Meta-Reasoning in a Concurrent Logical Framework

Iliano Cervesato and Jorge Luis Sacchini

Carnegie Mellon University

Chalmers University, 16 Oct 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)
- We follow the shallow approach, using CLF as our LF.

Outline

1. CLF
2. Substructural operational semantics
3. Safety for SSOS
4. Meta-CLF
5. Conclusions and future work

Outline

1. CLF
2. Substructural operational semantics
3. Safety for SSOS
4. Meta-CLF
5. Conclusions and future work

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:

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

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

CLF

- Synchronous types: $\otimes$ is associative and commutative, 1 is a neutral element.

$$\exists x : A.!B \otimes C \otimes 1 \quad \longrightarrow \quad !x : A,\ !y_1 : B,\ y_2 : C$$

- Arguments can be uncurrified.

$$\Pi x : A.B \to C \multimap \{D\} \quad \longrightarrow \quad \Pi !x : A, !y_1 : B, y_2 : C.\ \{D\}$$

- We can redefine types in CLF to use contexts:

$$\begin{array}{lcll} A & ::= & \Pi\Delta.P \mid \Pi\Delta.\{\Delta\} & \text{(Async types)} \\ \Delta & ::= & \cdot \mid x : A, \Delta \mid !x : A, \Delta & \text{(Contexts)} \end{array}$$

CLF

CLF combines:

- Asynchronous types (Π, $\multimap$, $\&$)
  - Linear Logical Framework
  - Backward chaining operational semantics
- Synchronous types ($\exists$, $\otimes$)
  - Encapsulated in a monad ($\{S\}$)
  - Forward chaining operational semantics

CLF

Asynchronous fragment

- Backward chaining (as in Twelf):

$$\begin{aligned} \text{plus} &: \text{nat} \to \text{nat} \to \text{type}. \\ \text{plus/z} &: \text{plus } 0\ N\ N. \\ \text{plus/s} &: \text{plus } (\text{s } M)\ N\ (\text{s } P) \\ &\quad \leftarrow \text{plus } M\ N\ P. \end{aligned}$$

- Example:

$$\text{plus } 2\ 2\ X \qquad X?$$

CLF

Asynchronous fragment

- Backward chaining (as in Twelf):

$$\begin{aligned} \text{plus} &: \text{nat} \to \text{nat} \to \text{type}. \\ \text{plus/z} &: \text{plus } 0\ N\ N. \\ \text{plus/s} &: \text{plus } (\text{s } M)\ N\ (\text{s } P) \\ &\quad \leftarrow \text{plus } M\ N\ P. \end{aligned}$$

- Example:

$$\frac{\text{plus } 1\ 2\ X_1}{\text{plus } 2\ 2\ X} \qquad X = \text{s } X_1$$

CLF

Asynchronous fragment

- Backward chaining (as in Twelf):

$$\begin{aligned} \text{plus} &: \text{nat} \to \text{nat} \to \text{type}. \\ \text{plus/z} &: \text{plus } 0\ N\ N. \\ \text{plus/s} &: \text{plus } (\text{s } M)\ N\ (\text{s } P) \\ &\quad \leftarrow \text{plus } M\ N\ P. \end{aligned}$$

- Example:

$$\dfrac{\dfrac{\text{plus } 0\ 2\ X_2}{\text{plus } 1\ 2\ X_1}}{\text{plus } 2\ 2\ X} \qquad X = \text{s } X_1 = \text{s } (\text{s } X_2)$$

CLF

Asynchronous fragment

- Backward chaining (as in Twelf):

$$\begin{aligned} \text{plus} &: \text{nat} \to \text{nat} \to \text{type}. \\ \text{plus/z} &: \text{plus } 0\ N\ N. \\ \text{plus/s} &: \text{plus } (\text{s } M)\ N\ (\text{s } P) \\ &\quad \leftarrow \text{plus } M\ N\ P. \end{aligned}$$

- Example:

$$\dfrac{\dfrac{\text{plus } 0\ 2\ X_2}{\text{plus } 1\ 2\ X_1}}{\text{plus } 2\ 2\ X} \qquad X = \text{s } X_1 = \text{s } (\text{s } X_2) = \text{s } (\text{s } 2)$$

CLF

Synchronous fragment

- Monadic types are used to encapsulate concurrent effects:

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

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

CLF

Synchronous fragment

- Monadic types are used to encapsulate concurrent effects:

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

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

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

CLF

Synchronous fragment

- Monadic types are used to encapsulate concurrent effects:

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

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

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

CLF

Synchronous fragment

- Monadic types are used to encapsulate concurrent effects:

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

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

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

CLF

Synchronous fragment

- Monadic types are used to encapsulate concurrent effects:

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

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

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

CLF

Synchronous fragment

- Monadic types are used to encapsulate concurrent effects:

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

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

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

CLF

Synchronous fragment

- Monadic types are used to encapsulate concurrent effects:

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

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

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

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 ε transforms (rewrites) Δ into Δ'"*

$$\frac{}{\Delta \vdash \diamond : \Delta} \qquad \frac{c : \Pi\Delta_0.\{\Delta_1\} \in \Sigma}{\Delta \bowtie \Delta_0 \vdash \{\Delta_1\}{\leftarrow}c \cdot \Delta_0 : \Delta \bowtie \Delta_1}$$

$$\frac{\Delta_0 \vdash \varepsilon_1 : \Delta_1 \qquad \Delta_1 \vdash \varepsilon_2 : \Delta_2}{\Delta_0 \vdash \varepsilon_1; \varepsilon_2 : \Delta_2}$$

- $\Delta = \Delta_0 \bowtie \Delta_1$ iff
  - Persistent declarations in Δ appear in both Δ_0 and Δ_1
  - Linear declarations in Δ appear in exactly one of Δ_0 and Δ_1

Traces

- Example: $\mathsf{in} : \mathsf{tick} \multimap \mathsf{count}\ N \multimap \{\mathsf{count}\ (s\ N)\}$

