# Towards Meta-Reasoning in the Concurrent Logical Framework CLF[*]

Iliano Cervesato Jorge Luis Sacchini
Carnegie Mellon University
iliano@cmu.edu sacchini@qatar.cmu.edu

The concurrent logical framework CLF is an extension of the logical framework LF designed to specify concurrent and distributed languages. While it can be used to define a variety of formalisms, reasoning about such languages within CLF has proved elusive. In this paper, we propose an extension of LF that allows us to express properties of CLF specifications. We illustrate the approach with a proof of safety for a small language with a parallel semantics.

## 1 Introduction

Due to the widespread availability of multi-core architectures and the growing demands of web applications and cloud-based computation models, primitives for programming concurrent and distributed systems are becoming essential features in modern programming languages. However, their semantics and meta-theory are not as well understood as those of sequential programming languages. This limits our assurance in the correctness of the systems written in them. Thus, just as in the case of sequential languages 40 years ago, there has been increasing interest in defining formal semantics that isolate and explain their quintessential features. Just as for sequential languages, such semantics hold the promises of developing, for example, provably-correct compilers and optimizations for such languages, as well as verification frameworks for concurrent applications written using them.

Logical frameworks are formalisms designed to specify and reason about the meta-theory of programming languages and logics. They are at the basis of tools such as Agda [10], Coq [7], Isabelle [9], and Twelf [12]. The current generation of logical frameworks were designed to study sequential programming languages, and specifying concurrent systems using these tools requires a large effort, as the user is forced to define ad-hoc concurrency models that are difficult to reuse and automate.

One way to deal with this problem is to design a logical framework that natively embeds a general-purpose concurrency model. This then provides native support for describing parallel execution and synchronization, for example, thus freeing the user from the delicate task of correctly encoding them and proving properties about them. One example of this approach is the concurrent logical framework CLF [4,13,15], an extension of the logical framework LF [6] designed for specifying concurrent, parallel, and distributed languages. One of its distinguishing features is its support for expressing concurrent traces, i.e., computations where independent steps can be permuted. For example, traces can represent sequences of evaluation steps in a parallel operational semantics, where executions that differ only in the order of independent steps are represented by the same object (modulo permutation). CLF has been used to encode a variety of systems such as Concurrent ML, the $\pi$-calculus, and Petri nets in a natural way [4].

However, unlike LF which permits specifying a system and its meta-theory within the same framework, CLF is not expressive enough for proving meta-theoretical properties about CLF specifications

[*]This work was supported by the Qatar National Research Fund under grant NPRP 09-1107-1-168.

(e.g., type preservation, or the correctness of program transformations). The main reason is that traces are not first-class values in CLF, and therefore cannot be manipulated. In this work we propose a logical framework that supports meta-reasoning over parallel, concurrent, and distributed specifications. Specifically, the main contributions of this paper are the following:

- We define an extension of LF, called Meta-CLF, that allows meta-reasoning over a CLF specification. It enriches LF with a type for concurrent traces and the corresponding constructor and destructors (via pattern-matching). This permits a direct manipulation of traces. Meta-theorems can be naturally represented as relations, similar to the way sequential programming languages are analyzed in LF.
- We illustrate the use of Meta-CLF by proving safety for a CLF specification of a small programming language with a parallel semantics.

The rest of the paper is organized as follows: in Sect. 2 we recall CLF and use it to define the operational semantics of a simple parallel language. In Sect. 3 we present Meta-CLF and use it to express a proof of safety for this language. We discuss related work in Sect. 4 and outline directions of future research in Sect. 5.

## 2 CLF

We begin by defining some key elements of CLF. For conciseness, we omit aspects of CLF that are not used in our examples. The results of this paper extend to the full language, however. The presentation given here follows the template proposed in [3] rather than the original definition of CLF [4, 15]; see also [13].

### 2.1 Syntax and Typing Rules

CLF is an extension of LF, or more precisely of the linear logical framework LLF [1], with a lax modality from lax logic [5] used to encapsulate the effects of concurrent computations. The introduction form of lax modality are witnessed by a form of proof term called *traces*. A trace is a sequence of computational steps where independent steps can be permuted.

The syntax of CLF is given by the following grammar:

$$
\begin{array}{rcll}
K & ::= & \mathsf{type} \mid \Pi !x{:}T.K & \text{(Kinds)} \\
P & ::= & a \cdot !S \mid \{\Delta\} & \text{(Base types)} \\
T & ::= & \Pi \square x{:}T.T \mid P & \text{(Types)} \\
N & ::= & \lambda \square x{:}T.N \mid H \cdot S \mid \{\varepsilon\} & \text{(Terms)} \\
\varepsilon & ::= & \diamond \mid \varepsilon_1 ; \varepsilon_2 \mid \{\Delta\} \leftarrow c \cdot S & \text{(Traces)} \\
S & ::= & \cdot \mid \square N ; S & \text{(Spines)} \\
\Delta & ::= & \cdot \mid \Delta, \square x : T & \text{(Contexts)} \\
\square & ::= & \downarrow \mid ! & \text{(Modalities)}
\end{array}
$$

In this paper, we only consider CLF's persistent (!) and linear (↓) substructural modalities. CLF also includes an affine modality, omitted here for space reasons.

Kinds are as in LF. Note that the argument in product kinds must be persistent. Base types are either *atomic* ($a \cdot !S$) which are formed by a constant applied to a persistent spine [2], or *monadic* which are a context enclosed in the lax modality, denoted with $\{\_\}$.

A type is either a product or a base type. We consider two different products: a persistent product, $\Pi!x{:}TU$, as in LF, and a linear product, $\Pi{\downarrow}x{:}TU$, from LLF. Note, however, that the typing rules prevent dependencies on linear products. We usually write $T \to U$ and $T \multimap U$ for $\Pi!x{:}TU$ and $\Pi{\downarrow}x{:}TU$, respectively, when $x$ is not free in $U$.

A term is either an abstraction (persistent and linear), an atomic term $H \cdot S$ formed by a variable $H$ applied to a list of arguments given by the spine $S$, or a trace $\{\varepsilon\}$.

A trace is either the empty trace ($\diamond$), a composition of traces ($\varepsilon_1; \varepsilon_2$) or an individual step of the form $\{\Delta\}{\leftarrow}c \cdot S$, where $c$ is a constant defined in the signature applied to a spine $S$, whose type must be monadic. This step *consumes* the linear variables in $S$ and *produces* the linear and persistent variables in $\Delta$. A step binds the variables defined in $\Delta$ in any trace that follows it.

Concurrent computation is expressed by endowing traces with a monoidal structure: the empty trace is the unit, and trace composition is associative. Furthermore, it allows permutation of independent steps. Step independence is defined on the basis of the notion of *trace interface*. The *input interface* of a trace, denoted $\bullet\varepsilon$, is the set of variables used by $\varepsilon$, i.e., its free variables. The *output interface* of a trace, denoted $\varepsilon\bullet$, is the set of variables defined by $\varepsilon$. They are given by the following equalities:

$$\begin{aligned} \bullet(\diamond) &= \emptyset & (\diamond)\bullet &= \emptyset \\ \bullet(\{\Delta\}{\leftarrow}c \cdot S) &= \mathrm{FV}(S) & (\{\Delta\}{\leftarrow}c \cdot S)\bullet &= \mathrm{dom}(\Delta) \\ \bullet(\varepsilon_1;\varepsilon_2) &= \bullet\varepsilon_1 \cup (\bullet\varepsilon_2 \setminus \varepsilon_1\bullet) & (\varepsilon_1;\varepsilon_2)\bullet &= \varepsilon_2\bullet \cup (\varepsilon_1\bullet \setminus \bullet\varepsilon_2) \cup !(\varepsilon_1\bullet) \end{aligned}$$

where $\mathrm{FV}(S)$ is the set of free variables in $S$, and $\mathrm{dom}(\Delta)$ is the set of variables declared in $\Delta$. In a trace composition, the output interface contains all the persistent variables introduced in $\varepsilon_1$, even if $\varepsilon_2$ uses them. In other words, persistent facts cannot be removed from the output once they are introduced. On the other hand, linear facts are effectively removed if $\varepsilon_2$ uses them.

Two traces $\varepsilon_1$ and $\varepsilon_2$ are *independent*, denoted $\varepsilon_1 \parallel \varepsilon_2$, if $\bullet\varepsilon_1 \cap \varepsilon_2\bullet = \emptyset$ and $\varepsilon_1\bullet \cap \bullet\varepsilon_2 = \emptyset$. Independent traces do not share variables and can therefore be executed in any order.

