Substructural Operational Semantics and Linear Destination-Passing Style

Frank Pfenning

Carnegie Mellon University

Invited Talk

Second Asian Symposium on Programming Languages and Systems (APLAS'04)

Taipei, Taiwan, November 5, 2004

Motivation

- Programming languages and new features are still designed and implemented, for example:
  - Spatial computation
  - Information flow
  - Probabilistic computation
  - Typed assembly language
  - Many more . . .
- Need effective means to specify languages
  - Type systems
  - Operational semantics

Goals

- Specification framework for operational semantics
  - Permit varying levels of abstraction
  - Support modularity
  - Rest on logical foundation
- Framework roles
  - Conceptual clarity and simplicity
  - Prototyping
  - Reasoning about languages

Dimensions of Operational Semantics

- Language features
  - Functions, pairs, recursion, references, exceptions, callcc, memoization, concurrency, parallelism, etc.
- Levels of abstraction
  - Substitution vs. environments
  - Large values vs. heaps
  - Term structure vs. stacks vs. continuations
  - Mobile terms vs. marshalling

Modularity in Operational Semantics

- Modularity for language features
  - Simple specification for simple features
  - Extend without rewriting earlier specification
- Some standard formalisms are non-modular
  - Structural operational semantics [Plotkin’81]
  - Natural semantics [Kahn’87]
- Some recent approaches
  - Modular structural operational semantics [Mosses’98,’04]
  - Modular rewriting semantics [Meseguer & Braga’03]

Outline

- Call-by-value λ-calculus
- Linear destination passing
- Call-with-current-continuation
- Call-by-need λ-calculus
- Synchronous π-calculus
- Substructural lax logic
- Related and future work
- Summary

Call-by-Value λ-Calculus

- Pure call-by-value λ-calculus
- Types are integral part of language definition
  - Ignored here to concentrate on operational aspects
- Variables x
- Expressions $e ::= x \mid \mathsf{lam}(x.e) \mid \mathsf{app}(e_1, e_2)$
- Values $v ::= \mathsf{lam}(x.e)$
- Binding $x.e$ and substitution $[e_1/x]e_2$
- Tacit renaming of bound variables

Computation as Proof Construction

- Atomic judgments
  - $\mathsf{eval}(e)$ evaluate e *active*
  - $\mathsf{ret}(v)$ return v *passive*
- Computation as bottom-up proof construction

$$\begin{array}{c}\mathsf{ret}(v)\\ \vdots \\ \mathsf{eval}(e)\end{array}$$

- Partial proofs as intermediate states
- Consider only single-premise rules (for now)

Rules of Computation

- Introduce frames as intermediate objects
  - $\mathsf{comp}(f)$ compute frame f *suspended*
- State is *ordered* $J_1 \cdots J_n$ (for now)
- Apply rules anywhere in state

$$\frac{\mathsf{ret}(\mathsf{lam}(x.e))}{\mathsf{eval}(\mathsf{lam}(x.e))} \qquad \frac{\mathsf{eval}(e_1) \cdot \mathsf{comp}(\mathsf{app}(\Box, e_2))}{\mathsf{eval}(\mathsf{app}(e_1, e_2))}$$

$$\frac{\mathsf{eval}(e_2) \cdot \mathsf{comp}(\mathsf{app}(v_1, \Box))}{\mathsf{ret}(v_1) \cdot \mathsf{comp}(\mathsf{app}(\Box, e_2))} \qquad \frac{\mathsf{eval}([v_2/x]e_1')}{\mathsf{ret}(v_2) \cdot \mathsf{comp}(\mathsf{app}(\mathsf{lam}(x.e_1'), \Box))}$$