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

$$\begin{array}{ll} \vdash & \{c_{11}\} \leftarrow \mathsf{in} \cdot t_1, c_1 \\ & \{c_{12}\} \leftarrow \mathsf{in} \cdot t_2, c_{11} \\ & \{c_{21}\} \leftarrow \mathsf{in} \cdot t_3, c_2 \\ & \{c_{13}\} \leftarrow \mathsf{in} \cdot t_4, c_{12} \\ & \{c_{22}\} \leftarrow \mathsf{in} \cdot t_5, c_{21} \\ & : c_{13} : \mathsf{count}\ 3, c_{22} : \mathsf{count}\ 2 \end{array}$$

Traces

- Equality on traces: α-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:

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

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

Traces

- Trace equality:

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

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

- Example:

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

Outline

1. CLF
2. Substructural operational semantics
3. Safety for SSOS
4. Meta-CLF
5. Conclusions and future work

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 λ-calculus

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

- HOAS representation in (C)LF:

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

$$\mathsf{lam} : (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}\,.$$
$$\mathsf{app} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}\,.$$

SSOS

- Linear-destination passing style (Pfenning04)
- Based on multiset rewriting; suitable for specifying in linear logic
- Multiset of facts:

eval e d	Evaluate expression e in destination d
ret e d	Return value e to destination d
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

SSOS

- Expressions represented as multiset of facts:

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

- In CLF:

$$\begin{aligned}
&\text{dest} : \text{type}.\\
&\text{eval} : \text{exp} \to \text{dest} \to \text{type}.\\
&\text{ret} \ \ : \text{exp} \to \text{dest} \to \text{type}.\\
&\text{fapp} : \text{dest} \to \text{dest} \to \text{dest} \to \text{type}.
\end{aligned}$$

Parallel evaluation semantics

- Transitions as multiset rewriting rules:

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

- In CLF:

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

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:

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

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

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

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

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

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

In CLF:

$$\begin{array}{l} !d : \mathsf{dest},\ x_0 : \mathsf{eval}\ (\mathsf{app}\ (\mathsf{lam}\ \lambda x.x)\ (\mathsf{lam}\ \lambda y.y))\ d \\ \quad \vdash \{!d_1, !d_2, x, y, z\} \leftarrow \mathsf{step/app}\ x_0; \\ \qquad\qquad \{x'\} \leftarrow \mathsf{step/eval}\ x; \end{array}$$

$$\begin{array}{l} : !d\ !d_1\ !d_2 : \mathsf{dest},\ x' : \mathsf{ret}\ (\mathsf{lam}\ \lambda x.x)\ d_1,\ y : \mathsf{eval}\ (\mathsf{lam}\ \lambda y.y)\ d_2 \\ \quad z : \mathsf{fapp}\ d_1\ d_2\ d \end{array}$$

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:

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

$$\begin{aligned}
&: !d\ !d_1\ !d_2 : \text{dest},\ x' : \text{ret}\ (\text{lam}\ \lambda x.x)\ d_1,\ y' : \text{ret}\ (\text{lam}\ \lambda y.y)\ d_2 \\
&\quad z : \text{fapp}\ d_1\ d_2\ d
\end{aligned}$$

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:

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

Example

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

In CLF:

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

Example

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

In CLF:

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

Outline

1. CLF
2. Substructural operational semantics
3. Safety for SSOS
4. Meta-CLF
5. Conclusions and future work

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 Δ is a well-typed multiset and $\Delta \rightsquigarrow \Delta'$ (step), then Δ' is a well-typed multiset.

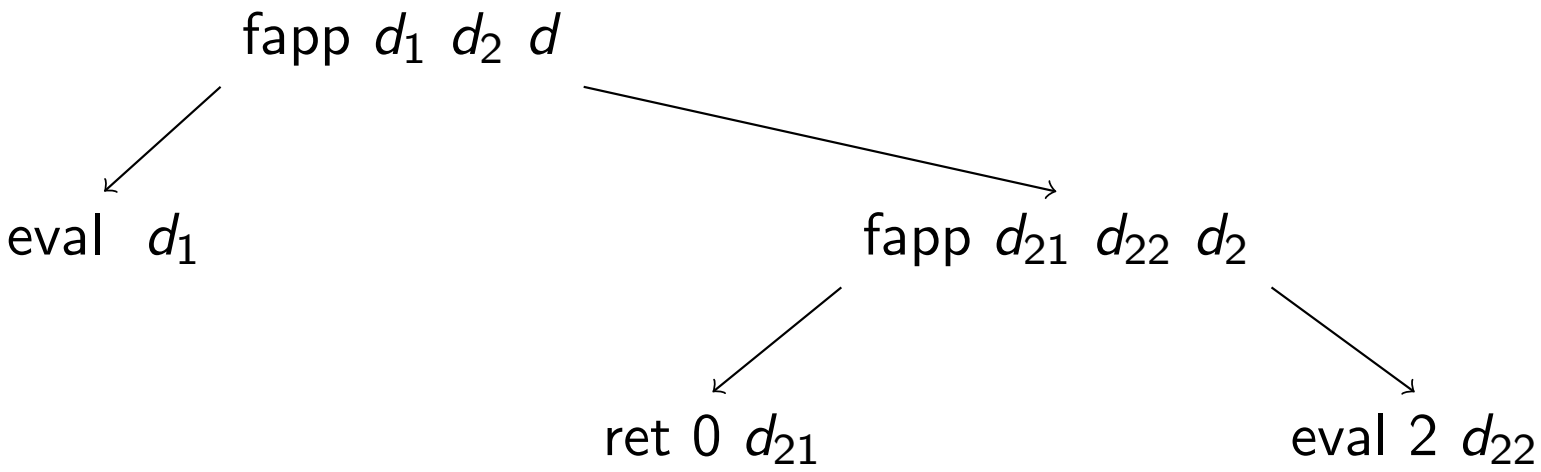
  - Progress: a well-typed multiset is either final (result) or is possible to take a step.