**Equality.** We denote with $\equiv$ the equality relation on kinds, types, terms, spines, and contexts. It is defined as $\alpha$-equality extended with trace equality, also denoted with $\equiv$, defined by the following rules:

$$\frac{}{\varepsilon;\diamond \equiv \varepsilon} \qquad \frac{}{\varepsilon \equiv \varepsilon;\diamond} \qquad \frac{}{\varepsilon_1;(\varepsilon_2;\varepsilon_3) \equiv (\varepsilon_1;\varepsilon_2);\varepsilon_3}$$

$$\frac{\varepsilon_1 \parallel \varepsilon_2}{\varepsilon_1;\varepsilon_2 \equiv \varepsilon_2;\varepsilon_1} \qquad \frac{\varepsilon_1 \equiv \varepsilon_1'}{\varepsilon_1;\varepsilon_2 \equiv \varepsilon_1';\varepsilon_2} \qquad \frac{\varepsilon_2 \equiv \varepsilon_2'}{\varepsilon_1;\varepsilon_2 \equiv \varepsilon_1;\varepsilon_2'}$$

These rules state that traces form a monoid and that independent steps can be permuted.

**Typing.** The typing rules of CLF rely on some auxiliary meta-level operators. We say that a context $\Delta$ *splits* into $\Delta_1$ and $\Delta_2$, denoted $\Delta = \Delta_1 \bowtie \Delta_2$, if each persistent declaration in $\Delta$ appears in both $\Delta_1$ and $\Delta_2$, and each linear declaration in $\Delta$ appears in exactly one of $\Delta_1$ and $\Delta_2$.

We write $X[N/x]$ for the *hereditary substitution* of variable $x$ by term $N$ in $X$ (where $X$ belongs to one of the CLF syntactic classes). Hereditary substitutions [16] normalizes a terms as the substitution is carried out, thereby allowing us to restrict the definition of CLF to canonical terms (in $\beta\eta$-normal form).

Its definition is type-directed and therefore terminating. The interested reader can find an exhaustive account in [16].

The typing rules of CLF are displayed in Fig. 1. We assume a fixed signature $\Sigma$ of global declarations for kinds and types. Typical of type theories, we use bidirectional typing rules where types of variables are inferred from the context, while terms are checked against a type.

The rules for kinds, types, spines and terms are standard [16]. Note that only persistent variables are dependent in types. The typing rules for traces show the intuition that a trace is a context transformer: we can read the judgment $\Delta \vdash \varepsilon : \Delta'$ as "*$\varepsilon$ transform the context $\Delta$ into $\Delta'$*". Note that the trace typing rules imply a form of the frame rule: in fact, it is easy to prove that if $\Delta_1 \vdash \varepsilon : \Delta_2$, then $\Delta_0 \bowtie \Delta_1 \vdash \varepsilon : \Delta_0 \bowtie \Delta_2$. The empty trace does not change the context, while traces can be composed if the internal interface matches. For a single step, part of the context is transformed: the spine $S$ consumes $\Delta_1$ and generates the context $\Delta'$ (or equivalently, $\Delta_2$).

## 2.2 Substructural Operational Semantics in CLF

In this section, we show how to use CLF to define a substructural operational semantics (SSOS) for a programming language. As a case study, we illustrate the approach on the simply-typed lambda calculus with an operational semantics that evaluates functions and their arguments in parallel.

SSOS specifications have two main features. The first is compositionality: extending a programming language with a new feature does not invalidate the SSOS already developed [14]. Second, SSOS specifications can naturally express parallel and concurrent semantics of a programming language [11].

The language of expressions and types for the simply-typed lambda calculus, $\lambda^{\rightarrow}$, is given by the following grammar:

$$\begin{aligned} e &::= x \mid \lambda x.e \mid e\,e && \text{(Expressions)} \\ \tau &::= o \mid \tau \rightarrow \tau && \text{(Types)} \end{aligned}$$

Expressions are either variables, abstractions or applications; types are either base types or function types. Typing is defined by the judgment $\Gamma \vdash e : \tau$, given by the following rules:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

Evaluation is given by $\beta$-reduction: $(\lambda x.e_1)\, e_2 \rightsquigarrow e_1[e_2/x]$.

In CLF (and LF as well), we can represent this language and its typing rules using a higher-order abstract syntax encoding as follows. For clarity, we use implicit arguments (which can be reconstructed):

$$\begin{aligned} \mathsf{exp} &: \mathsf{type} & \mathsf{of} &: \mathsf{exp} \rightarrow \mathsf{tp} \rightarrow \mathsf{type} \\ \mathsf{lam} &: (\mathsf{exp} \rightarrow \mathsf{exp}) \rightarrow \mathsf{exp} & \mathsf{of/app} &: \mathsf{of}\ (\mathsf{app}\ e_1\ e_2)\ t_1 \\ \mathsf{app} &: \mathsf{exp} \rightarrow \mathsf{exp} \rightarrow \mathsf{exp} & &\leftarrow \mathsf{of}\ e_1\ (\mathsf{arr}\ t_2\ t_1) \\ \mathsf{tp} &: \mathsf{type} & &\leftarrow \mathsf{of}\ e_2\ t_2 \\ \mathsf{arr} &: \mathsf{tp} \rightarrow \mathsf{tp} \rightarrow \mathsf{tp} & \mathsf{of/lam} &: \mathsf{of}\ (\mathsf{lam}\ e_2)\ (\mathsf{arr}\ t_1\ t_2) \\ \mathsf{value} &: \mathsf{exp} \rightarrow \mathsf{type} & &\leftarrow (\Pi x : \mathsf{exp}.\ \mathsf{of}\ x\ t_1 \rightarrow \mathsf{of}\ (e_2\ x)\ t_2) \\ \mathsf{value/lam} &: \mathsf{value}\ (\mathsf{lam}\ \lambda x.e\ x) & & \end{aligned}$$

Contexts: $\boxed{!\Delta \vdash \Delta'}$

$$\frac{}{!\Delta \vdash \cdot} \qquad \frac{!\Delta \vdash T : \mathsf{type} \qquad !(\Delta, \square x{:}T) \vdash \Delta'}{!\Delta \vdash \square x{:}T, \Delta'}$$

Kinds: $\boxed{!\Delta \vdash K : \mathsf{kind}}$

$$\frac{}{!\Delta \vdash \mathsf{type} : \mathsf{kind}} \qquad \frac{!\Delta \vdash T : \mathsf{type} \qquad !\Delta, !x{:}T \vdash K : \mathsf{kind}}{!\Delta \vdash \Pi !x{:}T.K : \mathsf{kind}}$$

Base types: $\boxed{!\Delta \vdash P : K}$

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

Types: $\boxed{!\Delta \vdash T : K}$

$$\frac{!\Delta \vdash T : \mathsf{type} \qquad !\Delta, !x{:}T \vdash T : \mathsf{type}}{!\Delta \vdash \Pi !x{:}T.T : \mathsf{type}} \qquad \frac{!\Delta \vdash T : \mathsf{type} \qquad !\Delta \vdash U : \mathsf{type} \qquad x \text{ fresh}}{!\Delta \vdash T \multimap U : \mathsf{type}}$$

Terms: $\boxed{\Delta \vdash N \Leftarrow T}$

$$\frac{\Delta, \square x{:}T \vdash N \Leftarrow U}{\Delta \vdash \lambda \square x{:}.N \Leftarrow \Pi \square x{:}T.U} \qquad \frac{\Delta \vdash \varepsilon : \Delta'}{\Delta \vdash \{\varepsilon\} \Leftarrow \{\Delta'\}}$$

$$\frac{!x{:}T \in \Delta \qquad \Delta \vdash S : T > U \qquad U \equiv U'}{\Delta \vdash x \cdot S \Leftarrow U'} \qquad \frac{\Delta_0, \Delta_1 \vdash S : T > U \qquad U \equiv U'}{\Delta_0, {\downarrow} x{:}T, \Delta_1 \vdash x \cdot S \Leftarrow U'}$$

$$\frac{c{:}T \in \Sigma \qquad \Delta \vdash S : T > U \qquad U \equiv U'}{\Delta \vdash c \cdot S \Leftarrow U'}$$

