Reversible Computing

Ivan Lanese
Focus research group
Computer Science and Engineering Department
University of Bologna/INRIA
Bologna, Italy
ρCCS: reminder

- Causal-consistent Reversible CCS
- Syntax:
  \[ M ::= k: P | [a, P, P', k, k', k'', k'''] | k < k', k'' | \\
  M|M' | νu M | 0 \]
  \[ P ::= a. P | \bar{a}. P | P + P' | P|P' | νa P | 0 \] 
- Reduction rules:
  \[ k: (\bar{a}. P + Q)|k': (a. P' + Q') \]
  \[ \rightarrow νh νh' h: P|h': P'|[a, Q, Q', k, k', h, h'] \]
  \[ h: P|h': P'|[a, Q, Q', k, k', h, h'] \]
  \[ \rightsquigarrow k: (\bar{a}. P + Q)|k': (a. P' + Q') \]
Example

\[ k: \bar{a}. P \mid k': (a. b. 0 + a. c. 0) \mid k'': \bar{b}. (Q|Q') \rightarrow \]
\[ vh \; vh'[a, 0, a. c. 0, k, k', h, h'] \mid h: P \mid h': b. 0 \mid k'': \bar{b}. (Q|Q') \]
\[ \rightarrow \]
\[ vh \; vh' \; vl \; vl'[a, 0, a. c. 0, k, k', h, h'] \mid h: P \mid [b, 0, 0, k'', h', l', l] \]
\[ l: 0 \mid l': (Q|Q') \rightarrow \]
\[ vh \; vh' \; vl \; vl'[a, 0, a. c. 0, k, k', h, h'] \mid h: P \mid h': b. 0 \mid k'': \bar{b}. (Q|Q') \]
\[ \rightarrow \]
\[ vh \; vh' \; vl \; vl' \; k: \bar{a}. P \mid k': (a. b. 0 + a. c. 0) \mid k'': \bar{b}. (Q|Q') \]
Programming in ρCCS

- We said that the programmer will write processes as usual, only the runtime support should change.
- Here we have a lot of additional information.
- Where does the additional information come from?
ρCCS vs CCS

- Given a CCS process $P$ we can generate a ρCCS configuration as $νk k: P$
  - No memories
  - No causal dependencies
- The programmer writes the CCS process and we can transform it into a ρCCS configuration
- Given a ρCCS configuration we can generate a CCS process by removing all the additional information
- The two transformations form a Galois connection
  - $α$ from ρCCS to CCS
  - $c$ from CCS to ρCCS
ρCCS vs CCS, behaviorally

- Forward reductions of ρCCS configurations are CCS reductions
  \[ M \rightarrow M' \text{ implies } \alpha(M) \rightarrow \alpha(M') \]

- Given a CCS reduction, this can be done by any ρCCS configuration mapped to it
  \[ P \rightarrow P' \text{ and } \alpha(M) = P \text{ implies } M \rightarrow M' \text{ and } \alpha(M') = P' \]
  - History information has no impact on forward reductions
Valid configurations

- Not all the configurations are valid
- E.g., if the configuration contains a connector $k < k', k''$ then $k'$ and $k''$ occur also as keys of a process, a memory or another connector
- Causality information should form a partial order
- A bit difficult to characterize syntactically valid configurations
- Semantic characterization: a configuration is valid iff it can be derived from a configuration of the form $\nu k \ k : P$
Parabolic Lemma

- Each computation is causally equivalent to a computation obtained by doing a backward computation followed by a forward computation.
- Intuitively, I undo all what I have done and then compute only forward.
  - Tries which are undone are not relevant.
- Useful for proving the Causal consistency Theorem.
Causal-consistent CCS in the literature

- In the literature there are two other causal-consistent reversible CCS
  - RCCS
    [Vincent Danos, Jean Krivine: Reversible Communicating Systems. CONCUR 2004]
    LTS based, histories attached to threads
  - CCSk
    [Iain C. C. Phillips, Irek Ulidowski: Reversing Algebraic Process Calculi. FoSSaCS 2006]
    LTS based, process is not consumed, part of it is just annotated as no more available
Are all those causal-consistent CCS equivalent?

- Yes!
- Reductions in ρCCS correspond to internal steps (τ moves) of the other approaches
- Essentially they provide different run time support definitions for the same language
- There exists a unique way to define a causal-consistent extension of a given language
  - Satisfying the expected properties
- Our approach is more easy to generalize
From ρCCS to Rhopi

- CCS is not expressive enough
- We want to consider more expressive languages
- We choose higher-order π-calculus
  - Allows processes to communicate
HOpi

- Syntax
  \[ P ::= a(P) | a(X) \triangleright P | P|Q | \nu a P | X | 0 \]
- Higher-order communication
- Asynchronous calculus
- You can imagine structural congruence
- A reduction rule
  \[ a(P)|a(X) \triangleright Q \rightarrow Q\{P/X\} \]
Infinite behaviors

- HO\(\pi\) can implement infinite behaviors
  - No need for operators for replication or recursion
- \(Q = a(X) \triangleright P | X | a\langle X \rangle\)
  - \(Q | a\langle Q \rangle\) reduces to \(P | Q | a\langle Q \rangle\)
- This allows one to generate an infinite amount of copies of \(P\)
How to make HOπ reversible?

