Asynchronous Functional Sessions: Cyclic and Concurrent

Bas van den Heuvel
(joint work with Jorge A. Pérez)

University of Groningen, The Netherlands

EXPRESS/SOS 2022, 12 September 2022

UNIFYING CORRECTNESS FOR COMMUNICATING SOFTWARE

Overview

- Curry-Howard correspondences between linear logic and session types: a solid foundation for message-passing concurrency.
- Pros: Deadlock-freedom by typing and clear connections with functional languages with concurrency. Cons: Limited expressiveness.
- Much interest in increasing the expressiveness of session-typed π-calculi. We seek to transfer these gains to the functional setting.
- Here: Concurrent GV (CGV), a new λ-calculus with sessions.
  - Solid design basis: APCP, an expressive session-typed π-calculus. Main features: asynchrony, arbitrary network topologies.
  - CGV's type system ensures *session fidelity* and *communication safety*, but not *deadlock-freedom*.
  - An operationally correct translation from CGV into APCP. Transfer deadlock-freedom to (a subset of) well-typed CGV programs.

- Three intertwined novelties that set CGV apart from its predecessors.
  - λ^{sess} (Gay and Vasconcelos, 2010);
  - GV (Wadler, 2012);
  - EGV (Fowler, Lindley, Morris and Decova, 2019);
  - PGV (Kokke and Dardha, 2021).
- Asynchronous communication:
  CGV uses buffers such that *outputs are non-blocking*.
- Configurations of threads in arbitrary topologies:
  CGV allows *cyclic* thread configurations.
- Highly concurrent evaluation strategy:
  CGV evaluates functions and their parameters *concurrently*.

CGV by example

	CGV	$\lambda^{\textsf{sess}}$	GV	EGV	PGV
Communication					
Topologies					
Deadlock-freedom					
Evaluation					

$$(\nu xy)(\nu vw)\left(\begin{array}{r}\text{let } x' = \text{send}\,(u, x) \text{ in} \\ \text{let}\,(q, v') = \text{recv}\; v \text{ in } q\end{array} \;\middle\|\; \begin{array}{r}\text{let } w' = \text{send}\,(q', w) \text{ in} \\ \text{let}\,(u', y') = \text{recv}\; y \text{ in } u'\end{array}\right)$$

- In CGV, communication is *asynchronous*:
  the messages sent on x and on w are placed in buffers.
- Messages are read from the buffers with the recvs on v and on y.
- Under *synchronous* communication, this program is deadlocked.

	CGV	λ^{sess}	GV	EGV	PGV
Communication	Async.	Async.	Sync.	Async.	Sync.
Topologies					
Deadlock-freedom					
Evaluation					

$$(\boldsymbol{\nu} xy)(\boldsymbol{\nu} vw)\left(\begin{aligned}&\mathsf{let}\,(u,x') = \mathsf{recv}\,x\;\mathsf{in}\\ &\quad\mathsf{let}\,v' = \mathsf{send}\,(q,v)\;\mathsf{in}\,()\end{aligned}\;\middle\|\;\begin{aligned}&\mathsf{let}\,y' = \mathsf{send}\,(u',y)\;\mathsf{in}\\ &\quad\mathsf{let}\,(q',w') = \mathsf{recv}\,w\;\mathsf{in}\,()\end{aligned}\right)$$

- Two sessions and two threads that are *cyclically connected*.
- The program is *deadlock-free*:
  first the send on y and recv on x, then the send on v and recv on w.
- Well-typed in CGV, guaranteed deadlock-free *via APCP*.

	CGV	λ^{sess}	GV	EGV	PGV
Communication	Async.	Async.	Sync.	Async.	Sync.
Topologies	Cyclic	Cyclic	Tree	Tree	Cyclic
Deadlock-freedom	APCP	None	Typing	Typing	Typing
Evaluation					

$$\left(\begin{array}{l}\lambda x\,.\,\mathsf{let}\,(u,y) = \mathsf{recv}\,y\;\mathsf{in} \\ \qquad\qquad \mathsf{let}\,x = \mathsf{send}\,(u,x)\;\mathsf{in}\;()\end{array}\right) \;(\mathsf{send}\,(v,z))$$

- In CGV, a function and its parameters are evaluated *concurrently*: no restriction on the order of the recv on y and the send on z.
- The evaluation strategy of CGV is reminiscent of *call-by-future*.
- Under call-by-value (CbV) strategies the function on x can only be applied after evaluating the send on z, blocking the recv on y.