Traces: $\boxed{\Delta \vdash \varepsilon : \Delta'}$

$$\frac{}{\Delta \vdash \diamond : \Delta} \qquad \frac{\Delta \vdash \varepsilon_1 : \Delta_1 \qquad \Delta_1 \vdash \varepsilon_2 : \Delta_2}{\Delta \vdash \varepsilon_1 ; \varepsilon_2 : \Delta_2}$$

$$\frac{c{:}T \in \Sigma \qquad \Delta_1 \vdash S : T > \{\Delta'\} \qquad \Delta_2 \equiv \Delta'}{\Delta_0 \bowtie \Delta_1 \vdash \{\Delta_2\} \leftarrow c \cdot S : \Delta_0, \Delta_2}$$

Spines: $\boxed{\Delta \vdash S : T > T'}$

$$\frac{}{\Delta \vdash \cdot : T > T} \qquad \frac{!\Delta_1 \vdash N \Leftarrow T \qquad \Delta_0 \vdash S : T_2[N/x] > T'}{\Delta_0 \bowtie !\Delta_1 \vdash (!N; S) : \Pi !x{:}TU > T'} \qquad \frac{{\downarrow}\Delta_1 \vdash N \Leftarrow T \qquad \Delta_0 \vdash S : U > T'}{\Delta_0 \bowtie {\downarrow}\Delta_1 \vdash ({\downarrow}N; S) : T \multimap U > T'}$$

Figure 1: CLF typing rules

We denote this signature with $\Sigma_{\lambda^{\rightarrow}}$. The syntax of $\lambda^{\rightarrow}$ is defined by the type $\mathsf{exp}$ with constructors $\mathsf{app}$ (representing function application) and $\mathsf{lam}$ (representing abstraction). The type $\mathsf{tp}$ encodes types from $\lambda^{\rightarrow}$ with $\mathsf{arr}$ being the function type from $\lambda^{\rightarrow}$. Variables are implicitly defined using CLF variables. Similarly, there is no explicit representation of the context for typing; instead, we use CLF's (persistent) context for this purpose. The typing relation is expressed by the CLF type family of relating $\lambda^{\rightarrow}$ expressions and types. We also define the predicate $\mathsf{value}$ stating that abstractions are values.

In the following, we define a SSOS for $\lambda^{\rightarrow}$ evaluation using destination-passing style [11]. The SSOS is given by a state and a set of rewriting rules. The state is a multiset whose elements have one of the following forms:

- $\mathsf{eval}\ e\ d$: evaluate expression $e$ in destination $d$. Destinations are virtual locations that store expressions to be evaluated and results.
- $\mathsf{ret}\ e\ d$: the result $e$ of an evaluation is stored at $d$. An invariant of the semantics ensures that $e$ is always a value.
- $\mathsf{fapp}\ d_1\ d_2\ d$: an application frame expecting the result of evaluating a function in $d_1$, its argument in $d_2$, and storing the result in $d$.

The rewriting rules encoding expression evaluation are as follows:

$$\begin{array}{rcll}
\mathsf{eval}\ e\ d & \rightsquigarrow & \mathsf{ret}\ e\ d & \text{if } \mathsf{value}(e) \\
\mathsf{eval}\ (e_1\, e_2)\ d & \rightsquigarrow & \mathsf{eval}\ e_1\ d_1,\ \mathsf{eval}\ e_2\ d_2,\ \mathsf{fapp}\ d_1\ d_2\ d & d_1, d_2 \text{ fresh} \\
\mathsf{ret}\ (\lambda x.e_1)\ d_1,\ \mathsf{ret}\ e_2\ d_2,\ \mathsf{fapp}\ d_1\ d_2\ d & \rightsquigarrow & \mathsf{eval}\ (e_1[e_2/x])\ d &
\end{array}$$

The first rule says that evaluating a value expression immediately returns the result. The second rule says that to evaluate an application, we evaluate the function and argument in two fresh destinations, and create a frame that connects the results. Evaluation of function and argument can proceed in parallel. The third rule computes the application once we have the values of the function and argument connected by a frame.

A complete evaluation of an expression $e$ to a value $v$ is given by a sequence of multisets of the form:

$$\mathscr{A}_0 = \{\mathsf{eval}\ e\ d\} \rightsquigarrow \mathscr{A}_1 \rightsquigarrow \ldots \rightsquigarrow \mathscr{A}_{n-1} \rightsquigarrow \mathscr{A}_n = \{\mathsf{ret}\ v\ d\}$$

where at each step $\mathscr{A}_i \rightsquigarrow \mathscr{A}_{i+1}$ part of $\mathscr{A}_i$ is rewritten using one of the evaluation rules given above.

This semantics can be faithfully represented in CLF using the linear context to represent the multiset state, where each element is a linear fact, and destinations are represented by persistent facts. The semantics is given by the following CLF signature:

$$\begin{array}{rl}
\mathsf{dest} : & \mathsf{type} \\
\mathsf{eval} : & \mathsf{exp} \to \mathsf{dest} \to \mathsf{type} \\
\mathsf{ret} : & \mathsf{exp} \to \mathsf{dest} \to \mathsf{type} \\
\mathsf{fapp} : & \mathsf{dest} \to \mathsf{dest} \to \mathsf{dest} \to \mathsf{type} \\
\mathsf{step/eval} : & \mathsf{eval}\ e\ d \multimap \mathsf{value}\ e \to \{\downarrow x{:}\mathsf{ret}\ e\ d\} \\
\mathsf{step/app} : & \mathsf{eval}\ (\mathsf{app}\ e_1\ e_2)\ d \multimap \{!d_1{:}\mathsf{dest}, !d_2{:}\mathsf{dest}, \downarrow x_1{:}\mathsf{eval}\ e_1\ d_1, \downarrow x_2{:}\mathsf{eval}\ e_2\ d_2, \downarrow x_3{:}\mathsf{fapp}\ d_1\ d_2\ d\} \\
\mathsf{step/beta} : & \mathsf{ret}\ (\mathsf{lam}\ e_1)\ d_1 \multimap\ \mathsf{ret}\ e_2\ d_2 \multimap \mathsf{fapp}\ d_1\ d_2\ d \multimap \{\downarrow x{:}\mathsf{eval}\ (e_1\ e_2)\ d\}
\end{array}$$

We denote with $\Sigma_{\mathsf{step}}$ the signature containing the declarations of the evaluation rules. Rule $\mathsf{step/eval}$ is effectively a conditional rewriting rule. In rule $\mathsf{step/app}$, new destinations ($d_1$ and $d_2$) are created to evaluate function and argument; these evaluations can proceed in parallel.

**Safety.** Safety for this language is proven by giving a suitable notion of what a valid state looks like [14]. Note that not all multisets are valid state. For example, the singleton $\{\mathsf{fapp}\ d_1\ d_2\ d\}$ is not valid, since there is no expression to evaluate at $d_1$ or $d_2$; similarly $\{\mathsf{eval}\ e_1\ d, \mathsf{eval}\ e_2\ d\}$ is not valid since there are two expressions evaluating on the same destination.

In a valid state, the elements should form a tree whose nodes are linked by destinations. Internal nodes have the form $\mathsf{fapp}\ d_1\ d_2\ d$, with two children (corresponding to $d_1$ and $d_2$) and the leaves are of the form $\mathsf{eval}\ e\ d$ or $\mathsf{ret}\ v\ d$ (where $v$ is a value). Furthermore, the types of the expressions should match.

Following Simmons [14], we define well-typed states by rewriting rules. The idea is to create the tree *top-down* starting at the root. We can write these rules in CLF as follows:

$$\begin{aligned}
\mathsf{gen} &: \mathsf{tp} \to \mathsf{dest} \to \mathsf{type}\\
\mathsf{gen/eval} &: \mathsf{gen}\ t\ d \multimap \mathsf{of}\ e\ t \to \{\downarrow x{:}\mathsf{eval}\ e\ d\}\\
\mathsf{gen/ret} &: \mathsf{gen}\ t\ d \multimap \mathsf{of}\ e\ t \to \mathsf{value}\ e \to \{\downarrow x{:}\mathsf{ret}\ e\ d\}\\
\mathsf{gen/fapp} &: \mathsf{gen}\ t\ d \multimap \{!d_1{:}\mathsf{dest}, !d_1{:}\mathsf{dest}, \downarrow x_0{:}\mathsf{fapp}\ d_1\ d_2\ d, \downarrow x_1{:}\mathsf{gen}\ (\mathsf{arr}\ t_1\ t)\ d_1, \downarrow x_2{:}\mathsf{gen}\ t_1\ d_2\}\\
\mathsf{gen/dest} &: \{!d{:}\mathsf{dest}\}
\end{aligned}$$

We denote with $\Sigma_{\mathsf{gen}}$ the signature containing these generation rules. A fact of the form $\mathsf{gen}\ t\ d$ is read as "*generate a tree with root at destination d and type t*". We have three ways to do this: by generating a leaf of either the form $\mathsf{eval}\ e\ d$ or $\mathsf{ret}\ e\ d$ (for an $e$ of the appropriate type), or by generating an internal node $\mathsf{fapp}\ d_1\ d_2\ d$ and then generating trees rooted at $d_1$ and $d_2$. Rule gen/dest is necessary to keep track of destinations that were created during evaluation (by rule step/app) but are not used anymore (after the application is reduced using step/beta, the destinations created to evaluate the function and the argument are not needed anymore; see Sect. 3.2).

