to a unification problem, since the reduction of a unification problem to an equation of the form (2) is not unique (as shown in the example below). However, all solutions can be found by reducing the problem to the form above.

Example Consider the following unification problem

$$\begin{pmatrix} \{x_2\}\leftarrow c\cdot x; \\ \{x_3\}\leftarrow c\cdot x_2; \\ \{z\}\leftarrow X_1[x_3] \end{pmatrix} \stackrel{?}{=} \begin{pmatrix} \{y_2\}\leftarrow X_2[x]; \\ \{y_3\}\leftarrow c\cdot y_2; \\ \{z\}\leftarrow c\cdot y_3 \end{pmatrix}$$

The input interface contains x and the output interface contains z. Applying rule $\mathsf{dec}_{\multimap}$-tr-inst-unif1, we obtain one solution

$$X_1 \leftarrow x_3^{-1}\{\mathsf{let}\ \{y_2\}\leftarrow Z[x_3]; \{y_3\}\leftarrow c\cdot y_2; \{z\}\leftarrow c\cdot y_3\ \mathsf{in}\ z\}$$
$$X_2 \leftarrow x_2^{-1}\{\mathsf{let}\ \{x_2\}\leftarrow c\cdot x; \{x_3\}\leftarrow c\cdot x_2; \{y_2\}\leftarrow Z[x_3]\ \mathsf{in}\ y_2\}$$

Another solution is found if we first match $\{x_3\}\leftarrow c\cdot x_2$ against $\{y_3\}\leftarrow c\cdot y_2$, thus reducing the problem to

$$\begin{pmatrix} \{w_1\}\leftarrow c\cdot x; \\ \{z\}\leftarrow X_1[w_2] \end{pmatrix} \stackrel{?}{=} \begin{pmatrix} \{w_1\}\leftarrow X_2[x]; \\ \{z\}\leftarrow c\cdot w_2 \end{pmatrix}$$

Applying rule $\mathsf{dec}_{\multimap}$-tr-inst-unif1, we obtain one solution

$$X_1 \leftarrow w_2^{-1}\{\mathsf{let}\ \{\cdot\}\leftarrow Z[\cdot]; \{z\}\leftarrow c\cdot w_2\ \mathsf{in}\ z\}$$
$$X_2 \leftarrow x^{-1}\{\mathsf{let}\ \{w_1\}\leftarrow c\cdot x; \{\cdot\}\leftarrow Z[\cdot]\ \mathsf{in}\ w_1\}$$

2.4 Graph Interpretation

A concurrent trace ϵ can be interpreted as a bipartite directed acyclic graph (BDAG) $G = (N_1, N_2, E)$ where N_1 is the set of steps δ in ϵ and N_2 is the set of the variables x mentioned in ϵ. For each $\delta = \{\Delta\}\leftarrow c\cdot\Delta'$, there is an edge from $x \in N_2$ to $\delta \in N_1$ if and only if x is declared in Δ', and there is an edge from δ to $y \in N_2$ iff y appears in Δ. Figure 4 shows an example of a trace and its graphical representation, where nodes in N_1 are drawn as rectangle and nodes in N_2 as circles. The BDAGs obtained in this way are exactly what we get by graphically unfolding the computation of a place/transition Petri net [24].

Equality over traces corresponds therefore to a graph isomorphism problem. Although the complexity of checking whether two BDAGs are isomorphic has not been determined as far as we know, the isomorphism problem for both bipartite graphs and directed acyclic graphs is known to be GI-complete, where GI is a complexity class between P and NP [35].

Trace equations correspond to isomorphism problems with holes in them. Specifically, every step $\{\Delta\}\leftarrow X[\theta]$ corresponds to an unknown subgraph with incoming edges from the N_2-nodes associated with the domain of θ and outgoing edges to the N_2-nodes determined by the variables in $\mathrm{dom}(\Delta)$. Note that each such step behaves like a node in N_1.

The matching problem over traces corresponds to overlaying a fully determined BDAG with a BDAG containing such holes (at most one in our discussion). Our algorithm spots and removes

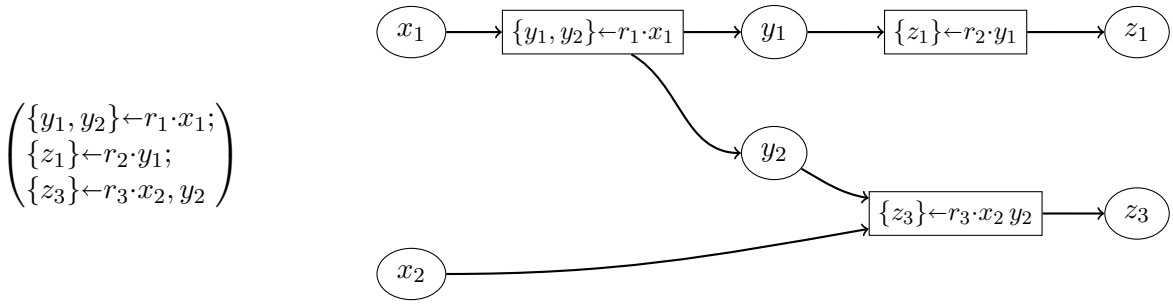


Figure 4: A Trace and its Graphical Representation

compatible N_1-nodes until just the hole is left on one side — the graph on the other side is then the solution to the problem. Incorrect guesses cause backtracking.

Trace unification corresponds to the problem of overlaying two BDAGs, both of which may contain holes. The algorithm outlined in the last section spots and removes compatible N_1-nodes until the hole on each side must cover every N_1-node on the other side (or a dead-end has been encountered).

The graphical interpretation just outlined applies to the other languages examined in this report. The corresponding graphs become structurally more and more complex, and so do the matching and unification problems. We will not draw the parallels between traces in these languages and the corresponding graphs, although this can be easily done.

3 CLF$_{@!}$: Affine and Persistent Functions

In this section we present CLF$_{@!}$, an extension of CLF$_{\multimap}$ with affine and persistent functions (Section 3.1). We analyze the matching problem for CLF$_{@!}$ in the presence of at most one logic variable (Section 3.2). The introduction of affine and persistent functions complicates the matching algorithm as we cannot rely on implicit α-conversion to match internal variables as we do in CLF$_{\multimap}$; since persistent variables can be used more than once we need to make sure that the matching of internal variables is one-to-one. We introduce explicit renamings between variables that are computed by the matching algorithm; variables are marked to ensure the renaming is one-to-one (Section 3.2).

3.1 Language

CLF$_{@!}$ extends CLF$_{\multimap}$ with affine and persistent functions. Variables in CLF$_{@!}$ are decorated with a modality indicating whether they are linear, affine, or persistent. Persistent variables may be reused in a trace; affine variables may be used at most once but, unlike linear variables, do not need to be consumed and may be simply ignored.

Contexts CLF$_{@!}$ extends the definition of contexts for CLF$_{\multimap}$ by including affine and persistent hypotheses. A *context* is a sequence of variable declarations of the form $□x{:}a$, where a is a base

type and $\square$ is one of three *modalities*: ! (persistent), @ (affine), or $\downarrow$ (linear).

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

The declarations $x{:}a$ of $\text{CLF}_{\multimap}$ are written as linear declarations $\downarrow x{:}a$ in $\text{CLF}_{@!}$.

We write $!\Delta$ for the largest subcontext of Δ containing only persistent variables. Similarly, $@\Delta$ is the largest subcontext of Δ containing only affine or persistent variables (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. We write $\mathsf{nolin}(\Delta)$ to mean that the context Δ contains no linear declaration.

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:

$$(\Delta_0, \square x{:}a, \Delta_1 \setminus\!\setminus x) = \begin{cases} (\Delta_0, \Delta_1) & \text{if } \square \in \{@, \downarrow\} \\ \Delta_0, \square x{:}a, \Delta_1 & \text{if } \square = ! \end{cases}$$

Modulo the update to the notion of contexts, the definition of signature is as in $\text{CLF}_{\multimap}$. We continue to assume an implicit global signature Σ, which declares the constants in use.

We redefine the splitting operation on contexts to account for affine and persistent hypotheses: $\Delta = \Delta_1 \bowtie \Delta_2$ if all persistent declarations in Δ may appear in both Δ_1 and Δ_2, linear declarations are included in exactly one of Δ_1 and Δ_2, and affine declarations are included in at most one of Δ_1 and Δ_2. Formally, splitting is defined by the following rules:

$$\frac{}{\cdot = \cdot \bowtie \cdot} \qquad \frac{\Delta = \Delta_1, \Delta_1' \bowtie \Delta_2, \Delta_2'}{\Delta, !x{:}a = \Delta_1, !x{:}a, \Delta_1' \bowtie \Delta_2, !x{:}a, \Delta_2'}$$

$$\frac{\Delta = \Delta_1 \bowtie \Delta_2}{\Delta, @x{:}a = \Delta_1, \Delta_2} \qquad \frac{\Delta = \Delta_1, \Delta_1' \bowtie \Delta_2}{\Delta, @x{:}a = \Delta_1, @x{:}a, \Delta_1' \bowtie \Delta_2} \qquad \frac{\Delta = \Delta_1 \bowtie \Delta_2, \Delta_2'}{\Delta, @x{:}a = \Delta_1 \bowtie \Delta_2, @x{:}a, \Delta_2'}$$

$$\frac{\Delta = \Delta_1, \Delta_1' \bowtie \Delta_2}{\Delta, \downarrow x{:}a = \Delta_1, \downarrow x{:}a, \Delta_1' \bowtie \Delta_2} \qquad \frac{\Delta = \Delta_1 \bowtie \Delta_2, \Delta_2'}{\Delta, \downarrow x{:}a = \Delta_1 \bowtie \Delta_2, \downarrow x{:}a, \Delta_2'}$$

We write $\Delta_1 \approx \Delta_2$ if $\Delta_1 = \Delta_2 \bowtie \cdot$. Note that, if $\Delta = \Delta_1 \bowtie \Delta_2$, then $!\Delta = !\Delta_1 = !\Delta_2$ and each affine or linear declaration in Δ appear in exactly one of Δ_1 and Δ_2

We define an partial-order relation between contexts: 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'.

Traces We redefine the notion of step in $\text{CLF}_{@!}$. Traces and steps are defined as follows:

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

Traces have the same structure as in $\text{CLF}_{\multimap}$. A step has the form $\{\Delta\} \leftarrow c \cdot S$ where S is a spine [6], and c is a constant from the signature. In the case of $\text{CLF}_{@!}$, a spine is just a list of variables annotated with a modality. In later systems, the notion of spine is redefined to allow more complex

terms. In an *expression* $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$, the context Δ collects all unused linear variables and, some affine and persistent variables.

The interface of a trace in $\mathrm{CLF}_{@!}$ reflects the fact that persistent variables can be reused. The input and output interface are given by the following rules:

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

The only difference with the rules of $\mathrm{CLF}_{\multimap}$ is the last rule of the output interface where persistent variables in ϵ_1 are not removed with the output interface of $\epsilon_1;\epsilon_2$.

Equality of traces and expressions is defined as in $\mathrm{CLF}_{\multimap}$. However, note that persistent variables are always in the output interface. This means that the trace $\{!x\}{\leftarrow}c;\{!y\}{\leftarrow}c{\cdot}!x$ is not equal to $\{!z\}{\leftarrow}c;\{!y\}{\leftarrow}c{\cdot}!z$, since they differ in the output interface. However, the expressions $\{\mathsf{let}\ \{!x\}{\leftarrow}c;\{!y\}{\leftarrow}c{\cdot}!x\ \mathsf{in}\ \cdot\}$ and $\{\mathsf{let}\ \{!z\}{\leftarrow}c;\{!y\}{\leftarrow}c{\cdot}!z\ \mathsf{in}\ \cdot\}$ are α-equivalent, since enclosing a trace inside a let-expression turns its type into a type of the form $\{\Delta\}$ where the names declared in Δ are local.

The global *signature* collects the constant declaration. As in $\mathrm{CLF}_{\multimap}$, each constant is declared with a type of the form $\Delta \multimap \{\Delta'\}$, where the scope of the declarations in Δ and $\{\Delta'\}$ is limited to the context itself.

Typing The typing rules of $\mathrm{CLF}_{@!}$ are defined by the following judgments:

$$\begin{array}{ll} \textit{Expressions:} & \Delta \vdash E \Leftarrow \{\Delta'\} \\ \textit{Traces:} & \Delta \vdash \epsilon : \Delta' \\ \textit{Spines:} & \Delta \vdash S : \Delta' \end{array}$$

The typing rules of $\mathrm{CLF}_{@!}$ are given in Figure 5. Note that the context Δ of an expression $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$ contains all linear hypotheses produced by the trace ϵ, but does not have to contain all the affine and persistent hypotheses.

The rules for spines state that a persistent hypotheses can only be fulfilled by a persistent variable, an affine hypotheses can only be fulfilled by a persistent or affine variable, while there are no restrictions for linear hypotheses.

Metatheory In addition to the properties stated in Section 2.1.1, the typing relation satisfies the properties stated below. Weakening is valid for affine and intuitionistic hypotheses.

Lemma 3.1 *If* $\Delta \vdash E \Leftarrow \{\Delta'\}$*, then* $\Delta, @\Delta_0 \vdash E \Leftarrow \{\Delta'\}$*, for* $\square \in \{@, !\}$.

Strengthening is also valid for affine and intuitionistic hypotheses.

Lemma 3.2 *Let* $\Delta_0, \square x{:}a, \Delta_1 \vdash E \Leftarrow \{\Delta'\}$ *be a valid judgment, where* $\square \in \{@, !\}$*. If* $x \notin \mathrm{FV}(E)$*, then* $\Delta_0, \Delta_1 \vdash E \Leftarrow \{\Delta'\}$.

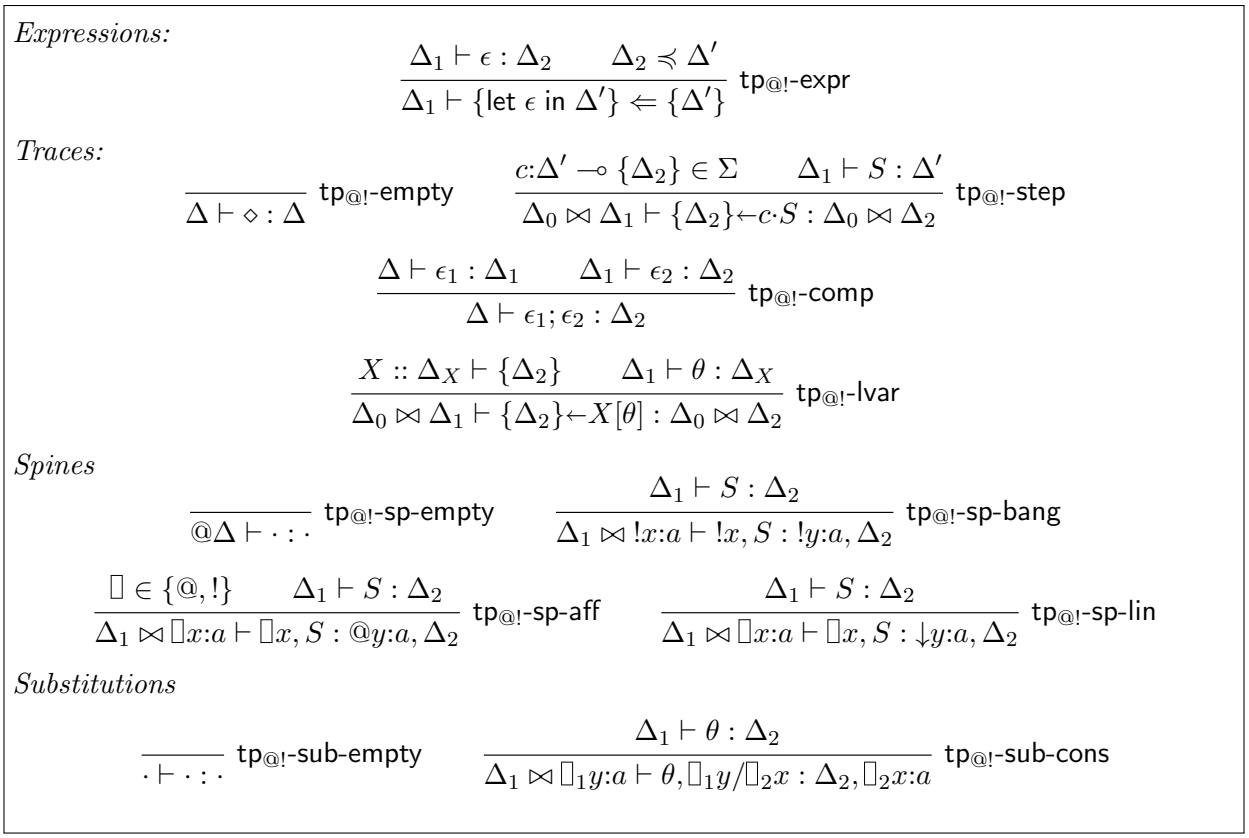


Figure 5: Typing Rules of $\text{CLF}_{@!}$

3.2 Matching

The main structure of the matching algorithm remains the same as in $\text{CLF}_{\multimap}$: individual steps are matched on both sides and removed until we reach the empty trace or a step with a logic variable. Recall that we assume that there is at most one logic variable. In this section, we describe the changes needed for logic variables, and substitutions. We then revisit the algorithm and proof that is sound and complete.

