CLF: A logical framework for concurrent systems

Thesis Proposal

Kevin Watkins

Carnegie Mellon University

Committee:
Frank Pfenning, CMU (Chair)
Stephen Brookes, CMU
Robert Harper, CMU
Gordon Plotkin, University of Edinburgh
Thesis: CLF enables succinct and straightforward specification and implementation of concurrent systems

CLF = Concurrent Logical Framework

Developed jointly with Iliano Cervesato, Frank Pfenning, and David Walker
What is a logical framework?

Generic, mechanizable system for specifying, computing with, and reasoning about deductive systems.

Consists of:

- **Language** based on logical formulas
- **Principles** for representing systems of interest
- **Algorithms** for mechanically manipulating the language

Applications:

- Logics
- Programming languages
What is a logical framework?

Basic idea:

- Specify at high level using *logical connectives*
- Find powerful connectives in logics richer than classical: *intuitionistic logic, linear logic, lax logic*

Why based on logic?

- Conceptually uniform (*same language* for specification and reasoning)
- Generic
- Long history (most studied kind of formal system)
Outline

- The LF logical framework
  - Modeling judgments and deductions
- Linear logic
  - Modeling state
- The CLF framework
  - Monadic type
  - Modeling concurrency
- Thesis statement
- Research plan
The LF logical framework

CLF extends LF = Logical Framework [Harper, Honsell, Plotkin 1987]

LF based on type theory:

- Syntax and deductions unified as objects
- Correctness of objects specified by types
- Type language based on intuitionistic logic
The LF logical framework

Why explicit objects for proofs?

- Meta-reasoning
- Applications (e.g. proof-carrying code)
- Reliability

Why types?

- Type checking is \textit{decidable}
- Type checking algorithm is \textit{efficient}
- Well-typed objects automatically \textit{compose}
- Proof checking = type checking!
Representing deductive systems in LF

Deductive system terminology:

- **Judgment** = statement subject to proof
- **Examples:**
  - “The proposition $A$ is true.”
  - “The expression $e$ evaluates to value $v$.”
  - “The principal $P$ knows secret key $\kappa$.”
- **Deduction** = object containing evidence of a judgment (tree of inferences)

\[
\frac{A \text{ true} \quad B \text{ true}}{A \land B \text{ true}} \quad \frac{e_1 \downarrow \text{ true} \quad e_2 \downarrow v}{(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \downarrow v}
\]
Representing deductive systems in LF

Formalize system of interest:

- Syntax
- Judgments
- Allowed rules of deduction

Formulate *representation function* mapping to LF:

- Syntax becomes LF objects
- Judgments become LF types
- Deductions become LF objects
Representing deductive systems in LF

Syntax built from LF constants having function types:

\[
\begin{align*}
\text{and} & : \ prop \to prop \to prop \\
\text{ite} & : \ exp \to exp \to exp \to exp \\
\end{align*}
\]

\[
\begin{align*}
\lnot A \land B & \equiv \text{and}(\lnot A, \lnot B) \\
\lnot \text{if } e_1 \text{ then } e_2 \text{ else } e_3 & \equiv \text{ite}(\lnot e_1, \lnot e_2, \lnot e_3) \\
\end{align*}
\]

Judgments become *dependent types* referring to particular objects. Examples:

\[
\begin{align*}
\text{true} & : \ prop \to \text{type} \\
\text{eval} & : \ exp \to \text{val} \to \text{type} \\
\text{knows} & : \ prin \to \text{key} \to \text{type} \\
\end{align*}
\]
Representing deductive systems in LF

Rules of inference become LF constants having function types:

\[ \land_1 : \text{true}(A) \rightarrow \text{true}(B) \rightarrow \text{true}(\text{and}(A, B)) \]
\[ \text{if\_true} : \text{eval}(e_1, \text{true}) \rightarrow \text{eval}(e_2, v) \rightarrow \text{eval}(\text{ite}(e_1, e_2, e_3), v) \]

Deductions mapped to compositions of these constants
Representing deductive systems in LF

Adequacy theorem:

- Bijection between syntax of system and LF objects of proper type
- Bijection between deductions of system and LF objects of proper type

LF features make this easier:

- Model variable binding with LF function types
- Model capture-avoiding substitution with LF function application
Language:
  • Dependent type theory
  • Based on intuitionistic logic

