From game semantics to automated verification of OCaml programs

Guilhem Jaber
Univ. Nantes, Équipe Inria Gallinette

Séminaire Cambium - Inria Paris

How to automatically check safety properties
for an higher-order programming language
with a polymorphic type system,
control operators,
and mutable dynamically-allocated resources?

Interactive semantics of programming languages

50 years of history:

- Bohm trees;
- Sequential algorithms (Berry & Curien);
- Game semantics (Abramsky, Jagadeesan & Malacaria; Hyland & Ong; Nickau).

Main successes:

- Fully-abstract semantics:
  - untyped CBN λ-calculus (Bohm trees, Levy-Longo trees)
  - PCF+exceptions (sequential algorithms)
  - Idealized Algol, RefML, ... (game semantics).
- Decidability results
  - model-checking μ-calculus formulas over higher-order recursion schemes
  - contextual equivalence for Idealized Algol and GroundML for programs of low-order type.

Introducing interactive semantics

- Interactions between the program and its environment is represented as *plays* (a.k.a. *traces*)
  - $\rightsquigarrow$ sequences of *moves* (a.k.a. *actions*) alternating between *Proponent* and *Opponent*.
- *Proponent* represents the program behavior
  - $\rightsquigarrow$ it is determined by the computation of the program.
- *Opponent* represents environment behavior
  - $\rightsquigarrow$ it is specified by rules of the *game*
  - $\rightsquigarrow$ built from the programming languages features: type system, (absence of) side-effects.
- Denotation of a program is formed by a *strategy*
  - $\rightsquigarrow$ the set of traces the program generates against any environment that behave according to the rules of the game.

In this talk:
operationally-presented game semantics

- *Labelled Transition Systems* (LTS) as the basic blocs
  - for computing the interaction
  - for representing the rules of the game.
- Proponent's behavior is computed via an operational semantics
  - rather than compositionally by induction over the typing derivation.
- Causality between moves is represented via a nominal encoding.

James Laird. "A Fully Abstract Trace Semantics for General References". In: *Proceedings of the 34th International Conference on Automata, Languages and Programming*. 2007

Definition

An LTS $\mathcal{L}$ is a triple $(\mathrm{States}, \mathrm{Actions}, \rightarrow)$ with

- States a set of states
  - called *configurations* when they are built over terms;
- A set of actions $\mathrm{Actions}$
  - visible actions are called *moves* m;
  - a silent action op, corresponding to internal computations.
- a labeled transition relation $\rightarrow \subseteq \mathrm{States} \times \mathrm{Actions} \times \mathrm{States}$
  - we write $\mathbb{C} \xrightarrow{\mathsf{a}} \mathbb{D}$ for $(\mathbb{C}, \mathsf{a}, \mathbb{D}) \in \rightarrow$.

Traces

- Traces are finite sequences of *moves* $\mathsf{p}_1\mathsf{o}_2 \cdot \mathsf{p}_k\mathsf{o}_k$
- that alternates between:
  - *Player* moves p representing the program behavior
  - *Opponent* moves o representing the environment behavior.

- Moves m are either *call* or *return* operations:

P-question	P-answer	O-question	O-answer
$\overline{f}(A)$	$\overline{\mathsf{ret}}(A)$	$f(A)$	$\mathsf{ret}(A)$

- Input (Opponent) / Output (Proponent) polarities of moves
- Duality operator switching polarities: $\bar{\mathsf{m}}$;
- Moves exchanges *abstract values* A, B;
  - defined from a characterization of the observational power of the programming language.

How to define the observational power of a programming language ?

Via **polarization**:

- *interact* with negative ($\ominus$) values;
- *observe* positive ($\oplus$) values, called patterns.

In this talk: a typed call-by-value λ-calculus with pattern-matching.

Negative ($\ominus$)	Positive ($\oplus$)
$\to$	$\mathtt{unit}, \mathtt{int}, \times, +$

Typing abstract values

- Abstract values A, B are *nominal ultimate patterns*:

Unit	Integer	Function names	Sum	Pairs
$()$	k	$\underline{f}$	$\mathtt{inj}_i(A)$	$\langle A, B\rangle$

- Typing judgment $\Delta \Vdash A : \tau$
  - with Δ a linear context of names.
  - and no names with positive types in Δ.

$$\frac{}{\varnothing \Vdash () : \mathtt{unit}} \qquad \frac{k \in \mathbb{Z}}{\varnothing \Vdash k : \mathtt{int}} \qquad \frac{\tau \text{ function type}}{f : \tau \Vdash \underline{f} : \tau}$$

$$\frac{\Delta \Vdash A : \tau_i}{\Delta \Vdash \mathtt{inj}_i(A) : \tau_1 + \tau_2} \qquad \frac{\Delta_1 \Vdash A_1 : \tau_1 \qquad \Delta_2 \Vdash A_2 : \tau_2}{\Delta_1 \cdot \Delta_2 \Vdash \langle A_1; A_2\rangle : \tau_1 \times \tau_2}$$