Equations We consider steps with logic variables as in the case of $\text{CLF}_{\multimap}$. Logic variables are declared in a contextual modal context (cf. Section 2.1.2). We extend the syntax of steps by allowing logic variables at the head:

$$\textit{Steps} \quad \delta \quad ::= \quad \{\Delta\} \leftarrow c \cdot \Delta' \mid \{\Delta\} \leftarrow X[\theta]$$

We redefine the notion of substitution by annotating variables with their modality:

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

In a substitution item $\square y / \square' x$, the modalities $\square$ and $\square'$ may be different: indeed, a linear variable may be replaced by a persistent term without violating typing (so that $!y/\downarrow x$ is allowed), while the opposite may yield an ill-typed term (e.g., $\downarrow y/!x$ may lead to multiple copies of $\downarrow y$). The valid

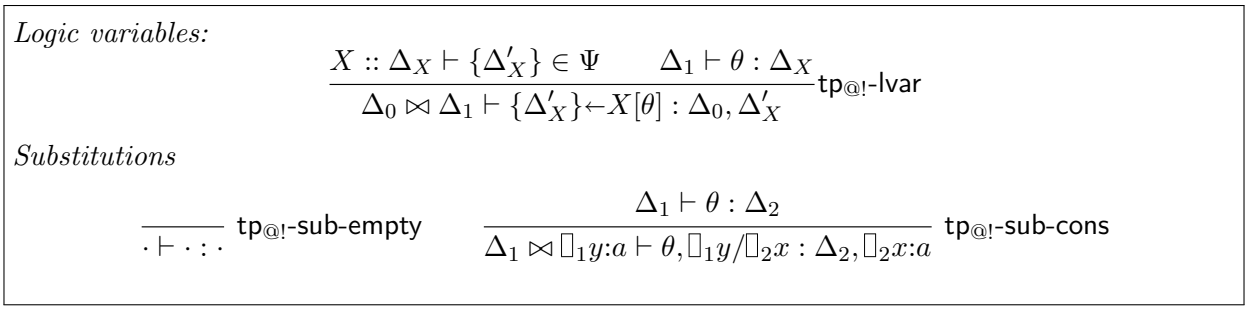


Figure 6: Typing Rules for Substitutions and Logic Variables of $\text{CLF}_{@!}$

values for $(\Box, \Box')$ are given by the reflexive-transitive closure of $\{(!, @), (@, \downarrow)\}$. Substitutions of this form are called *linear changing*.

Substitutions are type-checked using the same judgment as in $\text{CLF}_{\multimap}$: $\Delta \vdash \theta : \Delta'$. The updated typing rules for logic variables and substitutions are given in Figure 6. In rule $\mathsf{tp}_{@!}$-sub-cons, we split the context according to the modalities $(\Box_1, \Box_2)$ as explained above.

Different from $\text{CLF}_{\multimap}$, a substitution in $\text{CLF}_{@!}$ is not necessarily injective, since persistent variables can occur more than once. The lemma below states the conditions under which the inverse of a substitution can be applied to an expression. The statement relies on the following definition.

A variable $\Box x$ occurs in a *linear position* if it occurs in a step of the form $\{\Delta\}{\leftarrow}c{\cdot}(S_1, \Box x, S_2)$, where the type of c is of the form $\Delta_1, \downarrow y{:}a, \Delta_2 \multimap \{\Delta'\}$ with Δ_1 and Δ_2 of the same size as S_1 and S_2. Similarly, if the type of c is of the form $\Delta_1, @y{:}a, \Delta_2 \multimap \{\Delta'\}$ or $\Delta_1, !y{:}a, \Delta_2 \multimap \{\Delta'\}$, we say that $\Box x$ occurs in an *affine position* or *persistent position*, respectively.

The inverse of a linear-changing substitution can be applied to a term under certain conditions stated in the following lemma.

Lemma 3.3 (Inversion for $\text{CLF}_{@!}$) *Let E be an expression and θ a linear-changing pattern substitution. Then, there exists E' such that $E \equiv \theta E'$ iff the following conditions hold:*

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

Renamings The main issue in designing a matching algorithm for traces is how to deal with variables introduced in the trace. Recall that, differently from $\text{CLF}_{\multimap}$, in a $\text{CLF}_{@!}$ 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:

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

We write $\epsilon\{\varphi\}$ and $\Delta\{\varphi\}$ for the application of the renaming φ to the free variables *and bound variables* in a trace ϵ and a context Δ, respectively. Application of a renaming is defined by the following rules:

$$\begin{aligned}(\diamond)\{\varphi\} &= \diamond\\ (\{\Delta\}{\leftarrow}c{\cdot}S)\{\varphi\} &= \{\Delta\{\varphi\}\}{\leftarrow}c{\cdot}\varphi S\\ (\epsilon_1;\epsilon_2)\{\varphi\} &= \epsilon_1\{\varphi\};\epsilon_2\{\varphi\}\\ {\cdot}\{\varphi\} &= \cdot\\ \Delta,{\square}x{:}a\{\varphi\} &= \Delta\{\varphi\},{\square}\varphi x{:}a\end{aligned}$$

The composition $\varphi_1\varphi_2$ of two renamings φ_1 and φ_2 is defined as their union. Renamings are used to keep track of the one-to-one correspondence between local variables in a trace matching problem. When we compose two renamings during a matching problem, their domains are disjoint (once a variable has been renamed, it is not renamed again).

Design of the 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. Intuitively, the algorithm is based on the $\mathrm{CLF}_{\multimap}$-style judgment of the form $\Delta \vdash \epsilon_1 \stackrel{?}{=} \epsilon_2 \mapsto \sigma$ which attempts to match ϵ_1 against ϵ_2 (both well typed under context Δ) and results in the assignment σ (although we will update this judgment shortly). However, with the addition of affine and intuitionistic hypotheses, the invariants of the algorithm are different in $\mathrm{CLF}_{@!}$. We will now illustrate some of the design choices we made with some examples.

Let us consider the following matching problem (for simplicity deprived of logic variables):

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

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. Unlike $\mathrm{CLF}_{\multimap}$, the variables introduced by the two traces do not occur in each expression's context (here "$\cdot$"). Therefore, there is no obvious way to rename these variables at the outset so that the output interface of the two traces match. We will handle this problem by delaying giving these variable a common name: specifically, rather than relying on explicit α-renaming, we will incrementally compute a renaming for each side of the equation as the algorithm gathers information. Consequently, we will need to equip most matching judgments with additional arguments representing these renamings, as discussed below.

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

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

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

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

Now matching $\{\cdot\}\leftarrow c\,!z$ against $\{\cdot\}\leftarrow c\,!y_1$ 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 as $\underline{x}$). For example, the equation over expressions

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

reduces to the following trace equation, where all variables bound in a step have been marked:

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

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

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

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

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

which is derivable iff $\Delta_1\{\varphi_1\} \equiv \Delta_2\{\varphi_2\}$. It is an invariant of this judgment that the domains of φ_1 and φ_2 contain only marked variables and the codomains contain only unmarked variables. We call a renaming with this property a *matching renaming*.

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

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

derivable iff $(\sigma\epsilon_1)\{\varphi_1\} \equiv \epsilon_2\{\varphi_2\}$.

An invariant of our 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:

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

As the output of the trace on the right ($!\underline{y_1}$ and $!\underline{y_2}$) is not used, these variables 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: $X \leftarrow \{\mathsf{let}\ \{!\underline{y_1}\}\leftarrow c_1\,!x; \{!\underline{y_2}\}\leftarrow c_2\,!x\ \mathsf{in}\ \bigcirc\}$. However, the output $\bigcirc$ is missing. This problem has a solution only if $!x_1$ has the same type as either $!y_1$ or $!y_2$.

This example is a particular case of a matching problem of the form

$$\left(\{\Delta\}\leftarrow X[\theta]\right)\ \stackrel{?}{=}\ \epsilon$$

which 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 $\Delta_2\{\varphi_2\} \preccurlyeq \Delta\{\varphi_1\}$.

The algorithm It is defined by the following judgments:

$$\begin{aligned}
\text{Contexts: } \Delta_1 &\stackrel{?}{=} \Delta_2 \mapsto \varphi_1; \varphi_2 \\
\Delta_1 &\succcurlyeq \Delta_2 \mapsto \varphi_1; \varphi_2 \\
\text{Spines: } S_1 &\stackrel{?}{=} S_2 \mapsto \varphi_1; \varphi_2 \\
\text{Expressions: } \Delta \vdash E_1 &\stackrel{?}{=} E_2 \mapsto \sigma \\
\text{Traces: } \Delta \vdash \epsilon_1 &\stackrel{?}{=} \epsilon_2 \mapsto \sigma; \varphi_1; \varphi_2
\end{aligned}$$

The first context judgment is used for matching contexts introduced by steps, while the second judgment is used for matching the output contexts in equations of the form $(\{\Delta\} \leftarrow X[\theta]) \stackrel{?}{=} \epsilon$. They are defined in Figure 7. The output renamings are always matching renamings. This is an invariant of the matching algorithm (easily checked by induction on the rules). Rules $\mathsf{dec}_{@!}$-ctx-eq-* deal with context equality. A judgment $\Delta_1 \stackrel{?}{=} \Delta_2 \mapsto \varphi_1; \varphi_2$ is derivable if Δ_1 and Δ_2 have the same length, and variables in corresponding positions are either both marked (rule $\mathsf{dec}_{@!}$-ctx-eq-mark), or both unmarked (rule $\mathsf{dec}_{@!}$-ctx-eq-unmark).

The judgment $\Delta_1 \succcurlyeq \Delta_2 \mapsto \varphi_1; \varphi_2$, defined by the rules $\mathsf{dec}_{@!}$-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 $\mathsf{dec}_{@!}$-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 $\mathsf{dec}_{@!}$-ctx-weak-mark). Similarly, an unmarked variable in Δ_1 must be matched against an unmarked variable in Δ_2 (rule $\mathsf{dec}_{@!}$-ctx-weak-unmark).

The judgment $S_1 \stackrel{?}{=} S_2 \mapsto \varphi_1; \varphi_2$ is given by rules $\mathsf{dec}_{@!}$-sp-* in Figure 8. It is similar to context matching, except that spines contain no type information and variables may be repeated. We require that variables in corresponding positions in matching spines be both marked or both unmarked.

Matching on expressions and traces is defined in Figure 8. We write $\underline{\epsilon}$ for the trace obtained by marking every variable introduced in ϵ; similarly we write $\underline{\Delta}$ for the context obtained by replacing every variable x declared in Δ by $\underline{x}$. We write $\overline{\Delta}$ for the context obtained by removing all marked variables from Δ.

Matching on expressions is given by rule $\mathsf{dec}_{@!}$-expr. The problem is reduced to matching traces by renaming the part of the output interfaces that is collected in the expression (recall that renamings introduce fresh variables). This does not imply that both traces have the same interface, as they might contain persistent and affine hypotheses that are not present in the context. Unlike the case of $\mathrm{CLF}_{-\circ}$, we do not use implicit α-renaming but instead build a renaming between the interfaces on both traces incrementally.

Matching on traces is given by the rules $\mathsf{dec}_{@!}$-tr-*. In a matching equation of the form $\epsilon_1 \stackrel{?}{=} \epsilon_2$ we leave the order of ϵ_1 fixed, while we implicitly reorder ϵ_2 to match the order of ϵ_1. Rule $\mathsf{dec}_{@!}$-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 $\mathsf{dec}_{@!}$-tr-step is similar to the corresponding rule in $\mathrm{CLF}_{-\circ}$, except that we build explicit renamings and we keep track of the context where both traces are well typed.

Finally, rule $\mathsf{dec}_{@!}$-tr-inst handles the case of a monadic logic variable. The input context on both sides is the same, but the output context might differ. Hence, we need to ensure that all

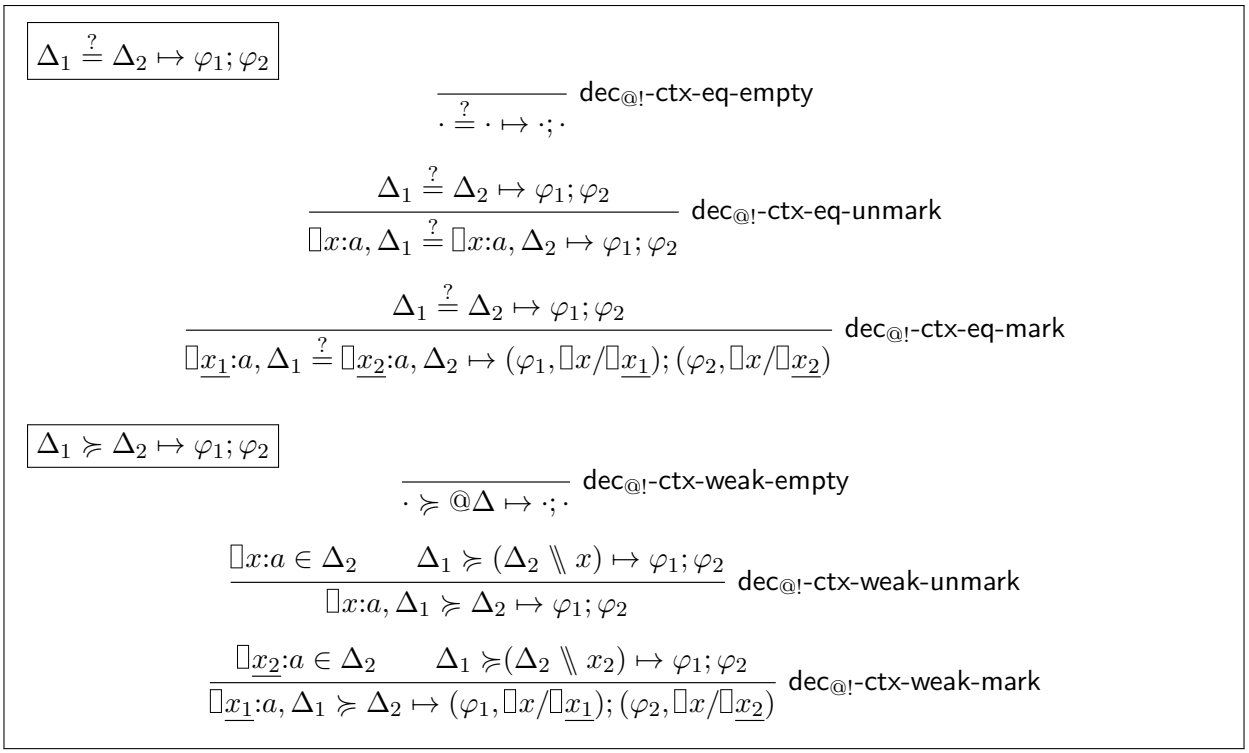


Figure 7: Matching on Contexts in CLF$_{@!}$

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 $\{\mathsf{let}\ \epsilon_2\{\varphi_2\}\ \mathsf{in}\ \Delta'\{\varphi_1\}\}$.

Correctness of the algorithm The proof of soundness of the matching algorithm proceeds in a similar way as in the case of CLF$_{\multimap}$. However, in the case of CLF$_{@!}$ we cannot enforce that both sides of a matching equation should have the same type (as shown in the examples above).

The following lemma states soundness of the matching algorithm for contexts.

Lemma 3.4 (Soundness of context matching for CLF$_{@!}$)

- *If* $\Delta_1 \stackrel{?}{=} \Delta_2 \mapsto \varphi_1; \varphi_2$, *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. □

Soundness of matching for spines is similar to context equality.

Lemma 3.5 (Soundness of spine matching for CLF$_{@!}$)

If $S_1 \stackrel{?}{=} S_2 \mapsto \varphi_1; \varphi_2$, *then* $\varphi_1 S_1 = \varphi_2 S_2$.

Proof: By induction on the given derivation. □

Finally, the next lemma states soundness of matching for expressions and traces.