- The main novelty is given by substitutions
- In ρCCS we can take the continuations from the configuration
- Here this is no more true
- From Q\{P/X\} I cannot recover Q or P
- Not even Q if I know P
  - P/X, X/P, P/P, X/X all produce the same result
- Not even P if I know Q
  - If Q does not contain X
Rhopi

- Syntax:
  \[ M ::= k: P \mid [\mu, k] \mid k < k', k'' \mid M\!|\!M' \mid \nu \mu M \mid 0 \]
  \[ \mu ::= k: a(P)\mid k': a(X) \triangleright Q \]

- Reduction rules:
  \[ k: a(P)\mid k': a(X) \triangleright Q \rightarrow \nu k'' k'': Q{P/X}\!|\![\mu, k''] \]
  \[ k'': R\!|\![\mu, k''] \rightsquigarrow \mu \]

- A unique continuation since the calculus is asynchronous

- I store the whole configuration
  - Not really memory efficient
  - But it works, and provides a simple semantics
  - I may optimize it later
Restriction

- It seems we do not consider restriction
- Indeed, this is what we do
- We can do it!
- Try what happens with
  \[ k: a \langle vb \ c \langle b \langle Q \rangle . \ 0 \rangle \rangle \ | \ k': a(X) \triangleright X \ | \ k'': c(Y) \triangleright Y \]
Summarizing

- We have been able to define reversible CCS and HO\(\pi\)
- Both causal consistent
- Using almost the same techniques
- But we are still at uncontrolled reversibility
Controlling reversibility

Power is nothing without control
Roll-π

- We want to use the roll operator to control reversibility in Rhopi
- We have to attach labels γ to some actions
  - We choose triggers
  - Since triggers have a continuation
- The challenge is to define the semantics of the roll operator
  - It involves an unbounded number of processes
Roll-π syntax

- $M ::= k: P \mid [\mu, k] \mid k < k', k'' \mid M|M' \mid \nu u M \mid 0$
- $P ::= a\langle P\rangle \mid a(X) \triangleright_{\gamma} P \mid P|Q \mid \nu a P \mid X \mid 0 | \text{roll } \gamma \mid \text{roll } k$
- $\mu ::= k: a\langle P\rangle|k': a(X) \triangleright_{\gamma} Q$
- Now $\gamma$ attached to triggers
- $\gamma$ is a binder
- At run-time $\gamma$ replaced by $k$
Roll-π semantics

- Little changes to the forward rule
  \[ k: a\langle P \rangle \mid k': a(X) \triangleright_\gamma Q \rightarrow \nu k'' k''': Q\{P/X\}{k''/\gamma}\mid [\mu, k''] \]

- A new, complex, backward rule
  \[ k > M \text{ complete}(M|\mu, k||k': \text{roll } k) \]
  \[ M|\mu, k||k': \text{roll } k \leadsto\mu|M \downarrow k \]

- The two preconditions require to involve only processes which depend on \( k \), and all of them

- We need to define the dependency relation
Exploiting causality

- Causal dependence: if in a term I have
  - \([k: a(P) \mid k': a(X) \triangleright Q, k''] \) then \(k > k''\) and \(k' > k''\)
  - \(k < k', k''\) then \(k > k'\) and \(k > k''\)

- \(k > M\) if \(k > h\) for all \(h: P, [\mu, h]\) and \(h < h', h''\) in \(M\)

- Completeness is essentially closure under consequences

- Completeness: if in a term I have
  - \([k: a(P) \mid k': a(X) \triangleright Q, k''] \) then there is another occurrence of \(k''\)
  - \(k < k', k''\) then there are other occurrences of \(k'\) and \(k''\)
Example

\[ k: a\langle P \rangle \mid k': a(X) \triangleright_\gamma b(Y) \triangleright \text{roll } \gamma \mid k'': b\langle P' \rangle \rightarrow \]

\[ \nu k'''[k: a\langle P \rangle \mid k': a(X) \triangleright_\gamma b(Y) \triangleright \text{roll } \gamma, k'''] \mid k''': b(Y) \]

\[ \triangleright \text{roll } k''' \mid k'': b\langle P' \rangle \rightarrow \]

\[ \nu k''''k'''''[k: a\langle P \rangle \mid k': a(X) \triangleright_\gamma b(Y) \triangleright \text{roll } \gamma, k'''''] \mid [k''': b(Y) \]

\[ \triangleright \text{roll } k''' \mid k'': b\langle P' \rangle, k''''' \mid k''''': \text{roll } k'''' \twoheadrightarrow \]

\[ \nu k''''k''''''k: a\langle P \rangle \mid k': a(X) \triangleright_\gamma b(Y) \triangleright \text{roll } \gamma \mid k'': b\langle P' \rangle \]
Releasing resources

- Processes which are not dependent on $k'''$ but are in memories dependent on $k'''$ can be seen as resources taken by the computation from the environment.
- They have to be restored in case of **roll** $k'''$
- This is done by $M \downarrow k'''$