Representation principles:
  • Judgments as types
  • Deductions as objects

Algorithms:
  • Type checking (= proof checking)
  • More . . .
Example: locking protocol

Model trivial locking protocol

- Multiple threads $t_1, \ldots, t_n$
- Multiple locks $l_1, \ldots, l_n$
- Each thread runs program
- Program = sequence of instructions
  - lock($l$)
  - unlock($l$)

Other details suppressed
Example: locking protocol

LF types for threads, locks, programs:

thread : type
lock : type
program : type

LF objects for programs:

exit : program
do_lock : lock → program → program
do_unlock : lock → program → program

Example: do_lock(l, do_unlock(l, exit)) has type program
Example: locking protocol

Great, we modeled the syntax. But how to model execution?

Need to model *state*:
- What program is each thread running?
- Which locks are locked?

Could introduce more syntax for states . . .

Better answer: extend the logic underlying LF
Linear logic

CLF related to Dual Intuitionistic Linear Logic (DILL) [Hodas, Miller 1994; Barber 1996]

“Dual” meaning two kinds of hypotheses:

- *Unrestricted* hypotheses
  - Can use more than once
  - Or not at all
- *Linear* hypotheses
  - Must use *exactly once*

Think linear hypotheses = *resources*
Unrestricted hypotheses already available via LF function type $A \rightarrow B$

New connectives:

- **Linear implication** $A \multimap B$ (create linear hypothesis = resource)
- **Multiplicative conjunction** $A \otimes B$ (join resources)
- **Multiplicative unit** $1$ (empty set of resources)
- More . . .
Unrestricted examples:

- More than once: $A \rightarrow A \otimes A$
- Not at all: $A \rightarrow 1$

Linear examples:

- Okay: $A \otimes B \rightarrow B \otimes A$
- No! $A \rightarrow A \otimes A$
- No! $A \rightarrow 1$
Representing state via linear logic

Richer logic allows simpler modeling of state

- State = set of linear hypotheses (resources)
- Inference rules modify state
  - Consume resources
  - Introduce new resources
New judgments (= types) for state:

- unlocked : lock $\rightarrow$ type
- locked : lock $\rightarrow$ thread $\rightarrow$ type
- run : thread $\rightarrow$ program $\rightarrow$ type

Inference rules modify state:

- $\text{run}(t, \text{exit}) \triangleright 1$
- $\text{run}(t, \text{do\_lock}(l, p)) \otimes \text{unlocked}(l) \triangleright \text{run}(t, p) \otimes \text{locked}(l, t)$
- $\text{run}(t, \text{do\_unlock}(l, p)) \otimes \text{locked}(l, t) \triangleright \text{run}(t, p) \otimes \text{unlocked}(l)$
Representing state via linear logic

Modeling reachability:

- Initialize with linear hypotheses for starting state
- Final state reachable iff there is a deduction of it

Not yet a type theory

- What about deductions-as-objects?
- Want \textit{bijection} between deductions and executions
- More precise than reachability
Linear logic as a framework

Prior work: Linear Logical Framework (LLF) [Cervesato, Pfenning 1996]

- Has unrestricted and linear hypotheses
- Has unrestricted and linear implication: \( \rightarrow \) and \( \leftarrow \)
- Also more connectives not discussed here: \( \Pi, \&, \top \)
- No *synchronous connectives*: \( \otimes, 1, \oplus, 0, !, \exists \)
Linear logic as a framework

No $\otimes$, 1 available in LLF

Instead use “continuation-passing style”:

- DILL style:
  \[
  \text{run}(t, \text{do\_lock}(l, p)) \otimes \text{unlocked}(l) \rightarrow \circ \text{run}(t, p) \otimes \text{locked}(l, t)
  \]

- LLF style:
  \[
  (\text{run}(t, p) \rightarrow \circ \text{locked}(l, t) \rightarrow \circ g) \rightarrow \circ
  \]
  \[
  (\text{run}(t, \text{do\_lock}(l, p)) \rightarrow \circ \text{unlocked}(l) \rightarrow \circ g)
  \]

Problem: CPS \textit{sequentializes} execution

- Too few deductions equal
- Proof search not concurrent

No good for concurrent systems!
Linear logic as a framework

Why no synchronous connectives?

