

# Meta-Reasoning in a Concurrent Logical Framework

Iliano Cervesato and Jorge Luis Sacchini

Carnegie Mellon University in Qatar

LIX Colloquium on  
Theory and Application of Formal Proofs  
7 Nov 2013

# Objectives

- Concurrency and distribution are essential features in modern systems.
- PLs are engineered (or retrofitted) to support them.
- Their formal semantics is not as well understood or studied as in the sequential case.
- Formal semantics will enable, e.g.,
  - ▶ formal verification,
  - ▶ static analyses,
  - ▶ verifying program transformations.

# Logical frameworks

- Formalisms used to specify PL and their metatheory.
  - ▶ Coq, Agda, Twelf, Beluga, Delphin, ...
- Our goal is to develop logical frameworks for specifying concurrent and distributed PL.
- Deep approach: specify a concurrency model in a general purpose LF (Coq, Agda)
- Shallow approach: provide direct support in a special purpose LF (Twelf, Beluga, Delphin, LLF, HLF, CLF, Abella)
- We follow the shallow approach, using CLF as our LF.

# CLF

- CLF is an extension of the Edinburgh logical framework (LF) designed to specify distributed and concurrent systems.
- Large number of examples: semantics of PL, Petri nets, voting protocols, etc.
- CLF extends LF with linear types and a monad to encapsulate concurrent effects:

$$\begin{array}{ll} K ::= \text{type} \mid \prod !x : A. K & \text{(Kinds)} \\ A ::= P \mid \prod x : A. B \mid A \rightarrow B \mid A \multimap B \mid \{S\} & \text{(Async types)} \\ S ::= 1 \mid !A \mid A \mid S \otimes S \mid \exists x : A. S & \text{(Sync types)} \end{array}$$

as well as proof terms for these types (more on that later)

# CLF

## Synchronous fragment

- Monadic types are used to encapsulate concurrent effects:

$$A \multimap B \multimap \{C\}$$

- (Multiset) Rewriting interpretation of linear logic.
- Example:  $\text{in} : \text{tick} \multimap \text{count } N \multimap \{\text{count } (N + 1)\}$

- Monadic types are used to encapsulate concurrent effects:

$$A \multimap B \multimap \{C\}$$

- (Multiset) Rewriting interpretation of linear logic.
- Example:  $\text{in} : \text{tick} \multimap \text{count } N \multimap \{\text{count } (N + 1)\}$

$t_1 : \text{tick}, t_2 : \text{tick}, t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 0, c_2 : \text{count } 0$

# CLF

## Synchronous fragment

- Monadic types are used to encapsulate concurrent effects:

$$A \multimap B \multimap \{C\}$$

- (Multiset) Rewriting interpretation of linear logic.
- Example:  $\text{in} : \text{tick} \multimap \text{count } N \multimap \{\text{count } (N + 1)\}$

$$\begin{aligned} t_1 : \text{tick}, t_2 : \text{tick}, t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 0, c_2 : \text{count } 0 \\ \rightsquigarrow t_2 : \text{tick}, t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 1, c_2 : \text{count } 0 \end{aligned}$$

- Monadic types are used to encapsulate concurrent effects:

$$A \multimap B \multimap \{C\}$$

- (Multiset) Rewriting interpretation of linear logic.
- Example:  $\text{in} : \text{tick} \multimap \text{count } N \multimap \{\text{count } (N + 1)\}$

$$\begin{aligned} t_1 : \text{tick}, t_2 : \text{tick}, t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 0, c_2 : \text{count } 0 \\ \rightsquigarrow t_2 : \text{tick}, t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 1, c_2 : \text{count } 0 \\ \rightsquigarrow t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 2, c_2 : \text{count } 0 \end{aligned}$$



# CLF

## Synchronous fragment

- Monadic types are used to encapsulate concurrent effects:

$$A \multimap B \multimap \{C\}$$

- (Multiset) Rewriting interpretation of linear logic.
- Example:  $\text{in} : \text{tick} \multimap \text{count } N \multimap \{\text{count } (N + 1)\}$

$$\begin{aligned} t_1 : \text{tick}, t_2 : \text{tick}, t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 0, c_2 : \text{count } 0 \\ \rightsquigarrow t_2 : \text{tick}, t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 1, c_2 : \text{count } 0 \\ \rightsquigarrow t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 2, c_2 : \text{count } 0 \\ \rightsquigarrow t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 2, c_2 : \text{count } 1 \end{aligned}$$

- Monadic types are used to encapsulate concurrent effects:

$$A \multimap B \multimap \{C\}$$

- (Multiset) Rewriting interpretation of linear logic.
- Example:  $\text{in} : \text{tick} \multimap \text{count } N \multimap \{\text{count } (N + 1)\}$

$$\begin{aligned}
 & t_1 : \text{tick}, t_2 : \text{tick}, t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 0, c_2 : \text{count } 0 \\
 & \rightsquigarrow t_2 : \text{tick}, t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 1, c_2 : \text{count } 0 \\
 & \rightsquigarrow t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 2, c_2 : \text{count } 0 \\
 & \rightsquigarrow t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 2, c_2 : \text{count } 1 \\
 & \rightsquigarrow t_5 : \text{tick}, c_1 : \text{count } 3, c_2 : \text{count } 1
 \end{aligned}$$

- Monadic types are used to encapsulate concurrent effects:

$$A \multimap B \multimap \{C\}$$

- (Multiset) Rewriting interpretation of linear logic.
- Example:  $\text{in} : \text{tick} \multimap \text{count } N \multimap \{\text{count } (N + 1)\}$

$$\begin{aligned}
 & t_1 : \text{tick}, t_2 : \text{tick}, t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 0, c_2 : \text{count } 0 \\
 & \rightsquigarrow t_2 : \text{tick}, t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 1, c_2 : \text{count } 0 \\
 & \rightsquigarrow t_3 : \text{tick}, t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 2, c_2 : \text{count } 0 \\
 & \rightsquigarrow t_4 : \text{tick}, t_5 : \text{tick}, c_1 : \text{count } 2, c_2 : \text{count } 1 \\
 & \rightsquigarrow t_5 : \text{tick}, c_1 : \text{count } 3, c_2 : \text{count } 1 \\
 & \rightsquigarrow c_1 : \text{count } 3, c_2 : \text{count } 2
 \end{aligned}$$

# Proof terms

- Monadic types are introduced by **traces**
- A trace is basically a sequence of rule applications:

$$\varepsilon ::= \diamond \mid \{\Delta\} \leftarrow c \cdot S \mid \varepsilon_1; \varepsilon_2$$

- Trace composition (;) is associative and  $\diamond$  is a neutral element
- Forward chaining with committed choice after every step.