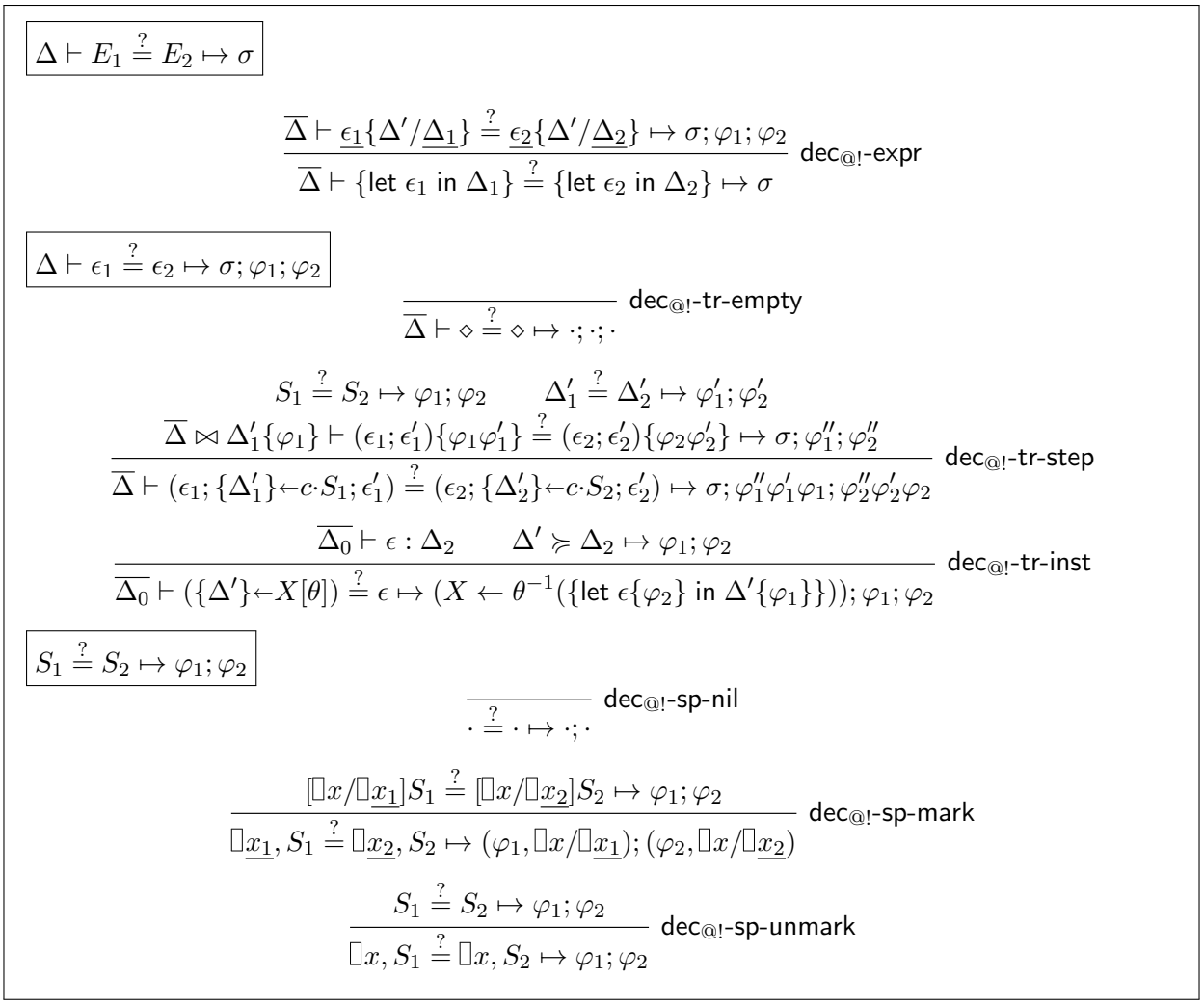


Figure 8: Matching Algorithm for $\text{CLF}_{@!}$

Lemma 3.6 (Soundness of matching for $\text{CLF}_{@!}$)

- *If* $\overline{\Delta} \vdash E_1, E_2 \Leftarrow \{\Delta'\}$ *and* $\Delta \vdash E_1 \stackrel{?}{=} E_2 \mapsto \sigma$, *then* $\sigma E_1 \equiv E_2$.
- *If* $\overline{\Delta} \vdash \epsilon_1 : \Delta_1$ *and* $\overline{\Delta} \vdash \epsilon_2 : \Delta_2$ *and* $\overline{\Delta} \vdash \epsilon_1 \stackrel{?}{=} \epsilon_2 \mapsto \sigma; \varphi_1; \varphi_2$, *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 only consider 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 $\mathsf{dec}_{@!}$**-tr-step.** Let $\overline{\Delta} \vdash \epsilon_1; \{\Delta_1'\}{\leftarrow}c{\cdot}S_1; \epsilon_1' : \Delta_1$ and $\overline{\Delta} \vdash \epsilon_2; \{\Delta_2'\}{\leftarrow}c{\cdot}S_2; \epsilon_2' : \Delta_2$. Assume that $c{:}\Delta_c \multimap \{\Delta_c'\} \in \Sigma$. By inverting on the typing derivation, there exists Δ_i^* and Δ_i^S (for $i = 1, 2$) such that

$$\overline{\Delta} \xrightarrow{\epsilon_i} \Delta_i^* \bowtie \Delta_i^S \xrightarrow{\{\Delta_i'\}\leftarrow c\cdot S_i} \Delta_i^* \bowtie \Delta_i' \xrightarrow{\epsilon_i'} \Delta_i$$

where $\Delta_i^S \vdash \{\Delta_i'\}{\leftarrow}c{\cdot}S_i : \Delta_i'$ and Δ_i^S is the minimum context such that $\Delta_i^S \vdash S_i : \Delta_c$. By IH, $\varphi_1 S_1 \equiv \varphi_2 S_2$, and $\Delta_1'\{\varphi_1'\} \equiv \Delta_2'\{\varphi_2'\}$. From the former, we observer that $\Delta_1^S\{\varphi_1\} \equiv \Delta_2^S\{\varphi_2\}$. Applying the frame rule to ϵ_i and ϵ_i' and combining the results, we have $\overline{\Delta} \bowtie \Delta_i' \vdash \epsilon_i; \epsilon_i' : \Delta_i \bowtie \Delta_i^S$. Applying the renaming $\varphi_i\varphi_i'$ we have that $\epsilon_i; \epsilon_i'\{\varphi_i\varphi_i'\}$ is well typed under context $\overline{\Delta} \bowtie \Delta_1'\{\varphi_1\}$ (equal to $\overline{\Delta} \bowtie \Delta_2'\{\varphi_2\}$).

By IH, $\sigma\epsilon_1; \epsilon_1'\{\varphi_1\varphi_1'\} \equiv \epsilon_2; \epsilon_2'\{\varphi_2\varphi_2'\}$. The result follows.

Rule $\mathsf{dec}_{@!}$**-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.

□

Next, we give completeness statements for the matching judgments. Recall that a matching renaming is a renaming whose domain contains only marked variables and its codomain contains only unmarked variables. Furthermore, a renaming is a bijection whose codomain contains only fresh variables.

Lemma 3.7 (Completeness of context matching for CLF$_{@!}$)

- *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 $\Delta_1 \stackrel{?}{=} \Delta_2 \mapsto \varphi_1'; \varphi_2'$.*
- *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 3.8 (Completeness of spine matching for CLF$_{@!}$) *Let S_1 and S_2 be spines and φ_1 and φ_2 matching renamings such that $\varphi_1 S_1 = \varphi_2 S_2$. Then there exists $\varphi_1' \subseteq \varphi_1$ and $\varphi_2' \subseteq \varphi_2$ such that $S_1 \stackrel{?}{=} S_2 \mapsto \varphi_1'; \varphi_2'$.*

Proof: By induction on the structure of the spine. □

Completeness of matching for traces is similar to the case of CLF$_{\multimap}$. However, we do not assume that both traces have the same output interface. We write $\Delta_1 \approx_{\downarrow} \Delta_2$ to mean Δ_1 and Δ_2 coincide (at least) in their linear declarations, i.e., if there exists Δ_0, Δ_1', and Δ_2', with $\mathsf{nolin}(\Delta_1')$ and $\mathsf{nolin}(\Delta_2')$, such that $\Delta_1 = \Delta_0 \bowtie \Delta_1'$ and $\Delta_2 = \Delta_0 \bowtie \Delta_2'$.

Lemma 3.9 (Completeness of trace matching for CLF$_{@!}$) *Let ϵ_1 and ϵ_2 be expressions such that ϵ_2 is ground and $\overline{\Delta} \vdash \epsilon_1 : \Delta_1$ and $\overline{\Delta} \vdash \epsilon_2 : \Delta_2$, with $\Delta_1 \approx_{\downarrow} \Delta_2$. Assume there exists matching 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 exists φ_1' and φ_2' such that $\overline{\Delta} \vdash \epsilon_1 \stackrel{?}{=} \epsilon_2 \mapsto \sigma; \varphi_1'; \varphi_2'$ is derivable and $\varphi_1'|_{\epsilon_2\bullet} = \varphi_1$ and $\varphi_2'|_{\epsilon_2\bullet} \subseteq \varphi_2$.*

Proof: Let Δ_0, Δ_1', and Δ_2' be contexts, with $\mathsf{nolin}(\Delta_1)$ and $\mathsf{nolin}(\Delta_2)$, such that $\Delta_1 = \Delta_0 \bowtie \Delta_1'$ and $\Delta_2 = \Delta_0 \bowtie \Delta_2'$. We can assume that all internal variables in ϵ_1 and ϵ_2 are marked, as well as the variables in $@\epsilon_1\bullet$ and $@\epsilon_2\bullet$. That is, we can consider the traces $\epsilon_i' = \underline{\epsilon_i}\{\overline{\Delta_0}/\underline{\Delta_0}\}$, for $i = 1, 2$, because ϵ_1' and ϵ_2' satisfy the same hypotheses as the lemma.

We proceed by induction on the length of ϵ_1. We have three cases.

1. Consider the case $\epsilon_1 = \delta; \epsilon_1'$, where $\delta = \{\Delta_*\}{\leftarrow}c{\cdot}S$. Then ϵ_2 must be of the form $\delta'; \epsilon_2'$ with $\delta' = \{\Delta_*'\}{\leftarrow}c{\cdot}S'$. Furthermore, there exist internal renamings φ and φ' such that $\delta\{\varphi_1\}\{\varphi\} \equiv \delta'\{\varphi_2\}\{\varphi'\}$ and $\varphi\epsilon_1'\{\varphi_1\} \equiv \varphi'\epsilon_2'\{\varphi_2\}$. We can choose φ and φ' such that $\varphi|_{\epsilon_1\bullet} \subseteq \epsilon_1$ and $\varphi'|_{\epsilon_1\bullet} \subseteq \epsilon_2$. Matching Δ_* and Δ_*' succeeds (Lemma 3.7). Note that $S = S'$ since they only include variables in $\overline{\Delta}$. Traces ϵ_1' and ϵ_2' satisfy the conditions of the lemma using renamings $\varphi_1|_{@\epsilon_1'\bullet}$ and $\varphi_2|_{@\epsilon_2'\bullet}$. By IH, there exists a derivation of $\overline{\Delta} \bowtie \Delta_*\{\varphi\} \vdash \epsilon_1'\{\varphi\} \stackrel{?}{=} \epsilon_2'\{\varphi'\}$. The result follows by applying rule $\mathsf{dec}_{@!}$-tr-step.

2. Consider the case $\epsilon_1 = \{\Delta_*\}{\leftarrow}X[\theta]; \epsilon_1'; \delta$, where where $\delta = \{\Delta_*\}{\leftarrow}c{\cdot}S$. Then ϵ_2 must be of the form $\epsilon_2'; \delta'$ with $\delta' = \{\Delta_*'\}{\leftarrow}c{\cdot}S'$. Furthermore, there exist internal renamings φ and φ' such that $\varphi\delta\{\varphi_1\} \equiv \varphi'\delta'\{\varphi_2\}$ and $(\{\Delta_*\}{\leftarrow}X[\theta]; \epsilon_1'\{\varphi_1\})\{\varphi\} \equiv \epsilon_2'\{\varphi_2\}\{\varphi'\}$. We have $\Delta_*\{\varphi_1\} \equiv \Delta_*\{\varphi_2\}$; matching the contexts succeeds. Furthermore, $\varphi\varphi_1 S = \varphi'\varphi_2 S'$; matching the spines succeeds. Similar to the previous case, the result follows from the IH and rule $\mathsf{dec}_{@!}$-tr-step.

3. Finally, consider the case $\epsilon_1 = \{\Delta_1'\}{\leftarrow}X[\theta]$. Let $\sigma = X \leftarrow \{\mathsf{let}\ \epsilon_0\ \mathsf{in}\ \Delta_0\}$. The hypothesis says that $(\theta\epsilon_0)\{\Delta_1'/\Delta_0\{\theta\}\}\{\varphi_1\} \equiv \epsilon_2\{\varphi_2\}$. Now, $(\theta\epsilon_0)\{\Delta_1'/\Delta_0\{\theta\}\}\{\varphi_1\} \equiv (\theta\epsilon_0)\{\Delta_1'\{\varphi_1\}/\Delta_0\{\theta\}\}$. Then, the term $\{\mathsf{let}\ \epsilon_2\{\varphi_2\}\ \mathsf{in}\ \Delta_1'\{\varphi_1\}\}$ is well typed. Then $\Delta_1'\{\varphi_1\} \succcurlyeq \Delta_2\{\varphi_2\}$. The result follows from completeness of context matching.

□

Lemma 3.10 (Completeness of expression matching for CLF$_{@!}$) *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 $\Delta \vdash E_1 \stackrel{?}{=} E_2 \mapsto \sigma$ is derivable.*

Proof: Follows directly from the previous lemma. □

3.3 Unification

As in CLF$_{\multimap}$, we restrict ourselves to the subset of CLF$_{@!}$ where there is at most one logic variable occurrence in each side of the equation, and these variables are distinct. The unification algorithm proceeds like matching, reducing the problem to the base case

$$\overline{\Delta} \vdash \epsilon_1; (\{\Delta_1\}{\leftarrow}X_1[\theta_1]) \stackrel{?}{=} (\{\Delta_2\}{\leftarrow}X_2[\theta_2]); \epsilon_2$$

The intention is, as in CLF$_{\multimap}$, to assign ϵ_2 to X_1 and ϵ_1 to X_2. We assume that the linear parts of the output coincide, but the affine and persistent do not have to coincide. We apply renamings on both sides for the output interfaces, as well as the internal interfaces between ϵ_1 and $\{\Delta_1\}{\leftarrow}X_1[\theta_1]$, and between $(\{\Delta_2\}{\leftarrow}X_2[\theta_2])$ and ϵ_2.

Similar to the case of CLF$_{\multimap}$ the general solution to this problem is the following:

$$X_1 \leftarrow (\varphi_1\theta_1)^{-1}\{\mathsf{let}\ \{\Delta'\}{\leftarrow}Z[\theta'];\epsilon_2\{\varphi_2\}\ \mathsf{in}\ \Delta_1\{\varphi_1\}\}$$
$$X_2 \leftarrow (\varphi_2\theta_2)^{-1}\{\mathsf{let}\ \epsilon_1\{\varphi_1\};\{\Delta'\}{\leftarrow}Z[\theta']\ \mathsf{in}\ \Delta_2\{\varphi_2\}\}$$

The following conditions ensure that both terms above are well typed.

$$\begin{aligned}
\mathrm{FV}(\varphi_1\theta_1) &\preccurlyeq \mathrm{FV}(\theta') \cup (\bullet\epsilon_2\{\varphi_2\} \setminus \mathrm{dom}(\Delta')) \\
\mathrm{FV}(\varphi_2\theta_2) &\preccurlyeq \bullet\epsilon_1\{\varphi_1\} \cup (\mathrm{FV}(\theta') \setminus \epsilon_1\bullet) \\
\mathrm{dom}(\Delta_1\{\varphi_1\}) &\succcurlyeq \epsilon_2\{\varphi_2\}\bullet \cup (\mathrm{dom}(\Delta') \setminus \bullet\epsilon_2\{\varphi_2\}) \cup !\mathrm{dom}(\Delta') \\
\mathrm{dom}(\Delta_2\{\varphi_2\}) &\succcurlyeq \mathrm{dom}(\Delta') \cup (\epsilon_1\bullet \setminus \mathrm{FV}(\theta')) \cup !(\epsilon_1\bullet)
\end{aligned}$$

As in CLF$_{\multimap}$, there is an infinite number of solutions to these equations, whenever any exists. Assume that $(\varphi_1, \varphi_2, \theta', \Delta')$ is a solution. Then, $(\varphi_1, \varphi_2, \theta', (\Delta', @\Delta_0))$ is also a solution for any $@\Delta_0$. However, these solution are not useful. We can restrict Δ' so that they do not occur. Specifically, we only search for solutions that satisfy $\mathrm{dom}(\Delta') \subseteq \mathrm{dom}(\Delta_2\{\varphi_2\})$. Note from the first equation that $\mathrm{dom}(\theta') \subseteq \mathrm{dom}(\theta_1)$. The search space is thus finite, modulo equivalent renamings. Recall, however, that the reduction of a unification problem to a problem of the form (3.3) is not unique.