Sample Computation

- Will always work on left end (by invariant)
- Example computation

$$
\begin{array}{c}
\mathsf{ret}(\mathsf{lam}(y.y)) \\ \hline
\mathsf{eval}(\mathsf{lam}(y.y)) \\ \hline
\mathsf{ret}(\mathsf{lam}(y.y)) \cdot \mathsf{comp}(\mathsf{app}(\mathsf{lam}(x.x), \Box)) \\ \hline
\mathsf{eval}(\mathsf{lam}(y.y)) \cdot \mathsf{comp}(\mathsf{app}(\mathsf{lam}(x.x), \Box)) \\ \hline
\mathsf{ret}(\mathsf{lam}(x.x)) \cdot \mathsf{comp}(\mathsf{app}(\Box, \mathsf{lam}(y.y))) \\ \hline
\mathsf{eval}(\mathsf{lam}(x.x)) \cdot \mathsf{comp}(\mathsf{app}(\Box, \mathsf{lam}(y.y))) \\ \hline
\mathsf{eval}(\mathsf{app}(\mathsf{lam}(x.x), \mathsf{lam}(y.y)))
\end{array}
$$

Parallel Application

- Evaluate function and argument in “parallel”
- Judgment order guarantees proper interaction

$$\frac{\mathsf{ret}(\mathsf{lam}(x.e))}{\mathsf{eval}(\mathsf{lam}(x.e))} \qquad \frac{\mathsf{eval}(e_1) \cdot \mathsf{eval}(e_2) \cdot \mathsf{comp}(\mathsf{app}(\Box, \Box))}{\mathsf{eval}(\mathsf{app}(e_1, e_2))}$$

$$\frac{\mathsf{eval}([v_2/x]e_1')}{\mathsf{ret}(\mathsf{lam}(x.e_1')) \cdot \mathsf{ret}(v_2) \cdot \mathsf{comp}(\mathsf{app}(\Box, \Box))}$$

- Any interleaving of steps is valid

Assessment

- Sequential languages and simple control constructs are easy
  - Pairs, sums, recursion, polymorphism, exceptions, etc.
  - Modular for these features
- Complex control and semantic objects are difficult
  - Callcc, communication
  - Mutable store, heap, multiple hosts
- Currently still under investigation

Generalization to Linear Judgments

- Permit exchange among judgments in state
  - Atomic judgments eval, ret, and comp are now linear
- Write $J_1\ ,\ \ldots\ ,\ J_n$
- Now a value could be returned to the wrong frame!
- Introduce destinations d (names)
  - $e \mapsto d$ evaluate e with destination d *active*
  - $d{=}v$ return v to destination d *passive*
  - $f \rightarrowtail d$ compute f with destination d *suspended*

Linear Destination-Passing Style

- Functions

$$\frac{d{=}\mathsf{lam}(x.e)}{\mathsf{lam}(x.e) \mapsto d} \qquad \frac{e_1 \mapsto d_1 \ ,\ \mathsf{app}(d_1, e_2) \rightarrowtail d}{\mathsf{app}(e_1, e_2) \mapsto d}[d_1]$$

$$\frac{e_2 \mapsto d_2 \ ,\ \mathsf{app}(v_1, d_2) \rightarrowtail d}{d_1{=}v_1 \ ,\ \mathsf{app}(d_1, e_2) \rightarrowtail d}[d_2] \qquad \frac{[v_2/x]e_1' \mapsto d}{d_2{=}v_2 \ ,\ \mathsf{app}(\mathsf{lam}(x.e_1'), d_2) \rightarrowtail d}$$

- New destinations indicated by $[d]$

Overall Computation

- Overall computation now has form

$$\begin{array}{c} d_0{=}v \\ \vdots \\ e \mapsto d_0 \end{array}$$

- All evaluation ($e \mapsto d$), return ($d{=}v$) and computation ($f \rightarrowtail d$) judgments are linear
  - Remove matching linear judgments from the state
  - None (except $d_0{=}v$) may be left at the end

Call-With-Current-Continuation

- callcc$(x.e_1)$ binds x to the current continuation then evaluates e_1
- throw(e_1, e_2) evaluates e_1 to a continuation and throws the value of e_2 to it
- Use destination d to represent continuation
- May return to a destination more than once

Unrestricted Judgments

- To model callcc, either
  - Explicitly copy and delete frames (more modular)
  - Make all frames unrestricted (more direct)
- Mirrors real implementation choices
- Unrestricted judgments
  - Not consumed when used (may be reused)
  - May be left at the end

Callcc, Ctd.