# Traces

- Typing rules for traces:

$$\Delta \vdash \varepsilon : \Delta'$$

can be read as “*trace  $\varepsilon$  transforms (rewrites)  $\Delta$  into  $\Delta'$* ”

$$\frac{}{\Delta \vdash \diamond : \Delta} \quad \frac{c : \prod \Delta_0. \{\Delta_1\} \in \Sigma}{\Delta \bowtie \Delta_0 \vdash \{\Delta_1\} \leftarrow c \cdot \Delta_0 : \Delta \bowtie \Delta_1}$$
$$\frac{\Delta_0 \vdash \varepsilon_1 : \Delta_1 \quad \Delta_1 \vdash \varepsilon_2 : \Delta_2}{\Delta_0 \vdash \varepsilon_1; \varepsilon_2 : \Delta_2}$$

- $\Delta = \Delta_0 \bowtie \Delta_1$  iff
  - ▶ Persistent declarations in  $\Delta$  appear in both  $\Delta_0$  and  $\Delta_1$
  - ▶ Linear declarations in  $\Delta$  appear in exactly one of  $\Delta_0$  and  $\Delta_1$

# Traces

- Equality on traces:  $\alpha$ -equivalence modulo permutation of **independent** subtraces.
- Allows encoding of concurrent and distributed features.
- Two traces are independent ( $\varepsilon_1 \parallel \varepsilon_2$ ) if they operate on different sets of variables.
- Trace interface:

$$\begin{array}{l|l} \bullet(\diamond) = \emptyset & (\diamond)\bullet = \emptyset \\ \bullet(\{\Delta\} \leftarrow_c \cdot S) = \text{FV}(S) & (\{\Delta\} \leftarrow_c \cdot S)\bullet = \text{dom}(\Delta) \\ \bullet(\varepsilon_1; \varepsilon_2) = \bullet\varepsilon_1 \cup (\bullet\varepsilon_2 \setminus \varepsilon_1\bullet) & (\varepsilon_1; \varepsilon_2)\bullet = \varepsilon_2\bullet \cup (\varepsilon_1\bullet \setminus \bullet\varepsilon_2) \cup !(\varepsilon_1\bullet) \end{array}$$

- $\varepsilon_1 \parallel \varepsilon_2 \iff \bullet\varepsilon_1 \cap \varepsilon_2\bullet = \varepsilon_1\bullet \cap \bullet\varepsilon_2 = \emptyset.$

# Traces

- Trace equality:

$$\begin{array}{c} \overline{\varepsilon; \diamond} \equiv \varepsilon \\ \overline{\varepsilon} \equiv \varepsilon; \diamond \\ \overline{\varepsilon_1; (\varepsilon_2; \varepsilon_3)} \equiv (\varepsilon_1; \varepsilon_2); \varepsilon_3 \\ \frac{\varepsilon_1 \parallel \varepsilon_2}{\varepsilon_1; \varepsilon_2 \equiv \varepsilon_2; \varepsilon_1} \\ \frac{\varepsilon_1 \equiv \varepsilon'_1}{\varepsilon_1; \varepsilon_2 \equiv \varepsilon'_1; \varepsilon_2} \\ \frac{\varepsilon_2 \equiv \varepsilon'_2}{\varepsilon_1; \varepsilon_2 \equiv \varepsilon_1; \varepsilon'_2} \end{array}$$

- Example:

$$\begin{array}{l} \{c_{11}\} \leftarrow \text{in} \cdot t_1, c_1 \\ \{c_{12}\} \leftarrow \text{in} \cdot t_2, c_{11} \\ \{c_{21}\} \leftarrow \text{in} \cdot t_3, c_2 \\ \{c_{13}\} \leftarrow \text{in} \cdot t_4, c_{12} \\ \{c_{22}\} \leftarrow \text{in} \cdot t_5, c_{21} \end{array} \equiv \begin{array}{l} \{c_{11}\} \leftarrow \text{in} \cdot t_1, c_1 \\ \{c_{21}\} \leftarrow \text{in} \cdot t_3, c_2 \\ \{c_{12}\} \leftarrow \text{in} \cdot t_2, c_{11} \\ \{c_{13}\} \leftarrow \text{in} \cdot t_4, c_{12} \\ \{c_{22}\} \leftarrow \text{in} \cdot t_5, c_{21} \end{array}$$

# Substructural operational semantics

- Substructural operational semantics combines
  - ▶ Structural operational semantics
  - ▶ Substructural logics
- Extensible: we can add features without breaking previous developments
- Expressive: wide variety of concurrent and distributed mechanisms (Simmons12).