3.4 Extensions of CLF$_{@!}$

We briefly consider an extension of CLF$_{@!}$ that extends the syntax of expressions to allow spines in place of the trailing context of an expression. Expressions are then defined by the grammar:

$$E ::= \{\mathsf{let}\ \epsilon\ \mathsf{in}\ S\}$$

For example, in this extension the expression $\{\mathsf{let}\ \{!x\}{\leftarrow}c\ \mathsf{in}\ !x, !x\}$ is well typed. The typing rule for expressions is adapted as expected, typing the spine in the output context of the trace:

$$\frac{\Delta_1 \vdash \epsilon : \Delta_2 \qquad \Delta_2 \vdash S : \Delta'}{\Delta_1 \vdash \{\mathsf{let}\ \epsilon\ \mathsf{in}\ S\} \Leftarrow \{\Delta'\}}$$

This implies that the spine must use all linear resources produced by the trace, but might not mention all affine or persistent resources.

Matching We adapt the matching algorithm to this extension. The main change is in the matching of trailing spines. It is now possible to match a marked persistent variable against an unmarked persistent variable. The matching rules for spines are the following:

$$\frac{}{\cdot \stackrel{?}{=} \cdot \mapsto \cdot;\cdot}\ \mathsf{dec}_{@!}\text{-}\mathsf{sp}\text{-}\mathsf{nil}$$

$$\frac{[\Box x/\Box \underline{x_1}]S_1 \stackrel{?}{=} [\Box x/\Box \underline{x_2}]S_2 \mapsto \varphi_1;\varphi_2}{\Box \underline{x_1}, S_1 \stackrel{?}{=} \Box \underline{x_2}, S_2 \mapsto (\varphi_1, \Box x/\Box \underline{x_1});(\varphi_2, \Box x/\Box \underline{x_2})}\ \mathsf{dec}_{@!}\text{-}\mathsf{sp\text{-}mark}$$

$$\frac{S_1 \stackrel{?}{=} S_2 \mapsto \varphi_1;\varphi_2}{\Box x, S_1 \stackrel{?}{=} \Box x, S_2 \mapsto \varphi_1;\varphi_2}\ \mathsf{dec}_{@!}\text{-}\mathsf{sp\text{-}unmark}$$

$$\frac{[!x_2/!\underline{x_1}]S_1 \stackrel{?}{=} S_2 \mapsto \varphi_1;\varphi_2}{!\underline{x_1}, S_1 \stackrel{?}{=} !x_2, S_2 \mapsto (\varphi_1, !x_2/!\underline{x_1});\varphi_2}\ \mathsf{dec}_{@!}\text{-}\mathsf{sp\text{-}mark\text{-}unmark}$$

With respect to Figure 8, we added rule $\mathsf{dec}_{@!}$-sp-mark-unmark to deal with cases such as the one at the beginning of this section, where different variables on the left will be assigned to the same variable on the right. Let us illustrate this point with an example. Consider the following matching problem, where all variables are marked as usual

$$\begin{pmatrix} \{!\underline{x_1}, !\underline{x_2}\}\leftarrow X[\cdot]; \\ \{\cdot\}\leftarrow c\cdot!\underline{x_1} \\ \{\cdot\}\leftarrow c\cdot!\underline{x_2} \end{pmatrix} \stackrel{?}{=} \begin{pmatrix} \{!\underline{y}\}\leftarrow c_0; \\ \{\cdot\}\leftarrow c\cdot!\underline{y} \\ \{\cdot\}\leftarrow c\cdot!\underline{y} \end{pmatrix}$$

Matching the last step on both traces reduces the problem to

$$\begin{pmatrix} \{!\underline{x_1}, !y\}\leftarrow X[\cdot]; \\ \{\cdot\}\leftarrow c\cdot!\underline{x_1} \end{pmatrix} \stackrel{?}{=} \begin{pmatrix} \{!y\}\leftarrow c_0; \\ \{\cdot\}\leftarrow c\cdot!y \end{pmatrix}$$

Matching again the last two steps involves using rule $\mathsf{dec}_{@!}$-sp-mark-unmark since we need to match $\underline{x_1}$ with y. The problem is reduced to

$$\{!y, !y\}\leftarrow X[\cdot]; \stackrel{?}{=} \{!y\}\leftarrow c_0;$$

This gives the solution $X \leftarrow \{\mathsf{let}\ \{!y\}\leftarrow c_0\ \mathsf{in}\ !y, !y\}$, by applying rule $\mathsf{dec}_{@!}$-sp-mark. Note that this rule and rules for context matching are unchanged.

However, note that the last equation is not well formed because of the context produced by X. To prove that this algorithm is correct, we need to relax the typing rules to allow this kind of context. Specifically, for steps of the form $\{\Delta\}\leftarrow X[\theta]$, we allow repeated persistent variables in Δ if they have the same type. We leave for future work to adapt to this extension the proofs of soundness and completeness given in the previous section.

4 CLF_{Π}: Dependent Types

CLF_{Π} is an extension of $\mathrm{CLF}_{@!}$ with dependent types and terms. CLF_{Π} is a significant fragment of full CLF [31, 7]; many real-world specifications fit in this fragment (see the examples in Section 4.3).

The main structure of the matching algorithm is the same as in $\mathrm{CLF}_{@!}$ and $\mathrm{CLF}_{\multimap}$. However, given the presence of terms in CLF_{Π}, we need to combine the matching algorithm for traces with traditional techniques of higher-order matching based on pattern substitutions [19]. Furthermore, because of dependent types, we cannot anymore remove steps from the middle of a trace, since this might break dependencies. Instead, the algorithm proceeds by removing steps from the beginning or the end of a trace (Section 4.2).

4.1 Language

CLF_Π is an extension of $\text{CLF}_{@!}$ with dependent types and terms. CLF_Π includes types and kinds as in LF, as well as full λ-terms. We use the LF-style presentation based on canonical forms introduced in [31]. However, to simplify the presentation of the matching algorithm, we keep more type information in the terms. This type information can be inferred by type reconstruction, so it is not necessary for the user to provide it.

Contexts Similarly to $\text{CLF}_{@!}$, a *context* is a sequence of variable declarations of the form $\square x{:}A$, where A is a type (CLF_Π types are defined below) and $\square$ is one of the three modalities of $\text{CLF}_{@!}$: ! (persistent), @ (affine), or $\downarrow$ (linear). In CLF_Π, context variables have dependent types instead of the base types of $\text{CLF}_{@!}$.

$$\textit{Contexts:}\quad \Delta \quad ::= \quad \cdot \mid \Delta, \square x{:}A$$

As usual, 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 dependencies on persistent, but not affine or linear declarations.

We use the same operations defined in Section 3.1 for $\text{CLF}_{@!}$. In particular, $!\Delta$, $@\Delta$, $\downarrow\Delta$, and $\Delta \setminus\!\setminus x$ have the same definition.

Terms The language of terms of our language combines aspects of the spine calculus of [6] and of the pattern-based presentation of CLF in [28]. 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 variable x or a constant c from the global signature Σ (whose definition remains as in previous systems). 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 Curry-style λ-calculus. An atomic term is closely related to the monadic terms of [31, 28]. 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 reduces the result at once to canonical form [31, 28]. 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 Traces are essentially the same as in $\text{CLF}_{@!}$. In particular, the argument of steps is a spine.

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

As in $\text{CLF}_{@!}$, the context Δ of an expression $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$ collects all unused linear variables and, some affine and persistent variables. In full CLF, expressions can appear in terms. We disallow this here. Terms of the form $\widehat{\lambda}\Delta.\, E$ are not considered since they play no part in our matching algorithm, although we occasionally use them in the examples (see Section 4.3). Note that the head of a step must be a constant from the signature — we will allow variables in Section 5.

Types and kinds While $\text{CLF}_{\multimap}$ and $\text{CLF}_{@!}$ only mentioned constant types and only featured traces, CLF_{Π} includes dependent types and full terms.

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

$$\begin{array}{rrcl} \textit{Base types:} & P & ::= & a{\cdot}S \mid \{\Delta\} \\ \textit{Types:} & A & ::= & \Pi\Delta.P \\ \textit{Kinds:} & K & ::= & \Pi!\Delta.\mathsf{type} \end{array}$$

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 [29]. As in previous systems, 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: we write type for $\Pi \cdot .\mathsf{type}$ and P for $\Pi \cdot .P$.

Equality Equality for terms, spines, types and contexts, denoted $\equiv$, is defined in the usual way, up to α-equivalence; its definition is omitted. Equality for traces and expressions is defined as in $\text{CLF}_{@!}$. The notions of trace interface and trace independence are also as in $\text{CLF}_{@!}$ (Section 3.1).

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

$$\begin{array}{rl} \textit{Kinds:} & !\Delta \vdash K : \mathsf{kind} \\ \textit{Base types:} & !\Delta \vdash P : \mathsf{type} \\ \textit{Types:} & !\Delta \vdash A : \mathsf{type} \\ \textit{Contexts:} & !\Delta \vdash \Delta' \\ \textit{Signatures:} & \vdash \Sigma \end{array} \qquad \begin{array}{rl} \textit{Expressions:} & \Delta \vdash E \Leftarrow \{\Delta\} \\ \textit{Traces:} & \Delta \vdash \epsilon : \Delta' \\ \textit{Spines:} & \Delta \vdash S \Leftarrow \Delta' \\ \textit{Terms:} & \Delta \vdash N \Leftarrow A \end{array}$$

Their definition, in Figures 9 and 10, relies on equality $\equiv$ (defined below) and the two additional relations already seen in Section 3.1: $\Delta = \Delta_1 \bowtie \Delta_2$ and $\Delta_1 \preccurlyeq \Delta_2$. Recall that if $\Delta = \Delta_1 \bowtie \Delta_2$, then $!\Delta = !\Delta_1 = !\Delta_2$ and each affine or linear declaration in Δ appear in exactly one of Δ_1 and

Kinds:

$$\frac{!\Delta \vdash !\Delta'}{!\Delta \vdash \Pi!\Delta'.\mathsf{type} : \mathsf{kind}}\ \mathsf{tp}_\Pi\text{-kind}$$

Base types:

$$\frac{a{:}\Pi!\Delta'.\mathsf{type} \in \Sigma \qquad !\Delta \vdash S : !\Delta'}{!\Delta \vdash a{\cdot}S : \mathsf{type}}\ \mathsf{tp}_\Pi\text{-tp-atom} \qquad \frac{!\Delta \vdash \Delta'}{!\Delta \vdash \{\Delta'\} : \mathsf{type}}\ \mathsf{tp}_\Pi\text{-tp-mon}$$

Types:

$$\frac{!\Delta \vdash \Delta' \qquad !(\Delta, \Delta') \vdash P : \mathsf{type}}{!\Delta \vdash \Pi\Delta'.P : \mathsf{type}}\ \mathsf{tp}_\Pi\text{-tp-prod}$$

Contexts:

$$\frac{}{!\Delta \vdash \cdot}\ \mathsf{tp}_\Pi\text{-ctx-empty} \qquad \frac{!\Delta \vdash A : \mathsf{type} \qquad !(\Delta, \Box x{:}\Box A) \vdash \Delta'}{!\Delta \vdash \Box x{:}\Box A, \Delta'}\ \mathsf{tp}_\Pi\text{-ctx-cons}$$

Signatures:

$$\frac{}{\vdash \cdot}\ \mathsf{tp}_\Pi\text{-sig-empty} \qquad \frac{\vdash \Sigma \qquad \cdot \vdash A : \mathsf{type}}{\vdash \Sigma, c{:}A}\ \mathsf{tp}_\Pi\text{-sig-type} \qquad \frac{\vdash \Sigma \qquad \cdot \vdash K : \mathsf{kind}}{\vdash \Sigma, a{:}K}\ \mathsf{tp}_\Pi\text{-sig-kind}$$

Figure 9: Typing Rules for Types and Kinds of CLF_Π

Δ_2; and $\Delta \preccurlyeq \Delta'$ 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 implicitly rely on α-renaming to ensure that a context extension or concatenation does not declare duplicate variable names.

The rules for types and kinds in Figure 9 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 persistent variables while ! applied to constructed contexts filters out non-persistent declarations. Context typing works similarly.

The typing rules for terms and spines are a minor variant of the semantics of [6, 28] for these entities. The typing rules for traces reflect that a trace can be seen as a state transformation. The empty trace does not change the state. A step transforms a part of the state. The spine S in rule tp_Π-step uses Δ_1 and produces the 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 [31, 28].

4.2 Matching

The infrastructure needed to carry out matching in CLF_Π differs from that of $\mathrm{CLF}_{@!}$ mainly by the addition of judgments that handle terms. The matching algorithm is extended with some of the standard means to solve equations for atomic variables. Dependencies also limit matching steps at

Terms:

$$\frac{!\Delta \vdash \Delta' \qquad \Delta, \Delta' \vdash H \cdot S \Leftarrow P}{\Delta \vdash \widehat{\lambda}\Delta'.\, H \cdot S \Leftarrow P}\ \mathsf{tp}_\Pi\text{-lam}$$

$$\frac{\Box x{:}\Pi\Delta'.a \cdot S' \in \Delta \qquad \Delta \setminus\!\setminus \Box x \vdash S \Leftarrow \Delta' \qquad P \equiv a \cdot S'[S/\Delta']}{\Delta \vdash \Box x \cdot S \Leftarrow P}\ \mathsf{tp}_\Pi\text{-var}$$

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

Spines:

$$\frac{}{@\Delta \vdash \cdot : \cdot}\ \mathsf{tp}_\Pi\text{-sp-empty}$$

$$\frac{!\Delta_1 \vdash N \Leftarrow P \qquad \Delta_0 \vdash S[!N/!x] : \Delta_2[!N/!x]}{\Delta_0 \bowtie @\Delta_1 \vdash !N, S \Leftarrow !x{:}A, \Delta_2}\ \mathsf{tp}_\Pi\text{-sp-bang}$$

$$\frac{@\Delta_1 \vdash N \Leftarrow P \qquad \Delta_0 \vdash S[@N/@x] : \Delta_2[@N/@x]}{\Delta_0 \bowtie @\Delta_1 \vdash @N, S \Leftarrow @x{:}A, \Delta_2}\ \mathsf{tp}_\Pi\text{-sp-aff}$$

$$\frac{\downarrow\Delta_1 \vdash N \Leftarrow P \qquad \Delta_0 \vdash S[\downarrow N/\downarrow x] : \Delta_2[\downarrow N/\downarrow x]}{\Delta_0 \bowtie \downarrow\Delta_1 \vdash \downarrow N, S \Leftarrow \downarrow x{:}A, \Delta_2}\ \mathsf{tp}_\Pi\text{-sp-lin}$$

Traces:

$$\frac{}{\Delta \vdash \diamond : \Delta}\ \mathsf{tp}_\Pi\text{-empty} \qquad \frac{\Delta \vdash \epsilon_1 : \Delta_1 \qquad \Delta_1 \vdash \epsilon_2 : \Delta_2}{\Delta \vdash \epsilon_1; \epsilon_2 : \Delta_2}\ \mathsf{tp}_\Pi\text{-comp}$$

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

Expressions:

$$\frac{\Delta_0 \vdash \epsilon : \Delta_1 \qquad \Delta_1 \preccurlyeq \Delta'}{\Delta_0 \vdash \{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta'\} \Leftarrow \{\Delta'\}}\ \mathsf{tp}_\Pi\text{-expr}$$

Figure 10: Typing Rules for Terms and Traces of CLF_Π

the beginning or at the end of a trace only.

Equations As in the cases of $\mathrm{CLF}_{\multimap}$ and $\mathrm{CLF}_{@!}$, we consider steps with logic variables. However, since CLF_Π also includes full terms, we need to consider logic variables as head of terms as well.

We slightly redefine the notion of contextual modal context to accommodate both kinds of logic variables. Concretely, a contextual modal context is a sequence of logic variable declarations, formally defined as follows:

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

where A can be either a monadic type (for logic variables occurring as head of steps), or an atomic type (for logic variables occurring as head of terms). We say that a logic variable is *atomic* (resp. *monadic*) if its type is atomic (resp. monadic).

As in the cases of $\mathrm{CLF}_{\multimap}$ and $\mathrm{CLF}_{@!}$, 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). The notion of substitution is also redefined to include full terms:

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

As in $\text{CLF}_{@!}$, we allow linear-changing substitutions.

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

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

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 11.