- All frames are unrestricted

$$\frac{[\mathsf{cont}(d)/x]e_1 \mapsto d}{\mathsf{callcc}(x.e_1) \mapsto d} \qquad \frac{d{=}\mathsf{cont}(d_2)}{\mathsf{cont}(d_2) \mapsto d}$$

$$\frac{e_1 \mapsto d_1 \;,\; \mathsf{throw}(d_1, e_2) \rightarrowtail d}{\mathsf{throw}(e_1, e_2) \mapsto d}\,[d_1]$$

$$\frac{e_2 \mapsto d_2}{d_1{=}\mathsf{cont}(d_2) \;,\; \mathsf{throw}(d_1, e_2) \rightarrowtail d}$$

- $e \mapsto d$ and $d{=}v$ remain linear

Overall Computation

- Frames remain at end of computation

$$\begin{array}{c} d_0{=}v\ ,\ f_i \rightarrowtail d_i \\ \vdots \\ e \mapsto d_0 \end{array}$$

- $f_i \rightarrowtail d_i$ represents all frames created during computation
- It is possible to model garbage collection

Call-by-Need

- Alternative semantics for λ-calculus
  - Delay argument evaluation in *thunk*
  - Force evaluation of thunk when value is needed
  - Memoize value to avoid future computation
- Model thunks as destinations t with delay frame
- Thunk is affine (it may never be forced)
  - May be left at the end
- Memoized value is unrestricted
  - May be referenced many times, left at the end

Call-by-Need, Ctd.

- Thunks as destinations t (eliding one rule)

$$\frac{e_1 \mapsto d_1 \ , \ \mathsf{app}(d_1, e_2) \rightarrowtail d}{\mathsf{app}(e_1, e_2) \mapsto d} [d_1]$$

$$\frac{\mathsf{delay}(e_2) \rightarrowtail t_2 \ , \ [\mathsf{thunk}(t_2)/x]e_1' \mapsto d}{d_1{=}\mathsf{lam}(x.e_1') \ , \ \mathsf{app}(d_1, e_2) \rightarrowtail d} [t_2]$$

$$\frac{e_2 \mapsto t_2 \ , \ \mathsf{thunk}(t_2) \rightarrowtail d}{\mathsf{delay}(e_2) \rightarrowtail t_2 \ , \ \mathsf{thunk}(t_2) \mapsto d} \qquad \frac{t_2{=}v_2 \ , \ d{=}v_2}{t_2{=}v_2 \ , \ \mathsf{thunk}(t_2) \rightarrowtail d}$$

- $\mathsf{delay}(e_2) \rightarrowtail t_2$ is affine, $t_2{=}v_2$ unrestricted

Overall Computation

- Overall computation now has form

$$d_0{=}v \;,\; \mathsf{delay}(e_i) \rightarrowtail t_i \;,\; t_j{=}v_j$$
$$\vdots$$
$$e \mapsto d_0$$

- $\mathsf{delay}(e_i) \rightarrowtail t_i$ denote unforced thunks
- $t_j{=}v_j$ denotes evaluated thunks

Assessment

- Computation as proof construction
  - Logical interpretation later in this talk
- Ordered, linear, affine, unrestricted judgments
- Rules are applied to parts of state
  - Modular extension by new constructs
  - Can model varying levels of abstraction

Concurrency

- Exemplify with synchronous π-calculus
- Structural congruences as properties of state
  - Linearity for ordinary processes
  - Unrestricted judgment for process replication
- Asynchronous π-calculus is simpler
- Concurrency can be added modularly (á la CML or JO'Caml)

π-Calculus

- Names a, b, bound variables x, y
- Syntax

$$\begin{array}{llll} \text{Procs} & P & ::= & (P_1|P_2) \mid 0 \mid \text{new}\, x.P \mid !P \mid M \\ \text{Sums} & M & ::= & \bar{a}\langle b\rangle.P \mid a(x).P \mid M_1 + M_2 \end{array}$$