	CGV	λ^{sess}	GV	EGV	PGV
Communication	Async.	Async.	Sync.	Async.	Sync.
Topologies	Cyclic	Cyclic	Tree	Tree	Cyclic
Deadlock-freedom	APCP	None	Typing	Typing	Typing
Evaluation	Concur.	CbV	CbV	CbV	CbV

$$(\boldsymbol{\nu} xy)\left(\begin{array}{l}\text{let}\,(v,w)=\text{new in}\\ \quad\text{let}\,x'=\text{send}\,(\text{send}\,(u,w),x)\,\text{in}\\ \qquad\text{let}\,(u',v')=\text{recv}\,v\,\text{in}\,u'\end{array}\;\middle\|\;\text{let}\,(s,y')=\text{recv}\,y\,\text{in}\,s\right)$$

$$\rightarrow^{*}(\boldsymbol{\nu} vw)\left(\text{let}\,(u',v')=\text{recv}\,v\,\text{in}\,u' \;\middle\|\; \text{send}\,(u,w)\right)$$

- Using higher-order message-passing, threads can send whole terms.
- The left thread sends to the right an output on a new channel.
- After receiving the output, the right thread executes it, to be received by the left thread.
- In prior works, only *values* can be sent (e.g., variables and functions).

- CGV is a typed calculus, with *functional types* and *session types*.

$$x : !(T \multimap (\mathbf{1} \times T)) . S \vdash \mathsf{send}\,(\lambda z . ((), z), x) : S$$

- Typing ensures *session fidelity* and *communication safety*, but not *deadlock-freedom*.

$$(\boldsymbol{\nu} xy)(\boldsymbol{\nu} vw) \left(\begin{array}{l} \mathsf{let}\,(u, x') = \mathsf{recv}\, x \,\mathsf{in} \\ \quad \mathsf{let}\, v' = \mathsf{send}\,(u, v)\,\mathsf{in}\,() \end{array} \;\Big\|\; \begin{array}{l} \mathsf{let}\,(q, w') = \mathsf{recv}\, w \,\mathsf{in} \\ \quad \mathsf{let}\, y' = \mathsf{send}\,(q, y)\,\mathsf{in}\,() \end{array} \right)$$

- Well-typed in CGV, but not deadlock-free.
- APCP to the rescue.

- In recent work (ICE'21), we developed APCP: a session type system for π-calculus processes.
- Key features:
  cyclic process networks, *asynchronous communication*, and *recursion*.
- Follows and extends the Curry-Howard correspondences between linear logic and session types. A very solid design basis for CGV.
- Priorities on types are used to rule out circular dependencies in processes (Kobayashi, 2006; Padovani, 2014; Dardha and Gay, 2018).
- Key properties:
  session fidelity, *communication safety*, and *deadlock-freedom*.
- APCP is expressive enough for a decentralized analysis of Multiparty Session Types (cf. our journal paper Sci. Comput. Program., 2022).

- Recall our first CGV example:

$$(\boldsymbol{\nu} xy)(\boldsymbol{\nu} vw) \begin{pmatrix} \text{let } x' = \text{send}\,(u, x) \text{ in} \\ \quad \text{let}\,(q, v') = \text{recv } v \text{ in } q \end{pmatrix} \parallel \begin{pmatrix} \text{let } w' = \text{send}\,(q', w) \text{ in} \\ \quad \text{let}\,(u', y') = \text{recv } y \text{ in } u' \end{pmatrix}$$

- Analogous example in APCP:

$$(\boldsymbol{\nu} xy)(\boldsymbol{\nu} vw) \begin{pmatrix} (\boldsymbol{\nu} ax')(\boldsymbol{\nu} bu)(x[a, b] \mid v(q', v') \,.\, \mathbf{0}) \\ \mid (\boldsymbol{\nu} cw')(\boldsymbol{\nu} dq)(w[c, d] \mid y(u', y') \,.\, \mathbf{0}) \end{pmatrix}$$

Outputs are standalone and parallel, session order is maintained by means of continuation-passing.

- Deadlock-free in APCP due to asynchronous communication; Deadlocked under synchronous communication.

- Terms and types translated:

$$[\![\Gamma \vdash M : T]\!]z = [\![M]\!]z \vdash \overline{[\![\Gamma]\!]}, z : [\![T]\!]$$

- For example, translation of pairs (M, N)—omitting types:

$$[\![(M, N)]\!]z = (\boldsymbol{\nu} ab)(\boldsymbol{\nu} cd)(\; z[a, c] \mid b(e, b')\; [\![M]\!]e \mid d(f, d')\; [\![N]\!]f \;)$$

The translations of M and N are not blocked by the output.
Additional inputs are required.

- Translation of function abstraction $\lambda x \,.\, M$:

$$[\![\lambda x \,.\, M]\!]z = z(a, b) \,.\, (\boldsymbol{\nu} cx)\big((\boldsymbol{\nu} ef)\; a[c, e] \mid [\![M]\!]b\big)$$

Here, an additional output is required, to activate the function's parameter which is blocked by an input.

Translating CGV into APCP

- The translation preserves well-typedness, *up to priorities*: we ignore priorities for translations of CGV programs with cyclic dependencies.

- The translation is *operationally complete*:
  Reductions in CGV programs are mimicked by their APCP translations.

- We also desire *operational soundness*:
  Any reductions in APCP should be reflected by source CGV programs.

- However, APCP's semantics is *too eager* for soundness.
  We state soundness in terms of an alternative *lazy* semantics.

- Our characterization of deadlock-freedom in CGV:
  M is deadlock-free if $[\![M]\!]z$ is well-typed *including priorities*.
- Those translations are deadlock-free in APCP, but only under the standard semantics.
- By analyzing the shapes of M and $[\![M]\!]z$, we prove that the translation is also deadlock-free under the lazy semantics.
- Hence, deadlock-freedom in APCP transfers to CGV through operational soundness.