For the rest of the report, we assume that all logic variables have base types. We write $X[\theta]$ instead of $X[\theta]\cdot(\cdot)$. Logic variables with function types can indeed be *lowered* [21], 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 $\widehat{\lambda}\Delta.\, Y[\text{id}]$, where id is the identity substitution over the appropriate context.

Pattern substitutions A *pattern substitution* 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 $\theta^{-1}N$ [19]. However, it may have no solution if N contains variables not present in θ.

For linear-changing pattern substitutions the existence of the inverse applied to a term is subject to the same conditions on the occurrences of variables as in $\text{CLF}_{@!}$. In particular, 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. $@N$, $\downarrow N$). The following inversion result is similar to Lemma 3.3.

Lemma 4.1 (Inversion [30]) *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 $!y/\downarrow x \in \theta$, variable $!y$ occurs exactly once in T in a linear position;*
- *for every $!y/@x \in \theta$, variable $!y$ occurs at most once in T in a linear or affine position;*
- *for every $@y/\downarrow x \in \theta$, variable $@y$ occurs exactly once in T in a linear position.*

Design of the matching algorithm As for other systems in this report, we consider trace matching problems that have at most one monadic logic variable:

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

Logic variables:

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

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

Substitutions:

$$\frac{}{@\Delta \vdash \cdot : @\Delta}\ \mathsf{tp}_\Pi\text{-sub-empty} \qquad \frac{\Delta_0 \vdash \theta : \Delta_2 \qquad \square_1\Delta_1 \vdash N \Leftarrow A}{\Delta_0 \bowtie \square_1\Delta_1 \vdash \theta, \square_1 N/\square_2 x : \Delta_2, \square_2 x{:}A}\ \mathsf{tp}_\Pi\text{-sub-cons}$$

Figure 11: Typing Rules for Substitutions and Logic Variables of CLF_Π

where δ_i, δ_i' have the form $\{\Delta\}{\leftarrow}c{\cdot}S$, where S might now contain atomic logic variables. Assuming that all logic variables in a trace matching problem are applied to (linear-changing) pattern substitutions, matching is decidable for the reasons outlined in Section 2.2: in $\epsilon_1 \stackrel{?}{=} \epsilon_2$, any solution instantiates the monadic logic variables in ϵ_1 to subtraces of ϵ_2 and, since there are only finitely many subtraces, one can try all possible partitions of ϵ_2 among these monadic logic variables.

The basis of the algorithm for CLF_Π is similar to the algorithms for $\mathrm{CLF}_{\multimap}$ and $\mathrm{CLF}_{@!}$: matching individual steps until an empty trace is left on both sides, or a step with a monadic logic variable is left on the left side. However, unlike $\mathrm{CLF}_{\multimap}$ and $\mathrm{CLF}_{@!}$, steps have to be matched at either end of the trace. In the presence of dependent types, removing steps from the middle of the trace may lead to ill-typed traces. For example, consider the trace

$$\begin{pmatrix} \{!x_2, {\downarrow}y_2\}{\leftarrow}c{\cdot}!x_1; \\ \{!x_3, {\downarrow}y_3\}{\leftarrow}c{\cdot}!x_2; \\ \{!x_4, {\downarrow}y_4\}{\leftarrow}c{\cdot}!x_3 \end{pmatrix}$$

in a context containing ${\downarrow}!x_1{:}a$ and $c{:}\Pi!x{:}a.\{!x'{:}a, {\downarrow}y{:}b\,!x\}$. Removing the middle step leads to an ill-typed trace. Rule *-tr-step from the previous algorithms is divided in two rules: dec_Π-tr-step-hd and dec_Π-tr-step-tl matching steps at the beginning and at the end of the trace, respectively.

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 in Section 3.4. This is because substituting a trace can trigger reductions. 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 [10] 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

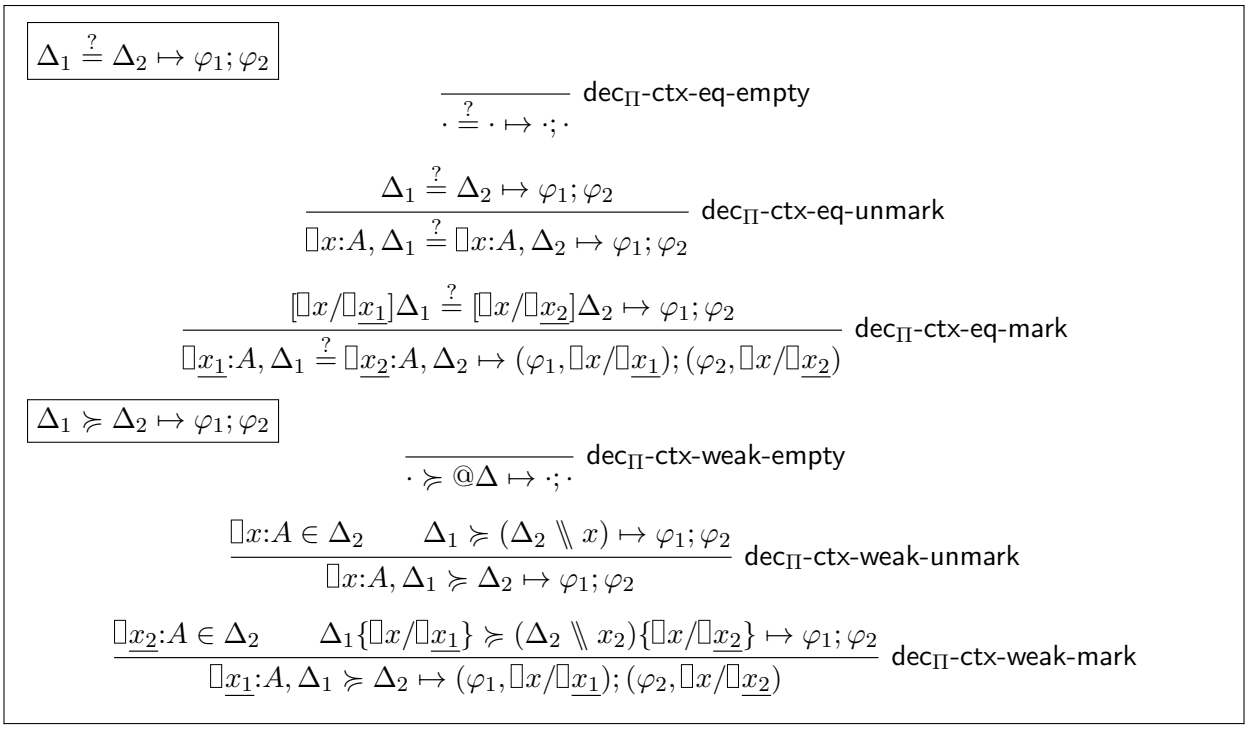


Figure 12: Matching on Contexts in CLF_Π

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:

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

Context matching is similar to $\text{CLF}_{@!}$, except for the presence of dependent types; the rules have the same structure.

When matching spines (and terms), each side is well typed in its own context. This is necessary for matching steps at the end of a trace: in the matching equation $\epsilon_1; \{\Delta_1\} \leftarrow c \cdot S_1 \stackrel{?}{=} \epsilon_2; \{\Delta_2\} \leftarrow c \cdot S_2$, spines S_1 and S_2 are well typed in different contexts that contain the variables introduced by ϵ_1 and ϵ_2, respectively. Similarly to $\text{CLF}_{@!}$, 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 13.

$\boxed{(\Delta_1 \vdash S_1) \stackrel{?}{=} (\Delta_2 \vdash S_2) \mapsto \sigma; \varphi_1; \varphi_2}$

$$\frac{}{(\cdot \vdash \cdot) \stackrel{?}{=} (\cdot \vdash \cdot) \mapsto \cdot; \cdot; \cdot} \ \mathsf{dec}_\Pi\text{-sp-nil}$$

$$\frac{(\square_1, \square_2) \in \{(@, !), (@, @), (\downarrow, \downarrow)\} \qquad (\Delta_1''|_{N_1} \vdash N_1) \stackrel{?}{=} (\Delta_2''|_{N_2} \vdash N_2) \mapsto \sigma; \varphi_1; \varphi_2 \quad (\sigma\Delta_1'\{\varphi_1\} \vdash \sigma\varphi_1 S_1) \stackrel{?}{=} (\Delta_2'\{\varphi_2\} \vdash \varphi_2 S_2) \mapsto \sigma'; \varphi_1'; \varphi_2'}{(\Delta_1' \bowtie \square_2\Delta_1'' \vdash \square_1 N_1, S_1) \stackrel{?}{=} (\Delta_2' \bowtie \square_2\Delta_2'' \vdash \square_1 N_2, S_2) \mapsto \sigma'\sigma; \varphi_1'\varphi_1; \varphi_2'\varphi_2} \ \mathsf{dec}_\Pi\text{-sp-cons}$$

$\boxed{(\Delta_1 \vdash N_1) \stackrel{?}{=} (\Delta_2 \vdash N_2) \mapsto \sigma; \varphi_1; \varphi_2}$

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

$\boxed{(\Delta_1 \vdash H_1 \cdot S_1) \stackrel{?}{=} (\Delta_2 \vdash H_2 \cdot S_2) \mapsto \sigma; \varphi_1; \varphi_2}$

$$\frac{(\Delta_1|_{S_1} \vdash S_1) \stackrel{?}{=} (\Delta_2|_{S_2} \vdash S_2) \mapsto \sigma; \varphi_1; \varphi_2}{(\Delta_1 \vdash c \cdot S_1) \stackrel{?}{=} (\Delta_2 \vdash c \cdot S_2) \mapsto \sigma; \varphi_1; \varphi_2} \ \mathsf{dec}_\Pi\text{-head-const}$$

$$\frac{(\Delta_1 \setminus\!\setminus x|_{S_1} \vdash S_1) \stackrel{?}{=} (\Delta_2 \setminus\!\setminus x|_{S_2} \vdash S_2) \mapsto \sigma; \varphi_1; \varphi_2}{(\Delta_1 \vdash x \cdot S_1) \stackrel{?}{=} (\Delta_2 \vdash x \cdot S_2) \mapsto \sigma; \varphi_1; \varphi_2} \ \mathsf{dec}_\Pi\text{-head-var-unmark}$$

$$\frac{((\Delta_1 \setminus\!\setminus \underline{x_1})\{x/\underline{x_1}\}|_{S_1} \vdash S_1) \stackrel{?}{=} ((\Delta_2 \setminus\!\setminus \underline{x_2})\{x/\underline{x_2}\}|_{S_2} \vdash S_2) \mapsto \sigma; \varphi_1; \varphi_2}{(\Delta_1 \vdash \underline{x_1} \cdot S_1) \stackrel{?}{=} (\Delta_2 \vdash \underline{x_2} \cdot S_2) \mapsto \sigma; (\varphi_1, x/\underline{x_1}); (\varphi_2, x/\underline{x_2})} \ \mathsf{dec}_\Pi\text{-head-var-mark}$$

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

Figure 13: Matching on terms in CLF_Π

The judgment $(\Delta_1 \vdash S_1) \stackrel{?}{=} (\Delta_2 \vdash S_2) \mapsto \sigma; \varphi_1; \varphi_2$ is given by rules dec_Π-sp-* and defined by induction on the structure of the spines. In rule dec_Π-sp-nil we require both spines to be empty (to ensure that both 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 spine. We write $\Delta|_N$, where N is a term well typed under Δ, to mean the minimum subcontext Δ_0 of Δ such that N is well typed under Δ_0. Because of dependencies, this is not necessarily $\Delta|_{\mathrm{FV}(N)}$.

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

$\boxed{\overline{\Delta} \vdash E_1 \stackrel{?}{=} E_2 \mapsto \sigma}$

$$\frac{\overline{\Delta} \vdash \underline{\epsilon_1}\{\overline{\Delta_2}/\underline{\Delta_1}\} \stackrel{?}{=} \underline{\epsilon_2}\{\overline{\Delta_2}/\underline{\Delta_2}\} \mapsto \sigma; \varphi_1; \varphi_2}{\overline{\Delta} \vdash \{\mathsf{let}\ \epsilon_1\ \mathsf{in}\ \Delta_1\} \stackrel{?}{=} \{\mathsf{let}\ \epsilon_2\ \mathsf{in}\ \Delta_2\} \mapsto \sigma}\ \mathsf{dec}_\Pi\text{-expr}$$

$\boxed{\overline{\Delta} \vdash \epsilon_1 \stackrel{?}{=} \epsilon_2 \mapsto \sigma; \varphi_1; \varphi_2}$

$$\frac{}{\overline{\Delta} \vdash \diamond \stackrel{?}{=} \diamond \mapsto \cdot; \cdot; \cdot}\ \mathsf{dec}_\Pi\text{-tr-empty}$$

$$\frac{(\overline{\Delta}|_{S_1} \vdash S_1) \stackrel{?}{=} (\overline{\Delta}|_{S_2} \vdash S_2) \mapsto \sigma; \cdot; \cdot \qquad \sigma\Delta_1 \stackrel{?}{=} \Delta_2 \mapsto \varphi_1; \varphi_2 \qquad \overline{\Delta} \bowtie \sigma\Delta_1\{\varphi_1\} \vdash \sigma\varphi_1\epsilon_1 \stackrel{?}{=} \varphi_2\epsilon_2 \mapsto \sigma'; \varphi_1'; \varphi_2'}{\overline{\Delta} \vdash (\{\Delta_1\} \leftarrow c \cdot S_1; \epsilon_1) \stackrel{?}{=} (\{\Delta_2\} \leftarrow c \cdot S_2; \epsilon_2) \mapsto \sigma'\sigma; \varphi_1'\varphi_1; \varphi_2'\varphi_2}\ \mathsf{dec}_\Pi\text{-tr-step-hd}$$

$$\frac{\overline{\Delta} \vdash \epsilon_1 : \Delta_1' \qquad \overline{\Delta} \vdash \epsilon_2 : \Delta_2' \qquad (\Delta_1'|_{S_1} \vdash S_1) \stackrel{?}{=} (\Delta_2'|_{S_2} \vdash S_2) \mapsto \sigma; \varphi_1; \varphi_2 \qquad \sigma\varphi_1\Delta_1 \stackrel{?}{=} \varphi_2\Delta_2 \mapsto \varphi_1'; \varphi_2' \qquad \overline{\Delta} \vdash \sigma\epsilon_1\{\varphi_1\} \stackrel{?}{=} \epsilon_2\{\varphi_2\} \mapsto \sigma'; \varphi_1''; \varphi_2''}{\overline{\Delta} \vdash (\epsilon_1; \{\Delta_1\} \leftarrow c \cdot S_1) \stackrel{?}{=} (\epsilon_2; \{\Delta_2\} \leftarrow c \cdot S_2) \mapsto \sigma'\sigma; \varphi_1''\varphi_1'\varphi_1; \varphi_2''\varphi_2'\varphi_2}\ \mathsf{dec}_\Pi\text{-tr-step-tl}$$

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

Figure 14: Matching on traces in CLF_Π

Marked variables are only used for variables introduced by a trace.

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

Finally, rule dec_Π-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-hand side must be of the form $H \cdot S$ (actually $\widehat{\lambda}(\cdot).\, H \cdot S$). The solution is essentially $X \leftarrow \theta^{-1}(H \cdot S)$ (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 4.1 are satisfied.

Matching on expressions and traces is defined in Figure 14. As for $\mathrm{CLF}_{@!}$, 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 expressions is given by rule dec_Π-expr, which is the same as in $\mathrm{CLF}_{@!}$. Matching on traces is given by the rules dec_Π-tr-*. Rule dec_Π-tr-empty matches the empty trace on both sides (same as $\mathrm{CLF}_{\multimap}$ and $\mathrm{CLF}_{@!}$).

Rule dec_Π-tr-step-hd matches a step at the beginning of the traces. As explained above, we

assume that the trace on the right side can be reordered (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 $\overline{\Delta}$. We write $\Delta|_S$, where S is a spine well typed under Δ, to mean the minimum subcontext Δ_0 of Δ such that S is well typed under Δ_0. 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 dec_Π-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 dec_Π-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 dec_Π-tr-inst handles the case of a monadic logic variable. The input context on both sides is the same, but the output context 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 $\{\mathsf{let}\ \epsilon_2\{\varphi_2\}\ \mathsf{in}\ \Delta'\{\varphi_1\}\}$.

Correctness of the algorithm We prove that the algorithm in Figures 12, 13, and 14 is sound and complete.

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 a matching renaming 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 matching renamings.

The following lemmas state the soundness of the various matching judgments.