A generic tree is built by a sequence of rewriting steps starting from a single $\mathsf{gen}\ t\ d$: $\mathscr{A}_0 = \{\mathsf{gen}\ t\ d\} \rightsquigarrow \mathscr{A}_1 \rightsquigarrow \ldots \rightsquigarrow \mathscr{A}_n$, where $\mathscr{A}_n$ does not contain facts of the form $\mathsf{gen}\ t\ d$.

This kind of *generative rules* for describing valid states generalizes context-free grammars, with gen being a non-terminal, and eval, ret, and fapp are terminal symbols [14]. Generative rules (also called generative grammars) are very powerful allowing to express a wide variety of invariants.

With this definition of a well-typed state, we can prove that the language is safe. Safety is given by two properties: type preservation (i.e., evaluation preserves well-typed states), and progress (i.e., either the state contains the final result, or it is possible to make a step), as stated in the following theorem. We write $\mathscr{A} \rightsquigarrow^*_\Sigma \mathscr{A}'$ to mean a *maximal* rewrite sequence from $\mathscr{A}$ to $\mathscr{A}'$ using the rules in $\Sigma$. The sequence is maximal in the sense that $\mathscr{A}'$ does not contain non-terminal symbols. We write $\mathscr{A} \rightsquigarrow^1_\Sigma \mathscr{A}'$ to mean a *step* from $\mathscr{A}$ to $\mathscr{A}'$ using one of the rules in $\Sigma$.

**Theorem 1.** *The language defined by the signatures $\Sigma_{\lambda\to}$, $\Sigma_{\mathsf{step}}$ and $\Sigma_{\mathsf{gen}}$ is safe, i.e., it satisfies the following properties:*

**Preservation** *If $\{\mathsf{gen}\ t\ d\} \rightsquigarrow^*_{\Sigma_{\mathsf{gen}}} \Delta$ and $\Delta \rightsquigarrow^1_{\Sigma_{\mathsf{step}}} \Delta'$, then $\{\mathsf{gen}\ t\ d\} \rightsquigarrow^*_{\Sigma_{\mathsf{gen}}} \Delta'$.*

**Progress** *If $\{\mathsf{gen}\ t\ d\} \rightsquigarrow^*_{\Sigma_{\mathsf{gen}}} \Delta$, then either $\Delta$ is of the form $\{\mathsf{ret}\ v\ d\}$ with $\mathsf{value}\ v$, or there exists $\Delta'$ such that $\Delta \rightsquigarrow_{\Sigma_{\mathsf{step}}} \Delta'$.*

*Proof sketch.* (See [14] for details.) Preservation proceeds by case analysis on the rewriting step and inversion on the generated trace. Let us consider the case step/eval. We have that $\Delta$ must be of the form $\Delta_0, \mathsf{eval}\ e\ d$, and $\Delta'$ must be of the form $\Delta_0, \mathsf{ret}\ e\ d$, where $\mathsf{value}\ e$. Then, the generation trace for $\Delta$ must contain a step using gen/eval to construct $\mathsf{eval}\ e\ d$. The generation trace for $\Delta'$ is constructed by replacing this step with a gen/ret step.

Progress proceed by induction on the length of the generating trace and case analysis on the first step. The interesting case is when this step is $\mathsf{gen}/\mathsf{fapp}$. The rest of the trace can be split in two parts, one generating trace for each child of the $\mathsf{fapp}$ generated in the first step. The proof follows by induction on these subtraces. $\square$

## 3 Meta-CLF

Meta-theorems such as preservation and progress cannot be expressed in CLF, since it lacks primitives for manipulating traces as first-class objects. For example, we cannot talk about the type of generated traces of the form

$$\{\Delta\} \rightsquigarrow^*_{\Sigma_{\mathsf{gen}}} \{\Delta'\}$$

which is essential to express preservation.

Furthermore, CLF lacks abstractions over context, which prevents us from defining a trace type that is parametric over its interface. For example, in CLF we can define a relation on traces

$$\mathsf{rel} : (A \multimap \{\downarrow x{:}B\}) \to (A \multimap \{\downarrow x{:}B\}) \to \mathsf{type}$$

that relates two traces that transform an $A$ into a $B$. However, we cannot define the type of traces as a transformation between two generic contexts $\Delta_1$ and $\Delta_2$. For this, we need to quantify over contexts. With a dependent product that takes contexts as arguments, we can define of all traces that generate valid states starting from a seed:

$$\Pi t : \mathsf{tp}.\Pi\psi : \mathsf{ctx}.\ (\Pi !d{:}\mathsf{dest}.\mathsf{gen}\ d\ t \multimap \{\psi\})$$

We use these ideas for designing a logical framework that permits meta-reasoning on CLF specifications. The resulting framework, which we call Meta-CLF, is an extension of LF with trace types and quantification over contexts. Trace types have the form

$$\{\Delta\}\,\Sigma\,\{\Delta'\}$$

where $\Delta$ and $\Delta'$ are CLF contexts and $\Sigma$ is a CLF signature that contains (monadic) rewriting rules (e.g., $\Sigma_{\mathsf{gen}}$ and $\Sigma_{\mathsf{step}}$). A term of this type is a trace of the form:

$$\delta_1; \ldots; \delta_n$$

where each step $\delta_i$ is either $\{\Delta_i\} \leftarrow c_i \cdot S_i$ with $c_i$ declared in $\Sigma$, or $x_i \cdot S_i$ where $x_i$ is a (Meta-CLF) variable that, applied to $S_i$, returns a trace. The interface of the whole trace is given by $\Delta$ and $\Delta'$.

Meta-CLF includes two different trace types: $\{\Delta\}\,\Sigma^*\,\{\Delta'\}$ and $\{\Delta\}\,\Sigma^1\,\{\Delta'\}$. The former defines maximal traces, while the latter defines traces of exactly one step. We write $\{\Delta\}\,\Sigma\,\{\Delta'\}$ to refer to either of these types.

### 3.1 Syntax and Typing Rules

Meta-CLF is an extension of LF with trace and context types, parameterized over a CLF signature, denoted $\Sigma_0$. The kinds, types, and contexts of Meta-CLF are given by the following grammar:

$$\begin{array}{rll} K ::= & \mathsf{type} \mid \Pi x{:}A.K \mid \Pi\psi{:}\mathsf{ctx}.K \mid \widehat{\Pi} x{:}T.K \mid \nabla x.K & \text{(Kinds)} \\ A ::= & a \cdot S \mid \Pi x{:}A.A \mid \Pi\psi{:}\mathsf{ctx}.A \mid \widehat{\Pi} x{:}T.A \mid \nabla x.A & \\ & \mid \{\Delta\}\,\Sigma^*\,\{\Delta\} \mid \{\Delta\}\,\Sigma^1\,\{\Delta\} & \text{(Types)} \\ \Delta ::= & \cdot \mid \psi, \Delta \mid \downarrow x{:}A, \Delta \mid !x{:}A, \Delta & \text{(Contexts)} \end{array}$$

Kinds include, besides the constructions derived from LF, products over contexts ($\Pi\psi{:}\mathsf{ctx}.K$), products over CLF types from the signature $\Sigma_0$ ($\widehat{\Pi}x{:}T K$), and products over names ($\nabla x.K$). In $\widehat{\Pi}x{:}T K$, the type $T$ must be well-typed in the signature $\Sigma_0$, using the CLF typing rules. Quantification over names is necessary for composing traces, to ensure that names declared in the interface match. The notation is taken from [8].

Types in Meta-CLF include the analogous kinds contructors applied to the type level, with the addition of the trace types $\{\Delta\}\,\Sigma^*\,\{\Delta'\}$ and $\{\Delta\}\,\Sigma^1\,\{\Delta'\}$. Contexts are sequences consisting of linear declarations, persistent declarations, and context variables, whose types are CLF types checked in the signature $\Sigma_0$. Names of declared variables must be introduced using $\nabla$.

