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:

  \[
  A ::= a \cdot S | \Pi x : A.A | A \rightarrow B | A \rightarrowo B | \{ \Delta \}
  \]
  \[
  \Delta ::= \cdot | \Delta, \downarrow x : A | \Delta, !x : A
  \]

  Example: \( A \rightarrowo B \rightarrowo \{ x : B, y : C \} \).
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 $\lambda$-calculus

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

- In (C)LF:

\[
\begin{align*}
\text{exp} : & \text{type}. \\
\text{lam} : & (\text{exp} \to \text{exp}) \to \text{exp}. \\
\text{app} : & \text{exp} \to \text{exp} \to \text{exp}.
\end{align*}
\]
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 d₁ d₂ d` Application: expects the function and argument to be evaluated in `d₁` and `d₂`, and the result is evaluated in `d`

Evaluation rules transform multisets of facts
Multiset of facts:

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

In CLF:

\[
\begin{align*}
\text{dest} : & \text{type.} \\
\text{eval} : & \text{exp} \rightarrow \text{dest} \rightarrow \text{type.} \\
\text{ret} : & \text{exp} \rightarrow \text{dest} \rightarrow \text{type.} \\
\text{fapp} : & \text{dest} \rightarrow \text{dest} \rightarrow \text{dest} \rightarrow \text{type.}
\end{align*}
\]
Evaluation rules

- Multiset rewriting rules:
  \[ \text{eval } e \, d \rightsquigarrow \text{ret } e \, d \quad \text{if } e \text{ is a value} \]

- In CLF:
  \[ \text{step/eval} : \text{eval } e \, d \rightarrow \{ \text{ret } e \, d \} . \]
Evaluation rules

- Multiset rewriting rules:

  \[ \text{eval} \ (e_1 \ e_2) \ d \leadsto \text{eval} \ e_1 \ d_1, \text{eval} \ e_2 \ d_2, \text{fapp} \ d_1 \ d_2 \ d \]

  where \( d_1, d_2 \) fresh

- In CLF:

  \[
  \text{step/app} : \text{eval} \ (\text{app} \ e_1 \ e_2) \ d
  \leadsto \{ !d_1 \ !d_2 : \text{dest},
  \begin{align*}
  x_1 : & \text{eval} \ e_1 \ d_1, \ x_2 : \text{eval} \ e_2 \ d_2, \\
  y : & \text{fapp} \ d_1 \ d_2 \ d
  \end{align*}
  \}.
  \]
Evaluation rules

- **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:**

\[
\text{step/beta : ret } \text{(lam } e_1) d_1 \\
\quad \rightarrow \text{ret } e_2 d_2 \\
\quad \rightarrow \text{fapp } d_1 d_2 d \\
\quad \rightarrow \{ \text{eval } (e_1 e_2) d \}
\]
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:

\[ \varepsilon ::= \emptyset \mid \{ \Delta \} \leftarrow c \cdot S \mid \varepsilon_1 ; \varepsilon_2 \]

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

\[ \Delta \vdash \varepsilon : \Delta' \]
Equality on traces: $\alpha$-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 $\text{dom}(\Delta_1) \cap \text{FV}(S_2) = \text{dom}(\Delta_2) \cap \text{FV}(S_1) = \emptyset$. 
Example

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

In CLF:

(!d : dest)(x_0 : eval (app (lam \lambda x.x) (lam \lambda y.y)) d)
\vdash \diamond

: (!d : dest)(x_0 : eval (app (lam \lambda x.x) (lam \lambda y.y)) d)
Example

\[
eval (((\lambda x.x)(\lambda y.y))\ d) \leadsto eval (\lambda x.x)\ d_1, eval (\lambda y.y)\ d_2, fapp\ d_1\ d_2\ d
\]

In CLF:

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

\[
: (!d, !d_1, !d_2 : dest)(x : eval (lam \lambda x.x)\ d_1)(y : eval (lam \lambda y.y)\ d_2) (z : fapp\ d_1\ d_2\ d)
\]
Example

\[
\text{eval } ((\lambda x.x)(\lambda y.y)) \ d \ \leadsto \ \text{eval } (\lambda x.x) \ d_1, \ \text{eval } (\lambda y.y) \ d_2, \ \text{fapp } d_1 \ d_2 \ d \\
\leadsto \ \text{ret } (\lambda x.x) \ d_1, \ \text{eval } (\lambda y.y) \ d_2, \ \text{fapp } d_1 \ d_2 \ d
\]

In CLF:

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

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

eval ((\(x\).\(x\))(\(y\).\(y\))) \(d\) \(\leadsto\) eval (\(x\).\(x\)) \(d_{1}\), eval (\(y\).\(y\)) \(d_{2}\), fapp \(d_{1}\) \(d_{2}\) \(d\)