- $\otimes$, $1$, $\oplus$, $0$, $!$, $\exists$ involve let-style elimination forms
- *Commuting conversions* push let bindings around
- Example: $(\text{let } y_1 \otimes y_2 = x \text{ in } M_1) \ M_2$ versus
  $\text{let } y_1 \otimes y_2 = x \text{ in } (M_1 \ M_2)$

No obvious way to define canonical forms

- LLF solution: rule out synchronous connectives
- CLF solution: *segregate* $\otimes$, $1$, $!$, $\exists$ using *Monad*
Segregate more restrictive from less restrictive language

Prior work:

- Segregate *effectful* from *non-effectful* computations in functional programming [Moggi 1988]
- Logical view: *lax logic* [Benton, Bierman, de Paiva 1998]
- Judgmental view [Pfenning, Davies 2000]

Two kinds of judgments:

- “A true”: can prove $A$ in *more* restrictive language
- “A lax”: can prove $A$ in *less* restrictive language
New monadic type constructor \{\textendash\}

Moving between judgments:

- If “\(A\) true” holds, then “\(A\) lax” holds
- If “\(A\) lax” holds, then “\{\(A\)\} true” holds

CLF idea: confine \(\otimes\), \(\top\), \(!\), \(\exists\) to lax judgment

Syntactic restriction on types

- From this: \(A \otimes 1 \otimes !B \rightarrow C \otimes 1 \otimes !D\)
- To this: \(A \rightarrow B \rightarrow \{C \otimes 1 \otimes !D\}\)
Example revisited

DILL style:

\[ \text{run}(t, \text{exit}) \rightarrow 1 \]
\[ \text{run}(t, \text{do\_lock}(l, p)) \otimes \text{unlocked}(l) \rightarrow \text{run}(t, p) \otimes \text{locked}(l, t) \]
\[ \text{run}(t, \text{do\_unlock}(l, p)) \otimes \text{locked}(l, t) \rightarrow \text{run}(t, p) \otimes \text{unlocked}(l) \]

CLF style:

\[ \text{run}(t, \text{exit}) \rightarrow \{1\} \]
\[ \text{run}(t, \text{do\_lock}(l, p)) \rightarrow \text{unlocked}(l) \rightarrow \{\text{run}(t, p) \otimes \text{locked}(l, t)\} \]
\[ \text{run}(t, \text{do\_unlock}(l, p)) \rightarrow \text{locked}(l, t) \rightarrow \{\text{run}(t, p) \otimes \text{unlocked}(l)\} \]
Types:

Atomic \( P ::= a \mid P \; N \)

Asynch \( A ::= P \mid \Pi x : A. \; A \mid A \rightarrow A \mid A \& A \mid \top \mid \{ S \} \)

Synch \( S ::= S \otimes S \mid 1 \mid \exists x : A. \; S \mid !A \mid A \)

\((A \rightarrow B \text{ special case of } \Pi x : A. \; B)\)
Objects (only canonical forms):

**Atomic**  
\[ R ::= c \mid x \mid R \, N \mid R^\top \, N \mid \pi_1 \, R \mid \pi_2 \, R \]

**Normal**  
\[ N ::= R \mid \lambda x. \, N \mid \hat{\lambda} \, x. \, N \mid \langle N, N \rangle \mid \langle \rangle \mid \{ E \} \]

**Expr**  
\[ E ::= \text{let } \{ p \} = R \text{ in } E \mid M \]

**Monadic**  
\[ M ::= M \otimes M \mid 1 \mid [N, M] \mid !N \mid N \]

**Pattern**  
\[ p ::= p \otimes p \mid 1 \mid [x, p] \mid !x \mid x \]

Truth judgment: atomic objects, normal objects, monadic objects

Lax judgment: expressions
Monad eliminates commutative conversions

- Example: \( \{ \text{let } \{ y_1 \otimes y_2 \} = x \text{ in } M_1 \} M_2 \} \) ruled out by judgments
- Example: \( \{ \text{let } \{ y_1 \otimes y_2 \} = x \text{ in } M_1 \} M_2 \) not well typed
- Must have \( \{ \text{let } \{ y_1 \otimes y_2 \} = x \text{ in } (M_1 M_2) \} \)