    If Δ is a well-typed multiset, then either $\Delta = \{ret\ e\ d\}$ or there exists Δ' such that $\Delta \rightsquigarrow \Delta'$.

- We need a notion of well-typed multiset for SSOS specifications.

Well-formed multisets

- Example: eval e_1 d, *eval* e_2 d ✖ (repeated destination)
  fapp d_1 d_2 d, eval e_1 d_1 ✖ (no fact for d_2)
- Well-formed multisets form a tree:



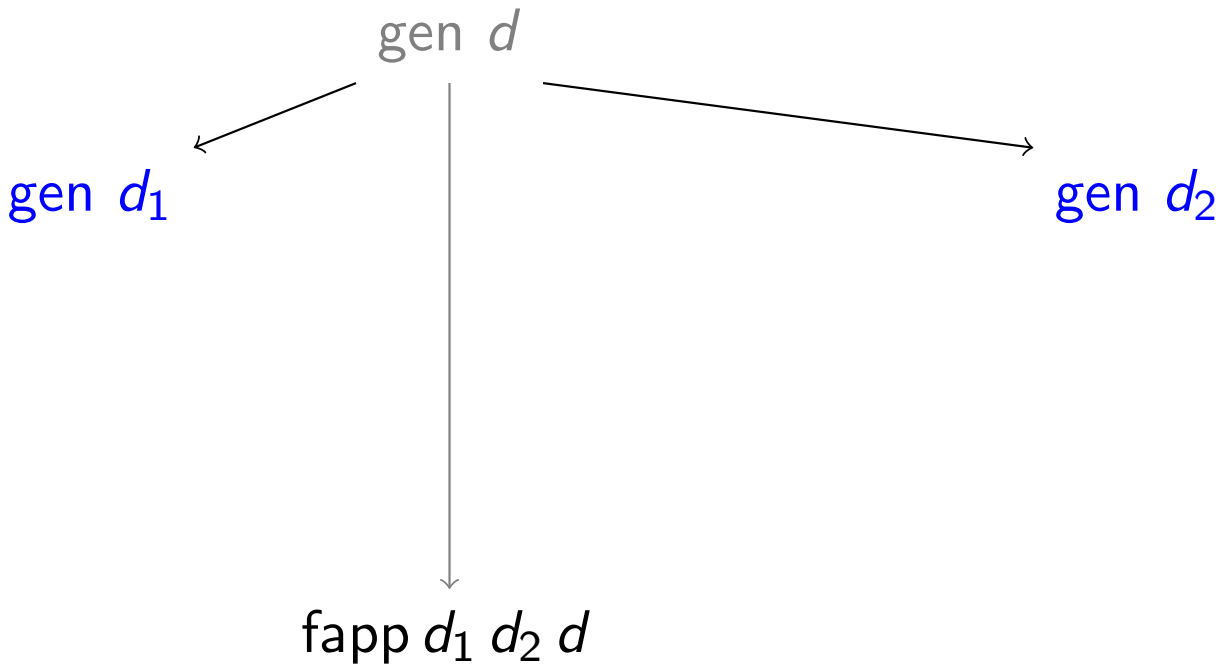
Well-formed multisets

- Well-formed multisets can be described by rewriting rules.
- gen t d means *"generate a term of type t rooted at d"*

gen d

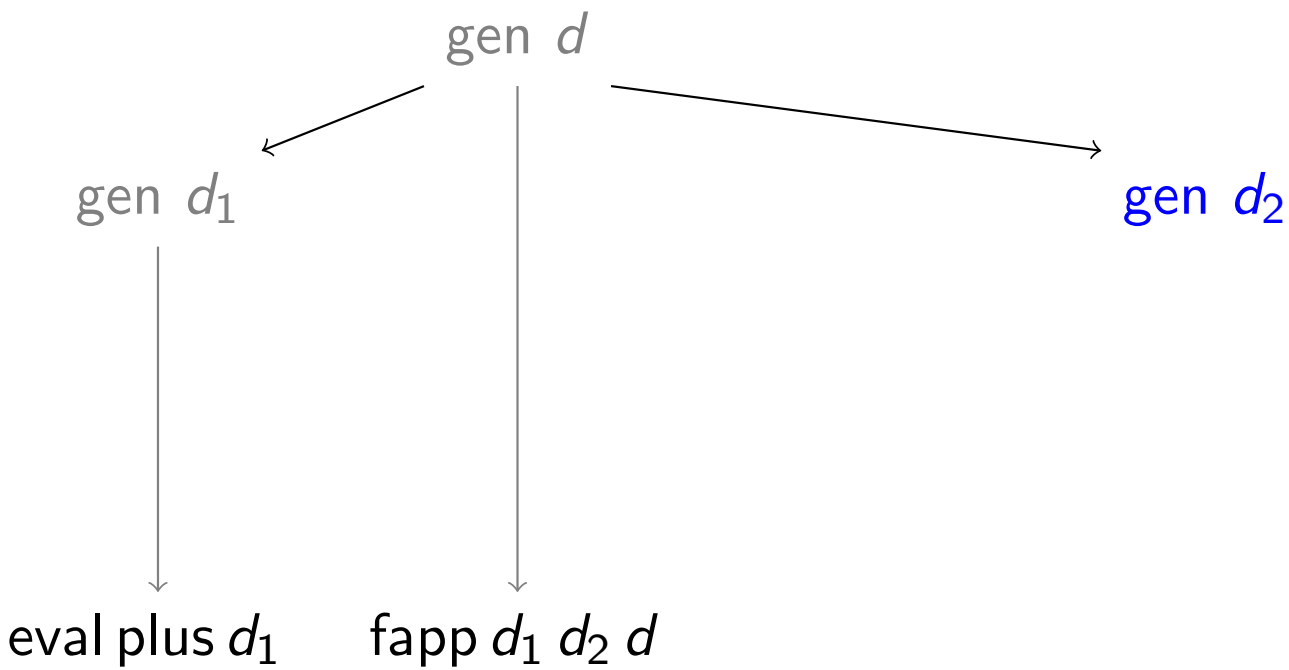
Well-formed multisets

- Well-formed multisets can be described by rewriting rules.
- gen t d means *"generate a term of type t rooted at d"*