\(\leadsto\) ret (\(x\).\(x\)) \(d_{1}\), eval (\(y\).\(y\)) \(d_{2}\), fapp \(d_{1}\) \(d_{2}\) \(d\)

\(\leadsto\) ret (\(x\).\(x\)) \(d_{1}\), ret (\(y\).\(y\)) \(d_{2}\), fapp \(d_{1}\) \(d_{2}\) \(d\)

In CLF:

\(((\!d : \text{dest})(x_{0} : \text{eval (app (lam \(\lambda\) \(x\).\(x\)) (lam \(\lambda\) \(y\).\(y\))) \(d\))))\)

\(\vdash \{!d_{1},!d_{2},x,y,z\} \leftrightarrow \text{step/app } x_{0};\)

\(\{x'\} \leftrightarrow \text{step/eval } x;\)

\(\{y'\} \leftrightarrow \text{step/eval } y;\)

\( : ((\!d,!d_{1},!d_{2} : \text{dest})(x' : \text{ret (lam \(\lambda\) \(x\).\(x\)) \(d_{1}\)})(y' : \text{ret (lam \(\lambda\) \(y\).\(y\)) \(d_{2}\)})\)

\(z : \text{fapp } d_{1} \ d_{2} \ d)\)
Example

\[
eval ((\lambda x.x)(\lambda y.y)) d \rightsquigarrow \eval (\lambda x.x) d_1, \eval (\lambda y.y) d_2, \fapp d_1 d_2 d
\]

\[
\rightsquigarrow \ret (\lambda x.x) d_1, \eval (\lambda y.y) d_2, \fapp d_1 d_2 d
\]

\[
\rightsquigarrow \ret (\lambda x.x) d_1, \ret (\lambda y.y) d_2, \fapp d_1 d_2 d
\]

\[
\rightsquigarrow \eval (\lambda y.y) d
\]

In CLF:

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

\[
\vdash \{!d_1, !d_2, x, y, z\} \leftarrow \text{step/app } x_0;
\]

\[
\{x'\} \leftarrow \text{step/eval } x;
\]

\[
\{y'\} \leftarrow \text{step/eval } y;
\]

\[
\{w\} \leftarrow \text{step/beta } x' y' z;
\]

\[
: (,!d, !d_1, !d_2 : \text{dest})(w : \eval (\text{lam } \lambda y.y) d)
\]
Example

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

In CLF:

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

eval ((\lambda x.x)(\lambda y.y)) \ d \ \leadsto \ eval (\lambda x.x) \ d_1, \ eval (\lambda y.y) \ d_2, \ fapp \ d_1 \ d_2 \ d

\leadsto \ ret (\lambda x.x) \ d_1, \ eval (\lambda y.y) \ d_2, \ fapp \ d_1 \ d_2 \ d

\leadsto \ ret (\lambda x.x) \ d_1, \ ret (\lambda y.y) \ d_2, \ fapp \ d_1 \ d_2 \ d

\leadsto \ eval (\lambda y.y) \ d

\leadsto \ ret (\lambda y.y) \ d

In CLF:

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

\vdash \{!d_1, !d_2, x, y, z\} \leftarrow \text{step/app } x_0;

\{y'\} \leftarrow \text{step/eval } y;

\{x'\} \leftarrow \text{step/eval } x;

\{w\} \leftarrow \text{step/beta } x' \ y' \ z;

\{w'\} \leftarrow \text{step/eval } w;

: (\!d, !d_1, !d_2 : \text{dest})(w' : \text{ret (lam } \lambda y.y) \ d)
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, `eval e_1 d, 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:**

  \[
  \text{gen} : \text{tp} \rightarrow \text{dest} \rightarrow \text{type}.
  \]

  \[
  \text{gen/eval} : \text{gen t d} \circ \text{of e t} \rightarrow \{\text{eval e d}\}.
  \]

  \[
  \text{gen/ret} : \text{gen t d} \circ \text{of e t} \rightarrow \{\text{ret e d}\}.
  \]

  \[
  \text{gen/fapp} : \text{gen t d} \circ \{!d_1 !d_2 : \text{dest},
  \text{fapp }d_1 d_2 d,
  \text{gen (arr t_1 t) }d_1,
  \text{gen }t_1 d_2\}.
  \]

- **Generating well-typed states:**

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

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

Lemma (Safety)

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

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

\[ A ::= \ldots | \{\Delta\} \Sigma^* \{\Delta\} | \{\Delta\} \Sigma^1 \{\Delta\} | \Pi \psi : \text{ctx.} A | \nabla x. A \]