Still have *permutative conversions* inside expressions

- Example: \( \{ \text{let } \{ p_1 \} = R_1 \text{ in let } \{ p_2 \} = R_2 \text{ in } E \} \) versus \( \{ \text{let } \{ p_2 \} = R_2 \text{ in let } \{ p_1 \} = R_1 \text{ in } E \} \)

- *Equal objects* in CLF (presuming variables don’t get detached from their bindings)
CLF type theory

CLF equality on objects given by:

- $\alpha$-conversion
- Permutative conversions

Also need *instantiation algorithm* to compute canonical forms while typing

Payoff:

- $\alpha$-conversion models variable binding
- Instantiation algorithm models capture-avoiding substitution
- New: Permutative conversions model concurrency!
Basic idea:

- Concurrent execution becomes sequence of `let` bindings
- Independent computation steps are `let` bindings with no common linear variables
- Because of permutative conversions, can’t observe order in which independent computation steps occur

More details in proposal document

Still need to axiomatize more sophisticated relations (e.g. $\pi$-calculus bisimulation)
Language:
- Dependent type theory
- New: Based on linear logic plus monad

Representation principles:
- Deductions are objects
- Judgments are types
- State as linear hypotheses
- New: Concurrent computations are monadic expressions

Conservatively extends LF and LLF
Thesis: CLF enables succinct and straightforward specification and implementation of concurrent systems
In detail:

- Succinct: don’t have to reason explicitly about serializations
- Straightforward: just add monad brackets to your DILL formulas
- Analogy: in LF, don’t have to reason about variable binding

Not only interested in specification; must be possible to create mechanized tools for computing with and reasoning about specifications
Completed work:

- [Definition of CLF]
- Theory of CLF
- Example specifications

Proposed work:

- Framework extensions
- Semantics of proof search
- Tools
Theory of CLF

Key points (see proposal document):

- Includes all connectives of DILL except $\oplus$, 0 (future work)
- Conservatively extends LF and LLF
- New presentation of LF restricts to canonical forms
  - No redices allowed
  - *Instantiation* algorithm works on ill-typed objects
  - No mutual dependence of equality and typing
Instantiation and typing:

\[
\frac{\Gamma \vdash R \Rightarrow \Pi x : A. \ B \quad \Gamma \vdash N \leftarrow A}{\Gamma \vdash R \ N \Rightarrow \text{inst}_a A \ (x. \ B, \ N)} \quad \text{ΠE}
\]

Example:

\[
\text{inst}_a a \rightarrow a (x. \ b \ (\lambda y. \ c \ (x \ (x \ y))), \ \lambda z. \ d \ z) \equiv b \ (\lambda y. \ c \ (d \ (d \ y)))
\]
Example specifications

Already done:

- Petri nets
- The $\pi$-calculus [Milner]
- ML with references, suspensions, futures, concurrency à la Concurrent ML [Reppy]

Future work:

- MSR (security protocols) [Cervesato]
- Forum [Miller]
- Action calculi [Milner]
Framework extensions

Full DILL language: \((\text{add } \oplus, 0)\)

- Which equality is right? (need more examples)

Syntactic extensions:

- Notational definitions
- Explicit substitutions

More judgments:

- Ordered hypotheses [Polakow]
- Proof irrelevance [Pfenning]
Proof search

Prior work: Elf language [Pfenning 1994]
- Interpret LF specification as logic program
- Operational semantics for proof search
- Generalizes Prolog
- Requires unification algorithm

New issues for CLF:
- Non-determinism associated with concurrency
- Linear unification algorithm (prior work: pre-unification [Cervesato, Pfenning 1997])
Key algorithms:

- Type-checking
- Type reconstruction
- Proof search

Prior work: Twelf system [Pfenning et al.]
Research plan

First:

- Implement checker
- More example specifications

Informed by examples:

- Framework extensions
- Semantics of proof search
- Implement search (restricted unification) and experiment
If time permits:

- Full unification
- Methods of representing meta-proofs
Natural progression:

- **LF**: judgments as types, deductions as objects
  - Internalizes $\alpha$-conversion, capture-avoiding substitution
- **LLF**: state as linear hypotheses
  - Internalizes state
- **CLF**: concurrent computations as monadic expressions
  - Internalizes concurrent equality