Lemma 4.2 (Soundness of matching for contexts)

- *If* $\Delta_1 \stackrel{?}{=} \Delta_2 \mapsto \varphi_1; \varphi_2$, *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 4.3 (Soundness of matching for terms)

- *Let* N_1 *and* N_2 *be well typed terms under contexts* Δ_1 *and* Δ_2, *respectively, such that* $\Delta_1 \equiv \Delta_1|_{N_1}$ *and* $\Delta_2 \equiv \Delta_2|_{N_2}$. *If* $(\Delta_1 \vdash N_1) \stackrel{?}{=} (\Delta_2 \vdash N_2) \mapsto \sigma; \varphi_1; \varphi_2$, *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, such that* $\Delta_1 \equiv \Delta_1|_{S_1}$ *and* $\Delta_2 \equiv \Delta_2|_{S_2}$. *If* $(\Delta_1 \vdash S_1) \stackrel{?}{=} (\Delta_2 \vdash S_2) \mapsto \sigma; \varphi_1; \varphi_2$, *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 4.4 (Soundness of matching for traces)

- *If* $\overline{\Delta} \vdash E_1, E_2 \Leftarrow \{\Delta\}$ *and* $\Delta \vdash E_1 \stackrel{?}{=} E_2 \mapsto \sigma$, *then* $\sigma E_1 \equiv E_2$

- *If* $\overline{\Delta} \vdash \epsilon_1 : \Delta_1$, $\overline{\Delta} \vdash \epsilon_2 : \Delta_2$ *and* $\overline{\Delta} \vdash \epsilon_1 \stackrel{?}{=} \epsilon_2 \mapsto \sigma; \varphi_1; \varphi_2$, *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_Π-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 dec_Π-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 dec_Π-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 $\sigma\Delta_1\{\varphi_1\varphi_1'\} \equiv \Delta_2\{\varphi_2\varphi_2'\}$. (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 dec_Π-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. □

Next, we give completeness statements for the matching algorithm for CLF_Π.

Lemma 4.5 (Completeness of context matching for $\mathbf{CLF}_\Pi$)

- *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* $\Delta_1 \stackrel{?}{=} \Delta_2 \mapsto \varphi_1'; \varphi_2'$.
- *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 4.6 (Completeness of term matching for $\mathbf{CLF}_\Pi$) *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 terms under contexts Δ_1 and Δ_2, respectively, such that $\Delta_1 \equiv \Delta_1|_{F_1}$ and $\Delta_2 \equiv \Delta_2|_{F_2}$. Assume there exists matching renamings φ_1 and φ_2, and an assignment σ such that $\mathrm{dom}(\varphi_1) \subseteq \mathrm{FV}(F_1)$, $\mathrm{dom}(\varphi_2) \subseteq \mathrm{FV}(F_2)$, $\sigma\varphi_1 F_1 = \varphi_2 F_2$, and $\Delta_1\{\varphi_1\} \preccurlyeq \Delta_2\{\varphi_2\}$. Then, there exists φ_1' and φ_2' such that $(\Delta_1 \vdash F_1) \stackrel{?}{=} (\Delta_2 \vdash F_2) \mapsto \sigma; \varphi_1'; \varphi_2'$ 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. □

Completeness of trace matching is similar to the case of $\mathrm{CLF}_{@!}$.

Lemma 4.7 (Completeness of trace matching for $\mathbf{CLF}_\Pi$) *Let ϵ_1 and ϵ_2 be expressions such that ϵ_2 is ground and $\overline{\Delta} \vdash \epsilon_1 : \Delta_1$ and $\overline{\Delta} \vdash \epsilon_2 : \Delta_2$, with $\Delta_1 \approx_\downarrow \Delta_2$. Assume there exists matching 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 exists φ_1' and φ_2' such that $\overline{\Delta} \vdash \epsilon_1 \stackrel{?}{=} \epsilon_2 \mapsto \sigma; \varphi_1'; \varphi_2'$ is derivable and $\varphi_1'|_{\epsilon_2\bullet} = \varphi_1$ and $\varphi_2'|_{\epsilon_2\bullet} \subseteq \varphi_2$.*

Proof: We proceed in a similar way as in Lemma 3.9. 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 dec_Π-tr-step-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 dec_Π-tr-step-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 dec_Π-tr-inst for the last step containing the logic variable. □

Lemma 4.8 (Completeness of expression matching for $\mathbf{CLF}_\Pi$) *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 $\Delta \vdash E_1 \stackrel{?}{=} E_2 \mapsto \sigma$ is derivable.*

Proof: Follows directly from the previous lemma. □

4.3 Examples

We consider two examples of specifications: a π-calculus with correspondence assertions and Kruskal's algorithm for computing minimum spanning trees. Running this examples in a moded semantics based on matching involves solving the kind of trace matching problems that we address in this section.

π-calculus We consider a formalization of the asynchronous π-calculus with correspondence assertions [34], taken from [32]. We only present part of the code, the full version can be found in Appendix A.1.

The syntax of processes is given by:

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

The standard meaning of these operators will be reviewed shortly as we describe their CLF encoding.

CLF supports, actually encourages, 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:

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

The process stop corresponds to $(\mathbf{0})$ and represents a finished process $(\mathbf{0})$; $\mathsf{par}{\cdot}Q_1\,Q_2$ corresponds to $(Q_1|Q_2)$ and represents a concurrent execution of Q_1 and Q_2; $\mathsf{repeat}{\cdot}Q$ corresponds to $!Q$ and represents a process that can create copies of itself; $\mathsf{new}{\cdot}(\lambda x.Q)$ 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); $\mathsf{choose}{\cdot}Q_1\,Q_2$ corresponds to $Q_1 + Q_2$ and represents a nondeterministic choice between Q_1 and Q_2; $\mathsf{inp}{\cdot}x\,(\lambda y.Q)$ and $\mathsf{out}\,x\,y$ correspond to $x(y).Q$ and $x\langle y\rangle$, respectively, and represent communication between processes; finally, $\mathsf{begin}{\cdot}L\,Q$ and $\mathsf{end}{\cdot}L\,Q$ represent correspondence assertions that have to be matched in well-formed processes.

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:

$$\begin{aligned}
\mathsf{ev_stop} &: \mathsf{run}{\cdot}\mathsf{stop} \multimap \{\cdot\} \\
\mathsf{ev_par} &: \mathsf{run}{\cdot}(\mathsf{par}{\cdot}Q_1\,Q_2) \multimap \{@\mathsf{run}{\cdot}Q_1, @\mathsf{run}{\cdot}Q_2\} \\
\mathsf{ev_repeat} &: \mathsf{run}\,(\mathsf{repeat}{\cdot}Q) \multimap \{!\mathsf{run}{\cdot}Q\} \\
\mathsf{ev_sync} &: \mathsf{run}{\cdot}(\mathsf{out}{\cdot}X\,Y) \multimap \mathsf{run}{\cdot}(\mathsf{inp}{\cdot}X\,(\lambda y.Q{\cdot}y)) \\
&\qquad \multimap \{@\mathsf{run}{\cdot}(Q{\cdot}Y)\} \\
\mathsf{ev_begin} &: \mathsf{run}\,(\mathsf{begin}{\cdot}L\,Q) \multimap \{@\mathsf{run}{\cdot}Q\}
\end{aligned}$$

A process state is modeled by declarations of type $@\mathsf{run}{\cdot}Q$ and $!\mathsf{run}{\cdot}Q$; the latter represents a process that can be executed repeatedly. The objects of type $\mathsf{run}{\cdot}Q \multimap \{\cdot\}$ represent computations starting from a process Q. Computations that only differ in the order of independent steps are represented by the same object.

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. We consider a second operational semantics taken from Gordon and Jeffrey [11] 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:

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

This second 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:

$$\begin{aligned} \mathsf{abst\_sync} : \quad & \mathsf{abst}\,\{\mathsf{let}\ \begin{pmatrix} \{@r\} \leftarrow \mathsf{ev\_sync}\cdot @R_1\, @R_2; \\ \{\cdot\} \leftarrow E\cdot r \end{pmatrix}\ \mathsf{in}\ \cdot\}\ (\mathsf{sint}\cdot !s) \\ & \leftarrow (\Pi r.\mathsf{abst}\cdot(E\cdot @r)\, !s) \\ \mathsf{abst\_begin} : \quad & \mathsf{abst}\,\{\mathsf{let}\ \begin{pmatrix} \{@r\} \leftarrow \mathsf{ev\_begin}\cdot !L\, @R; \\ \{\cdot\} \leftarrow E\cdot r \end{pmatrix}\ \mathsf{in}\ \cdot\}\ (\mathsf{sbegin}\cdot !L\, !s) \\ & \leftarrow (\Pi r.\mathsf{abst}\cdot(E\cdot @r)\, !s) \\ \mathsf{abst\_end} : \quad & \mathsf{abst}\,\{\mathsf{let}\ \begin{pmatrix} \{@r\} \leftarrow \mathsf{ev\_end}\cdot !L\, @R; \\ \{\cdot\} \leftarrow E\cdot r \end{pmatrix}\ \mathsf{in}\ \cdot\}\ (\mathsf{send}\cdot !L\, !s) \\ & \leftarrow (\Pi r.\mathsf{abst}\cdot(E\cdot @r)\, !s) \\ \mathsf{abst\_new} : \quad & \mathsf{abst}\,\{\mathsf{let}\ \begin{pmatrix} \{!x, @r_1\} \leftarrow \mathsf{ev\_new}\cdot @R; \\ \{\cdot\} \leftarrow E\cdot x, r_1 \end{pmatrix}\ \mathsf{in}\ \cdot\}\ (\mathsf{sgen}\cdot(\lambda !x.s\, x) \\ & \leftarrow (\Pi x, r_1.\mathsf{abst}\cdot(E\cdot x, r_1 c)\, (s\, x)) \end{aligned}$$

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 logic 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 $\mathsf{abst}\cdot e\, X$, where e is a

ground computation and X is a logic variable, reduces to solving matching problems of the form (*) on page 33. The variable E in the above clauses represents an unknown computation (a trace in CLF). For example, consider the process P and execution E_P given above. 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:

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

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

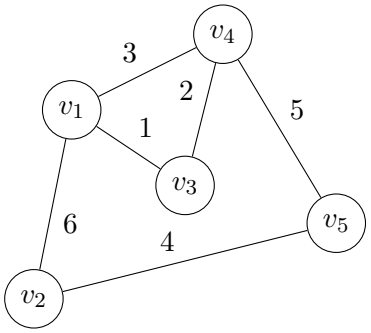
Kruskal's algorithm Next, we consider a CLF specification of Kruskal's algorithm. This example uses additive conjunctions which is not included in the subsystems of CLF that we consider in this work. However, additive conjunctions are unproblematic for the problem of trace matching.

Kruskal's algorithm is a greedy algorithm used to find a minimum spanning tree in a weighted graph. It proceeds by picking the edge with lowest weight in the graph, removing it, and adding it to a partially constructed tree if it does not create cycles. The process is repeated for all edges of the graph. The partial tree is actually a set of connected components, each of them a tree. Adding an edge does not create a cycle iff the vertices belong to different components.

This example also shows the use of traces to represent multisets. Weighted graphs are represented using the following signature:

$$\begin{aligned}
\mathsf{node} &: \mathsf{type}.\\
\mathsf{isnode} &: \mathsf{node} \to \mathsf{type}.\\
\mathsf{edge} &: \mathsf{nat} \to \mathsf{node} \to \mathsf{node} \to \mathsf{type}.
\end{aligned}$$

We omit the representation of natural numbers and their usual operations. A graph is represented in the context by a set of *affine* resources of the form $\mathsf{isnode}\, V$ to represent the vertices and a set of *linear* resources of the form $\mathsf{edge}\, N\, V\, W$. Let us illustrate this representation with an example:



$$\begin{aligned}
&@v_1 : \mathsf{isnode},\ \ @v_2 : \mathsf{isnode},\ \ @v_3 : \mathsf{isnode},\\
&@v_4 : \mathsf{isnode},\ \ @v_5 : \mathsf{isnode},\ \ @v_6 : \mathsf{isnode},\\
&{\downarrow}e_1 : \mathsf{edge}\, 6\, v_1\, v_2,\ \ {\downarrow}e_2 : \mathsf{edge}\, 1\, v_1\, v_3,\ \ {\downarrow}e_3 : \mathsf{edge}\, 3\, v_1\, v_4,\\
&{\downarrow}e_4 : \mathsf{edge}\, 4\, v_2\, v_5,\ \ {\downarrow}e_5 : \mathsf{edge}\, 2\, v_3\, v_4,\ \ {\downarrow}e_6 : \mathsf{edge}\, 5\, v_4\, v_5
\end{aligned}$$

A tree is represented as a multiset of nodes and edges, using the following signature:

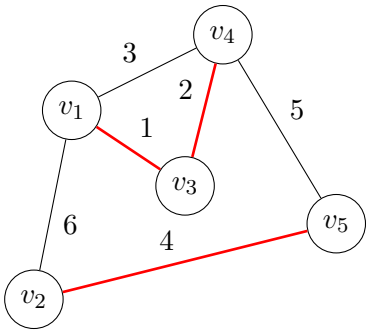
$$\begin{aligned}
\mathsf{ktree} &: \mathsf{type}.\\
\mathsf{itree} &: \mathsf{ktree} \to \mathsf{node} \to \{\cdot\}.\\
\mathsf{kedge} &: \mathsf{ktree} \to \mathsf{nat} \to \mathsf{node} \to \mathsf{node} \to \{\cdot\}.\\
\mathsf{component} &: (\mathsf{ktree} \to \{\cdot\}) \to \mathsf{type}.
\end{aligned}$$

The canonical terms of type $\mathsf{ktree} \to \{\cdot\}$ have the form

$$\lambda(!c : \mathsf{ktree}).\{\mathsf{let}\ \{\cdot\}{\leftarrow}H_1 \cdot c, S_1; \ldots; \{\cdot\}{\leftarrow}H_n \cdot c, S_n\ \mathsf{in}\ \cdot\},$$

where each H_i is either itree or kedge. To ensure that only itree and kedge are used as constructors in the tree we use the flag ktree. Each monadic type is *guarded* by a similar flag. The above term has a flag c that restrict us to use only constructors whose type includes a flag of the same type as c. In the full program (Appendix A.2) we use another flag to find the minimal edge in a graph.

The following diagram shows a partial tree with two components and its representation in CLF:



$$\begin{aligned}
&\mathsf{component}\ (\lambda!c.\ \mathsf{let}\ \{\{\cdot\}{\leftarrow}\mathsf{itree}\, c\, v_1;\\
&\qquad\qquad\qquad\qquad \{\cdot\}{\leftarrow}\mathsf{itree}\, c\, v_3;\\
&\qquad\qquad\qquad\qquad \{\cdot\}{\leftarrow}\mathsf{itree}\, c\, v_4;\\
&\qquad\qquad\qquad\qquad \{\cdot\}{\leftarrow}\mathsf{kedge}\, c\, 1\, v_1\, v_3;\\
&\qquad\qquad\qquad\qquad \{\cdot\}{\leftarrow}\mathsf{kedge}\, c\, 2\, v_3\, v_4\ \mathsf{in}\ \cdot\})\\
&\mathsf{component}\ (\lambda!c.\ \mathsf{let}\ \{\{\cdot\}{\leftarrow}\mathsf{itree}\, c\, v_2;\\
&\qquad\qquad\qquad\qquad \{\cdot\}{\leftarrow}\mathsf{itree}\, c\, v_5;\\
&\qquad\qquad\qquad\qquad \{\cdot\}{\leftarrow}\mathsf{kedge}\, c\, 4\, v_2\, v_5\ \mathsf{in}\ \cdot\})
\end{aligned}$$

Note there is no dependency between the steps that belong to a component. Steps in a component can be reordered in any way, effectively encoding a multiset.