Abstract values via Focusing

- Any value V can be decomposed into a nominal ultimate pattern A and an environment γ such that $A\{\gamma\} = V$.
  - Corresponds to large-step focusing: Noam Zeilberger. "Focusing and Higher-Order Abstract Syntax". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2008
- This decomposition $V \nearrow (A, \gamma)$ is defined by the following rules:

$$\frac{}{() \nearrow ((), \varepsilon)} \qquad \frac{}{k \nearrow (k, \varepsilon)}$$

$$\frac{}{\lambda x.M \nearrow (\underline{f}; [f \mapsto \lambda x.M])} \qquad \frac{}{f \nearrow (\underline{g}; [g \mapsto f])}$$

$$\frac{V \nearrow (A, \gamma) \quad W \nearrow (B, \gamma')}{\langle V, W\rangle \nearrow (\langle A, B\rangle, \gamma \cdot \gamma')} \qquad \frac{V \nearrow (A, \gamma)}{\mathtt{inj}_i(V) \nearrow (\mathtt{inj}_i(A), \gamma)}$$

Justification pointer

- A move *binds* the function names introduced in the abstract value A it exchanges.
- Taking $\mathrm{t} = \mathrm{t}_1\ f(A)\ \mathrm{t}_2$, the question $f(A)$ is *justified in* t by the move of t_1 that introduces f.
- Taking $\mathrm{t} = \mathrm{t}_1\ \mathrm{ret}(A)\ \mathrm{t}_2$, the answer $\mathrm{ret}(A)$ is *justified in* t by the last unanswered question of t_1.
  - $\rightsquigarrow$ a question of t is *unanswered* when it does not justifies an answer in t.
- Example:

$$\overline{\mathrm{ret}}(\langle 3, f\rangle)\ f(g)\ \overline{g}()\ f(g')\ \overline{\mathrm{ret}}(5)\ \mathrm{ret}(9)\ \overline{\mathrm{ret}}(12)$$

CPS and Well-bracketing

- Transform answers into calls to *continuation names c*

P-question	P-answer	O-question	O-answer
$\overline{f}(A, c)$	$\overline{c}(A)$	$f(A, c)$	$c(A)$

- Taking $\mathrm{t} = \mathrm{t}_1\ c(A)\ \mathrm{t}_2$, the question $c(A)$ is *justified in* t by the move of t_1 that introduces c.
- CPS translation of traces:

$$\overline{\mathrm{ret}}(\langle 3, f\rangle)\ f(g)\ \overline{g}()\ f(g')\ \overline{\mathrm{ret}}(5)\ \mathrm{ret}(9)\ \overline{\mathrm{ret}}(12)$$

into

$$\overline{c_0}(\langle 3, f\rangle)\ f(g, c_1)\ \overline{g}(c_2)\ f(g', c_3)\ \overline{c_3}(5)\ c2(9)\ \overline{c_1}(12)$$

- Reversing the CPS translation?
  - only on *well-bracketed* traces

Definition

A trace is well-bracketed if all its answers are justified by the previously unanswered question in the trace.

Introducing the OGS LTS

Definition (Synchronization)

Taking $\mathcal{L}_1 = (\mathrm{States}_1, \mathrm{Actions}, \to_1)$ and $\mathcal{L}_2 = (\mathrm{States}_2, \mathrm{Actions}, \to_2)$ two LTSs sharing the same set of actions, their *synchronization* $\mathcal{L}_1 \bowtie \mathcal{L}_2$, is the LTS $(\mathrm{States}_1 \times \mathrm{States}_2, \mathrm{Actions}, \to)$ with $\to$ defined as:

$$\frac{\mathbb{I} \xrightarrow{\mathsf{op}}_1 \mathbb{J}}{(\mathbb{I};\mathbb{S}) \xrightarrow{\mathsf{op}} (\mathbb{J};\mathbb{S})} \qquad \frac{\mathbb{S} \xrightarrow{\mathsf{op}}_2 \mathbb{T}}{(\mathbb{I};\mathbb{S}) \xrightarrow{\mathsf{op}} (\mathbb{I};\mathbb{T})} \qquad \frac{\mathbb{I} \xrightarrow{\mathsf{m}}_1 \mathbb{J} \qquad \mathbb{S} \xrightarrow{\mathsf{m}}_2 \mathbb{T}}{(\mathbb{I};\mathbb{S}) \xrightarrow{\mathsf{m}} (\mathbb{J};\mathbb{T})}$$

The OGS LTS is built as the parallel composition $\mathcal{L}_{\mathsf{I}} \bowtie \mathcal{L}_{\mathsf{Ty}} \bowtie \mathcal{L}_{\mathsf{WB}} \bowtie \mathcal{L}_{\mathsf{discl}}$ with:

- the Interactive LTS $\mathcal{L}_{\mathsf{I}}$ that computes Proponent's interactions using operational semantics;
- the Typing LTS $\mathcal{L}_{\mathsf{Ty}}$ that enforces Opponent's interactions to be well-typed
- History LTSs $\mathcal{L}_{\mathsf{WB}}$ and $\mathcal{L}_{\mathsf{discl}}$ that enforces Opponent's interactions to be well-bracketed and *non-omniscient*.