Type preservation for the language described in Sect. 2.2 can then be stated in Meta-CLF as follows:

$$\begin{aligned}&\widehat{\Pi}t{:}\mathsf{tp}.\ \nabla d.\ \nabla g.\ \Pi\psi_1{:}\mathsf{ctx}.\ \Pi\psi_2{:}\mathsf{ctx}.\\ &\quad \{!d:\mathsf{dest}, \downarrow g:\mathsf{gen}\ d\ t\}\,\Sigma^*_{\mathsf{gen}}\,\{\psi_1\} \to \{\psi_1\}\,\Sigma^1_{\mathsf{step}}\,\{\psi_2\} \to \{!d:\mathsf{dest}, \downarrow g:\mathsf{gen}\ d\ t\}\,\Sigma^*_{\mathsf{gen}}\,\{\psi_2\} \to \mathsf{type}\end{aligned}$$

This type family can be read functionally as follows: given a trace that generates $\psi_1$ (using $\Sigma_{\mathsf{gen}}$), and a step from $\psi_1$ to $\psi_2$, we can obtain a trace that generates $\psi_2$ (using $\Sigma_{\mathsf{gen}}$).

Terms, spines, and traces in Meta-CLF are defined by the following grammar:

$$\begin{aligned}N &::= \lambda x.N \mid H\cdot S \mid \lambda\psi.N \mid \widehat{\lambda}x.N \mid \{\varepsilon\} && \text{(Terms)}\\ H &::= x \mid c && \text{(Heads)}\\ S &::= \cdot \mid N;S \mid \Delta;S \mid \#;S \mid \langle M\rangle && \text{(Spines)}\\ \varepsilon &::= \diamond \mid \varepsilon_1;\varepsilon_2 \mid \{\Delta\}\leftarrow c\cdot S \mid x\cdot S && \text{(Traces)}\end{aligned}$$

Terms include the introduction forms of Meta-CLF type ($\lambda x.N$), contexts ($\lambda\psi.N$), CLF types ($\widehat{\lambda}x.N$), as well as atomic terms (a variable applied to a spine) and traces ($\{\varepsilon\}$). Spines are sequence formed by terms ($N$), contexts ($\Delta$), CLF terms ($\langle M\rangle$), and fresh names (#).

A trace is either empty ($\diamond$), a composition of two traces ($\varepsilon_1;\varepsilon_2$), a single step ($\{\Delta\}\leftarrow c\cdot S$) where $c$ is defined in the CLF signature $\Sigma_0$, or a (Meta-CLF) trace variable applied to a spine ($x\cdot S$).

**Typing rules.** Typing judgments are parameterized by the CLF signature $\Sigma_0$. The typing rules are defined by the following judgments:

$$\begin{aligned}&\Sigma;\Gamma;\Xi \vdash_{\Sigma_0} K : \mathsf{kind} && \text{(Kinds)}\\ &\Sigma;\Gamma;\Xi \vdash_{\Sigma_0} A : K && \text{(Types)}\\ &\Sigma;\Gamma;\Xi \vdash_{\Sigma_0} \Delta\ \mathsf{ctx} && \text{(Contexts)}\\ &\Sigma;\Gamma;\Xi \vdash_{\Sigma_0} M : A && \text{(Terms)}\\ &\Sigma;\Gamma;\Xi \vdash_{\Sigma_0} S : A' > A && \text{(Spines)}\\ &\Sigma;\Gamma;\Xi \vdash_{\Sigma_0} \varepsilon : \{\Delta\}\,\Sigma\,\{\Delta'\} && \text{(Traces)}\end{aligned}$$

The signature $\Sigma$ is a Meta-CLF signature, while $\Sigma_0$ is a CLF signature. We usually omit them for clarity. The context $\Gamma$ contains the declarations of Meta-CLF variables, contexts, and CLF variables. The context $\Xi$ contains name declarations.

These contexts are defined by the following grammar:

$$\begin{aligned}\Gamma &::= \cdot \mid \Gamma, x{:}A \mid \Gamma, \psi{:}\mathsf{ctx} \mid \Gamma, x\widehat{:}T\\ \Xi &::= \cdot \mid \Xi, x\end{aligned}$$

The typing rules are defined in Fig. 2. We only show the rules related to the new constructions. We use the typing judgments from CLF to check products over CLF types. We also need a filtering operation on contexts, denoted $|\_|$ that keeps declarations of CLF types. It is defined by $|\cdot| = \cdot$, $|\Gamma, x\hat{:}T| = |\Gamma|, !x{:}T$, and $|\Gamma,\gamma| = |\Gamma|$ for declarations $\gamma$ of Meta-CLF types and contexts.

## 3.2 Safety for SSOS Specifications

We illustrate the use of Meta-CLF by stating and proving safety for the language introduced in Sect. 2.2. We use Meta-CLF over the signature $\Sigma_{\lambda^{\rightarrow}}$ defining the language $\lambda^{\rightarrow}$ and its static semantics.

Let us recall the type family expressing preservation of types in Meta-CLF:

$$\begin{array}{l}\mathsf{tpres} : \widehat{\Pi}t{:}\mathsf{tp}.\ \nabla d.\ \nabla g.\ \Pi\psi_1{:}\mathsf{ctx}.\ \Pi\psi_2{:}\mathsf{ctx}.\\ \quad \{!d : \mathsf{dest}, \downarrow g : \mathsf{gen}\ d\ t\}\,\Sigma^*_{\mathsf{gen}}\,\{\psi_1\} \to \{\psi_1\}\,\Sigma^1_{\mathsf{step}}\,\{\psi_2\} \to \{!d : \mathsf{dest}, \downarrow g : \mathsf{gen}\ d\ t\}\,\Sigma^*_{\mathsf{gen}}\,\{\psi_2\} \to \mathsf{type}\end{array}$$

The proof proceeds by case analysis on the type $\{\psi_1\}\,\Sigma^1_{\mathsf{step}}\,\{\psi_2\}$ (we follow essentially the same reasoning as described in the proof sketch of Theorem 1). We have three cases, one case for each rule in $\Sigma_{\mathsf{step}}$. For each case, we apply inversion on the trace of type $\{!d : \mathsf{dest}, \downarrow g : \mathsf{gen}\ d\ t\}\,\Sigma^*_{\mathsf{gen}}\,\{\psi_1\}$.

Consider the case step/eval; in this case $\psi_1$ must be of the form $\Delta, x{:}\mathsf{eval}\ e\ d_0$ and $\psi_2$ must be of the form $\Delta, y{:}\mathsf{ret}\ e\ d_0$, for some value expression $e$. By inversion, in the trace generating $\psi_1$, there must be a step that uses gen/eval to generate the declaration of $x$. That is, the generating trace has the form

$$X_1; \{\downarrow x\} \leftarrow \mathsf{gen/eval}\ e\ d_0\ g_0\ H; X_2$$

where $g_0{:}\mathsf{gen}\ d_0\ t_0$ for some type $t_0$ is generated by $X_1$ and $H$ is a proof that $e$ has type $t_0$ (i.e. $H : \mathsf{of}\ e\ t_0$). Note that $X_2$ cannot consume $x$, since eval is a terminal in the grammar defined by $\Sigma_{\mathsf{gen}}$. Then, the step generating $x$ can be permuted towards the end of the trace, so that $X_2$ can be taken to be the empty trace $\diamond$. To construct the trace that generates $\psi_2$, we only need to replace this last step by a gen/ret step.