# Example

- Simply-typed  $\lambda$ -calculus

$$e ::= x \mid \lambda x.e \mid e e$$

- HOAS representation in (C)LF:

$\text{exp} : \text{type}.$

$\text{lam} : (\text{exp} \rightarrow \text{exp}) \rightarrow \text{exp}.$

$\text{app} : \text{exp} \rightarrow \text{exp} \rightarrow \text{exp}.$

- Linear-destination passing style (Pfenning04)
- Based on multiset rewriting; suitable for specifying in linear logic
- Multiset of facts:
  - $\text{eval } e \ d$  Evaluate expression  $e$  in destination  $d$
  - $\text{ret } e \ d$  Return value  $e$  to destination  $d$
  - $\text{fapp } d_1 \ d_2 \ d$  Application: expects the function and argument to be evaluated in  $d_1$  and  $d_2$ , and the result is evaluated in  $d$
- Evaluation rules transform multisets of facts

- Expressions represented as multiset of facts:

$$\text{eval } e \ d, \quad \text{ret } e \ d, \quad \text{fapp } d_1 \ d_2 \ d$$

- In CLF:

$\text{dest} : \text{type}.$

$\text{eval} : \text{exp} \rightarrow \text{dest} \rightarrow \text{type}.$

$\text{ret} : \text{exp} \rightarrow \text{dest} \rightarrow \text{type}.$

$\text{fapp} : \text{dest} \rightarrow \text{dest} \rightarrow \text{dest} \rightarrow \text{type}.$

# Parallel evaluation semantics

- Transitions as multiset rewriting rules:

$$\text{eval } e \ d \rightsquigarrow \text{ret } e \ d \quad \text{if } e \text{ is a value}$$

- In CLF:

$$\text{step/eval} : \text{eval } e \ d \multimap \text{value } e \rightarrow \{\text{ret } e \ d\}.$$

# Parallel evaluation semantics

- Transitions as multiset rewriting rules:

$$\text{eval } (e_1 \ e_2) \ d \rightsquigarrow \text{eval } e_1 \ d_1, \text{eval } e_2 \ d_2, \text{fapp } d_1 \ d_2 \ d$$

where  $d_1, d_2$  fresh

- In CLF:

$$\text{step/app: eval } (\text{app } e_1 \ e_2) \ d \multimap \{ !d_1 \ !d_2 : \text{dest}, \\ \text{eval } e_1 \ d_1, \\ \text{eval } e_2 \ d_2, \\ \text{fapp } d_1 \ d_2 \ d \}.$$

## Parallel evaluation semantics

- Transitions as multiset rewriting rules:

$$\text{ret } (\lambda x. e_1) d_1, \text{ret } e_2 d_2, \text{fapp } d_1 d_2 d \rightsquigarrow \text{eval } (e_1[e_2/x]) d$$

- In CLF:

$$\begin{aligned} \text{step/beta} : \text{ret } (\text{lam } e_1) d_1 &\multimap \\ &\text{ret } e_2 d_2 \multimap \\ &\text{fapp } d_1 d_2 d \multimap \{ \text{eval } (e_1 e_2) d \} \end{aligned}$$

## Example

$\text{eval } ((\lambda x.x)(\lambda y.y)) d$

In CLF:

$!d : \text{dest}, x_0 : \text{eval } (\text{app } (\text{lam } \lambda x.x) (\text{lam } \lambda y.y)) d$   
 $\vdash \diamond$

$: !d : \text{dest}, x_0 : \text{eval } (\text{app } (\text{lam } \lambda x.x) (\text{lam } \lambda y.y)) d$

## Example

$\text{eval } ((\lambda x.x)(\lambda y.y)) d \rightsquigarrow \text{eval } (\lambda x.x) d_1, \text{eval } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d$

In CLF:

$!d : \text{dest}, x_0 : \text{eval } (\text{app } (\text{lam } \lambda x.x) (\text{lam } \lambda y.y)) d$   
 $\vdash \{!d_1, !d_2, x, y, z\} \leftarrow \text{step/app } x_0;$

$: !d !d_1 !d_2 : \text{dest}, x : \text{eval } (\text{lam } \lambda x.x) d_1, y : \text{eval } (\text{lam } \lambda y.y) d_2,$   
 $z : \text{fapp } d_1 d_2 d$



## Example

$$\begin{aligned} \text{eval } ((\lambda x.x)(\lambda y.y)) d &\rightsquigarrow \text{eval } (\lambda x.x) d_1, \text{eval } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \\ &\rightsquigarrow \text{ret } (\lambda x.x) d_1, \text{eval } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \end{aligned}$$

In CLF:

$$\begin{aligned} !d : \text{dest}, x_0 : \text{eval } (\text{app } (\text{lam } \lambda x.x) (\text{lam } \lambda y.y)) d \\ \vdash \{!d_1, !d_2, x, y, z\} \leftarrow \text{step/app } x_0; \\ \quad \{x'\} \leftarrow \text{step/eval } x; \end{aligned}$$
$$\begin{aligned} : !d !d_1 !d_2 : \text{dest}, x' : \text{ret } (\text{lam } \lambda x.x) d_1, y : \text{eval } (\text{lam } \lambda y.y) d_2 \\ z : \text{fapp } d_1 d_2 d \end{aligned}$$

## Example

$$\begin{aligned} \text{eval } ((\lambda x.x)(\lambda y.y)) d &\rightsquigarrow \text{eval } (\lambda x.x) d_1, \text{eval } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \\ &\rightsquigarrow \text{ret } (\lambda x.x) d_1, \text{eval } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \\ &\rightsquigarrow \text{ret } (\lambda x.x) d_1, \text{ret } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \end{aligned}$$

In CLF:

$$\begin{aligned} !d : \text{dest}, x_0 : \text{eval } (\text{app } (\text{lam } \lambda x.x) (\text{lam } \lambda y.y)) d \\ \vdash \{!d_1, !d_2, x, y, z\} \leftarrow \text{step/app } x_0; \\ \quad \{x'\} \leftarrow \text{step/eval } x; \\ \quad \{y'\} \leftarrow \text{step/eval } y; \end{aligned}$$
$$\begin{aligned} : !d !d_1 !d_2 : \text{dest}, x' : \text{ret } (\text{lam } \lambda x.x) d_1, y' : \text{ret } (\text{lam } \lambda y.y) d_2 \\ z : \text{fapp } d_1 d_2 d \end{aligned}$$

## Example

$$\begin{aligned} \text{eval } ((\lambda x.x)(\lambda y.y)) d &\rightsquigarrow \text{eval } (\lambda x.x) d_1, \text{eval } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \\ &\rightsquigarrow \text{ret } (\lambda x.x) d_1, \text{eval } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \\ &\rightsquigarrow \text{ret } (\lambda x.x) d_1, \text{ret } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \\ &\rightsquigarrow \text{eval } (\lambda y.y) d \end{aligned}$$