- Elide silent action for brevity

Judgments

- $\mathsf{proc}(P)$ computing process P
- $\mathsf{sync}(M, N, P, Q)$ synch M and N yields P and Q
  - M and N are sums from which communicating processes are selected non-deterministically
  - Auxiliary judgment has separate proof system

Synchronization

- Separate logical rules for synchronization

$$\frac{\mathsf{sync}(M_1, N, P, Q)}{\mathsf{sync}(M_1 + M_2, N, P, Q)} \qquad \frac{\mathsf{sync}(M_2, N, P, Q)}{\mathsf{sync}(M_1 + M_2, N, P, Q)}$$

$$\frac{}{\mathsf{sync}(\bar{a}\langle b\rangle.P, a(x).Q, P, [b/x]Q)}$$

- Elide three symmetric rules
- These rules are *don't-know non-deterministic*

Process Expressions

- Process evolution is *don't-care non-deterministic*

$$\frac{\mathsf{proc}(P)\ ,\ \mathsf{proc}(Q)}{\mathsf{proc}(P|Q)} \qquad \frac{.}{\mathsf{proc}(0)}$$

$$\frac{\mathsf{proc}([a/x]P)}{\mathsf{proc}(\mathsf{new}\, x.P)}\,[a] \qquad \frac{\mathsf{proc}(P)}{\mathsf{proc}(!P)}$$

- Form $\mathsf{proc}(P)$ is unrestricted form of $\mathsf{proc}(P)$
- No explicit structural congruences required

Communication

- Communication is trickiest part of specification

$$\frac{[\ \mathsf{sync}(M, N, P, Q)\ ] \qquad \mathsf{proc}(P)\ ,\ \mathsf{proc}(Q)}{\mathsf{proc}(M)\ ,\ \mathsf{proc}(N)}$$

- Left premise appears “special” (don’t-know ndt)
- Right premise is main branch (don’t-care ndt)

Overall Computation

- Computation may not terminate
- Model state evolution, not value-oriented computation

$$\vdots$$
$$\mathsf{proc}(P)$$

- State is terminal if there is no possible transition
  - Either has no linear processes (“terminated”)
  - Or has linear processes (“deadlocked”)

A Logical Reading: Challenges

- Ordered, linear, affine, unrestricted judgments
- Introduction of new destinations (parameters)
- Main branch vs. auxiliary judgments
- Don't-care vs. don't-know interpretation of rules
- Bijection between computations and proofs
- Tractable proof theory

Substructural Logics

- Hypotheses subject to usage constraints
  - Exchange (x), weakening (w), contraction (c)
- Forms of substructural hypothetical judgments
  - Ordered: $J_1 \cdots J_n \vdash J$
  - Linear (x): $J_1 \ , \ldots , \ J_n \vdash J$
  - Affine (x, w): $J_1 \ , \ldots , \ J_n \vdash J$
  - Unrestricted (x, w, c): $J_1 \ , \ldots , \ J_n \vdash J$
- Different forms of hypotheses may be mixed
- Modal operators promote

Representation of State

- State is translated to *hypotheses*
- Conclusion provable in final state, for example:

$$
\begin{array}{c} d{=}v \\ \vdots \\ e \mapsto d \end{array}
\quad \text{to} \quad
\begin{array}{c} \dfrac{\dfrac{}{d{=}v \vdash d{=}v}\,\text{hyp}}{d{=}v \vdash \exists x.\ d{=}x}\,\exists\text{I} \\ \vdots \\ e \mapsto d \vdash \exists x.\ d{=}x \end{array}
$$