Interactive LTS

- Configurations are either active $\langle [c]M; \gamma \rangle$ or passive $\langle \gamma \rangle$;
- with c a continuation name and M a term
- γ an environment mapping:
  - function names f to values
  - continuation names c to *named evaluation contexts* $K = [c']E$.

$$\text{op}\ \frac{M \mapsto_{\text{op}} N}{\langle [c]M; \gamma \rangle \xrightarrow{\text{op}} \langle [c]N; \gamma \rangle}$$

$$\text{PQ}\ \frac{V \nearrow (A; \gamma')}{\langle K[fV]; \gamma \rangle \xrightarrow{\overline{f}(A,c)} \langle \gamma \cdot \gamma' \cdot [c \mapsto K] \rangle} \qquad \frac{V \nearrow (A; \gamma')}{\langle [c]V; \gamma \rangle \xrightarrow{\overline{c}(A)} \langle \gamma \cdot \gamma' \rangle}\ \text{PA}$$

$$\text{OQ}\ \frac{}{\langle \gamma \rangle \xrightarrow{f(A,c)} \langle [c]\gamma(f)A; \gamma \rangle} \qquad \frac{}{\langle \gamma \rangle \xrightarrow{c(A)} \langle \gamma(c)[A]; \gamma \rangle}\ \text{OA}$$

Typing LTS

- Configurations $\mathbb{S}$ of $\mathcal{L}_{\mathsf{Ty}}$ keep track of typing of names:
  - $\langle \Delta_{\mathsf{O}} \mid \perp; \Delta_{\mathsf{P}} \rangle$ for Player configurations;
  - $\langle \Delta_{\mathsf{O}} \mid \Delta_{\mathsf{P}} \rangle$ for Opponent configurations.
- Transition checks typing constraints of patterns exchanged using $\Vdash$
  - Named evaluations contexts $[c]E$ have types $\neg\tau$, with τ the type of the hole of E.
  - Negation of types: $\begin{array}{lcl} (\tau \to \sigma)^{\perp} & \triangleq & \tau \times \neg\sigma \\ (\neg\tau)^{\perp} & \triangleq & \tau \end{array}$

James Laird. "A Curry-style Semantics of Interaction: From Untyped to Second-Order Lazy $\lambda\mu$-Calculus". In: *International Conference on Foundations of Software Science and Computation Structures.* FoSSaCS'20

Transitions of the Typing LTS

$$\text{PQ}\ \frac{\Delta \Vdash \langle A, c\rangle : \Delta_{\mathsf{O}}(f)^{\perp}}{\langle \Delta_{\mathsf{O}} \mid \perp; \Delta_{\mathsf{P}}\rangle \xrightarrow{\overline{f}(A,c)}_{\mathsf{Ty}} \langle \Delta_{\mathsf{O}} \mid \Delta_{\mathsf{P}} \cdot \Delta\rangle}$$

$$\text{PA}\ \frac{\Delta \Vdash A : \Delta_{\mathsf{O}}(c)^{\perp}}{\langle \Delta_{\mathsf{O}} \mid \perp; \Delta_{\mathsf{P}}\rangle \xrightarrow{\overline{c}(A)}_{\mathsf{Ty}} \langle \Delta_{\mathsf{O}} \mid \Delta_{\mathsf{P}} \cdot \Delta\rangle}$$

$$\text{OQ}\ \frac{\Delta \Vdash \langle A, c\rangle : \Delta_{\mathsf{P}}(f)^{\perp}}{\langle \Delta_{\mathsf{O}} \mid \Delta_{\mathsf{P}}\rangle \xrightarrow{f(A,c)}_{\mathsf{Ty}} \langle \Delta_{\mathsf{O}} \cdot \Delta \mid \perp; \Delta_{\mathsf{P}}\rangle}$$

$$\mathrm{OA}\ \frac{\Delta \Vdash A : \Delta_{\mathsf{P}}(c)^{\perp}}{\langle \Delta_{\mathsf{O}} \mid \Delta_{\mathsf{P}}\rangle \xrightarrow{c(A)}_{\mathsf{Ty}} \langle \Delta_{\mathsf{O}} \cdot \Delta \mid \perp; \Delta_{\mathsf{P}}\rangle}$$

Enforcing well-bracketing

Configurations of $\mathcal{L}_{\mathsf{WB}}$ are *stacks* π of continuation names introduced by Proponent.

$$\mathsf{PQ}\ \frac{}{\langle \pi \rangle \xrightarrow{\overline{f}(A,c)} \langle c :: \pi \rangle}$$

$$\mathsf{PA}\ \frac{}{\langle \pi \rangle \xrightarrow{\overline{c}(A)} \langle \pi \rangle}$$

$$\mathsf{OQ}\ \frac{}{\langle \pi \rangle \xrightarrow{f(A,c)} \langle \pi \rangle}$$

$$\mathsf{OA}\ \frac{}{\langle c :: \pi \rangle \xrightarrow{c(A)} \langle \pi \rangle}$$