In Meta-CLF, we can write this case of the proof as follows, where we omit the dependent arguments for clarity (like in LF and CLF, we expect that implicit arguments can be reconstructed):

$$\begin{array}{rl}\mathsf{tpres/ret} : \mathsf{tpres} & (X_1; \{\downarrow x\} \leftarrow \mathsf{gen/eval}\ e\ d_0\ g_0\ H)\\ & (\{\downarrow y\} \leftarrow \mathsf{step/eval}\ e\ d_0\ x\ H_v)\\ & (X_1; \{\downarrow y\} \leftarrow \mathsf{gen/ret}\ e\ d_0\ g_0\ H\ H_v)\end{array}$$

where $X_1 : \{!d : \mathsf{dest}, \downarrow g : \mathsf{gen}\ d\ t\}\,\Sigma^*_{\mathsf{gen}}\,\{\psi_1', !d_0 : \mathsf{dest}, \downarrow g_0 : \mathsf{gen}\ d_0\ t_0\}$ and $H_v{:}\mathsf{value}\ e$. The full type making explicit all arguments is the following:

$$\begin{array}{rl}\mathsf{tpres/ret} : & \nabla x.\nabla y.\nabla d.\nabla g.\nabla d_0.\nabla g_0.\Pi\psi_1' : \mathsf{ctx}.\widehat{\Pi}e : \mathsf{exp}.\widehat{\Pi}t : \mathsf{tp}.\widehat{\Pi}t_0 : \mathsf{tp}.\\ & \Pi X : \{!d : \mathsf{dest}, \downarrow g : \mathsf{gen}\ d\ t\}\,\Sigma^*_{\mathsf{gen}}\,\{\psi_1', !d_0 : \mathsf{dest}, \downarrow g_0 : \mathsf{gen}\ d_0\ t_0\}.\\ & \Pi H{:}\mathsf{of}\ e\ t_0.\Pi H_v{:}\mathsf{value}\ e.\\ & \mathsf{tpres}\ t\ d\ g\ (\psi_1', !d_0 : \mathsf{dest}, \downarrow x : \mathsf{eval}\ e\ d_0)\ (\psi_1', !d_0 : \mathsf{dest}, \downarrow y : \mathsf{ret}\ e\ d_0)\\ & \quad (X_1; \{\downarrow x : \mathsf{eval}\ e\ d_0\} \leftarrow \mathsf{gen/eval}\ e\ d_0\ g_0\ H)\\ & \quad (\{\downarrow y : \mathsf{ret}\ e\ d_0\} \leftarrow \mathsf{step/eval}\ e\ d_0\ x\ H_v)\\ & \quad (X_1; \{\downarrow y : \mathsf{ret}\ e\ d_0\} \leftarrow \mathsf{gen/ret}\ e\ d_0\ g_0\ H\ H_v)\end{array}$$

Kinds: $\boxed{\Sigma;\Gamma;\Xi \vdash_{\Sigma_0} K : \mathsf{kind}}$

$$\frac{\Gamma,\psi{:}\mathsf{ctx};\Xi \vdash K : \mathsf{kind}}{\Gamma;\Xi \vdash \Pi\psi{:}\mathsf{ctx}.K : \mathsf{kind}} \qquad \frac{\Gamma;\Xi,x \vdash K : \mathsf{kind}}{\Gamma;\Xi \vdash \nabla x.K : \mathsf{kind}} \qquad \frac{\Sigma_0;|\Gamma| \vdash_{\mathrm{CLF}} T : \mathsf{type} \qquad \Gamma,x{:}T;\Xi \vdash K : \mathsf{kind}}{\Gamma;\Xi \vdash \widehat{\Pi}x{:}T.K : \mathsf{kind}}$$

Types: $\boxed{\Sigma;\Gamma;\Xi \vdash_{\Sigma_0} A : K}$

$$\frac{\Gamma,\psi{:}\mathsf{ctx};\Xi \vdash A : \mathsf{type}}{\Gamma;\Xi \vdash \Pi\psi{:}\mathsf{ctx}.A : \mathsf{type}} \qquad \frac{\Gamma;\Xi,x \vdash A : K}{\Gamma;\Xi \vdash \nabla x.A : K}$$

$$\frac{\Gamma;\Xi \vdash \Delta_1\ \mathsf{ctx} \qquad \Gamma;\Xi \vdash \Delta_2\ \mathsf{ctx} \qquad |\Gamma| \vdash_{\mathrm{CLF}} \Sigma}{\Gamma;\Xi \vdash \{\Delta_1\}\Sigma\{\Delta_2\} : \mathsf{type}} \qquad \frac{|\Gamma| \vdash_{\mathrm{CLF}} T : \mathsf{type} \qquad \Gamma,x{:}T;\Xi \vdash A : \mathsf{type}}{\Gamma;\Xi \vdash \widehat{\Pi}x{:}T.A : \mathsf{type}}$$

Contexts: $\boxed{\Sigma;\Gamma;\Xi \vdash_{\Sigma_0} \Delta\ \mathsf{ctx}}$

$$\frac{}{\Gamma;\Xi \vdash \cdot\ \mathsf{ctx}} \qquad \frac{\begin{array}{c}\Gamma;\Xi \vdash \Delta\ \mathsf{ctx} \\ x \in \Xi \setminus \mathrm{dom}(\Delta) \qquad \Sigma_0;|\Gamma|,!\Delta \vdash_{\mathrm{CLF}} A : \mathsf{type}\end{array}}{\Gamma;\Xi \vdash \Delta,\Box x{:}A\ \mathsf{ctx}} \qquad \frac{\Gamma;\Xi \vdash \Delta\ \mathsf{ctx} \qquad \psi{:}\mathsf{ctx} \in \Gamma}{\Gamma;\Xi \vdash \Delta,\psi\ \mathsf{ctx}}$$

Terms: $\boxed{\Sigma;\Gamma;\Xi \vdash_{\Sigma_0} M : A}$

$$\frac{\Gamma,x:A;\Xi \vdash M : B}{\Gamma;\Xi \vdash \lambda x.M : \Pi x:A.B} \qquad \frac{c{:}B \in \Sigma \qquad \Gamma;\Xi \vdash S : B > A}{\Gamma;\Xi \vdash c \cdot S : A} \qquad \frac{x{:}B \in \Gamma \qquad \Gamma;\Xi \vdash S : B > A}{\Gamma;\Xi \vdash x \cdot S : A}$$

$$\frac{\Gamma,\psi{:}\mathsf{ctx};\Xi \vdash N : A}{\Gamma \vdash \lambda\psi.N : \Pi\psi{:}\mathsf{ctx}.A} \qquad \frac{\Gamma,x\hat{:}T;\Xi \vdash N : A}{\Gamma;\Xi \vdash \widehat{\lambda}x.N : \widehat{\Pi}x{:}T.A} \qquad \frac{\Gamma;\Xi \vdash \varepsilon : \{\Delta_1\}\Sigma\{\Delta_2\}}{\Gamma;\Xi \vdash \{\varepsilon\} : \{\Delta_1\}\Sigma\{\Delta_2\}}$$

Spines: $\boxed{\Sigma;\Gamma;\Xi \vdash_{\Sigma_0} S : A' > A}$

$$\frac{}{\Gamma;\Xi \vdash \cdot : A > A} \qquad \frac{\Gamma;\Xi \vdash \Delta\ \mathsf{ctx} \qquad \Gamma;\Xi \vdash S : A[\Delta/\psi] > B}{\Gamma;\Xi \vdash \Delta;S : \Pi\psi{:}\mathsf{ctx}.A > B} \qquad \frac{\alpha \text{ fresh} \qquad \Gamma;\Xi \vdash S : A[\alpha/x] > B}{\Gamma;\Xi \vdash \#;S : \nabla x.A > B}$$

$$\frac{\Gamma;\Xi \vdash N : A_1 \qquad \Gamma;\Xi \vdash S : A_2[N/x] > B}{\Gamma;\Xi \vdash N;S : \Pi x{:}A_1.A_2 > B} \qquad \frac{\Sigma_0;|\Gamma| \vdash_{\mathrm{CLF}} M : T \qquad \Gamma;\Xi \vdash S : A[\langle M\rangle c/x] > B}{\Gamma;\Xi \vdash \langle M\rangle;S : \Pi x{:}T.A > B}$$

Traces: $\boxed{\Sigma;\Gamma;\Xi \vdash_{\Sigma_0} \varepsilon : \{\Delta\}\Sigma^*\{\Delta'\}}$

$$\frac{}{\Gamma;\Xi \vdash \diamond : \{\Delta\}\Sigma^*\{\Delta\}} \qquad \frac{\Gamma;\Xi \vdash \varepsilon_1 : \{\Delta_1\}\Sigma^*\{\Delta\} \qquad \Gamma;\Xi \vdash \varepsilon_2 : \{\Delta\}\Sigma^*\{\Delta_2\}}{\Gamma;\Xi \vdash \varepsilon_1;\varepsilon_2 : \{\Delta_1\}\Sigma^*\{\Delta_2\}}$$

$$\frac{c{:}A \in \Sigma \qquad \Sigma_0;!|\Gamma|,\Delta_1 \vdash_{\mathrm{CLF}} S : A > \Delta_1}{\Gamma;\Xi \vdash \{\Delta_2\}{\leftarrow}c \cdot S : \{\Delta_0 \bowtie \Delta_1\}\Sigma\{\Delta_0,\Delta_2\}} \qquad \frac{x{:}A \in \Sigma \qquad \Gamma;\Xi \vdash S : A > \{\Delta_1\}\Sigma^*\{\Delta_2\}}{\Gamma;\Xi \vdash x \cdot S : \{\Delta_1\}\Sigma^*\{\Delta_2\}}$$

Figure 2: Typing rules for terms and traces in Meta-CLF

Although we do not treat implicit argument inference in this paper, we expect that we can infer implicit arguments by extending the LF type reconstruction algorithm. As we see from the case above, implicit arguments greatly increase the usability of the system, allowing to write more concise and clear proofs.