- Proof must respect structural properties

Logical Expression of Rules, 1st Try

- Use linear logic, for example:

$$\frac{d{=}\mathsf{lam}(x.e)}{\mathsf{lam}(x.e) \mapsto d} \qquad \frac{e_1 \mapsto d_1 \;,\; \mathsf{app}(d_1, e_2) \rightarrowtail d}{\mathsf{app}(e_1, e_2) \mapsto d}[d_1]$$

$$\mathsf{eval}\ (\mathsf{lam}\ (\lambda x.\ e\ x))\ d \qquad \mathsf{eval}\ (\mathsf{app}\ e_1\ e_2)\ d$$

$$\multimap \mathsf{ret}\ (\mathsf{lam}\ (\lambda x.\ e\ x))\ d \qquad \multimap \exists d_1.\ \mathsf{eval}\ e_1\ d_1 \otimes$$

$$\mathsf{comp}\ (\mathsf{app}\ d_1\ e_2)\ d$$

- Free variables implicitly universally quantified
- Existential quantification generates new names

Assessment

- First try satisfies:
  - Ordered, linear, affine, unrestricted judgments
  - Introduction of new destinations (parameters)
- First try does not satisfy:
  - Main branch vs. auxiliary judgments
  - Don’t-care vs. don’t-know interpretation of rules
  - Bijection between computations and proofs
  - Tractable proof theory

Lax Substructural Logic

- Solution: introduce monad $\{A\}$
- Logical foundation is lax logic
  [Fairtlough & Mendler'95] [Pf. & Davies'01]
- New judgment form $\Delta \vdash A$ lax
- Hypotheses Δ here substructural
- Rules

$$\frac{\Delta \vdash A \text{ true}}{\Delta \vdash A \text{ lax}} \qquad \frac{\Delta \vdash A \text{ lax}}{\Delta \vdash \{A\} \text{ true}}$$

$$\frac{\Delta \vdash \{A\} \text{ true} \quad \Delta_L \cdot A \text{ true} \cdot \Delta_R \vdash C \text{ lax}}{\Delta_L \cdot \Delta \cdot \Delta_R \vdash C \text{ lax}}$$

Logical Expression of Rules, 2nd Try

- Transitions take place in monad

$$\frac{d{=}\mathsf{lam}(x.e)}{\mathsf{lam}(x.e) \mapsto d} \qquad \frac{e_1 \mapsto d_1 \;,\; \mathsf{app}(d_1, e_2) \rightarrowtail d}{\mathsf{app}(e_1, e_2) \mapsto d}[d_1]$$

$$\mathsf{eval}\ (\mathsf{lam}\ (\lambda x.\ e\ x))\ d \multimap \{\mathsf{ret}\ (\mathsf{lam}\ (\lambda x.\ e\ x))\ d\}$$

$$\mathsf{eval}\ (\mathsf{app}\ e_1\ e_2)\ d \multimap \{\exists d_1.\ \mathsf{eval}\ e_1\ d_1 \otimes \mathsf{comp}\ (\mathsf{app}\ d_1\ e_2)\ d\}$$

Logical Expression of Rules, Ctd.

- Rule for argument evaluation

$$\frac{e_2 \mapsto d_2 \;,\; \mathsf{app}(v_1, d_2) \rightarrowtail d}{d_1 = v_1 \;,\; \mathsf{app}(d_1, e_2) \rightarrowtail d}\,[d_2]$$

$$\mathsf{ret}\ v_1\ d_1 \otimes \mathsf{comp}\ (\mathsf{app}\ d_1\ e_2)\ d$$
$$\multimap \{\exists d_2.\ \mathsf{eval}\ e_2\ d_2 \otimes \mathsf{comp}\ (\mathsf{app}\ v_1\ d_2)\ d\}$$

- Prove left-hand side from current state
- Assume right-hand side into current state

Logical Expression of Rules, Ctd.