The algorithm itself is represented by the following signature:

$$\begin{aligned}
&\mathsf{kruskal} : (\mathsf{ktree} \to \{\cdot\}) \to \mathsf{type}.\\
&\quad \mathsf{build} : \mathsf{nat} \to \mathsf{node} \to \mathsf{node} \to (\mathsf{ktree} \to \{\cdot\}) \to \mathsf{type}.
\end{aligned}$$

The type $\mathsf{kruskal}\, C$ is inhabited when C is a component representing the minimum spanning tree of the graph represented in the context. To construct $\mathsf{kruskal}\, C$ we repeatedly call $\mathsf{build}\, N\, A\, B\, C$ with each edge (N, A, B). Let us consider some constructors of build.

$$\begin{aligned}
&\mathsf{build}/1 : \mathsf{build}\, N\, A\, B\, C\\
&\qquad \circ\!\!- \ \mathsf{edge}\, N\, A\, B\\
&\qquad \circ\!\!- \ \mathsf{isnode}\, A\\
&\qquad \circ\!\!- \ \mathsf{isnode}\, B\\
&\qquad \circ\!\!- \ (\mathsf{component}\ (\lambda!c.\ \{\mathsf{let}\ \{\cdot\}{\leftarrow}\mathsf{itree}\, c\, A;\\
&\qquad\qquad\qquad\qquad\qquad\qquad \{\cdot\}{\leftarrow}\mathsf{itree}\, c\, B;\\
&\qquad\qquad\qquad\qquad\qquad\qquad \{\cdot\}{\leftarrow}\mathsf{kedge}\, c\, N\, A\, B\\
&\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \cdot\}\\
&\qquad\qquad\qquad\qquad\qquad\ \ {-\!\!@}\ \mathsf{kruskal}\, C)).
\end{aligned}$$

The edge (N, A, B) is the edge with minimum weight in the graph (we show below how to find the minimum edge). In this case, the nodes A and B are not in the tree yet, so we build an affine

component containing only the edge (N, A, B) and proceed by calling kruskalC. Note that the edge and the nodes are consumed.

$$
\begin{array}{l}
\mathsf{build}/2 : \mathsf{build}\ N\ A\ B\ C \\
\quad \circ\!\!- \ \mathsf{edge}\ N\ A\ B \\
\quad \circ\!\!- \ \mathsf{isnode}\ A \\
\quad @\!\!- \ (\mathsf{component}\ (\lambda!c.\ \{\mathsf{let}\ \{\cdot\}\!\leftarrow\!\mathsf{itree}\ c\ B; \\
\qquad\qquad\qquad\qquad\qquad\qquad \{\cdot\}\!\leftarrow\!(K\ c) \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{in}\ \cdot\})) \\
\quad \circ\!\!- \ (\mathsf{component}\ (\lambda!c.\ \{\mathsf{let}\ \{\cdot\}\!\leftarrow\!\mathsf{itree}\ c\ A; \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \{\cdot\}\!\leftarrow\!\mathsf{itree}\ c\ B; \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \{\cdot\}\!\leftarrow\!\mathsf{kedge}\ c\ N\ A\ B; \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \{\cdot\}\!\leftarrow\!(K\ c) \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{in}\ \cdot\} \\
\qquad -\!@\ \mathsf{kruskal}\ C)).
\end{array}
$$

In this case, the node A is not in the tree, but the node B is in a component of the form $\lambda!c.\ \{\mathsf{let}\ \{\cdot\}\!\leftarrow\!\mathsf{itree}\ c\ B; \{\cdot\}\!\leftarrow\!(K\ !c)\ \mathsf{in}\ \cdot\}$. The component is consumed and a new component generated adding the node A and the edge (N, A, B). To execute this program we need to match the trace representing the tree to find a value for K. A similar constructor considers the symmetric case where node A is in the tree while node B is not.

$$
\begin{array}{l}
\mathsf{build}/5 : \mathsf{build}\ N\ A\ B\ C \\
\quad \circ\!\!- \ \mathsf{edge}\ N\ A\ B \\
\quad @\!\!- \ (\mathsf{component}\ (\lambda!c.\ \mathsf{let}\ \{\{\cdot\}\!\leftarrow\!\mathsf{itree}\ c\ A; \\
\qquad\qquad\qquad\qquad\qquad\qquad \{\cdot\}\!\leftarrow\!(K_1\ c)\ \mathsf{in}\ \cdot\} \\
\quad @\!\!- \ (\mathsf{component}\ (\lambda!c.\ \mathsf{let}\ \{\{\cdot\}\!\leftarrow\!\mathsf{itree}\ c\ B; \\
\qquad\qquad\qquad\qquad\qquad\qquad \{\cdot\}\!\leftarrow\!(K_2\ c)\ \mathsf{in}\ \cdot\} \\
\quad \circ\!\!- \ (\mathsf{component}\ (\lambda!c.\ \mathsf{let}\ \{\{\cdot\}\!\leftarrow\!\mathsf{itree}\ c\ A; \\
\qquad\qquad\qquad\qquad\qquad\qquad \{\cdot\}\!\leftarrow\!\mathsf{itree}\ c\ B; \\
\qquad\qquad\qquad\qquad\qquad\qquad \{\cdot\}\!\leftarrow\!\mathsf{kedge}\ c\ N\ A\ B; \\
\qquad\qquad\qquad\qquad\qquad\qquad \{\cdot\}\!\leftarrow\!(K_1\ c); \\
\qquad\qquad\qquad\qquad\qquad\qquad \{\cdot\}\!\leftarrow\!(K_2\ c);\ \mathsf{in}\ \cdot\} -\!@\ \mathsf{kruskal}\ C).
\end{array}
$$

In this case, both nodes A and B are in the tree, but in different components. A new component is generated that unites the components containing A and B. Executing this constructor implies solving two matching problems on traces to find the components containing A and B.

Finally, the type family kruskal is defined by the following constructors:

$$\begin{aligned}&\mathsf{run} : \mathsf{kruskal}\, C\\ &\quad \circ\!\!- (\mathsf{mflag} \to \{\mathsf{min}\, N\, A\, B\}) \;\&\; \mathsf{build}\, N\, A\, B\, C.\\ &\mathsf{stop} : \mathsf{kruskal}\, C\\ &\quad @\!\!- \mathsf{component}\, C.\end{aligned}$$

Clause run considers if it is possible to add edge (N, A, B) to the tree. Clause stop can only be called when no linear hypotheses are in the context; in particular, this means that all edges have been considered. At this point there is only one component in the context. The flag mflag is used to compute the minimum edge of the graph by restricting the constructors used in terms of type $\{\mathsf{min}\ N\ A\ B\}$ to use this flag. This means, for example, that itree and kedge cannot be used to construct terms of this type, since they use a different flag (ktree). See Appendix A.2 for the full code of this example in Celf.

Running this algorithm in a well-moded semantics would reduce to solving the kind of matching problems we solve in this paper. For example, to apply rule build/5 we need to find a term of the form $\mathsf{component}\cdot(\lambda!c.E\cdot c)$, where E is an expression, and solve the matching problem

$$E(c) \stackrel{?}{=} \{\mathsf{let} \begin{pmatrix}\{\cdot\}\leftarrow \mathsf{itree}\ c\ A;\\ \{\cdot\}\leftarrow K_1\, c;\end{pmatrix}\ \mathsf{in}\ \cdot\}$$

where K_1 is a logic variable (we do not need to solve A since it is given by the edge to be tried). We also need to solve the matching problem

$$E'(c) \stackrel{?}{=} \{\mathsf{let} \begin{pmatrix}\{\cdot\}\leftarrow \mathsf{itree}\ c\ B;\\ \{\cdot\}\leftarrow K_2\, c;\end{pmatrix}\ \mathsf{in}\ \cdot\}$$

for a term of the form $\mathsf{component}\cdot(\lambda!c.E'\cdot c)$. If both matching problems succeed, it means that vertices A and B belong to different components, so adding the edge between them to the partial tree will not create cycles. The search for the minimum spanning tree continues by consuming both components and building one that combines both together with the edge between A and B. We do not delve into the details of the operational semantics of Celf; see [17] for the semantics of Lollimon on which Celf is based.

5 CLF$_x$: Variable Heads

In this section, we extend CLF$_\Pi$ with embedded clauses in traces. In previous systems, steps have had the form $\{\Delta\}\leftarrow c\cdot S$ where c is a constant from the signature. In CLF$_x$ steps can also have the form $\{\Delta\}\leftarrow x\cdot S$ where x is a variable in the context (e.g., introduced in a previous step).

Given that CLF$_x$ is close to CLF$_\Pi$ we only present a summary of the language highlighting the differences (Section 5.1). We present a comparison between CLF$_x$ and full CLF (Section 5.2). We adapt the matching algorithm from CLF$_\Pi$ to handle embedded clauses (Section 5.3).

Traces:

$$\frac{}{\Delta \vdash \diamond : \Delta}\ \mathsf{tp}_x\text{-empty} \qquad \frac{\Delta \vdash \epsilon_1 : \Delta_1 \qquad \Delta_1 \vdash \epsilon_2 : \Delta_2}{\Delta \vdash \epsilon_1; \epsilon_2 : \Delta_2}\ \mathsf{tp}_x\text{-comp}$$

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

$$\frac{x{:}\Pi\Delta'.\{\Delta''\} \in \Delta_0 \qquad \Delta_1 \vdash S \Leftarrow \Delta' \qquad \Delta_2 \equiv \Delta''[S/\Delta']}{\Delta_0 \bowtie \Delta_1 \vdash \{\Delta_2\}{\leftarrow}x{\cdot}S : (\Delta_0 \setminus\!\setminus x), \Delta_2}\ \mathsf{tp}_x\text{-step-var}$$

Figure 15: Typing Rules for Traces of CLF_x

5.1 Language

Syntax The language CLF_x (including logic variables) is defined by the following syntax:

Kinds:	K	$::=$	$\Pi!\Delta.\mathsf{type}$	*Heads:*	H	$::=$	$x \mid c \mid X[\theta]$
Base types:	P	$::=$	$a{\cdot}S \mid \{\Delta\}$	*Spines:*	S	$::=$	$\cdot \mid \Box N, S$
Types:	A	$::=$	$\Pi\Delta.P$	*Terms:*	N	$::=$	$\widehat{\lambda}\Delta.\, H{\cdot}S$
Expressions:	E	$::=$	$\{\mathsf{let}\ \epsilon\ \mathsf{in}\ \Delta\}$	*Modalities:*	$\Box$	$::=$	$\downarrow \mid @ \mid !$
Traces:	ϵ	$::=$	$\diamond \mid \delta \mid \epsilon_1; \epsilon_2$	*Contexts:*	Δ	$::=$	$\cdot \mid \Gamma, \Box x{:}\Box A$
Steps:	δ	$::=$	$\{\Delta\}{\leftarrow}H{\cdot}S$				

The only difference with respect to CLF_x is that a step can have a variable as a head. This allows us to write forward-chaining specifications that use embedded clauses. Typical examples that include this feature include shallow embedding of process calculi.

Typing The typing rules are given by the following judgments:

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

The typing rules for types, kinds, and contexts are the same as in CLF_Π (cf. Figure 9). The typing rules for terms, spines, and expressions are also the same as in CLF_Π (cf. Figure 10). The typing rules for traces are extended to deal with variables as heads and are given in Figure 15. Rules tp_x-step-const and tp_x-step-var cover the cases where the variable at the head of a step is a constant or a variable, respectively.

5.2 Comparison with CLF

The languages presented in this report, ultimately CLF_x, differs from CLF [31, 28] in a several ways.

1. CLF includes additive conjunctions at the level of types and, correspondingly, pairs and projections at the level of terms. We decided to exclude it from our language since it is orthogonal to the problem of matching. Reintroducing them is unproblematic.

2. In a step, 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.

3. In CLF, expressions have the form $\{\mathsf{let}\ \epsilon\ \mathsf{in}\ M\}$, where M is a *monadic term* (essentially a spine). In our case, we only allow a context (basically a sequence of variables) instead of terms. This is the biggest difference with respect to CLF. We make this restriction for two reasons. First, most of the CLF specifications fit in the fragment presented in this paper. Second, matching in full CLF (with monadic terms) is a much more difficult problem than the one considered here. This is because substitution of traces can trigger more reductions. As mentioned in Section 3.4, in CLF the equivalent rule for substitutions

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

   does not give necessarily a canonical term, as $[M/\Delta]\epsilon_2$ might have redexes. These reductions can have the effect of removing steps in ϵ_2; in particular, after reductions, $[M/\Delta]\epsilon_2$ could be reduced to the empty trace.

   The language presented here allows us to develop a simpler algorithm for matching that is described in the next section in the case of CLF_x, our most complete fragment. We leave for future work the problem of matching in full CLF.

5.3 Matching

Trace matching in CLF_x is similar to matching in CLF_Π: individual steps at the beginning or at the end of the trace are matched until a step with a monadic logic variable is obtained. The trace matching problem is defined by

$$\overline{\Delta} \vdash \epsilon_1 \stackrel{?}{=} \epsilon_2$$

where ϵ_1 and ϵ_2 are well typed traces in the (unmarked) context $\overline{\Delta}$. Assume that $\epsilon_1 = \{\Delta_1\}{\leftarrow}H{\cdot}S_1; \epsilon_1'$. The head H is either a constant c from the signature, or a variable x from $\overline{\Delta}$. In both cases, ϵ_2 is equivalent to a trace of the form $\{\Delta_2\}{\leftarrow}H{\cdot}S_2; \epsilon_2'$ with the same head H in the first step. Matching steps at the beginning of the trace follows the same procedure as in CLF_Π.

Let us consider the case of matching steps at the end of the trace. Assume that ϵ_1 has the form $\epsilon_1'; \{\Delta_1\}{\leftarrow}H_1{\cdot}S_1$. Then ϵ_2 is equivalent to a trace of the form $\epsilon_2'; \{\Delta_2\}{\leftarrow}H_2{\cdot}S_2$, where the last step corresponds to the last step of ϵ_1. If H_1 is a constant c from the signature, then H_2 must also be c. If H_1 is a variable x_1 introduced in ϵ_1, then H_2 must be a variable x_2 introduced in ϵ_2. In the latter case, matching the last step would identify x_1 with x_2. However, note that the types of x_1 and x_2 do not necessarily match, since they may depend on other variables introduced in the trace and not yet matched. For example, consider the following matching problem:

$$\begin{pmatrix}\{!x_1{:}a, x_2{:}\Pi x'{:}a.R{\cdot}x_1, x'\}{\leftarrow}c_0{\cdot}!x_0;\\ \{x_3\}{\leftarrow}x_2{\cdot}!z\end{pmatrix} \stackrel{?}{=} \begin{pmatrix}\{!y_1{:}a, y_2{:}\Pi y'{:}a.R{\cdot}y_1, y'\}{\leftarrow}c_0{\cdot}!y_0;\\ \{y_3\}{\leftarrow}y_2{\cdot}!z\end{pmatrix}$$

Note that in $\{x_3\}{\leftarrow}x_2{\cdot}!z$ and $\{y_3\}{\leftarrow}y_2{\cdot}!z$, the variables x_2 and y_2 have different types; they depend on x_1 and y_1, respectively. We could also match the types to have more assurance that we have chosen the correct step. This approach would reduce the search space, but it is not strictly necessary; if the wrong step was chosen, matching will eventually fail and backtrack to this point.

We proceed in a similar way as in CLF_Π, by matching the spines and the contexts produced by the step. Note, however, that when matching the contexts we cannot force the types to be the same on both sides. For example, x_3 and y_3 do not have the same type in the example above. Hence, we remove types from the context for matching this kind of steps. We define the operation $|\Delta|$ removes all types from Δ, changing them to a fixed type α. Formally, it is defined by the rules

$$\begin{array}{lcl} |\cdot| & = & \cdot \\ |\Delta, \square x{:}A| & = & |\Delta|, \square x{:}\alpha \end{array}$$

Although α is an abstraction that conflates arbitrary types, types in matching position have to match because everything else around them does: for example, in rule x:-tr-step-tl-unmark, if ϵ_1 and ϵ_2 match, and S_1 and S_2 match, then the types of Δ_1 and Δ_2 also match, since it is the same variable applied to the same spines.

The trace matching rules of CLF_x are given in Figure 16. The matching rules for terms and contexts are the same as for CLF_Π (Figures 13 and 12, respectively). As explained above, there are two rules for matching at the end of traces, depending if the heads are marked (dec_x-tr-step-tl-mark) or unmarked (dec_x-tr-step-tl-unmark).

Correctness of the algorithm The proof of soundness and completeness follow the same pattern as for CLF_Π.

We need to slightly modify the soundness and completeness for spines and terms, since we cannot enforce matching on the full contexts: consider a spine S well typed under a context Δ; then Δ might declare variables that do not occur in S (due to type dependencies). The matching judgment only ensures that the spine is matched.

Lemma 5.1 (Soundness of matching for terms)

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

Proof: By simultaneous induction on the given derivation. □

The following lemma states the soundness of matching for traces and expressions.

Lemma 5.2 (Soundness of matching for traces)

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

Proof: We proceed by induction on the given derivation. We only consider the case of rule dec_x-tr-step-tl-mark. The case of rule dec_x-tr-step-tl-unmark is similar, while the other cases are similar to the proof of soundness for CLF_Π (Lemma 4.4).