The other two cases of the proof, corresponding to $\mathsf{step/app}$ and $\mathsf{step/beta}$, are given below:

$$\begin{array}{l}\mathsf{tpres/app} : \mathsf{tpres}\ (X_1; \{\downarrow x\}{\leftarrow}\mathsf{gen/eval}\ (\mathsf{app}\ e_1\ e_2)\ d_0\ g_0\ H)\\ \qquad (\{!d_1, !d_2, \downarrow x_1, \downarrow x_2, \downarrow f\}{\leftarrow}\mathsf{step/app}\ e_1\ e_2\ d_0\ x)\\ \qquad (X_1; \{!d_1, !d_2, \downarrow f, \downarrow g_1, \downarrow g_2\}{\leftarrow}\mathsf{gen/fapp}\ d_0\ g_0;\\ \qquad\quad \{\downarrow x_1\}{\leftarrow}\mathsf{gen/eval}\ e_1\ d_1\ g_1\ H_1;\ \{\downarrow x_2\}{\leftarrow}\mathsf{gen/eval}\ e_2\ d_2\ g_2\ H_2)\end{array}$$

$$\begin{array}{l}\mathsf{tpres/beta} : \mathsf{tpres}\ (X_1; \{!d_1, !d_2, \downarrow f, \downarrow g_1, \downarrow g_2\}{\leftarrow}\mathsf{gen/fapp}\ d_0\ g_0;\\ \qquad\quad \{\downarrow x_1\}{\leftarrow}\mathsf{gen/ret}\ (\mathsf{lam}\ e_1)\ d_1\ g_1\ H_1\ H_{v_1};\ \{\downarrow x_2\}{\leftarrow}\mathsf{gen/ret}\ e_2\ d_2\ g_2\ H_2\ H_{v_2})\\ \qquad (\{\downarrow y\}{\leftarrow}\mathsf{step/beta}\ e_1\ e_2\ d_1\ d_2\ d\ x_1\ x_2\ f)\\ \qquad (X_1; \{\downarrow y\}{\leftarrow}\mathsf{gen/eval}\ (e_1\ e_2)\ d_0\ g_0\ H; \{!d_1\}{\leftarrow}\mathsf{gen/dest}; \{!d_2\}{\leftarrow}\mathsf{gen/dest})\end{array}$$

The latter case is the most interesting. By inversion, the generated state $\psi_1$ must be of the form

$$\begin{array}{lll}X_1; & \{!d_1, !d_2, \downarrow g_1, \downarrow g_2, \downarrow f\}{\leftarrow}\mathsf{gen/fapp}\ d_0\ g_0; & X_2;\\ & \{\downarrow x_1\}{\leftarrow}\mathsf{gen/ret}\ (\mathsf{lam}\ e_1)\ d_1'\ g_1'\ H_1\ H_{v_1}; & X_3;\\ & \{\downarrow x_2\}{\leftarrow}\mathsf{gen/ret}\ e_2\ d_2'\ g_2'\ H_2\ H_{v_2}; & X_4\end{array}$$

for some traces $X_1$, $X_2$, $X_3$, $X_4$. It must be that $d_1 = d_1'$ and $d_2 = d_2'$, and also $g_1 = g_1'$ and $g_2 = g_2'$. Note that $X_2$, $X_3$, and $X_4$ cannot use $g_1$ and $g_2$, so the trace can be reordered as

$$\begin{array}{ll}X_1; X_2; X_3; X_4; & \{!d_1, !d_2, \downarrow g_1, \downarrow g_2, \downarrow f\}{\leftarrow}\mathsf{gen/fapp}\ d_0\ g_0;\\ & \{\downarrow x_1\}{\leftarrow}\mathsf{gen/ret}\ (\mathsf{lam}\ e_1)\ d_1\ g_1\ H_1\ H_{v_1};\\ & \{\downarrow x_2\}{\leftarrow}\mathsf{gen/ret}\ e_2\ d_2\ g_2\ H_2\ H_{v_2})\end{array}$$

and $X_1$, $X_2$, $X_3$, and $X_4$ can be collapsed into one trace variable. In the generated trace after the rewriting step, we simply replace the generation of $\mathsf{fapp}$ and the two $\mathsf{ret}$s with a single $\mathsf{eval}$ fact. We also need two $\mathsf{gen/dest}$ steps for the destinations $d_1$ and $d_2$ which are not used anymore.

For proving progress, we first need to define a sum type that encodes the result: either we are at a final state or we can take a step.

$$\begin{array}{rl}\mathsf{result} : & \mathsf{ctx} \to \mathsf{type}\\ \mathsf{res/final} : & \{!d{:}\mathsf{dest}, \downarrow x{:}\mathsf{gen}\ d\ t\}\,\Sigma^*_{\mathsf{gen}}\,\{\psi, !d{:}\mathsf{dest}, \downarrow x{:}\mathsf{ret}\ e\ d\} \to \mathsf{result}\ (\psi, \downarrow x{:}\mathsf{ret}\ e\ d)\\ \mathsf{res/step} : & \{\psi_1\}\,\Sigma^1_{\mathsf{step}}\,\{\psi_2\} \to \mathsf{result}\ \psi_1\end{array}$$

Note in $\mathsf{res/final}$ that the generated trace has only one $\mathsf{ret}$ fact containing the final value at destination $d$, while $\psi$ contains only destinations (obtained from $\mathsf{gen/dest}$ steps) that are not used anymore. These are destinations that were generated during the evaluation of an expression by the $\mathsf{step/app}$ rule.

The progress theorem relates a well-typed state with a result:

$$\mathsf{progress} : \nabla d.\nabla g.\widehat{\Pi} t : \mathsf{tp}.\Pi\psi : \mathsf{ctx}.\{!d : \mathsf{dest}, \downarrow g : \mathsf{gen}\ d\ t\}\,\Sigma^*_{\mathsf{gen}}\,\{\psi\} \to \mathsf{result}\ \psi \to \mathsf{type}$$

We proceed by case analysis on the first step of the trace. If it is $\mathsf{gen/ret}$, then we are at a final state:

$$\mathsf{p/ret} : \mathsf{progress}\ (\{\downarrow x\}{\leftarrow}\mathsf{gen/ret}\ e\ d\ g\ H\ H_v; X)\ (\mathsf{res/final}\ (\{\downarrow x\}{\leftarrow}\mathsf{gen/ret}\ e\ d\ g\ H\ H_v; X))$$

If the trace starts with a gen/eval step, then we can make a step depending on which expression is generated. If the expression generated is an abstraction, we can make a step using step/eval since abstractions are values. If the expression generated is an application, we can make a step using step/app.

$$\begin{aligned}
\mathsf{p/ev\text{-}lam} : \mathsf{progress}\ & (\{\downarrow x\}\leftarrow\mathsf{gen/eval}\ (\mathsf{lam}\ e)\ d\ H; X) \\
& (\mathsf{res/step}\ (\{\downarrow x\}\leftarrow\mathsf{step/eval}\ (\mathsf{lam}\ e)\ d\ x\ (\mathsf{value/lam}\ e))) \\
\mathsf{p/ev\text{-}app} : \mathsf{progress}\ & (\{\downarrow x\}\leftarrow\mathsf{gen/eval}\ (\mathsf{app}\ e_1\ e_2)\ d\ H; X) \\
& (\mathsf{res/step}\ (\{!d_1, !d_2, \downarrow x_1, \downarrow x_2, \downarrow f\}\leftarrow\mathsf{step/app}\ e_1\ e_2\ x\ H))
\end{aligned}$$

Finally, we consider the case where the first step is gen/fapp. The generated trace has the form:

$$\{!d_1, !d_2, \downarrow f, \downarrow g_1, \downarrow g_2\}\leftarrow\mathsf{gen/fapp}\ d\ g; X_1\ d_1\ g_1; X_2\ d_2\ g_2$$

where the traces $X_1$ and $X_2$ generate trees rooted at $d_1$ and $d_2$ respectively. Note that, because $\Sigma_{\mathsf{gen}}$ is a generative grammar, these traces are independent.