In CLF:

$$\begin{aligned} !d : \text{dest}, x_0 : \text{eval } (\text{app } (\text{lam } \lambda x.x) (\text{lam } \lambda y.y)) d \\ \vdash \{!d_1, !d_2, x, y, z\} \leftarrow \text{step/app } x_0; \\ \quad \{x'\} \leftarrow \text{step/eval } x; \\ \quad \{y'\} \leftarrow \text{step/eval } y; \\ \quad \{w\} \leftarrow \text{step/beta } x' y' z; \end{aligned}$$
$$: !d !d_1 !d_2 : \text{dest}, w : \text{eval } (\text{lam } \lambda y.y) d$$

## Example

$$\begin{aligned} \text{eval } ((\lambda x.x)(\lambda y.y)) d &\rightsquigarrow \text{eval } (\lambda x.x) d_1, \text{eval } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \\ &\rightsquigarrow \text{ret } (\lambda x.x) d_1, \text{eval } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \\ &\rightsquigarrow \text{ret } (\lambda x.x) d_1, \text{ret } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \\ &\rightsquigarrow \text{eval } (\lambda y.y) d \\ &\rightsquigarrow \text{ret } (\lambda y.y) d \end{aligned}$$

In CLF:

$$\begin{aligned} !d : \text{dest}, x_0 : \text{eval } (\text{app } (\text{lam } \lambda x.x) (\text{lam } \lambda y.y)) d \\ \vdash \{!d_1, !d_2, x, y, z\} \leftarrow \text{step/app } x_0; \\ \quad \{x'\} \leftarrow \text{step/eval } x; \\ \quad \{y'\} \leftarrow \text{step/eval } y; \\ \quad \{w\} \leftarrow \text{step/beta } x' y' z; \\ \quad \{w'\} \leftarrow \text{step/eval } w; \\ : !d !d_1 !d_2 : \text{dest}, w' : \text{ret } (\text{lam } \lambda y.y) d \end{aligned}$$

## Example

$$\begin{aligned} \text{eval } ((\lambda x.x)(\lambda y.y)) d &\rightsquigarrow \text{eval } (\lambda x.x) d_1, \text{eval } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \\ &\rightsquigarrow \text{ret } (\lambda x.x) d_1, \text{eval } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \\ &\rightsquigarrow \text{ret } (\lambda x.x) d_1, \text{ret } (\lambda y.y) d_2, \text{fapp } d_1 d_2 d \\ &\rightsquigarrow \text{eval } (\lambda y.y) d \\ &\rightsquigarrow \text{ret } (\lambda y.y) d \end{aligned}$$

In CLF:

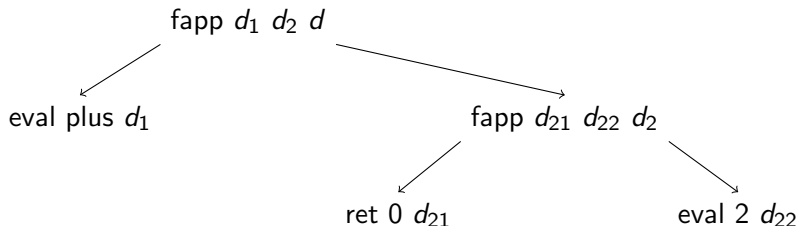
$$\begin{aligned} !d : \text{dest}, x_0 : \text{eval } (\text{app } (\text{lam } \lambda x.x) (\text{lam } \lambda y.y)) d \\ \vdash \{!d_1, !d_2, x, y, z\} \leftarrow \text{step/app } x_0; \\ \quad \{y'\} \leftarrow \text{step/eval } y; \\ \quad \{x'\} \leftarrow \text{step/eval } x; \\ \quad \{w\} \leftarrow \text{step/beta } x' y' z; \\ \quad \{w'\} \leftarrow \text{step/eval } w; \\ : !d !d_1 !d_2 : \text{dest}, w' : \text{ret } (\text{lam } \lambda y.y) d \end{aligned}$$

# Safety

- Recall that the previous semantics is parallel (more complex languages can have concurrent and distributed semantics as well).
- Safety.
  - ▶ Preservation: evaluation preserves well-typed multisets.  
*If  $\Delta$  is a well-typed multiset and  $\Delta \rightsquigarrow \Delta'$  (step), then  $\Delta'$  is a well-typed multiset.*
  - ▶ Progress: a well-typed multiset is either final (result) or is possible to take a step.  
*If  $\Delta$  is a well-typed multiset, then either  $\Delta = \{\text{ret } e\ d\}$  or there exists  $\Delta'$  such that  $\Delta \rightsquigarrow \Delta'$ .*
- We need a notion of well-typed multiset for SSOS specifications.

## Well-formed multisets

- Example:  $\text{eval } e_1 \ d, \text{eval } e_2 \ d$  ✘ (repeated destination)  
 $\text{fapp } d_1 \ d_2 \ d, \text{eval } e_1 \ d_1$  ✘ (no fact for  $d_2$ )
- Well-formed multisets form a tree:



## Well-formed multisets

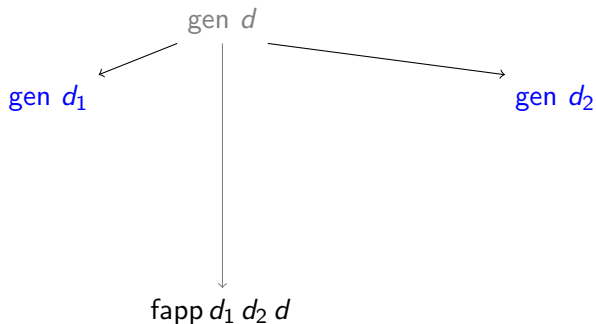
- Well-formed multisets can be described by rewriting rules.
- $\text{gen } t \ d$  means “generate a term of type  $t$  rooted at  $d$ ”

