CAVOC: Compositional Automated Verification of OCaml

Guilhem Jaber
Gallinette Team, LS2N, Univ. Nantes

Journée scientifique Inria - Nomadic Labs
1st of June

CAVOC in a nutshell

- Started in September 2021
- Members:
  - Hamza Jaâfar (PhD student, Gallinette team)
  - Guilhem Jaber (maître de conférences, Nantes Univ., Gallinette team)
  - Gabriel Radanne (Inria researcher, Cash team)
- External member:
  - Laure Gonnord (professor, ESISAR, INP Grenoble, LCIS)

- Objective:

  Automatically check module-safety of OCaml code

- Based on:
  - Game semantics
  - Higher-order model checking
  - Constrained Horn clauses

Module-safety ?

```
module M : sig
  type t
  val get : unit → t
  val check : t → unit
end = struct
  type t = int
  let c = ref 0
  let get () = c :=!c + 1; !c
  let check x = assert (x > 0)
end
```

Check that no OCaml code that uses this module trigger the assert

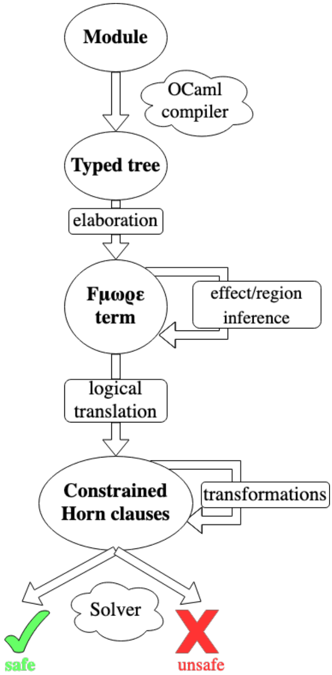
Long-term goal: Safety of the Buffer module

"This module implements buffers that automatically expand as necessary."

```
(* Excerpt of the signature:*)
type t
val create : int -> t
val contents : t -> string
val add_string : t -> string -> unit

(* Example of code *)
let concat_strings ss =
  let b = Buffer.create 16 in
    List.iter (Buffer.add_string b) ss;
    Buffer.contents b
```

- Abstract type `t` is implemented as a record with some mutable `byte` field;
- Resizing done by `add_string` is safe: no out of range access in the byte field.


Module
OCaml
compiler
Typed tree
elaboration
Fμωρε
term
effect/region
inference
logical
translation
Constrained
Horn clauses
transformations
Solver
safe
unsafe

F$\mu\omega\rho\varepsilon$

- F: Church-style parametric polymorphism
- μ: iso-recursive types
- ω: type constructors, higher-order polymorphism
- ρ: mutable references and region type system
- ε: exceptions and effect type system

From OCaml to F$\mu\omega\rho\varepsilon$

- Use the OCaml compiler to parse and infer types
  - producing a `Typedtree`, the typed abstract syntax tree used internally by the compiler
- Elaborate the `Typedtree` into F$\mu\omega\rho\varepsilon$
  - introduce variants and records construction into F$\mu\omega\rho\varepsilon$
  - translate GADT via equality constraints and existential types
  - applicative/generative functors via the F-ing module methodology

Effect and Region inference for F$\mu\omega\rho\varepsilon$

Annotate function types with their associated effects:

- uncaught exceptions
  - using row polymorphism
  - useful for benign exceptions (like `Not_found`)
- mutable references
  - using regions associated to syntactic allocation points
  - detect location disclosure
  - provide aliasing information
- detect pure (effect-free) functions
  - easier to analyze

A fully-abstract game model for F$\mu\omega\rho\varepsilon$

- Interaction between a F$\mu\omega\rho\varepsilon$ program and its environment is represented as a **play** between **Proponent** (the program) and **Opponent** (the environment).
- Denotation of a program is formed by all the possible interactions with any environment.
- Plays in a denotation should be in exact correspondence with environment written in F$\mu\omega\rho\varepsilon$: **full-abstraction**.