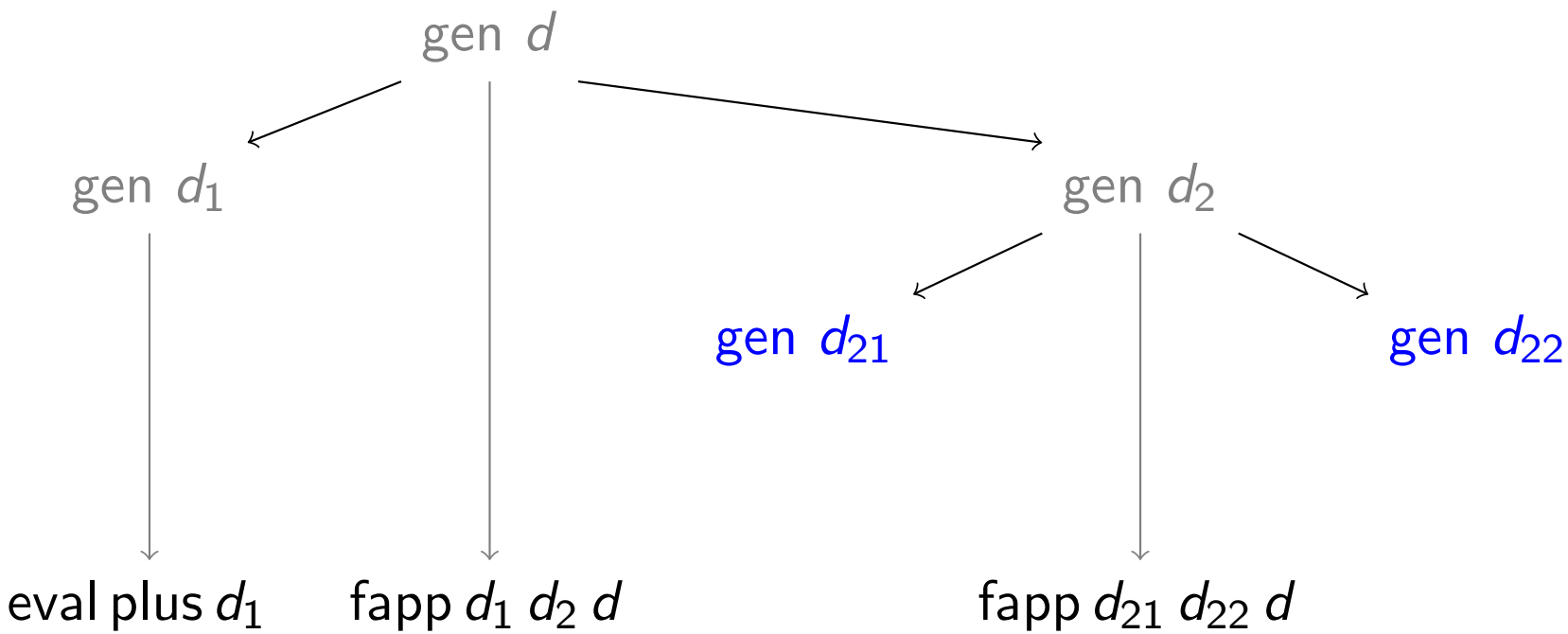
Well-formed multisets

- Well-formed multisets can be described by rewriting rules.
- gen t d means *"generate a term of type t rooted at d"*