- Rule for function invocation

$$\frac{[v_2/x]e'_1 \mapsto d}{d_2{=}v_2\ ,\ \mathsf{app}(\mathsf{lam}(x.e'_1), d_2) \rightarrowtail d}$$

$$\mathsf{ret}\ v_2\ d_2 \otimes \mathsf{comp}\ (\mathsf{app}\ (\mathsf{lam}\ (\lambda x.\ e'_1\ x))\ d_2)\ d$$
$$\multimap \{\mathsf{eval}\ (e'_1\ v_2)\ d\}$$

- Use higher-order abstract syntax for binding
- Use meta-level application for substitution

Representation Summary

- Computation inside the monad
- Laws of lax logic linearize proof
- Example: complete encoding of cbv λ-calculus

$$\mathsf{eval}\ (\mathsf{lam}\ (\lambda x.\ e\ x))\ d \multimap \{\mathsf{ret}\ (\mathsf{lam}\ (\lambda x.\ e\ x))\ d\}$$

$$\mathsf{eval}\ (\mathsf{app}\ e_1\ e_2)\ d \multimap \{\exists d_1.\ \mathsf{eval}\ e_1\ d_1 \otimes \mathsf{comp}\ (\mathsf{app}\ d_1\ e_2)\ d\}$$

$$\begin{aligned}&\mathsf{ret}\ v_1\ d_1 \otimes \mathsf{comp}\ (\mathsf{app}\ d_1\ e_2)\ d \\ &\qquad \multimap \{\exists d_2.\ \mathsf{eval}\ e_2\ d_2 \otimes \mathsf{comp}\ (\mathsf{app}\ v_1\ d_2)\ d\}\end{aligned}$$

$$\mathsf{ret}\ v_2\ d_2 \otimes \mathsf{comp}\ (\mathsf{app}\ (\mathsf{lam}\ (\lambda x.\ e_1'\ x))\ d_2)\ d \multimap \{\mathsf{eval}\ (e_1'\ v_2)\ d\}$$

Lax Ordered Logic

- Lambek calculus for grammars/parsing [Lambek'58]
- Extended to ordered logic [Polakow&Pf'99]
- $A \bullet B$ for ordered conjunction (fuse)
- $A \multimap\!\!\bullet\ B$ for ordered implication
  - Choice of left or right implication not important here
- Add monad $\{A\}$
  - Achieves bijection between proofs and computations

Lax Ordered Logic, Example

- Sample rules for parallel application

$$\frac{\mathsf{eval}(e_1) \cdot \mathsf{eval}(e_2) \cdot \mathsf{comp}(\mathsf{app}(\Box, \Box))}{\mathsf{eval}(\mathsf{app}(e_1, e_2))}$$

$$\mathsf{eval}\ (\mathsf{app}\ e_1\ e_2) \multimap \{\mathsf{eval}\ e_1 \bullet \mathsf{eval}\ e_2 \bullet \mathsf{comp}\ \mathsf{app}_2\}$$

$$\frac{\mathsf{eval}([v_2/x]e_1')}{\mathsf{ret}(\mathsf{lam}(x.e_1')) \cdot \mathsf{ret}(v_2) \cdot \mathsf{comp}(\mathsf{app}(\Box, \Box))}$$

$$\mathsf{ret}\ (\mathsf{lam}\ (\lambda x.\ e_1'\ x)) \bullet \mathsf{ret}\ v_2 \bullet \mathsf{comp}\ \mathsf{app}_2 \multimap \{\mathsf{eval}\ (e_1'\ v_2)\}$$

π-Calculus Revisited

- Represent rules for process expressions

$$\frac{\mathsf{proc}(P)\ ,\ \mathsf{proc}(Q)}{\mathsf{proc}(P|Q)} \qquad \frac{.}{\mathsf{proc}(0)}$$

$$\mathsf{proc}\ (P|Q) \multimap \{\mathsf{proc}\ P \otimes \mathsf{proc}\ Q\} \qquad \mathsf{proc}\ 0 \multimap \{1\}$$

$$\frac{\mathsf{proc}([a/x]P)}{\mathsf{proc}(\mathsf{new}\, x.P)}\,[a] \qquad \frac{\mathsf{proc}(P)}{\mathsf{proc}(!P)}$$

$$\mathsf{proc}\ (\mathsf{new}\ (\lambda x.\, P\ x)) \multimap \{\exists a.\, \mathsf{proc}\ (P\ a)\} \qquad \mathsf{proc}\ (!P) \multimap \{!\ \mathsf{proc}\ P\}$$