$\text{gen } d$



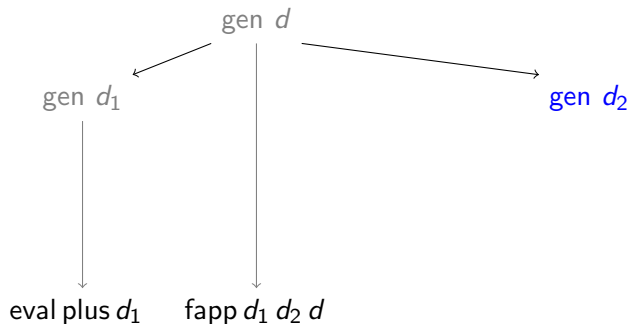
## Well-formed multisets

- Well-formed multisets can be described by rewriting rules.
- $\text{gen } t \ d$  means “generate a term of type  $t$  rooted at  $d$ ”



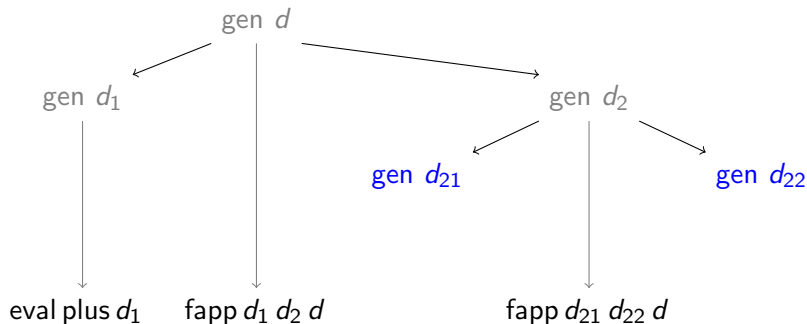
## Well-formed multisets

- Well-formed multisets can be described by rewriting rules.
- $\text{gen } t \ d$  means “generate a term of type  $t$  rooted at  $d$ ”



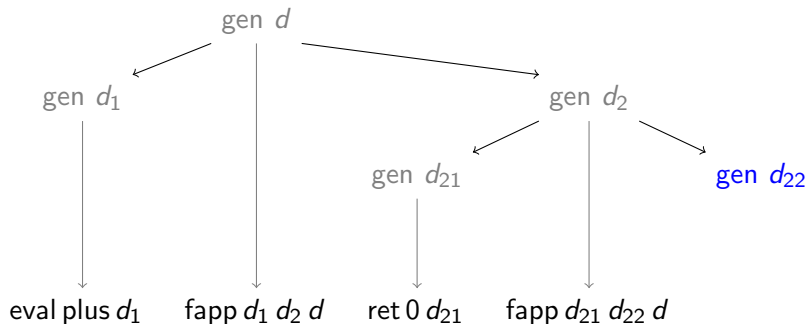
## Well-formed multisets

- Well-formed multisets can be described by rewriting rules.
- $\text{gen } t \ d$  means “generate a term of type  $t$  rooted at  $d$ ”



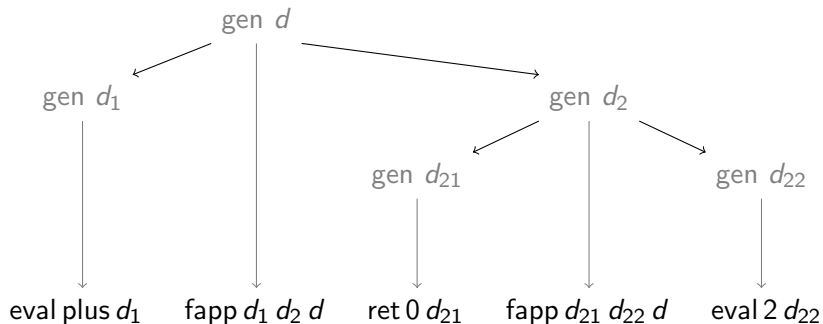
## Well-formed multisets

- Well-formed multisets can be described by rewriting rules.
- $\text{gen } t \ d$  means “generate a term of type  $t$  rooted at  $d$ ”



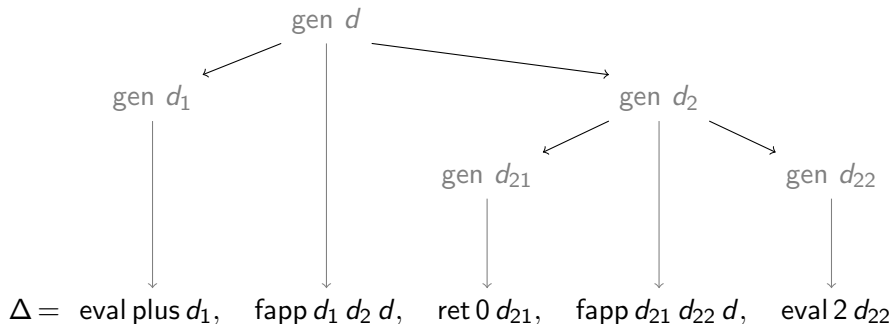
## Well-formed multisets

- Well-formed multisets can be described by rewriting rules.
- $\text{gen } t \ d$  means “generate a term of type  $t$  rooted at  $d$ ”



## Well-formed multisets

- Well-formed multisets can be described by rewriting rules.
- $\text{gen } t \ d$  means “generate a term of type  $t$  rooted at  $d$ ”



# Well-typed multisets

- Generating well-typed states:

$$\text{gen } t \ d \rightsquigarrow^* \mathcal{A}$$

where  $\mathcal{A}$  contains no fact of the form  $\text{gen } t_0 \ d_0$ .