Polymorphism

Church style System F:

$$\frac{\Gamma, X : \mathtt{Type} \vdash M : \tau}{\Gamma \vdash \Lambda X.M : \forall X.\tau} \qquad \frac{\Gamma \vdash M : \forall X.\tau}{\Gamma \vdash M\tau : \tau'\{X := \tau\}}$$

$$\frac{\Gamma \vdash M : \tau\{X := \tau'\}}{\Gamma \vdash \langle \tau'; M \rangle : \exists X.\tau}$$

$$\frac{\Gamma \vdash M : \exists X.\tau \qquad \Gamma, X : \mathtt{Type}, x : \tau \vdash N : \tau'}{\Gamma \vdash \mathtt{match}\ M\ \mathtt{with}\ (X, x) \Rightarrow N : \tau'}$$

Polarization of type variables

- We tag type variables:
  - $X^{\oplus}$ when Proponent choose the type associated to X;
  - $X^{\ominus}$ when Opponent choose the type associated to X.

- Via a type translation $\mathsf{pol}^{\kappa}_{\rho}(\cdot) : \mathrm{Types} \to \mathrm{Types}$, for $\kappa \in \{\oplus, \ominus\}$, defined as

$$
\begin{array}{lcl}
\mathsf{pol}^{\kappa}_{\rho}(\tau \to \sigma) & \triangleq & \mathsf{pol}^{\kappa^{\updownarrow}}_{\rho}(\tau) \to \mathsf{pol}^{\kappa}_{\rho}(\sigma) \\
\mathsf{pol}^{\kappa}_{\rho}(\forall X.\tau) & \triangleq & \forall X.\mathsf{pol}^{\kappa}_{\rho\cdot[X \mapsto \kappa^{\updownarrow}]}(\theta) \\
\mathsf{pol}^{\kappa}_{\rho}(X) & \triangleq & X^{\rho(X)} \\
\mathsf{pol}^{\kappa}_{\rho}(\exists X.\tau) & \triangleq & \exists X.\mathsf{pol}^{\kappa}_{\rho\cdot[X \mapsto \kappa]}(\theta)
\end{array}
$$

with $(\cdot)^{\updownarrow} \triangleq \begin{cases} \oplus \mapsto \ominus \\ \ominus \mapsto \oplus \end{cases}$

Abstract values for Polymorphism

- Types exchanged by the two players are represented by *type names* α, β.
  - Type generativity to distinguish multiple uses of the same existential type $\exists X.\tau$.
- Values exchanged as type X (i.e. polymorphic values) are represented by *polymorphic names* p, q.
  - Boxing discipline for polymorphic values.
  - Polymorphic names introduced by a Player can be replayed multiple times by the other player.

Negative ($\ominus$)	Positive ($\oplus$)	P-Positive	O-Positive	Neutral ($\odot$)
$\to, \forall$	$\mathtt{unit}, \mathtt{int}, \times, +, \exists$	$X^{\oplus}$	$X^{\ominus}$	$\mathtt{Type}$

Generating Polymorphic Abstract Values

- The decomposition $V \nearrow (A, \gamma)$ depends on the type of V:
  - We have $V \nearrow (p, [p \mapsto V])$ when V is of type $\alpha^{\oplus}$.
  - Three possible implementations:
    1. Define $\mathcal{L}_{\mathsf{I}}|\mathcal{L}_{\mathsf{Ty}}$ as one basic blocks that uses a typed focusing relation $(V, \tau) \nearrow (A, \gamma)$
    2. Uses an untyped focusing relation that can perform boxing non-deterministically, and uses $\mathcal{L}_{\mathsf{Ty}}$ to choose the right focusing according to the type.
    3. Compile System F to a language with explicit boxing.

- In abstract values provided by Opponent, one has to replace Proponent polymorphic names p by their concrete values $\gamma(p)$
  - reversing the focusing via a reduction relation $(A, \gamma) \searrow V$.

Typing abstract values

- Linear/non-linear typing judgment $\Gamma|\Delta \Vdash A : \tau_i$ with:
  - Δ the linear context for bound names;
  - Γ the non-linear context for free names.

$$\frac{\tau \text{ function type}}{\Gamma|f : \tau \Vdash \underline{f} : \tau} \qquad \frac{}{\Gamma|p : \alpha^{\oplus} \Vdash \underline{p} : \alpha^{\oplus}} \qquad \frac{\Gamma(p) = \alpha^{\ominus}}{\Gamma|\varnothing \Vdash p : \alpha^{\ominus}}$$

$$\frac{\Gamma|\Delta \Vdash A : \tau_i}{\Gamma|\Delta \Vdash \mathtt{inj}_i(A) : \tau_1 + \tau_2} \qquad \frac{\Gamma|\Delta_1 \Vdash A_1 : \tau_1 \qquad \Gamma|\Delta_2 \Vdash A_2 : \tau_2}{\Gamma|\Delta_1 \cdot \Delta_2 \Vdash \langle A_1; A_2\rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma|\Delta \Vdash A : \tau\{X := \alpha\}}{\Gamma|\Delta, \alpha : \mathrm{Types} \Vdash \langle \alpha; A\rangle : \exists X.\tau}$$