- \{\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:

\[
\text{preservation} : \nabla d. \nabla g. \Pi \psi_1 : \text{ctx.} \Pi \psi_2 : \text{ctx.}
\]

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

\[
\{!d : \text{dest}, g : \text{gen} \ d \ t\} \Sigma_{\text{gen}}^* \{\psi_2\} \rightarrow \text{type.}
\]
The safety proof in Meta-CLF follows closely the paper proof.

\[ A, \text{eval } e \ d \overset{\text{step}}{\Rightarrow} A, \text{ret } e \ d \]
Meta-CLF

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

\[ \text{gen } t_0 \ d_0 \]

\[ \mathcal{A}, \text{eval } e \ d \overset{\text{step}}{\sim} \mathcal{A}, \text{ret } e \ d \]
The safety proof in Meta-CLF follows closely the paper proof.

\[
\text{gen } t_0 \ d_0 \\
\Downarrow^* \\
\text{gen} \\
\mathcal{A}, \text{gen } t \ d \\
\Downarrow \\
\text{gen} \\
\mathcal{A}, \text{eval } e \ d \xrightarrow{\text{step}} \mathcal{A}, \text{ret } e \ d
\]
The safety proof in Meta-CLF follows closely the paper proof.

\[
\begin{align*}
gen t_0 d_0 \quad &\quad gen t_0 d_0 \\
\downarrow_{\mathcal{A}, gen t d} \quad &\quad \uparrow_{\mathcal{A}, gen}\ \\
\mathcal{A}, eval e d &\rightsquigarrow_{\text{step}} \mathcal{A}, ret e d
\end{align*}
\]
The safety proof in Meta-CLF follows closely the paper proof.

\[
\begin{align*}
g & \vdash_{gen} \ast \\
A, \, \text{gen } t \, d & \vdash_{gen} \ast \\
A, \, \text{eval } e \, d \quad & \rightsquigarrow_{\text{step}} \\
A, \, \text{ret } e \, d
\end{align*}
\]
The safety proof in Meta-CLF follows closely the paper proof.

\[
\begin{align*}
\text{gen } t_0 \ d_0 & \quad \text{gen } t_0 \ d_0 \\
\xrightarrow{\text{gen}^*} & \quad \xrightarrow{\text{gen}^*} \\
\mathcal{A}, \text{gen } t \ d & \quad \mathcal{A}\text{.gen } t \ d \\
\xleftarrow{\text{gen}} & \quad \xleftarrow{\text{gen}} \\
\mathcal{A}, \text{eval } e \ d & \quad \mathcal{A}, \text{ret } e \ d
\end{align*}
\]

In Meta-CLF:

\[
\begin{align*}
\text{tpres/ret} : \text{tpres } (X_1; \{\downarrow x\} \leftarrow \text{gen/eval } e \ d_0 \ g_0 \ H) \\
(\{\downarrow y\} \leftarrow \text{step/eval } e \ d_0 \times H_{\nu}) \\
(X_1; \{\downarrow y\} \leftarrow \text{gen/ret } e \ d_0 \ g_0 \ H \ H_{\nu})
\end{align*}
\]
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.)
We can extend this semantics with other features without invalidating the previous rules

Example: store, futures, call/cc, communication,…

location : type.
loc : location → exp.
get : exp → exp.
ref : exp → exp.
set : exp → exp → exp.

cell : location → exp → type.
step/ref : eval (ref e) d ↦ { !d_1 : dest, !l : loc,
                       fref d_1 l, eval e d_1, ret (loc l) d}.
step/fref : ret e d ↦ fref d l ↦ {cell l e}.
We can extend this semantics with other features without invalidating the previous rules

Example: store, futures, call/cc, communication, ...

\[
\begin{align*}
\text{future} & : \text{exp} \to \text{exp} . \\
\text{promise} & : \text{dest} \to \text{exp} . \\
\text{deliver} & : \text{exp} \to \text{dest} \to \text{type}. \\
\text{step/fut} & : \text{eval} \ (\text{future} \ e) \ d \to \{!d_1 : \text{dest}, \\
& \quad \text{eval} \ e \ d_1, \text{fdel} \ d_1, \\
& \quad \text{ret} \ (\text{promise} \ d_1) \ d\}. \\
\text{step/fdel} & : \text{ret} \ e \ d \to \text{fdel} \ d_1 \to \{!\text{deliver} \ e \ d\}. \\
\text{step/promise} & : \text{ret} \ (\text{promise} \ d_1) \ d \to \text{delivee} \ e \ d_1 \to \text{ret} \ e \ d.
\end{align*}
\]
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!