$\boxed{\overline{\Delta} \vdash E_1 \stackrel{?}{=} E_2 \mapsto \sigma}$

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

$\boxed{\overline{\Delta} \vdash \epsilon_1 \stackrel{?}{=} \epsilon_2 \mapsto \sigma; \varphi_1; \varphi_2}$

$$\frac{\Delta_1 \vdash S_1 \stackrel{?}{=} \Delta_2 \vdash S_2 \mapsto \sigma; \varphi_1; \varphi_2 \qquad \varphi_1 \sigma \Delta_1' \stackrel{?}{=} \varphi_2 \Delta_2' \mapsto \varphi_1'; \varphi_2'}{\Delta_1 \vdash \{\Delta_1'\} \leftarrow \overline{H} \cdot S_1 \stackrel{?}{=} \Delta_2 \vdash \{\Delta_2'\} \leftarrow \overline{H} \cdot S_2 \mapsto \sigma; \varphi_1' \varphi_1; \varphi_2' \varphi_2}\ \mathsf{dec}_x\text{-step}$$

$$\frac{}{\overline{\Delta} \vdash \diamond \stackrel{?}{=} \diamond \mapsto \cdot; \cdot; \cdot}\ \mathsf{dec}_x\text{-tr-empty}$$

$$\frac{\begin{array}{c}(\overline{\Delta}|_{S_1} \vdash \{\Delta_1\} \leftarrow \overline{H} \cdot S_1) \stackrel{?}{=} (\overline{\Delta}|_{S_2} \vdash \{\Delta_2\} \leftarrow \overline{H} \cdot S_2) \mapsto \sigma; \varphi_1; \varphi_2 \\ (\overline{\Delta} \setminus\setminus \overline{H}) \bowtie \sigma\Delta_1\{\varphi_1\} \vdash \sigma\varphi_1\epsilon_1 \stackrel{?}{=} \varphi_2\epsilon_2 \mapsto \sigma'; \varphi_1'; \varphi_2'\end{array}}{\overline{\Delta} \vdash (\{\Delta_1\} \leftarrow \overline{H} \cdot S_1; \epsilon_1) \stackrel{?}{=} (\{\Delta_2\} \leftarrow \overline{H} \cdot S_2; \epsilon_2) \mapsto \sigma'\sigma; \varphi_1' \varphi_1; \varphi_2' \varphi_2}\ \mathsf{dec}_x\text{-tr-step-hd}$$

$$\frac{\begin{array}{c}\overline{\Delta} \vdash \epsilon_1 : \Delta_1' \qquad \overline{\Delta} \vdash \epsilon_2 : \Delta_2' \\ (\Delta_1'|_{S_1} \vdash S_1) \stackrel{?}{=} (\Delta_2'|_{S_2} \vdash S_2) \mapsto \sigma; \varphi_1; \varphi_2 \qquad |\Delta_1| \stackrel{?}{=} |\Delta_2| \mapsto \varphi_1'; \varphi_2' \\ \overline{\Delta} \vdash \sigma\epsilon_1\{\varphi_1, x/\underline{x_1}\} \stackrel{?}{=} \epsilon_2\{\varphi_2, x/\underline{x_2}\} \mapsto \sigma'; \varphi_1''; \varphi_2''\end{array}}{\overline{\Delta} \vdash (\epsilon_1; \{\Delta_1\} \leftarrow \underline{x_1} \cdot S_1) \stackrel{?}{=} (\epsilon_2; \{\Delta_2\} \leftarrow \underline{x_2} \cdot S_2) \mapsto \sigma'\sigma; \varphi_1'' \varphi_1' \varphi_1, x/\underline{x_1}; \varphi_2'' \varphi_2' \varphi_2, x/\underline{x_2}}\ \mathsf{dec}_x\text{-tr-step-tl-mark}$$

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

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

Figure 16: Matching on traces in CLF_x

By IH on the spines, we have $\sigma S_1\{\varphi_1\} \equiv S_2\{\varphi_2\}$. By IH on the trace, $\sigma'\sigma\epsilon_1\{\varphi_1, x/\underline{x_1}\}\{\varphi_1''\} \equiv \epsilon_2\{\varphi_2, x/\underline{x_2}\}\{\varphi_2''\}$, and $\sigma'\sigma\Delta_1'\{\varphi_1, x/\underline{x_1}\}\varphi_1'' \equiv \Delta_2'\{\varphi_2, x/\underline{x_2}\}\varphi_2''$. This means that $\sigma(x \cdot S_1)\{\varphi_1\} \equiv x \cdot S_2\{\varphi_2\}$. Furthermore, this ensures that the contexts restricted to S_1 and S_2 also match, as well as Δ_1 and Δ_2 (when applied to the resulting assignments and renamings). By IH on the context, $|\Delta_1|\{\varphi_1'\} = |\Delta_2|\{\varphi_2'\}$. The result follows. □

Completeness of the matching algorithm follows the same pattern as in CLF_Π. As in the case of soundness, we slightly change the completeness theorem for terms and spines.

Lemma 5.3 (Completeness of matching for terms)

- *Let N_1 and N_2 be well typed terms under contexts Δ_1 and Δ_2. Assume there exists matching renamings φ_1 and φ_2, and an assignment σ such that* $\mathrm{dom}(\varphi_1) \subseteq \mathrm{FV}(N_1)$, $\mathrm{dom}(\varphi_2) \subseteq \mathrm{FV}(N_2)$, *and* $\sigma\varphi_1 N_1 \equiv N_2$. *Then, there exists φ_1' and φ_2' such that* $(\Delta_1 \vdash N_1) \stackrel{?}{=} (\Delta_2 \vdash N_2) \mapsto \sigma; \varphi_1'; \varphi_2'$, $\varphi_1' \subseteq \varphi_1$ *and* $\varphi_2' = \varphi_2$.
- *Let S_1 and S_2 be well typed terms under contexts Δ_1 and Δ_2. Assume there exists matching renamings φ_1 and φ_2, and an assignment σ such that* $\mathrm{dom}(\varphi_1) \subseteq \mathrm{FV}(S_1)$, $\mathrm{dom}(\varphi_2) \subseteq \mathrm{FV}(S_2)$, *and* $\sigma\varphi_1 S_1 \equiv S_2$. *Then, there exists φ_1' and φ_2' such that* $(\Delta_1 \vdash S_1) \stackrel{?}{=} (\Delta_2 \vdash S_2) \mapsto \sigma; \varphi_1'; \varphi_2'$, $\varphi_1' \subseteq \varphi_1$ *and* $\varphi_2' = \varphi_2$.

Proof: By simultaneous induction on the given derivation. □

Completeness of trace and expression matching is stated in the following two lemmas.

Lemma 5.4 (Completeness of trace matching for CLF$_x$) *Let ϵ_1 and ϵ_2 be well-typed expressions under context Δ such that ϵ_2 is ground. Assume there exists 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 exists φ_1' and φ_2' such that* $\Delta \vdash \epsilon_1 \stackrel{?}{=} \epsilon_2 \mapsto \sigma; \varphi_1'; \varphi_2'$ *is derivable.*

Proof: Similar to Lemma 4.7. □

Lemma 5.5 (Completeness of expression matching for CLF$_x$) *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* $\Delta \vdash E_1 \stackrel{?}{=} E_2 \mapsto \sigma$ *is derivable.*

Proof: Follows directly from the previous lemma. □

6 Related work

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 [18], which studies a specific form of matching for Mazurkiewicz traces [26]. Mazurkiewicz traces are partially commutative strings over a given alphabet of symbols representing the atomic steps of a concurrent computation: like ordinary strings, concatenation is associative and has the empty string as its unit; unlike ordinary strings, it allows designated pairs of symbols to commute. These structures, called trace monoids, capture the computations in simple concurrent languages at an abstract level: in particular, the notion of independence in Mazurkiewicz traces is given by this fixed relation on symbols. The notion of concurrent trace studied in this report is more complex in that individual steps have the structured form $\{\Delta\}{\leftarrow}c{\cdot}S$ rather than just a symbol — this is akin to going from a propositional to a first-order calculus. The variables used in S and produced in Δ enable a finer notion of independence between concurrent steps that is based on explicit dependencies (modeled by the variables in S and Δ). This binding structure makes our computational traces both more concrete and more general than Mazurkiewicz traces.

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 [24], colored Petri nets [15] and various forms of multiset rewriting [2, 3]. CLF also allows arbitrary combinations of forward chaining, typical of concurrent computations, and backward chaining found in top-down logic programming. The operational semantics of CLF is based on the operational semantics of Lollimon [17]. In this paper, we focused on matching for forward chaining traces, as unification in the backward chaining sublanguage is well understood [21].

Matching and unification have been studied extensively for related equational theories [1, 9]. Examples include unification in a commutative monoid (ACU unification) which is the basis of any multiset rewriting framework and string unification (which is associative with a unit). The problem studied here is characterized by a partially commutative operation and, maybe surprisingly, is substantially harder than either. Indeed, it is more closely related to problems such as graph matching and isomorphism, analyzed for example in [35]: as we saw in Section 2.4, the dependency between computational steps is captured rather directly by the edges of a graph. In general, subgraph matching techniques (combined with higher-order matching on λ-terms) could be used to match the known steps of the trace.

7 Conclusions and Future Work

We have presented sound and complete algorithms to perform matching on a series of fragments of CLF, with at most one variable standing for an unknown concurrent trace. We showed that the matching problem is decidable for the fragments of CLF studied in this report. These fragments cover a large subset of CLF sufficient to express many real-world specifications. We exemplified the use of matching with two examples: π-calculus and Kruskal's algorithm.

We also studied the unification problem with at most one variable standing for an unknown trace on each side of the equation. We showed that the matching algorithm can be adapted to unification in the simply-typed case (systems $\text{CLF}_{\multimap}$ and $\text{CLF}_{@!}$). We leave for future work a more systematic study of unification.

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.

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. Linear Higher-Order Pre-Unification. In G. Winskel, editor, *12th Annual Symposium on Logic in Computer Science — LICS'97*, pages 422–433, Warsaw, Poland, 1997. IEEE Computer Society Press.

[5] Iliano Cervesato and Frank Pfenning. A Linear Logical Framework. *Information & Computation*, 179(1):19–75, 2002.

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

[7] 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.

[8] Iliano Cervesato and Andre Scedrov. Relating State-Based and Process-Based Concurrency through Linear Logic. *Information & Computation*, 207(10):1044–1077, 2009.

[9] 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.

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

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

[12] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

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

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

[15] 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.

[16] Daniel Le Métayer. Higher-order multiset programming. In *Proc. DIMACS workshop on specifications of parallel algorithms*. American Mathematical Society, DIMACS series in Discrete Mathematics, Vol. 18, 1994.

[17] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In Pedro Barahona and Amy P. Felty, editors, *PPDP*, pages 35–46. PUB-ACM, 2005.

[18] 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.

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

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

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

[22] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

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

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

[25] 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.

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

[27] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

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

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

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

A Examples in Celf

We show the Celf code for the examples considered in Section 4.3.

A.1 π-calculus

```
%%
%% Processes (untyped, asynchronous, non-polyadic)
%%

pr : type.
ch : type.

out : ch -> ch -> pr.
inp : ch -> (ch -> pr) -> pr.
choose : pr -> pr -> pr.
new : (ch -> pr) -> pr.
par : pr -> pr -> pr.
repeat : pr -> pr.
stop : pr.

label : type.

begin : label -> pr -> pr.
end : label -> pr -> pr.

%%
%% Executions
%%
%% Executions can stop at any time. (See exec_encap.)
%%

flag : type.

exec : flag -> pr -> type.
run : pr -> type.

ev_stop : flag -> run stop -o {1}.
ev_par : flag -> run (par P Q) -o {@run P * @run Q}.
ev_repeat : flag -> run (repeat P) -o {!run P}.
ev_choose1 : flag -> run (choose P Q) -o {@run P}.
ev_choose2 : flag -> run (choose P Q) -o {@run Q}.
ev_new : flag -> run (new \!x. P x) -o {Exists x. @run (P x)}.
ev_sync : flag -> run (out X Y) -o run (inp X \!y. Q !y) -o {@run (Q !Y)}.

ev_begin : flag -> run (begin L P) -o {@run P}.
ev_end : flag -> run (end L P) -o {@run P}.
```

```
exec_encap : Pi f. exec f P <- (run P -o {1}).

%%
%% Traces
%%
%% We capture bound channel names inp a trace so that labels can
%% depend on channel names if necessary.
%%

ev : type.

snil : ev.
sint : ev -> ev.
sbegin : label -> ev -> ev.
send : label -> ev -> ev.
sgen : (ch -> ev) -> ev.

%%
%% Abstraction
%%
%% We can stop abstracting at any time. Thus the set of traces
%% abstractable from an execution is prefix-closed. (See abst_nil.)
%%

abst : (flag -> {1}) -> ev -> type.

abst_nil : abst E snil.

abst_stop : abst (\!x. { let {1} = ev_stop x R in let {1} = E !x in 1 }) S
          <- abst (\!x. E !x) S.
abst_par : abst (\!x. { let {[@r1,@r2]} = ev_par x R in
                        let {1} = E !x r1 r2 in 1 }) S
         <- (Pi r1. Pi r2. abst (\!x. E !x r1 r2) S).
abst_repeat : abst (\!x. { let {!r1} = ev_repeat x R in
                           let {1} = E x r1 in 1 }) S
            <- (Pi r1. abst (\!x. E x r1) S).
abst_choose1 : abst (\!x. { let {@r1} = ev_choose1 x R in
                            let {1} = E x r1 in 1 }) (sint S)
             <- (Pi r1. abst (\!x. E x r1) S).
abst_choose2 : abst (\!x. { let {@r2} = ev_choose2 x R in
                            let {1} = E x r2 in 1 }) (sint S)
             <- (Pi r2. abst (\!x. E x r2) S).
abst_new : abst (\!f. { let {[!x,@r1]} = ev_new f R in
                        let {1} = E f !x r1 in 1 }) (sgen \!x. S x)
         <- (Pi x. Pi r1. abst (\!f. E f !x r1) (S x)).
abst_sync : abst (\!f. { let {@r} = ev_sync f R1 R2 in
                         let {1} = E f r in 1 }) (sint S)
          <- (Pi r. abst (\!f. E f r) S).

abst_begin : abst (\!f. { let {@r} = ev_begin f R in
```

```
                          let {1} = E f r in 1 }) (sbegin L S)
          <- (Pi r. abst (\!f. E f r) S).
abst_end : abst (\!f. { let {@r} = ev_end f R in
                        let {1} = E f r in 1 }) (send L S)
         <- (Pi r. abst (\!f. E f r) S).
```

A.2 Kruskal's Algorithm

```
%%
%% Natural numbers
%%

nat : type.
z : nat.
s : nat -> nat.

le : nat -> nat -> type.
le0 : le z N.
les : le (s N) (s M)
       <- le N M.

add : nat -> nat -> nat -> type.
add/z : add z N N.
add/s : add (s M) N (s K)
         <- add M N K.

%%
%% Graphs
%%

node: type.
isnode : node -> type.
edge : nat -> node -> node -> type.

%%
%% Trees. Connected components
%%

ktree : type.
itree : ktree -> node -> {1}.
kedge : ktree -> nat -> node -> node -> {1}.
component : (ktree -> {1}) -> type.

%%
%% Find the minimal edge in a graph
%%
min : nat -> node -> node -> type.
mflag : type.
min/edge : mflag -> edge N A B -o {min N A B}.
min/min  : mflag -> min N A B -o min M _ _ -o le N M -> {min N A B}.
```

```
%%
%% The algorithm
%%
kruskal : (ktree -> {1}) -> type.
build : nat -> node -> node -> (ktree -> {1}) -> type.

build/1 : build N A B C
           o- edge N A B
           o- isnode A
           o- isnode B
           o- (component (\!c. {let {1} = itree c A in
                               let {1} = itree c B in
                               let {1} = kedge c N A B in 1}) -@ kruskal C).

build/2 : build N A B C
           o- edge N A B
           o- isnode A
           @- component (\!c. {let {1} = itree c B in
                               let {1} = (K !c) in 1})
           o- (component (\!c. {let {1} = itree c B in
                                let {1} = itree c A in
                                let {1} = kedge c N A B in
                                let {1} = (K !c) in 1})  -@ kruskal C).

build/3 : build N A B C
           o- edge N A B
           o- isnode B
           @- component (\!c. {let {1} = itree c A in
                               let {1} = (K !c) in 1})
           o- (component (\!c. {let {1} = itree c A in
                                let {1} = itree c B in
                                let {1} = kedge c N A B in
                                let {1} = (K !c) in 1})  -@ kruskal C).

build/4 : build N A B C
           o- edge N A B
           @- component (\!c. {let {1} = itree c A in
                               let {1} = itree c B in
                               let {1} = (K !c) in 1})
           o- (component (\!c. {let {1} = itree c A in
                                let {1} = itree c B in
                                let {1} = (K !c) in 1}) -@ kruskal C).

build/5 : build N A B C
           o- edge N A B
           @- component (\!c. {let {1} = itree c A in
                               let {1} = (K1 !c) in 1})
           @- component (\!c. {let {1} = itree c B in
                               let {1} = (K2 !c) in 1})
           o- (component (\!c. {let {1} = itree c A in
                                let {1} = itree c B in
```

```
                                let {1} = kedge c N A B in
                                let {1} = (K1 !c) in
                                let {1} = (K2 !c) in 1}) -@ kruskal C).

run : kruskal C
       o- (mflag -> {min N A B}) & build N A B C.

stop  : kruskal C
         @- component C.
```