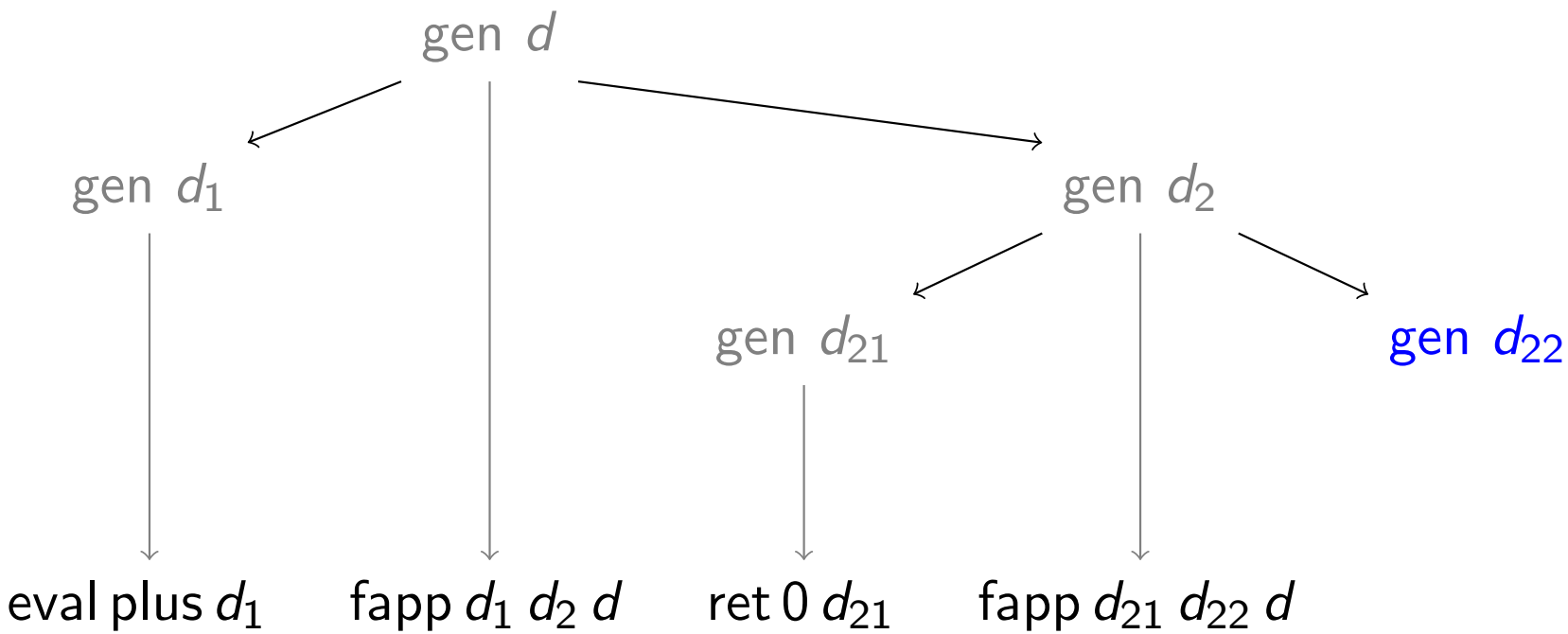
Well-formed multisets

- Well-formed multisets can be described by rewriting rules.
- gen t d means *"generate a term of type t rooted at d"*



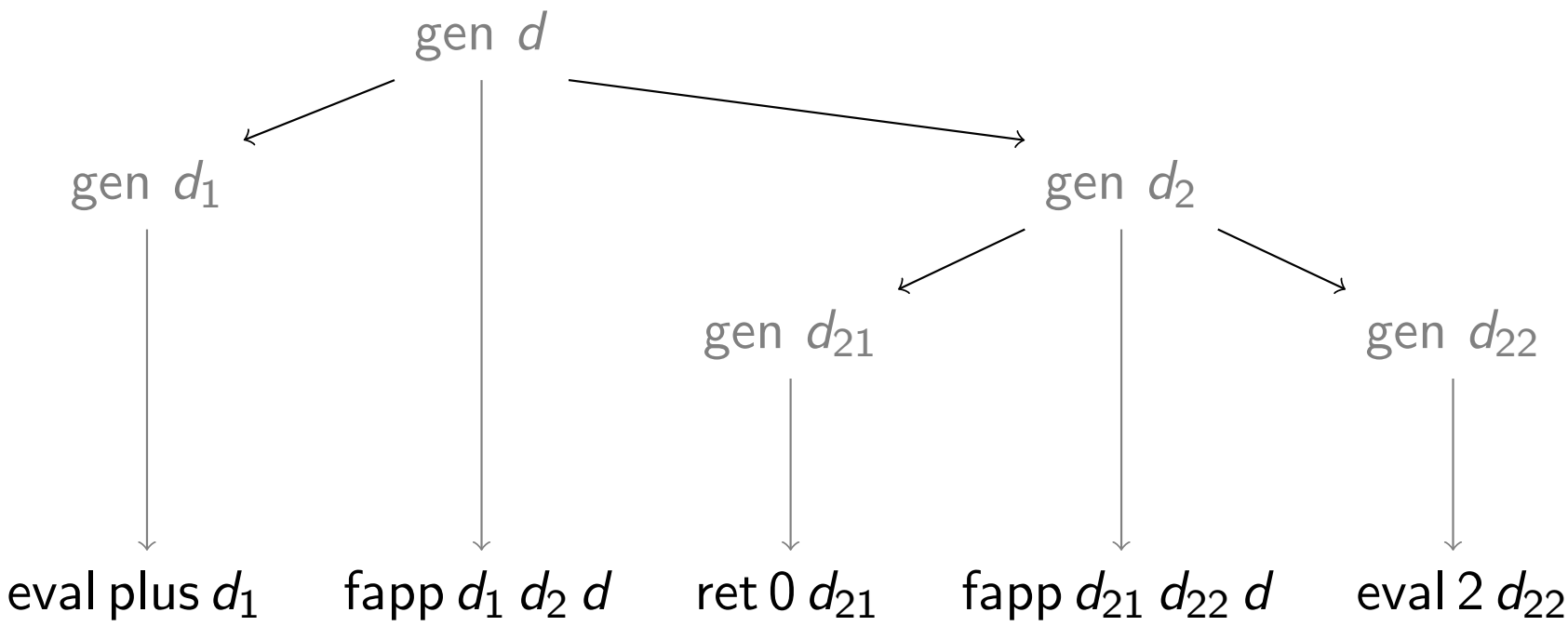
Well-formed multisets

- Well-formed multisets can be described by rewriting rules.
- gen t d means *"generate a term of type t rooted at d"*



