Towards Meta-Reasoning in the Concurrent Logical Framework CLF

Iliano Cervesato   Jorge Luis Sacchini

Carnegie Mellon University

August 26, 2013

Objectives

- Concurrency and distribution are essential features in modern PL.
- Their formal semantics is not as well understood or studied as in the sequential case.
- Formal semantics will enable, e.g., development of formal verification frameworks, verifying program transformations, etc.

Logical frameworks

- Logical frameworks are 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.
- Two main approaches
  - 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

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:

$$\begin{array}{rcl} A & ::= & a \cdot S \mid \Pi !x : A.A \mid A \rightarrow B \mid A \multimap B \mid \{\Delta\} \\ \Delta & ::= & \cdot \mid \Delta, \downarrow x : A \mid \Delta, !x : A \end{array}$$

Example: $A \multimap B \multimap \{x : B, y : C\}$.

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).

Higher-order abstract syntax

- Simply-typed λ-calculus

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

- 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	Value e in 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

- Multiset of facts:

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

- In CLF:

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

Evaluation rules

- Multiset rewriting rules:

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

- In CLF:

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

Evaluation rules

- Multiset rewriting rules:

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

where d_1, d_2 fresh

- In CLF:

$$\begin{aligned}\mathsf{step/app} : {} & \mathsf{eval}\ (\mathsf{app}\ e_1\ e_2)\ d \\ & \multimap \{!d_1\ !d_2 : \mathsf{dest}, \\ & \qquad x_1 : \mathsf{eval}\ e_1\ d_1, x_2 : \mathsf{eval}\ e_2\ d_2, \\ & \qquad y : \mathsf{fapp}\ d_1\ d_2\ d\}.\end{aligned}$$

Evaluation rules

- Multiset rewriting rules:

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

- In CLF:

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

Traces

- Evaluations (sequences of steps) are represented in CLF using traces.
- A trace is a sequence of computational steps, where independent steps can be permuted:

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

- $\{\Delta\}\leftarrow c \cdot S$ means apply rule c to arguments S returning a new context Δ; essentially a rewriting rule.
- Typing rules for traces:

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

Traces

- Equality on traces: α-equivalence modulo permutation of independent steps
- Two steps are independent if they operate on different variables:

$$\{\Delta_1\}\leftarrow c_1 \cdot S_1; \{\Delta_2\}\leftarrow c_2 \cdot S_2 \equiv \{\Delta_2\}\leftarrow c_2 \cdot S_2; \{\Delta_1\}\leftarrow c_1 \cdot S_1$$

if $\mathrm{dom}(\Delta_1) \cap \mathrm{FV}(S_2) = \mathrm{dom}(\Delta_2) \cap \mathrm{FV}(S_1) = \emptyset$.

Example

eval $((\lambda x.x)(\lambda y.y))\ d$

In CLF:

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

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

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:

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

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

Example

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

In CLF:

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

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

Example

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

In CLF:

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

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

Example

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

In CLF:

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

Example

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

In CLF:

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

Example

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

In CLF:

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

Safety

- Safety is the conjunction of the following properties:
  - Preservation: evaluation preserves well-typed states
  - Progress: a well-typed state is either final (result) or is possible to take a step
- Safety for SSOS can be proved by defining a suitable notion of well-typed multiset.
  For example, $\mathsf{eval}\ e_1\ d, \mathsf{eval}\ e_2\ d$ is not well typed.
- Well-typed states can be defined by rewriting rules.
- Well-typed states are generated following the structure of the term.

Safety

- Well-typed states:

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

- Generating well-typed states:

$$\mathsf{gen}\ t\ d \leadsto^* \mathcal{A}$$

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

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:

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

- $\{\Delta\}\,\Sigma^*\,\{\Delta'\}$ 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:

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

Meta-CLF

- The safety proof in Meta-CLF follows closely the paper proof.

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

Meta-CLF

- The safety proof in Meta-CLF follows closely the paper proof.

$$\text{gen}\ t_0\ d_0$$

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

Meta-CLF

- The safety proof in Meta-CLF follows closely the paper proof.

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

Meta-CLF

- The safety proof in Meta-CLF follows closely the paper proof.

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

Meta-CLF

- The safety proof in Meta-CLF follows closely the paper proof.

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

Meta-CLF

- The safety proof in Meta-CLF follows closely the paper proof.

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

- In Meta-CLF:

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

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,...

$$
\begin{aligned}
&\mathsf{location} && : \mathsf{type}.\\
&\mathsf{loc} && : \mathsf{location} \to \mathsf{exp}\,.\\
&\mathsf{get} && : \mathsf{exp} \to \mathsf{exp}\,.\\
&\mathsf{ref} && : \mathsf{exp} \to \mathsf{exp}\,.\\
&\mathsf{set} && : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}\,.
\end{aligned}
$$

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

SSOS

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

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

Conclusions and future work

- 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
  - Coverage checker
  - Termination checker
  - Implementation

Conclusions and future work

- 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
  - Coverage checker
  - Termination checker
  - Implementation

Thank you!