- In CLF:

$\text{gen} \quad : \text{tp} \rightarrow \text{dest} \rightarrow \text{type}.$

$\text{gen/eval} : \text{gen } t \ d \multimap \text{of } e \ t \rightarrow \{\text{eval } e \ d\}.$

$\text{gen/ret} : \text{gen } t \ d \multimap \text{of } e \ t \rightarrow \text{value } e \rightarrow \{\text{ret } e \ d\}.$

$\text{gen/fapp} : \text{gen } t \ d \multimap \{!d_1 \ !d_2 : \text{dest},$

$\text{fapp } d_1 \ d_2 \ d,$

$\text{gen } (\text{arr } t_1 \ t) \ d_1,$

$\text{gen } t_1 \ d_2\}.$

- We call these type of rules *generative invariants* (Simmons 12).

# Safety

## Lemma (Safety)

**Preservation** *If  $\{\text{gen } t \ d\} \rightsquigarrow_{\text{gen}}^* \mathcal{A}$  and  $\mathcal{A} \rightsquigarrow_{\text{step}} \mathcal{A}'$  then  $\{\text{gen } t \ d\} \rightsquigarrow_{\text{gen}}^* \mathcal{A}'$ .*

**Progress** *if  $\{\text{gen } t \ d\} \rightsquigarrow_{\text{gen}}^* \mathcal{A}$ , then either  $\mathcal{A}$  is of the form  $\{\text{ret } e \ d\}$  or there exists  $\mathcal{A}'$  such that  $\mathcal{A} \rightsquigarrow_{\text{step}} \mathcal{A}'$ .*

## Proof.

**Preservation** The proof proceeds by case analysis on the evaluation step.

**Progress** The proof proceeds by induction on the generating trace.





# Limitations of CLF

- In CLF it is not possible to express preservation and progress.
- CLF lacks support for first-order traces, and quantification over contexts.
- We propose an extension of LF with trace types: Meta-CLF.
- Similar approaches are taken in Beluga, Delphin, Abella (in the sense of using a two-level approach).

# Meta-CLF

- Meta-CLF is an extension of LF with trace types and quantification over contexts and names:

$$A ::= \dots \mid \{\Delta\} \Sigma^* \{\Delta\} \mid \{\Delta\} \Sigma^1 \{\Delta\} \mid \Pi \psi : \text{ctx}.A \mid \nabla x.A$$

- $\{\Delta\} \Sigma^* \{\Delta'\}$  is the type of all traces  $\varepsilon$  satisfying  $\Delta \vdash \varepsilon : \Delta'$  that use only rules in the signature  $\Sigma$ .
- $\{\Delta\} \Sigma^1 \{\Delta'\}$  is the type of all 1-step traces  $\varepsilon$  satisfying  $\Delta \vdash \varepsilon : \Delta'$  that use only rules in the signature  $\Sigma$ .

- In Meta-CLF we can express properties about traces:

preservation :  $\prod t : \text{tp}. \nabla d. \nabla g. \prod \psi_1 : \text{ctx}. \prod \psi_2 : \text{ctx}.$

$$\{!d : \text{dest}, g : \text{gen } d \ t\} \Sigma_{\text{gen}}^* \{\psi_1\} \rightarrow \{\psi_1\} \Sigma_{\text{step}}^1 \{\psi_2\} \rightarrow \\ \{!d : \text{dest}, g : \text{gen } d \ t\} \Sigma_{\text{gen}}^* \{\psi_2\} \rightarrow \text{type}.$$

*“If  $\psi_1$  is a well-typed state (generated from a single gen  $d$ ) and there is a step from  $\psi_1$  to  $\psi_2$ , then  $\psi_2$  is a well-typed state.”*

# Meta-CLF

- In Meta-CLF we can express properties about traces:

preservation :  $\prod t : \text{tp}. \nabla d. \nabla g. \prod \psi_1 : \text{ctx}. \prod \psi_2 : \text{ctx}.$

$$\{!d : \text{dest}, g : \text{gen } d \ t\} \Sigma_{\text{gen}}^* \{\psi_1\} \rightarrow \{\psi_1\} \Sigma_{\text{step}}^1 \{\psi_2\} \rightarrow \\ \{!d : \text{dest}, g : \text{gen } d \ t\} \Sigma_{\text{gen}}^* \{\psi_2\} \rightarrow \text{type}.$$

*“If  $\psi_1$  is a well-typed state (generated from a single gen  $d$ ) and there is a step from  $\psi_1$  to  $\psi_2$ , then  $\psi_2$  is a well-typed state.”*

- In Meta-CLF we can express properties about traces:

preservation :  $\prod t : \text{tp}. \nabla d. \nabla g. \prod \psi_1 : \text{ctx}. \prod \psi_2 : \text{ctx}.$

$$\{!d : \text{dest}, g : \text{gen } d \ t\} \Sigma_{\text{gen}}^* \{\psi_1\} \rightarrow \{\psi_1\} \Sigma_{\text{step}}^1 \{\psi_2\} \rightarrow \\ \{!d : \text{dest}, g : \text{gen } d \ t\} \Sigma_{\text{gen}}^* \{\psi_2\} \rightarrow \text{type}.$$

*“If  $\psi_1$  is a well-typed state (generated from a single gen  $d$ ) and there is a step from  $\psi_1$  to  $\psi_2$ , then  $\psi_2$  is a well-typed state.”*

- In Meta-CLF we can express properties about traces:

preservation :  $\prod t : \text{tp}. \nabla d. \nabla g. \prod \psi_1 : \text{ctx}. \prod \psi_2 : \text{ctx}.$

$$\{!d : \text{dest}, g : \text{gen } d \ t\} \Sigma_{\text{gen}}^* \{\psi_1\} \rightarrow \{\psi_1\} \Sigma_{\text{step}}^1 \{\psi_2\} \rightarrow \\ \{!d : \text{dest}, g : \text{gen } d \ t\} \Sigma_{\text{gen}}^* \{\psi_2\} \rightarrow \text{type}.$$

*“If  $\psi_1$  is a well-typed state (generated from a single gen  $d$ ) and **there is a step from  $\psi_1$  to  $\psi_2$** , then  $\psi_2$  is a well-typed state.”*