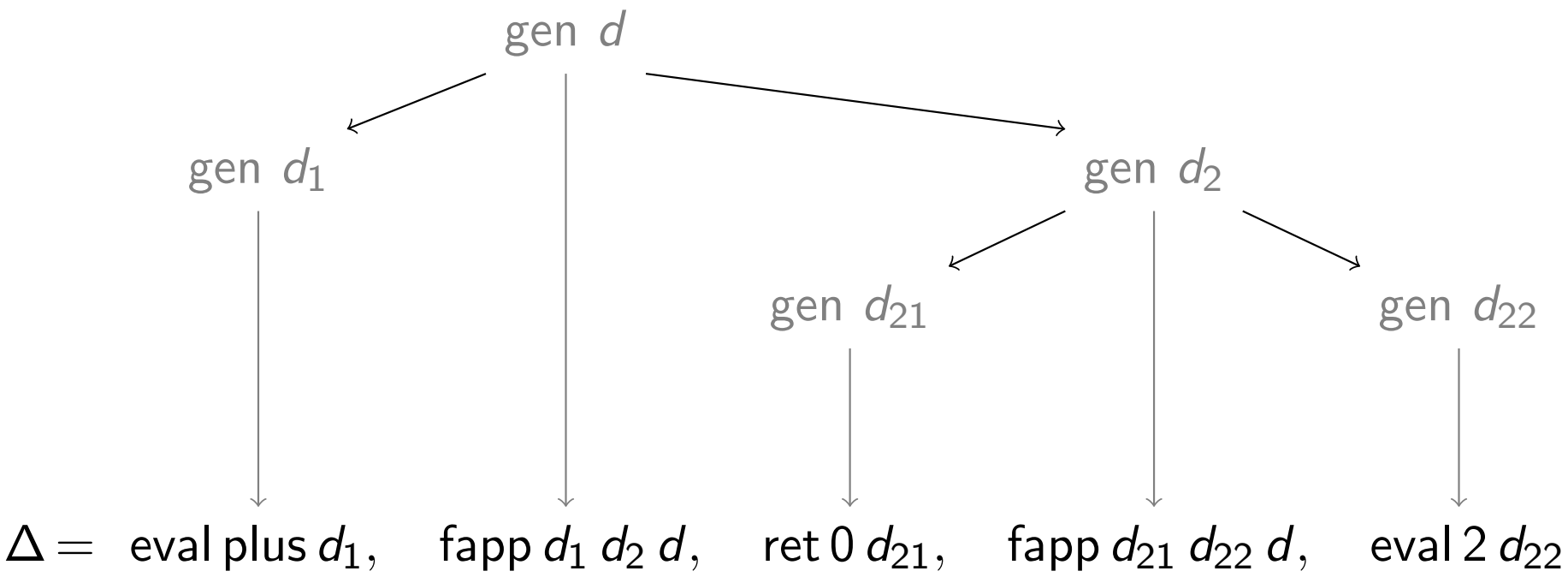
Well-formed multisets

- Well-formed multisets can be described by rewriting rules.
- 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.
- gen t d means *"generate a term of type t rooted at d"*



Well-typed multisets

- Add a type argument:

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

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

- In CLF:

$$\begin{array}{ll}
\text{gen} & : \text{tp} \to \text{dest} \to \text{type}. \\
\text{gen/eval} & : \text{gen } t\ d \multimap \text{of } e\ t \to \{\text{eval } e\ d\}. \\
\text{gen/ret} & : \text{gen } t\ d \multimap \text{of } e\ t \to \text{value } e \to \{\text{ret } e\ d\}. \\
\text{gen/fapp} & : \text{gen } t\ d \multimap \{!d_1\ !d_2 : \text{dest}, \\
 & \qquad \text{fapp } d_1\ d_2\ d, \\
 & \qquad \text{gen } (\text{arr } t_1\ t)\ d_1, \\
 & \qquad \text{gen } t_1\ d_2\}.
\end{array}$$

- 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).

Outline

1. CLF
2. Substructural operational semantics
3. Safety for SSOS
4. Meta-CLF
5. Conclusions and future work

Meta-CLF

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

$$A ::= \ldots \mid \{\Delta\}\,\Sigma^*\,\{\Delta\} \mid \{\Delta\}\,\Sigma^1\,\{\Delta\} \mid \Pi\psi : \mathsf{ctx}.A \mid \nabla x.A$$

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

Meta-CLF

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

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

"If ψ_1 is a well-typed state (generated from a single gen *d) and there is a step from ψ_1 to ψ_2, then ψ_2 is a well-typed state."*

Meta-CLF

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

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

"If ψ_1 is a well-typed state (generated from a single gen *d) and there is a step from ψ_1 to ψ_1, then ψ_2 is a well-typed state."*

Meta-CLF

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

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

"If ψ_1 is a well-typed state (generated from a single gen *d) and there is a step from ψ_1 to ψ_2, then ψ_2 is a well-typed state."*

Meta-CLF

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

$$\begin{aligned}&\text{preservation} : \Pi t : \text{tp}.\ \nabla d.\ \nabla g.\ \Pi \psi_1 : \text{ctx}.\ \Pi \psi_2 : \text{ctx}.\\ &\qquad \{!d : \text{dest}, g : \text{gen}\ d\ t\}\, \Sigma^*_{\text{gen}}\, \{\psi_1\} \rightarrow \{\psi_1\}\, \Sigma^1_{\text{step}}\, \{\psi_2\} \rightarrow\\ &\qquad\qquad \{!d : \text{dest}, g : \text{gen}\ d\ t\}\, \Sigma^*_{\text{gen}}\, \{\psi_2\} \rightarrow \text{type}.\end{aligned}$$

"If ψ_1 is a well-typed state (generated from a single gen *d) and there is a step from ψ_1 to ψ_2, then ψ_2 is a well-typed state."*

Meta-CLF

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

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

"If ψ_1 is a well-typed state (generated from a single gen *d) and there is a step from ψ_1 to ψ_2, then ψ_2 is a well-typed state."*