Transitions of the Typing LTS

$$\text{PQ}\ \frac{\Delta_{\mathsf{O}}|\Delta \Vdash (A, c) : \Delta_{\mathsf{O}}(f)^{\perp}}{\langle \Delta_{\mathsf{O}} \mid \perp; \Delta_{\mathsf{P}} \rangle \xrightarrow{\overline{f}(A,c)}_{\mathsf{Ty}} \langle \Delta_{\mathsf{O}} \mid \Delta_{\mathsf{P}} \cdot \Delta \rangle}$$

$$\text{PA}\ \frac{\Delta_{\mathsf{O}}|\Delta \Vdash A : \Delta_{\mathsf{O}}(c)^{\perp}}{\langle \Delta_{\mathsf{O}} \mid \perp; \Delta_{\mathsf{P}} \rangle \xrightarrow{\overline{c}(A)}_{\mathsf{Ty}} \langle \Delta_{\mathsf{O}} \mid \Delta_{\mathsf{P}} \cdot \Delta \rangle}$$

$$\text{OQ}\ \frac{\Delta_{\mathsf{P}}|\Delta \Vdash (A, c) : \Delta_{\mathsf{P}}(f)^{\perp^{\updownarrow}}}{\langle \Delta_{\mathsf{O}} \mid \Delta_{\mathsf{P}} \rangle \xrightarrow{f(A,c)}_{\mathsf{Ty}} \langle \Delta_{\mathsf{O}} \cdot \Delta \mid \perp; \Delta_{\mathsf{P}} \rangle}$$

$$\text{OA}\ \frac{\Delta_{\mathsf{P}}|\Delta \Vdash A : \Delta_{\mathsf{P}}(c)^{\perp^{\updownarrow}}}{\langle \Delta_{\mathsf{O}} \mid \Delta_{\mathsf{P}} \rangle \xrightarrow{c(A)}_{\mathsf{Ty}} \langle \Delta_{\mathsf{O}} \cdot \Delta \mid \perp; \Delta_{\mathsf{P}} \rangle}$$

An example of polymorphic interaction

- Interaction at type $\forall X.\exists Y.(X \to Y) \times (Y \to Y)$
- between the term $V \triangleq \Lambda X.\langle X; \langle \lambda x.x; \lambda x.x\rangle\rangle$.
- and the context
  $\mathtt{let}\ z\ = \bullet\mathtt{unit}\ \mathtt{in}\ \mathtt{match}\ z\ \mathtt{with}\langle Y; \langle w_1; w_2\rangle\rangle \Rightarrow w_2(w_1())$
- generates the trace

$$\overline{c_0}(f) \cdot f(\alpha, c_1) \cdot \overline{c_1}(\beta; \langle g_1; g_2\rangle) \cdot g_1(p, c_2) \cdot \overline{c_2}(q) \cdot g_2(q, c_3) \cdot \overline{c_3}(p)$$

Adding mutable references

- We extend the programming language with
  - reference creation $\texttt{ref}\, M$ of type $\tau\ \mathrm{ref}$,
  - assignation $M := N$
  - dereferencing $!M$
- Stores $\mathtt{S}$ are partial maps from locations ℓ to values used to keep track of the values of references
- Operational semantics is defined via the following reduction rules:

$$
\begin{array}{lcl}
(E[\texttt{ref}\, V]; \mathtt{S}) & \mapsto_{\mathsf{op}} & (E[\ell]; \mathtt{S} \cdot [\ell \mapsto V]) \\
(E[\ell := V; \mathtt{S}]) & \mapsto_{\mathsf{op}} & (E[()]; \mathtt{S}[\ell \mapsto V]) \\
(E[!\ell]; \mathtt{S}) & \mapsto_{\mathsf{op}} & (E[\mathtt{S}(\ell)]; \mathtt{S})
\end{array}
$$

- Locations ℓ are parts of patterns
  - to compare physical equality of locations

Interactive LTS in presence of references

$$\text{op}\ \frac{(M; \mathtt{S}) \mapsto_{\mathsf{op}} (N; \mathtt{T})}{\langle [c]M; \mathtt{S}; \gamma \rangle \xrightarrow{\mathsf{op}} \langle [c]N; \mathtt{T}; \gamma \rangle}$$

$$\text{PQ}\ \frac{V \nearrow (A; \gamma')}{\langle K[fV]; \mathtt{S}; \gamma \rangle \xrightarrow{\overline{f}(A,c)} \langle \mathtt{S}; \gamma \cdot \gamma' \cdot [c \mapsto K] \rangle} \qquad \frac{V \nearrow (A; \gamma')}{\langle [c]V; \mathtt{S}; \gamma \rangle \xrightarrow{\overline{c}(A)} \langle \mathtt{S}; \gamma \cdot \gamma' \rangle}\ \text{PA}$$

$$\text{OQ}\ \frac{}{\langle \mathtt{S}; \gamma \rangle \xrightarrow{f(A,c)} \langle [c]\gamma(f)A; \mathtt{S}; \gamma \rangle} \qquad \frac{}{\langle \mathtt{S}; \gamma \rangle \xrightarrow{c(A)} \langle \gamma(c)[M]; \mathtt{S}; \gamma \rangle}\ \text{OA}$$

For sake of simplicity here:

- Only locations ℓ of type $\mathtt{unit}\ \mathrm{ref}$ are allowed to appear in abstract values A
- General case: need move-with-abstract-store $(\mathsf{m}; R)$ with R representing the abstract values stored in the disclosed part of $\mathtt{S}$.