We have three subcases: either $X_1$ and $X_2$ generate final states (in which case we can make a step using step/beta, or we can make a step in either $X_1$ or $X_2$:

$$\begin{aligned}
\mathsf{p/fapp1} : \mathsf{progress}\ & (\{!d_1, !d_2, \downarrow f, \downarrow g_1, \downarrow g_2\}\leftarrow\mathsf{gen/fapp}\ d\ g; \\
& \{\downarrow x_1\}\leftarrow\mathsf{gen/ret}\ (\mathsf{lam}\ e_1)\ d_1\ g_1\ H_1\ H_{v_1}; \\
& \{\downarrow x_2\}\leftarrow\mathsf{gen/ret}\ e_2\ d_2\ g_2\ H_2\ H_{v_2}) \\
& (\mathsf{res/step}\ (\{\downarrow y\}\leftarrow\mathsf{step/beta}\ e_1\ e_2\ d_1\ d_2\ d\ x_1\ x_2\ f))
\end{aligned}$$

$$\begin{aligned}
\mathsf{p/fapp2} : \mathsf{progress}\ & (\{!d_1, !d_2, \downarrow f, \downarrow g_1, \downarrow g_2\}\leftarrow\mathsf{gen/fapp}\ d\ g; X_1\ d_1\ g_1; X_2\ d_2\ g_2) \\
& (\mathsf{res/step}\ z) \\
\leftarrow \mathsf{progress}\ & (X_1\ d_1\ g_1)\ (\mathsf{res/step}\ z)
\end{aligned}$$

$$\begin{aligned}
\mathsf{p/fapp3} : \mathsf{progress}\ & (\{!d_1, !d_2, \downarrow f, \downarrow g_1, \downarrow g_2\}\leftarrow\mathsf{gen/fapp}\ d\ g; X_1\ d_1\ g_1; X_2\ d_2\ g_2) \\
& (\mathsf{res/step}\ z) \\
\leftarrow \mathsf{progress}\ & (X_2\ d_2\ g_1)\ (\mathsf{res/step}\ z)
\end{aligned}$$

**Totality.** We showed how to encode, in Meta-CLF, proofs of safety for a small programming language with a parallel semantics. A natural question is: are these valid proofs? This amounts to check totality, which is the conjunction of two properties: coverage (i.e., all cases are considered) and termination. While we do not give a formal treatment of totality for Meta-CLF (which we leave for future work), we can show, informally, that both proofs given above are total.

It is easy to see that both proofs are terminating: in the case of preservation, the proof is not recursive, while for progress, recursive calls are performed on smaller traces.

Checking coverage is trickier, since it encompasses checking coverage for traces, which is a difficult problem because trace equality allows permuting steps.

In our proof of preservation, coverage checking manifests itself in the use of inversion: for each case of the step relation from context $\psi_1$ to $\psi_2$ we apply inversion to obtain a pattern that covers the generation of $\psi_1$. As we explained above, we only need one pattern for each case.

Note that coverage in the proof of preservation depends on the fact that $\Sigma_{\mathsf{gen}}$ is a generative grammar. In particular, that terminal symbols are not removed from the context once they are produced, and that fresh destinations are created for each non-terminal (this property is used in step/beta case).

In the proof of progress, coverage checking is directly performed over the generated trace. The interesting case is gen/fapp, since this case involves splitting the trace between the trees generated for each of the children of fapp. Again, this is possible because each non-terminal is associated with a fresh destination, so generated traces from starting from different non-terminals are independent.

While coverage checking for traces in general is a difficult problem, by restricting to traces generated using grammars, we expect to obtain a relatively simple algorithm to solve this problem.

It is important to note that these proofs would be similar if we consider a sequential semantics. Viewed in a different direction, having a parallel semantics does not change the proof with respect to the sequential case. The reason is the use of SSOS and trace equality, and the same holds true when considering other programming constructions [14]. The burden of the proof is shifted to the coverage checker. However, the use of generative grammars makes it possible to automate coverage checking.

## 4 Related Work

The original reports that introduced CLF [4, 15] include many applications, including SSOS of several programming languages features. However, no meta-reasoning is developed.

Attempts to perform meta-reasoning within CLF have proved to be unsatisfactory. Watkins et al. [17] define an encoding of the $\pi$-calculus and correspondence assertions for it in CLF. They define an abstraction relation that relates a concurrent computation with a sequence of events. However, due to lack of trace types, it is not possible to state the abstraction relation. This means also that coverage is difficult to establish. Schack-Nielsen [13] discusses the limitations of CLF to prove the equivalence between small-step and big-step semantics of MiniML.

Simmons [14] introduced the notion of generative grammar that generalizes both context-free grammars and regular worlds used in LF. He describes in detail the use generative grammars and SSOS. However, the proofs of safety are done only "on paper" since his framework is not expressive enough for this task. This work is an attempt to define a logical framework to carry out the proofs described in [14].

Finally, let us mention our previous work on matching traces in CLF [3] which provides the basis for defining a moded operational semantics for Meta-CLF. We expect to strengthen the results given in [3] by restricting the matching problem to traces produced using generative grammars.

## 5 Conclusions

We have developed a logical framework for meta-reasoning about specifications written in CLF. We show a typical use of this framework by proving type preservation and progress for a small programming language with a parallel semantics. Trace equality simplifies the proof as we do not have to worry about proving properties about step interleavings during the parallel execution. However, the downside of this approach is that coverage checking is more complicated than in the sequential case.

For future work, an immediate objective is to complete the meta-theoretical study of Meta-CLF itself. This involves proving the existence of canonical forms, type reconstruction of implicit arguments, and totality checking (coverage and termination). Then, of course, implementation is another obvious objective.

This framework is well suited for the kind of proofs of safety we developed in this paper and we expect it to perform well for other concurrent and parallel programming constructions (e.g., futures, communication). It will be interesting to see in what other domain we can use it. For example, semantics of relaxed memory models, or correctness of program transformations in the presence of threads.

## References

[1] Iliano Cervesato & Frank Pfenning (2002): *A Linear Logical Framework*. *Information & Computation* 179(1), pp. 19–75, doi:10.1006/inco.2001.2951.

[2] Iliano Cervesato & Frank Pfenning (2003): *A Linear Spine Calculus*. *Journal of Logic and Computation* 13(5), pp. 639–688, doi:10.1093/logcom/13.5.639.

[3] Iliano Cervesato, Frank Pfenning, Jorge Luis Sacchini, Carsten Schürmann & Robert J. Simmons (2012): *Trace Matching in a Concurrent Logical Framework*. In Adam Chlipala & Carsten Schürmann, editors: *7th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice — LFMTP'12*, Copenhagen, Denmark, doi:10.1145/2364406.2364408.

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

[5] Matt Fairtlough & Michael Mendler (1997): *Propositional Lax Logic*. *Information and Computation* 137(1), pp. 1–33, doi:10.1006/inco.1997.2627.

[6] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A framework for defining logics*. *Journal of the ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

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

[8] Dale Miller & Alwen Fernanto Tiu (2003): *A Proof Theory for Generic Judgments: An extended abstract*. In: *LICS*, IEEE Computer Society, pp. 118–127, doi:10.1109/LICS.2003.1210051.

[9] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, Springer, doi:10.1007/3-540-45949-9.

[10] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology.

[11] Frank Pfenning (2004): *Substructural Operational Semantics and Linear Destination-Passing Style (Invited Talk)*. In Wei-Ngan Chin, editor: *APLAS*, *LNCS* 3302, PUB-SP, p. 196, doi:10.1007/978-3-540-30477-7_13.

[12] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf - A Meta-Logical Framework for Deductive Systems*. In Harald Ganzinger, editor: *CADE*, *LNCS* 1632, Springer, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[13] Anders Schack-Nielsen (2011): *Implementing Substructural Logical Frameworks*. Ph.D. thesis, IT University of Copenhagen.

[14] Robert J. Simmons (2012): *Substructural Logical Specifications*. Ph.D. thesis, Carnegie Mellon University.

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

[16] Kevin Watkins, Iliano Cervesato, Frank Pfenning & David Walker (2004): *A Concurrent Logical Framework: The Propositional Fragment*. In Stefano Berardi, Mario Coppo & Ferruccio Damiani, editors: *TYPES*, *LNCS* 3085, PUB-SV, pp. 355–377, doi:10.1007/978-3-540-24849-1_23.

[17] Kevin Watkins, Iliano Cervesato, Frank Pfenning & David Walker (2008): *Specifying Properties of Concurrent Computations in CLF*. *ENTCS* 199, pp. 67–87, doi:10.1016/j.entcs.2007.11.013.