Operational presentation of game semantics

- Plays correspond to **traces** formed by calls and returns of functions exchanged between Proponent and Opponent.

- Such traces are computed by a labelled transition system representing the denotation of programs

$\leadsto$ by computing interaction on the fly using operational semantics

Typing as behavioral specification

Types specifies the rules of the game between Player and Opponent:

- based on a polarized interpretation:
  - Interacting with values of **negative** types (codata)
  - Observing values of **positive** types (data)
- abstract values for polymorphic types are represented as atoms that can only be exchanged
  - computational interpretation of parametricity as dynamic sealing
- effect and region annotations constrains the behavior of Opponent.

Scaling to more complex interaction models

- Asynchronous callbacks for signal handlers;
- Finalizers from garbage collected values;
- Asynch/Lwt's promises;
- Multiple domains running in parallel (OCaml 5);
- algebraic effect and handlers with one-shot continuation (OCaml 5).

Future work!

Towards symbolic representation of the interactive denotation of F$\mu\omega\rho\varepsilon$ terms

Main challenges:

- Disentangle the internal control flow (recursion, interprocedural) of a module with its interaction with Opponent;
- Reason symbolically on values exchanged between Proponent and Opponent;
- Represent dynamical allocation and the heap structure logically.

Constrained Horn Clauses

- A first-order formula of the shape

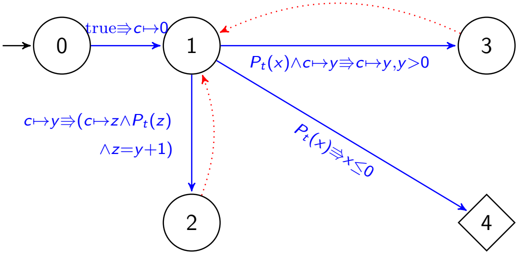
$$\forall \bar{x}. C \wedge B_1 \wedge \cdots \wedge B_n \Rightarrow H$$

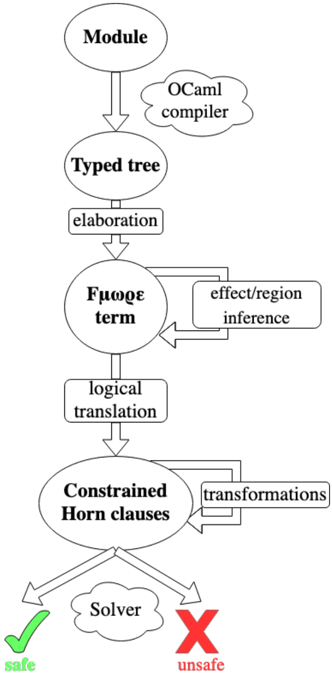
  - C a constraint formula written in a specified theory (linear integer arithmetic, arrays, algebraic data-types,...)
  - each B_i is a of the form $r(t_1, \ldots, t_m)$ with r an uninterpreted relation symbol;
  - H is either false or of the form $r(t_1, \ldots, t_m)$ as well.

- Solvers for checking satisfiability of CHC: z3, Eldarica, RInGen;
- Used for automated verification of various programming languages
  - C (SeaHorn), Java (JayHorn), Ada (AdaHorn); Rust (RustHorn)

- Simplification of CHCs via some transformations
  - Elimination of algebraic data-types, arrays, heaps
  - From non-linear to linear CHCs.

```
module M : sig
  type t
  val get : unit → t
  val check : t → unit
end = struct
  type t = int
  let c = ref 0
  let get () = c :=!c + 1; !c
  let check x = assert (x > 0)
end
```




Module
OCaml compiler
Typed tree
elaboration
Fμωρε term
effect/region inference
logical translation
Constrained Horn clauses
transformations
Solver
safe
unsafe