Enforcing non-omniscience

- Keep track of the locations that are disclosed by one player to the other
  - escape analysis of locations.
- Via an LTS $\mathcal{L}_{\mathsf{discl}}$ whose configurations are sets D of locations known by both P and O.
- Enforce Opponent's *non-omniscience*: O cannot play a Proponent's location that has not been disclosed.

$$\frac{\mathsf{fn}(\mathsf{m}) \subseteq D \qquad \mathsf{bn}(\mathsf{m}) \cap D = \varnothing}{\langle D \rangle \xrightarrow{\mathsf{m}} \langle D \cup \mathsf{bn}(\mathsf{m}) \rangle}$$

Exceptions and Effect Type System (j.w.w. with Hamza Jaâfar)

- We extend the programming language with:
  - exception creation $\texttt{exception e of } \tau$, of type $\tau \mathrm{exn}$,
  - which generates exception values e similar to locations ℓ;
  - raise operation of exceptions $\texttt{raise} V$, which inhabit any type τ.
- both $\mathsf{e}\ A$ and $\texttt{raise}\ (\mathsf{e}\ A)$ are part of patterns.
- Effect type system $\tau \rightarrow_{\varepsilon} \sigma$ with ε a set of exceptions $\{\mathsf{e}_1, \ldots, \mathsf{e}_i\}$
  - to restrict Opponent's behavior wrt the exceptions it can raise.
- Generalization to algebraic effects and handlers: WIP.

Composing OGS configurations

- Specifying both the behavior of Proponent and Opponent by programs.
- Via a synchronization process defined as a parallel composition plus hiding of LTSs.

Definition (Parallel composition with hiding)

Taking $\mathcal{L}_1 = (\mathrm{States}_1, \mathrm{Actions}, \rightarrow_1)$ and $\mathcal{L}_2 = (\mathrm{States}_2, \mathrm{Actions}, \rightarrow_2)$ two LTSs sharing the same set of actions, their *parallel composition with hiding* $\mathcal{L}_1 || \mathcal{L}_2$, is the LTS $(\mathrm{States}_1 \times \mathrm{States}_2, \{\mathsf{op}, \mathsf{sync}\}, \rightarrow)$ with $\rightarrow$ defined as:

$$\frac{\mathbb{I}_1 \xrightarrow{\mathsf{op}}_1 \mathbb{J}_1}{(\mathbb{I}_1; \mathbb{I}_2) \xrightarrow{\mathsf{op}} (\mathbb{J}_1; \mathbb{I}_2)} \qquad \frac{\mathbb{I}_1 \xrightarrow{\mathsf{op}}_2 \mathbb{J}_2}{(\mathbb{I}; \mathbb{S}) \xrightarrow{\mathsf{op}} (\mathbb{I}_1; \mathbb{J}_2)} \qquad \frac{\mathbb{I}_1 \xrightarrow{\mathsf{m}}_1 \mathbb{J}_1 \qquad \mathbb{I}_2 \xrightarrow{\bar{\mathsf{m}}}_2 \mathbb{J}_2}{(\mathbb{I}_1; \mathbb{I}_2) \xrightarrow{\mathsf{sync}} (\mathbb{J}_1; \mathbb{J}_2)}$$

Full-abstraction of trace equivalence

Theorem

OGS Trace equivalence and contextual equivalence of the programming language coincide.

- Soundness relies on a congruence and adequacy result for the parallel composition with hiding operator.
- Full-abstraction relies on a definability result to transform a trace t into a term that generates this trace;
  - mutable references is crucial for this result to hold;
  - needs to relax the notion of trace equivalence to *complete trace equivalence*: traces where all questions have been answered.

Guilhem Jaber and Nikos Tzevelekos. "Trace Semantics for Polymorphic References". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '16

Towards Model-Checking

How to automatically check safety properties of the OGS LTS?

- For example "The LTS does not reach a configuration of the shape $\langle [c_0]\texttt{raise e}; \texttt{S}; \gamma \rangle$ unless e has been raised by Opponent before".
- Difficult to prove automatically because the interactions can be arbitrary long due to
  - recursion for Proponent;
  - absence of restriction on the number of time function names can be called by Opponent.

Can we bound Opponent's behavior?

Linearizing the OGS LTS

- Opponent uses values and contexts provided by Opponent only once, when provided:

$$\text{op}\ \frac{M \mapsto_{\text{op}} N}{\langle [c]M \rangle \xrightarrow{\text{op}} \langle [c]N \rangle}$$

$$\text{PQ}\ \frac{V \nearrow (A;\gamma)}{\langle K[fV] \rangle \xrightarrow{\overline{f}(A,c)} \langle \gamma \cdot [c \mapsto K] \rangle} \qquad \frac{V \nearrow (A;\gamma)}{\langle [c]V \rangle \xrightarrow{\overline{c}(A)} \langle \gamma \rangle}\ \text{PA}$$

$$\text{OQ}\ \frac{}{\langle \gamma \rangle \xrightarrow{f(A,c)} \langle [c]\gamma(f)A \rangle} \qquad \frac{}{\langle \gamma \rangle \xrightarrow{c(A)} \langle \gamma(c)[A] \rangle}\ \text{OA}$$

- This LTS generates a call-by-value variant of Bohm trees.