- In Meta-CLF we can express properties about traces:

preservation :  $\prod t : \text{tp}. \nabla d. \nabla g. \prod \psi_1 : \text{ctx}. \prod \psi_2 : \text{ctx}.$

$$\{!d : \text{dest}, g : \text{gen } d \ t\} \Sigma_{\text{gen}}^* \{\psi_1\} \rightarrow \{\psi_1\} \Sigma_{\text{step}}^1 \{\psi_2\} \rightarrow \\ \{!d : \text{dest}, g : \text{gen } d \ t\} \Sigma_{\text{gen}}^* \{\psi_2\} \rightarrow \text{type}.$$

*“If  $\psi_1$  is a well-typed state (generated from a single gen  $d$ ) and there is a step from  $\psi_1$  to  $\psi_2$ , then  $\psi_2$  is a well-typed state.”*

## Safety proof in Meta-CLF

- Case step/eval :  $\text{eval } e \ d \dashv\vdash \text{value } e \rightarrow \{\text{ret } e \ d\}$ .

$$\mathcal{A}, \text{eval } e \ d \rightsquigarrow_{\text{step}} \mathcal{A}, \text{ret } e \ d$$



## Safety proof in Meta-CLF

- Case step/eval :  $\text{eval } e \ d \dashv\circ \text{value } e \rightarrow \{\text{ret } e \ d\}$ .

$\text{gen } t_0 \ d_0$

$\rightsquigarrow_{\text{gen}}^*$

*The multiset  $\mathcal{A}, \text{eval } e \ d$  is generated from  $\text{gen } t_0 \ d_0$ .*

$\mathcal{A}, \text{eval } e \ d \rightsquigarrow_{\text{step}} \mathcal{A}, \text{ret } e \ d$

## Safety proof in Meta-CLF

- Case  $\text{step/eval} : \text{eval } e \ d \dashv\circ \text{value } e \rightarrow \{\text{ret } e \ d\}$ .

$$\begin{array}{c} \text{gen } t_0 \ d_0 \\ \downarrow \text{gen}^* \\ \mathcal{A}, \text{gen } t \ d \\ \downarrow \text{gen} \\ \mathcal{A}, \text{eval } e \ d \rightsquigarrow_{\text{step}} \mathcal{A}, \text{ret } e \ d \end{array}$$

*There must be one step that generates  $\text{eval } e \ d$  using  $\text{gen/eval}$ ; this step can be moved to the end, since no other step depends on it.*

# Safety proof in Meta-CLF

- Case `step/eval` :  $\text{eval } e \ d \dashv\circ \text{value } e \rightarrow \{\text{ret } e \ d\}$ .

$$\begin{array}{ccc} \text{gen } t_0 \ d_0 & & \text{gen } t_0 \ d_0 \\ \downarrow \text{gen}^* & & \downarrow \text{gen}^*? \\ \mathcal{A}, \text{gen } t \ d & & \\ \downarrow \text{gen} & & \\ \mathcal{A}, \text{eval } e \ d & \rightsquigarrow_{\text{step}} & \mathcal{A}, \text{ret } e \ d \end{array}$$

*We want to show that it is possible to generate the multiset  $\mathcal{A}, \text{ret } e \ d$ .*

## Safety proof in Meta-CLF

- Case step/eval :  $\text{eval } e \ d \dashv\circ \text{value } e \rightarrow \{\text{ret } e \ d\}$ .

$\text{gen } t_0 \ d_0$

$\begin{array}{c} \text{gen} \\ \text{gen}^* \end{array}$

$\mathcal{A}, \text{gen } t \ d$

$\text{gen } t_0 \ d_0$

$\begin{array}{c} \text{gen} \\ \text{gen}^* \end{array}$

$\mathcal{A}, \text{gen } t \ d$

$\begin{array}{c} \text{gen} \\ \text{gen} \end{array}$   
 $\mathcal{A}, \text{eval } e \ d \rightsquigarrow_{\text{step}} \mathcal{A}, \text{ret } e \ d$

*We generate the multiset  $\mathcal{A}$  using the same rules as on the left.*

# Safety proof in Meta-CLF

- Case step/eval :  $\text{eval } e \ d \multimap \text{value } e \rightarrow \{\text{ret } e \ d\}$ .

$$\begin{array}{ccc} \text{gen } t_0 \ d_0 & & \text{gen } t_0 \ d_0 \\ \downarrow \text{gen}^* & & \downarrow \text{gen}^* \\ \mathcal{A}, \text{gen } t \ d & & \mathcal{A}, \text{gen } t \ d \\ \downarrow \text{gen} & & \downarrow \text{gen} \\ \mathcal{A}, \text{eval } e \ d & \rightsquigarrow_{\text{step}} & \mathcal{A}, \text{ret } e \ d \end{array}$$

*We generate the multiset  $\mathcal{A}$  using the same rules as on the left. But change the last step to generate  $\text{ret } e \ d$ .*

## Safety proof in Meta-CLF

- Case step/eval :  $\text{eval } e \ d \dashv\vdash \text{value } e \rightarrow \{\text{ret } e \ d\}$ .

$$\begin{array}{ccc} \text{gen } t_0 \ d_0 & & \text{gen } t_0 \ d_0 \\ \downarrow \text{gen}^* & & \downarrow \text{gen}^* \\ \mathcal{A}, \text{gen } t \ d & & \mathcal{A}, \text{gen } t \ d \\ \downarrow \text{gen} & & \downarrow \text{gen} \\ \mathcal{A}, \text{eval } e \ d & \rightsquigarrow_{\text{step}} & \mathcal{A}, \text{ret } e \ d \end{array}$$

- In Meta-CLF:

# Safety proof in Meta-CLF

- Case step/eval :  $\text{eval } e \ d \dashv\vdash \text{value } e \rightarrow \{\text{ret } e \ d\}$ .