Inside and Outside the Monad

- Auxiliary judgments remain outside the monad

$$\frac{[\ \mathsf{sync}(M, N, P, Q)\ ] \qquad \mathsf{proc}(P)\ ,\ \mathsf{proc}(Q)}{\mathsf{proc}(M)\ ,\ \mathsf{proc}(N)}$$

$$\mathsf{proc}\ M \otimes \mathsf{proc}\ N \otimes \mathsf{sync}\ M\ N\ P\ Q$$
$$\multimap \{\mathsf{proc}\ P \otimes \mathsf{proc}\ Q\}$$

- Monad forces main branch
- $\mathsf{sync}(M, N, P, Q)$ will be resource-neutral

Auxiliary Judgments

- Auxiliary judgments defined outside the monad
- For example

$$\frac{\mathsf{sync}(M_1, N, P, Q)}{\mathsf{sync}(M_1 + M_2, N, P, Q)}$$

$$\mathsf{sync}\ M_1\ N\ P\ Q \multimap \mathsf{sync}\ (M_1 + M_2)\ N\ P\ Q$$

$$\frac{}{\mathsf{sync}(\bar{a}\langle b\rangle.P, a(x).Q, P, [b/x]Q)}$$

$$\mathsf{sync}\ (\mathsf{out}\ a\ b\ P)\ (\mathsf{in}\ a\ (\lambda x.\, Q\ x))\ P\ (Q\ b)$$

Aside: Multi-Port π-Calculus

- Speculative, following [Abe'04]
- New form $a(x){:}P$ may synchronize with prefix of P that does not depend on x
- New rule (plus symmetric rule)

$$\frac{\mathsf{sync}([b/x]M, N, [b/x]P, Q)}{\mathsf{sync}(a(x){:}M, N, a(x){:}P, Q)}\ [b]$$

$$(\forall b.\, \mathsf{sync}\ (M\ b)\ N\ (P\,b)\ Q)$$

$$\multimap \{\mathsf{sync}\ (\mathsf{multi}\ a\ (\lambda x.M\ x))\ N\ (\mathsf{multi}\ a\ (\lambda x.P\ x))\ Q\}$$

- Correct?

Operational Interpretation

- Goal-directed backward chaining outside monad
- Committed choice forward chaining inside monad
- Applicable in wide variety of examples
- Realized in CLF [Watkins,Cervesato,Pfenning,Walker'03]
    - Only linear and unrestricted hypotheses so far
    - Reifies proofs as objects
    - True concurrency
    - Examples have been run with prototype implementation [Polakow]

Further Examples

- Exceptions
- Mutable store
- Heap semantics, garbage collection
- Concurrent ML
- Mobile calculi based on S4, S5
- Meta-interpreter for CLF
- Proof-carrying authorization

Some Related Work

- Structural operational semantics (SOS) [Plotkin'81]
- Natural semantics [Kahn'87]
- Modular SOS [Mosses'98]
- Contextual semantics [Wright & Felleisen'94]
- Abstract machines [Danvy et al.'03]
- Rewriting logic [Meseguer & Braga'03]
- Classical linear logic [Andreoli'90][Chirimar'95][Miller'96]
- Multiset rewriting [Cervesato'01]

Future Work

- More examples
- General concurrent systems
- Complete implementation of CLF
- Reasoning about specifications
  - Definitional reflection, (co-)induction [Tiu'04]
  - ∇ quantifier [Tiu & Miller'03]
  - Theorem proving [Chaudhuri]
  - Model checking

Summary

- Substructural operational semantics (SSOS)
  - Ordered, linear, affine, unrestricted judgments
  - Modular specifications
  - Structural properties as taxonomic device
  - Logical account of operational semantics
  - Widely applicable, realized in CLF
- Linear Destination-Passing (LDP)
  - Specification of functional and imperative computation
  - Destinations can embody return values, mutable references, heap locations, thunks, channels, hosts, etc.