Correspondence of bisimilarities

- Bisimilarity for this LTS is exactly Lassen's eager normal-form bisimilarity.
  Søren B. Lassen. "Eager normal form bisimulation". In: *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*. 2005

- For the pure cbv λ-calculus, OGS bisimilarity and eager normal-form bisimilarity coincide.
  Guilhem Jaber and Davide Sangiorgi. "Games, mobile processes, and functions". In: *30th EACSL Annual Conference on Computer Science Logic (CSL 2022)*. 2022

In presence of local references, this linearization is not sound anymore!

$$\text{op}\ \frac{(M;\mathtt{S}) \mapsto_{\text{op}} (N;\mathtt{T})}{\langle [c]M;\mathtt{S}\rangle \xrightarrow{\text{op}} \langle [c]N;\mathtt{T}\rangle}$$

$$\text{PQ}\ \frac{V \nearrow (A;\gamma)}{\langle K[fV];\mathtt{S}\rangle \xrightarrow{\overline{f}(A,c)} \langle \mathtt{S};\gamma\cdot[c\mapsto K]\rangle} \qquad \frac{V \nearrow (A;\gamma)}{\langle [c]V;\mathtt{S}\rangle \xrightarrow{\overline{c}(A)} \langle \mathtt{S};\gamma\rangle}\ \text{PA}$$

$$\text{OQ}\ \frac{}{\langle \mathtt{S};\gamma\rangle \xrightarrow{f(A,c)} \langle [c]\gamma(f)A;\mathtt{S}\rangle} \qquad \frac{}{\langle \mathtt{S};\gamma\rangle \xrightarrow{c(A)} \langle \gamma(c)[A];\mathtt{S}\rangle}\ \text{OA}$$

```
let count = ref 0 (* private *)
let inc () = count := !count+2
let check () = assert (not(odd !count))
```

One needs to test `inc` and `check` on all *reachable* values stored in `count`.

Store invariants

Consider formulas specifying sets of stored values, for example

$$\{[\texttt{count} \mapsto 2k] \mid k \in \mathbb{N}\}$$

$$\text{op}\ \frac{(M;\mathtt{S}) \mapsto_{\text{op}} (N;\mathtt{T})}{\langle M;\mathtt{S};\mathcal{I}\rangle \xrightarrow{\text{op}} \langle N;\mathtt{T};\mathcal{I}\rangle}$$

$$\text{PQ}\ \frac{V \nearrow (A;\gamma) \quad \mathtt{S} \in \mathcal{I}}{\langle K[fV];\mathtt{S};\mathcal{I}\rangle \xrightarrow{\overline{f}(A,c)} \langle \mathcal{I};\gamma \cdot [c \mapsto K]\rangle} \qquad \frac{V \nearrow (A;\gamma) \quad \mathtt{S} \in \mathcal{I}}{\langle [c]V;\mathtt{S};\mathcal{I}\rangle \xrightarrow{\overline{c}(A)} \langle \mathcal{I};\gamma\rangle}\ \text{PA}$$

$$\text{OQ}\ \frac{\mathtt{S} \in \mathcal{I}}{\langle \mathcal{I};\gamma\rangle \xrightarrow{f(A,c)} \langle [c]\gamma(f)A\rangle;\mathtt{S};\mathcal{I}} \qquad \frac{\mathtt{S} \in \mathcal{I}}{\langle \mathcal{I};\gamma\rangle \xrightarrow{c(A)} \langle \gamma(c)[A];\mathtt{S};\mathcal{I}\rangle}\ \text{OA}$$

Framing store invariants

- Introduce a separating conjunction $*$ between invariants
- provide a way to add new invariants during the interaction for freshly-allocated references.

$$\text{op}\ \frac{(M;\mathtt{S}) \mapsto_{\mathsf{op}} (N;\mathtt{T})}{\langle M;\mathtt{S};\mathcal{I}\rangle \xrightarrow{\mathsf{op}} \langle N;\mathtt{T};\mathcal{I}\rangle}$$

$$\frac{V \nearrow (A;\gamma) \quad \mathtt{S} \in \mathcal{I} * \mathcal{J}}{\langle K[fV];\mathtt{S};\mathcal{I}\rangle \xrightarrow{\bar{f}(A,c)} \langle \mathcal{I} * \mathcal{J};\gamma \cdot [c \mapsto K]\rangle} \qquad \frac{V \nearrow (A;\gamma) \quad \mathtt{S} \in \mathcal{I} * \mathcal{J}}{\langle [c]V;\mathtt{S};\mathcal{I}\rangle \xrightarrow{\bar{c}(A)} \langle \mathcal{I} * \mathcal{J};\gamma\rangle}$$

$$\text{OQ}\ \frac{\mathtt{S} \in \mathcal{I}}{\langle \mathcal{I};\gamma\rangle \xrightarrow{f(A,c)} \langle [c]\gamma(f)A\rangle;\mathtt{S};\mathcal{I}} \qquad \frac{\mathtt{S} \in \mathcal{I}}{\langle \mathcal{I};\gamma\rangle \xrightarrow{c(A)} \langle \gamma(c)[A];\mathtt{S};\mathcal{I}\rangle}\ \text{OA}$$