Safety proof in Meta-CLF

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

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

Safety proof in Meta-CLF

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

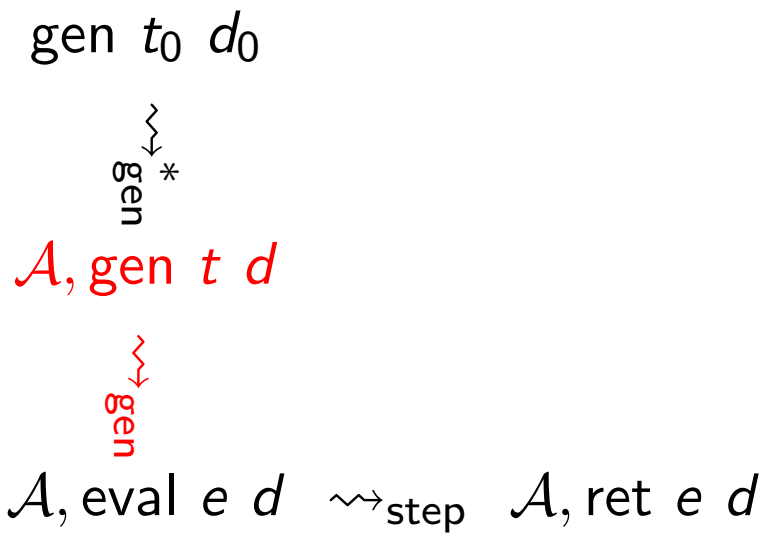
gen t_0 d_0

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

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

Safety proof in Meta-CLF

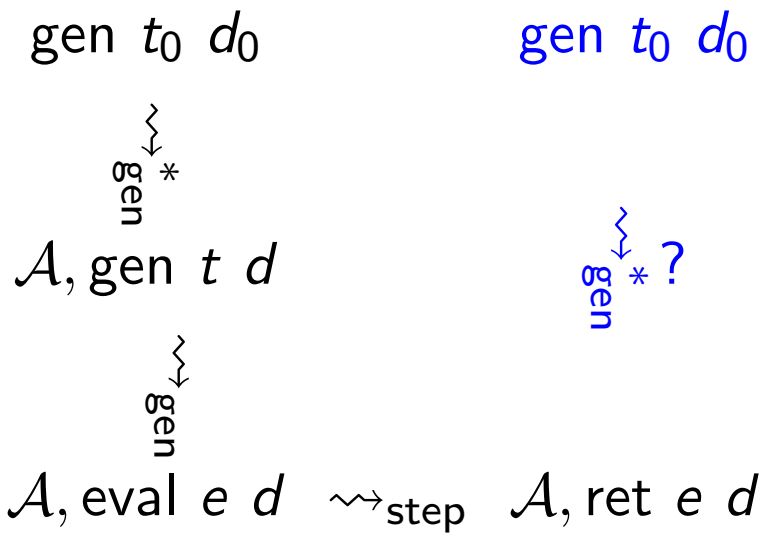
- Case step/eval : eval e d $\multimap$ value e $\to$ $\{$ret e $d\}$.



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

Safety proof in Meta-CLF

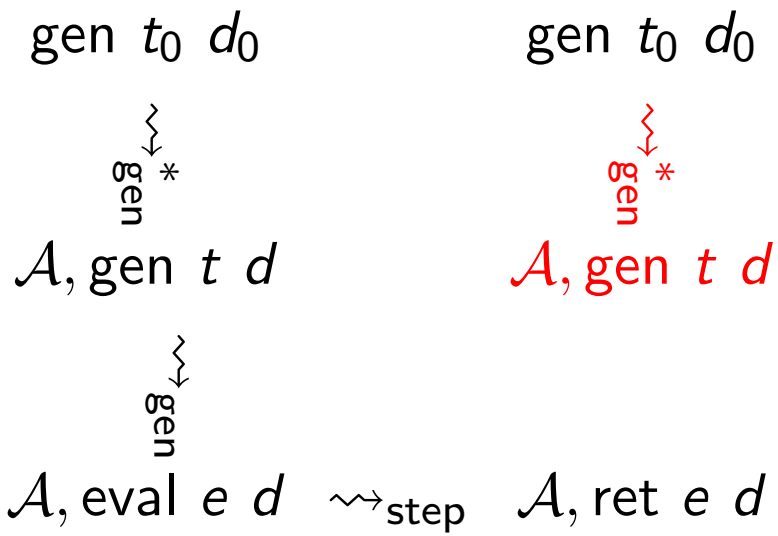
- Case step/eval : eval $e\ d \multimap$ value $e \to \{\text{ret}\ e\ d\}$.



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

Safety proof in Meta-CLF

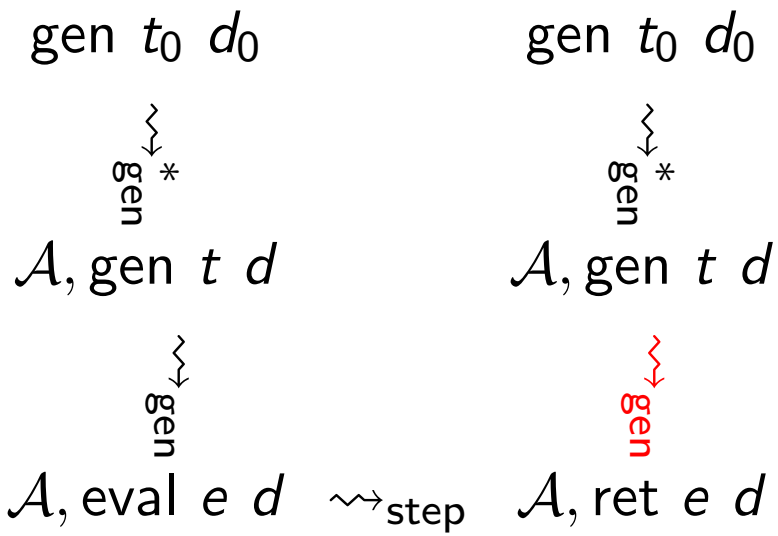
- Case step/eval : eval $e\ d \multimap$ value $e \to \{\text{ret}\ e\ d\}$.



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

Safety proof in Meta-CLF

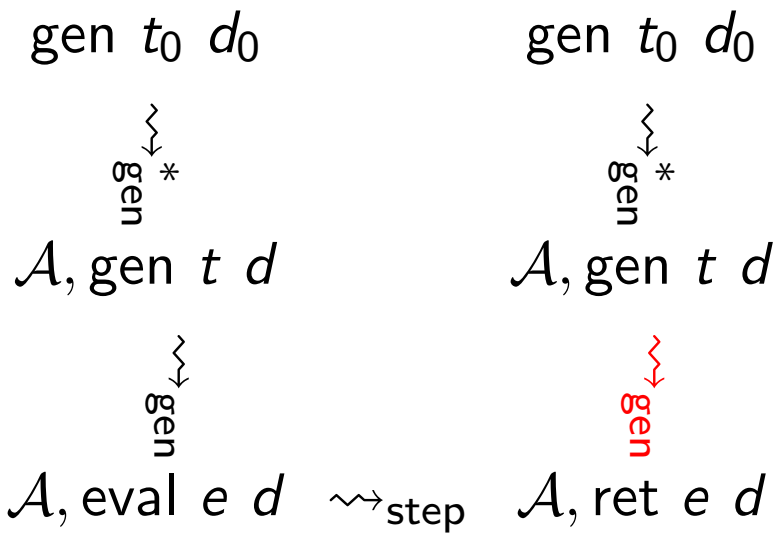
- Case step/eval : eval $e\ d \multimap$ value $e \to \{\text{ret}\ e\ d\}$.