$$\begin{array}{ccc} \text{gen } t_0 \ d_0 & & \text{gen } t_0 \ d_0 \\ \downarrow \text{gen}^* & & \downarrow \text{gen}^* \\ \mathcal{A}, \text{gen } t \ d & & \mathcal{A}, \text{gen } t \ d \\ \downarrow \text{gen} & & \downarrow \text{gen} \\ \mathcal{A}, \text{eval } e \ d & \rightsquigarrow_{\text{step}} & \mathcal{A}, \text{ret } e \ d \end{array}$$

- In Meta-CLF:

$$\begin{aligned} \text{pres/ret : preservation } & (X_1; \{\downarrow x\} \leftarrow \text{gen/eval } g_0 \ H) \\ & (\{\downarrow y\} \leftarrow \text{step/eval } x \ H_v) \\ & (X_1; \{\downarrow y\} \leftarrow \text{gen/ret } g_0 \ H \ H_v) \end{aligned}$$

where  $x : \text{eval } e \ d$ ,  $g_0 : \text{gen } t \ d$ ,  $H : \text{of } e \ t$ ,  $H_v : \text{value } e$ .

# Meta-CLF

- Both proofs of preservation and progress in Meta-CLF follow the pen-and-paper proofs.
- Preservation is performed by case analysis (no induction).
- Progress relies on induction, but termination is easy (size of the trace).
- However, we rely on coverage to ensure the proof is total.
- Coverage checking in the presence of traces is tricky, due to the possibility of permuting steps. (Left for future work.)



# Conclusions

- Our goal is to develop logical frameworks suitable for specifying concurrent and distributed systems.
- We introduced Meta-CLF, an extension of LF to reason about CLF specifications.
- We showed that it is expressive enough to write safety proofs of parallel/concurrent PL.

## Future work

- Decidability of type checking

# Future work

- Decidability of type checking
- Type reconstruction for implicit arguments (very important for usability)

pres/ret :

preservation

$$\begin{array}{l} (X_1; \{\downarrow x\} \leftarrow \text{gen/eval} \quad g_0 H) \\ (\{\downarrow y\} \leftarrow \text{step/eval} \quad x H_v) \\ (X_1; \{\downarrow y\} \leftarrow \text{gen/ret} \quad g_0 H H_v) \end{array}$$

# Future work

- Decidability of type checking
- Type reconstruction for implicit arguments (very important for usability)

pres/ret :  $\nabla x. \nabla y. \nabla d. \nabla g. \nabla d_0. \nabla g_0. \Pi \psi'_1 : \text{ctx}. \Pi e : \text{exp}.$   
 $\Pi t : \text{tp}. \Pi t_0 : \text{tp}. \Pi H : \text{of } e \ t_0. \Pi H_v : \text{value } e$   
 $\Pi X : \{!d : \text{dest}, \downarrow g : \text{gen } d \ t\} \Sigma_{\text{gen}}^* \{\psi'_1, !d_0 : \text{dest}, \downarrow g_0 : \text{gen } d_0 \ t_0\}.$   
preservation  $t \ d \ g \ (\psi'_1, !d_0 : \text{dest}, \downarrow x : \text{eval } e \ d_0)$   
 $(\psi'_1, !d_0 : \text{dest}, \downarrow y : \text{ret } e \ d_0)$   
 $(X_1; \{\downarrow x\} \leftarrow \text{gen/eval } e \ t \ d_0 \ g_0 \ H)$   
 $(\{\downarrow y\} \leftarrow \text{step/eval } e \ d_0 \ x \ H_v)$   
 $(X_1; \{\downarrow y\} \leftarrow \text{gen/ret } e \ t \ d_0 \ g_0 \ H \ H_v)$

# Future work

- Decidability of type checking
- Type reconstruction for implicit arguments (very important for usability)

pres/ret :  $\nabla x. \nabla y. \nabla d. \nabla g. \nabla d_0. \nabla g_0. \Pi \psi'_1 : \text{ctx}. \Pi e : \text{exp}.$   
 $\Pi t : \text{tp}. \Pi t_0 : \text{tp}. \Pi H : \text{of } e \ t_0. \Pi H_v : \text{value } e$   
 $\Pi X : \{!d : \text{dest}, \downarrow g : \text{gen } d \ t\} \Sigma_{\text{gen}}^* \{\psi'_1, !d_0 : \text{dest}, \downarrow g_0 : \text{gen } d_0 \ t_0\}.$   
preservation  $t \ d \ g \ (\psi'_1, !d_0 : \text{dest}, \downarrow x : \text{eval } e \ d_0)$   
 $(\psi'_1, !d_0 : \text{dest}, \downarrow y : \text{ret } e \ d_0)$   
 $(X_1; \{\downarrow x\} \leftarrow \text{gen/eval } e \ t \ d_0 \ g_0 \ H)$   
 $(\{\downarrow y\} \leftarrow \text{step/eval } e \ d_0 \ x \ H_v)$   
 $(X_1; \{\downarrow y\} \leftarrow \text{gen/ret } e \ t \ d_0 \ g_0 \ H \ H_v)$

- Implementation

# Future work

- Coverage checking

- ▶ Coverage checking for traces is difficult due to the equality relation.
- ▶ Easier when restricted to **generative invariants**
- ▶ GI look like a generalization of context-free grammars

$$\text{gen/eval} : \text{gen } t \ d \multimap \text{of } e \ t \rightarrow \{\text{eval } e \ d\}.$$
$$\text{gen/ret} : \text{gen } t \ d \multimap \text{of } e \ t \rightarrow \{\text{ret } e \ d\}.$$
$$\text{gen/fapp} : \text{gen } t \ d \multimap \{!d_1 \ !d_2 : \text{dest, fapp } d_1 \ d_2 \ d, \\ \text{gen } (\text{arr } t_1 \ t) \ d_1, \text{gen } t_1 \ d_2\}.$$

**Non-terminal:** gen. **Terminals:** eval, ret, fapp.

Example:

$$X_1; (\{y\} \leftarrow \text{gen/eval } x); X_2 \quad \equiv \quad X_1; X_2; (\{y\} \leftarrow \text{gen/eval } x)$$

- Termination (trace size)