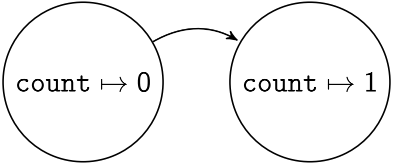
Invariants

- Computed via symbolic evaluation $(M; \mathcal{I}) \mapsto_{\mathsf{op}} (N; \mathcal{J})$
  - via *predicate transformer* semantics.
- For location disclosure ($\texttt{unit}\ \mathrm{ref}$): invariants are enough
  - WIP with Daniel Hirshckoff and Enguerrand Prebet.
- For polymorphic values, we *conjecture* that invariants are enough too
  - if true, this would have important consequences for full-abstraction of logical relations for System F.

Invariants are not enough for mutable store

```
let count = ref 0 (* private *)
let awk f = count:= 1; f(); assert (!count=1)
```

One needs a *transition system of invariants*



- $(\mathrm{Worlds}, \sqsubset)$ with worlds $\mathcal{W} \in \mathrm{Worlds}$ of the shape $(s, \mathcal{I})$ with s an abstract state and $\mathcal{I}$ an invariant.
- Allow Opponent to navigate arbitrarily far in this transition system
  - using the reflexive-transitive closure $\sqsubseteq^*$.

Derek Dreyer, Georg Neis, and Lars Birkedal. "The impact of higher-order state and control effects on local relational reasoning". In: *Journal of Functional Programming* 22.4-5 (2012), pp. 477–528

Kripke OGS

$$\text{op}\ \frac{(M;\mathtt{S}) \mapsto_{\mathsf{op}} (N;\mathtt{T})}{\langle M;\mathtt{S};\mathcal{W}\rangle \xrightarrow{\mathsf{op}} \langle N;\mathtt{T};\mathcal{W}\rangle}$$

$$\text{PQ}\ \frac{V \nearrow (A;\gamma) \quad \mathcal{W} \sqsubset \mathcal{W}' \quad \mathtt{S} \in \mathcal{W}'.\mathcal{I}}{\langle K[fV];\mathtt{S};\mathcal{W}\rangle \xrightarrow{\overline{f}(A,c)} \langle \mathcal{W}';\gamma\cdot[c \mapsto K]\rangle}$$

$$\text{PA}\ \frac{V \nearrow (A;\gamma) \quad \mathcal{W} \sqsubset \mathcal{W}' \quad \mathtt{S} \in \mathcal{W}'.\mathcal{I}}{\langle [c]V;\mathtt{S};\mathcal{W}\rangle \xrightarrow{\overline{c}(A)} \langle \mathcal{W};\gamma\rangle}$$

$$\text{OQ}\ \frac{\mathcal{W} \sqsubseteq^* \mathcal{W}' \quad \mathtt{S} \in \mathcal{W}'.\mathcal{I}}{\langle \mathcal{W};\gamma\rangle \xrightarrow{f,c(A)} \langle [c]\gamma(f)A;\mathtt{S};\mathcal{W}'\rangle} \qquad \frac{\mathcal{W} \sqsubseteq^* \mathcal{W}' \quad \mathtt{S} \in \mathcal{W}'.\mathcal{I}}{\langle \gamma\rangle \xrightarrow{c(A)} \langle \mathtt{S};\mathcal{W}';\gamma(c)[A]\rangle}\ \text{OA}$$

Kripke Eager Normal Form Bisimulation

- Bisimilarity over this LTS
  - using worlds with relational invariants over store
  - incorporates the well-bracketed constraints of $\mathcal{L}_{\mathrm{WB}}$ by distinguishing OQ and OA evolutions of worlds
- Fully-abstract for the a language with higher-order references.

Guilhem Jaber and Andrzej S. Murawski. "Compositional Relational Reasoning via Operational Game Semantics". In: *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*. 2021

Temporal reasoning (WIP with Andrzej Murawski)

- Abstract away the worlds
  - using temporal modalities $\diamond$ and $\Box$.
- Distinguish OQ and OA evolutions of worlds
  - OA transitions of worlds are well-bracketed
  - represented using visibly-pushdown modalities.
- Interactions inside `while`-loops represented using greatest fixpoints formulas of the μ-calculus.

Conclusion

- First steps towards automated verifications of effectul polymorphically typed programs.
  - Safety property as a model-checking problem over infinite trees
  - with linearized Opponent behavior
  - and temporal reasoning over resources.

- Future work:
  - Other OCaml's features
    - GADTs;
    - module system.
  - Invariant generation
    - via weakest precondition generators;
    - abstract interpretation;
    - abstract domains for ADTs.
  - Reducing the model-checking problem to satisfiability of Constrained Horn Clauses.
  - Implementation!