Summary

- Concurrent GV: a new λ-calculus with sessions that features *asynchrony* and *cyclic thread configurations*.
- CGV's type system ensures *session fidelity* and *communication safety*, but not *deadlock-freedom*
- A (typed) translation into APCP recovers deadlock-freedom for (a subset of) well-typed CGV programs.

	CGV	λ^{sess}	GV	EGV	PGV
Communication	Async.	Async.	Sync.	Async.	Sync.
Topologies	Cyclic	Cyclic	Tree	Tree	Cyclic
Deadlock-freedom	APCP	None	Typing	Typing	Typing
Evaluation	Concur.	CbV	CbV	CbV	CbV

- Omitted technical details (see https://arxiv.org/abs/2208.07644):
  - CGV's semantics with a runtime *configuration* layer: buffers and threads.
  - Translation of CGV types into APCP types.
  - APCP's lazy semantics for soundness of the translation.

CGV semantics (selected rules)

- Function application

$$(\lambda x \,.\, M)\ N \longrightarrow M\{\!\!\{ N/x \}\!\!\}$$

- Spawning a thread

$$\mathcal{F}[\mathsf{spawn}\,(M, N)] \longrightarrow \mathcal{F}[N] \parallel \diamond\, M$$

- Channel creation

$$\mathcal{F}[\mathsf{new}] \longrightarrow (\boldsymbol{\nu} x[\varepsilon\rangle y)(\mathcal{F}[(x, y)])$$

- Sending a message (structural congruence)

$$(\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[\mathsf{send}\,(M, x)] \parallel C) \equiv (\boldsymbol{\nu} x[M, \vec{m}\rangle y)(\mathcal{F}[x] \parallel C)$$

- Receiving a message

$$(\boldsymbol{\nu} x[\vec{m}, M\rangle y)(\mathcal{F}[\mathsf{recv}\, x] \parallel C) \longrightarrow (\boldsymbol{\nu} x[\vec{m}\rangle y)(\mathcal{F}[(M, y)] \parallel C)$$

Translation of CGV types into APCP types

$$\llbracket T \times U \rrbracket = (\llbracket T \rrbracket \parr \bullet) \otimes (\llbracket U \rrbracket \parr \bullet) \qquad \llbracket T \multimap U \rrbracket = (\overline{\llbracket T \rrbracket} \otimes \bullet) \parr \llbracket U \rrbracket$$

$$\llbracket \mathbf{1} \rrbracket = \bullet$$

$$\llbracket !T \,.\, S \rrbracket = (\overline{\llbracket T \rrbracket} \otimes \bullet) \parr \llbracket S \rrbracket \qquad \llbracket ?T \,.\, S \rrbracket = (\llbracket T \rrbracket \parr \bullet) \otimes \llbracket S \rrbracket$$

$$\llbracket \oplus\{i : T_i\}_{i \in I} \rrbracket = \&\{i : \llbracket T_i \rrbracket\}_{i \in I} \qquad \llbracket \&\{i : T_i\}_{i \in I} \rrbracket = \oplus\{i : \llbracket T_i \rrbracket\}_{i \in I}$$

$$\llbracket \mathsf{end} \rrbracket = \bullet$$

APCP's lazy semantics for soundness

Omitting branching/selection and closure rules:

$$
\begin{aligned}
(\overset{\leftrightarrow}{\boldsymbol{\nu}} yz)(x \leftrightarrow y \mid P) &\longrightarrow_{\mathtt{L}}^{(x,y)} P\{x/z\} \\
(\boldsymbol{\nu} xy)(x[a,b] \mid y(c,d) \,.\, P) &\longrightarrow_{\mathtt{L}}^{\cdot} P\{a/c, b/d\} \\
(\boldsymbol{\nu} xy)((\boldsymbol{\nu} uv)(x \leftrightarrow u \mid v[a,b]) & \\
\mid (\boldsymbol{\nu} wz)(y \leftrightarrow w \mid z(c,d) \,.\, P)) &\longrightarrow_{\mathtt{L}}^{\cdot} P\{a/c, b/d\}
\end{aligned}
$$

$$
\begin{aligned}
P \longrightarrow_{\mathtt{L}}^{\cdot} Q &\implies P \longrightarrow_{\mathtt{L}} Q \\
P \longrightarrow_{\mathtt{L}}^{(x,y)} Q \wedge \mathrm{bcont}_{x,y}(P) &\implies P \longrightarrow_{\mathtt{L}} Q
\end{aligned}
$$

The predicate $\mathrm{bcont}_{x,y}(P)$ holds iff $P \equiv \mathcal{E}[(\boldsymbol{\nu} xa)(x \leftrightarrow y \mid (\boldsymbol{\nu} cd)(c \leftrightarrow e \mid d[f,a]))]$ for some evaluation context $\mathcal{E}$ implies $P \equiv (\boldsymbol{\nu} eg)Q$ for some Q.