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

Safety proof in Meta-CLF

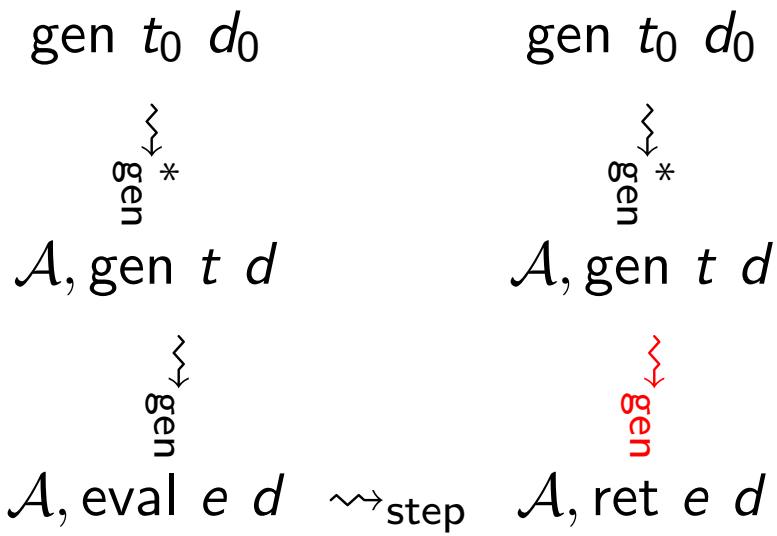
- Case step/eval : eval $e\ d \multimap$ value $e \to \{\text{ret}\ e\ d\}$.



- In Meta-CLF:

Safety proof in Meta-CLF

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



- In Meta-CLF:

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

where x : eval $e\,d$, g_0 : gen $t\,d$, H : of $e\,t$, H_v : 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.)

SSOS

- We can extend this semantics with other features without invalidating the previous rules
- Example: store, futures, call/cc, communication,...

$$
\begin{array}{ll}
\mathsf{location} & : \mathsf{type}. \\
\mathsf{loc} & : \mathsf{location} \rightarrow \mathsf{exp}\,. \\
\mathsf{get} & : \mathsf{exp} \rightarrow \mathsf{exp}\,. \\
\mathsf{ref} & : \mathsf{exp} \rightarrow \mathsf{exp}\,. \\
\mathsf{set} & : \mathsf{exp} \rightarrow \mathsf{exp} \rightarrow \mathsf{exp}\,. \\
 & \\
\mathsf{cell} & : \mathsf{location} \rightarrow \mathsf{exp} \rightarrow \mathsf{type}. \\
\mathsf{step/ref} & : \mathsf{eval}\ (\mathsf{ref}\ e)\ d \multimap \{!d_1 : \mathsf{dest}, !l : \mathsf{location}, \\
 & \qquad\qquad \mathsf{fref}\ d_1\ l, \mathsf{eval}\ e\ d_1, \mathsf{ret}\ (\mathsf{loc}\ l)\ d\}. \\
\mathsf{step/fref} & : \mathsf{ret}\ e\ d \multimap \mathsf{fref}\ d\ l \multimap \{\mathsf{cell}\ l\ e\}.
\end{array}
$$

SSOS

- We can extend this semantics with other features without invalidating the previous rules
- Example: store, futures, call/cc, communication,...

$$
\begin{array}{ll}
\text{future} & : \text{exp} \to \text{exp}\,. \\
\text{promise} & : \text{dest} \to \text{exp}\,. \\
\\
\text{deliver} & : \text{exp} \to \text{dest} \to \text{type}. \\
\\
\text{step/fut} & : \text{eval}\ (\text{future}\ e)\ d \multimap \{!d_1 : \text{dest}, \\
 & \qquad\qquad\qquad\qquad \text{eval}\ e\ d_1, \text{fdel}\ d_1, \\
 & \qquad\qquad\qquad\qquad \text{ret}\ (\text{promise}\ d_1)\ d\}. \\
\text{step/fdel} & : \text{ret}\ e\ d \multimap \text{fdel}\ d_1 \multimap \{!\text{deliver}\ e\ d\}. \\
\text{step/promise} & : \text{ret}\ (\text{promise}\ d_1)\ d \multimap \text{delivee}\ e\ d_1 \to \text{ret}\ e\ d.
\end{array}
$$

Outline

1. CLF
2. Substructural operational semantics
3. Safety for SSOS
4. Meta-CLF
5. Conclusions and 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

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

Future work

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

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

Future work

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

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

- 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

$$\begin{aligned}
&\text{gen/eval} : \text{gen}\ t\ d \multimap \text{of}\ e\ t \to \{\text{eval}\ e\ d\}.\\
&\text{gen/ret} \ \ : \text{gen}\ t\ d \multimap \text{of}\ e\ t \to \{\text{ret}\ e\ d\}.\\
&\text{gen/fapp} : \text{gen}\ t\ d \multimap \{!d_1\ !d_2 : \text{dest}, \text{fapp}\ d_1\ d_2\ d,\\
&\qquad\qquad\qquad\qquad \text{gen}\ (\text{arr}\ t_1\ t)\ d_1, \text{gen}\ t_1\ d_2\}.
\end{aligned}$$

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)