The Design and Implementation of An Abstract Interpreter for OCaml Programs

A Preliminary Report on the Salto Analyser

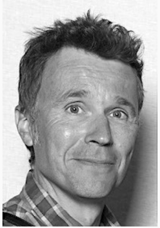
Benoît Montagu, Inria

ML workshop, Seattle — 2023, September 8th

The Salto Project

- **What:** static analysis for OCaml programs
  https://salto.gitlabpages.inria.fr/
- **Where:** Inria Rennes
- **Who:**

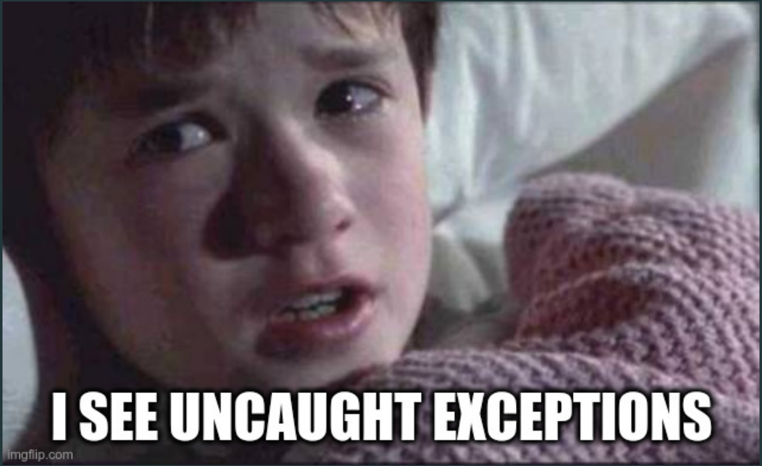

P. Lermusiaux


T. Genet


T. Jensen


B. Montagu

- **Funding:** Inria + Nomadic Labs


I SEE UNCAUGHT EXCEPTIONS
imgflip.com

Static Analysis of OCaml programs: What For?

Short-term goals:

- Detect uncaught exceptions
  - User-provided assertions
  - Missing exception handlers (e.g., `Division_by_zero`)
  - Out of bounds accesses for arrays, strings, ...
  - Polymorphic comparison on functions
- Detect illegal uses of unsafe functions (e.g., `String.unsafe_get`)

Static Analysis of OCaml programs: What For?

Short-term goals:

- Detect uncaught exceptions
  - User-provided assertions
  - Missing exception handlers (e.g., `Division_by_zero`)
  - Out of bounds accesses for arrays, strings, ...
  - Polymorphic comparison on functions
- Detect illegal uses of unsafe functions (e.g., `String.unsafe_get`)

Longer-term goals:

- Support most of the OCaml language
- Detect unhandled algebraic effects
- Detect some undefined behaviours (e.g., sensitivity to evaluation order)

Static Analysis of OCaml programs: What For?

Short-term goals:

- Detect uncaught exceptions
  - User-provided assertions
  - Missing exception handlers (e.g., `Division_by_zero`)
  - Out of bounds accesses for arrays, strings, …
  - Polymorphic comparison on functions
- Detect illegal uses of unsafe functions (e.g., `String.unsafe_get`)

Longer-term goals:

- Support most of the OCaml language
- Detect unhandled algebraic effects
- Detect some undefined behaviours (e.g., sensitivity to evaluation order)

Out of scope (for now):

- Concurrency, parallelism
- Support for the `Obj` module

Static Analysis of OCaml programs: What For?

Short-term goals:

This talk

- Detect uncaught exceptions
  - User-provided assertions
  - Missing exception handlers (e.g., `Division_by_zero`)
  - Out of bounds accesses for arrays, strings, ...
  - Polymorphic comparison on functions
- Detect illegal uses of unsafe functions (e.g., `String.unsafe_get`)

Longer-term goals:

- Support most of the OCaml language
- Detect unhandled algebraic effects
- Detect some undefined behaviours (e.g., sensitivity to evaluation order)

Out of scope (for now):

- Concurrency, parallelism
- Support for the `Obj` module

Two families of static analyses:

- Type and effect systems:
  - ⊕ Modular, good performance
  - ⊖ Limited precision for user-provided assertions

Xavier Leroy and François Pessaux. “Type-Based Analysis of Uncaught Exceptions”. In: *ACM Trans. Program. Lang. Syst.* 22.2 (2000), pp. 340–377. DOI: 10.1145/349214.349230

Two families of static analyses:

- Type and effect systems:
  - ➕ Modular, good performance
  - ➖ Limited precision for user-provided assertions

Xavier Leroy and François Pessaux. "Type-Based Analysis of Uncaught Exceptions". In: *ACM Trans. Program. Lang. Syst.* 22.2 (2000), pp. 340–377. DOI: 10.1145/349214.349230

- Extensions of control-flow analyses (CFA):
  - ➖ Not modular, more costly
  - ➕ Decent precision for user-provided assertions

Kwangkeun Yi. "Compile-time Detection of Uncaught Exceptions in Standard ML Programs". In: *Static Analysis, First International Static Analysis Symposium, SAS'94, Namur, Belgium, September 28-30, 1994, Proceedings*. Ed. by Baudouin Le Charlier. Vol. 864. Lecture Notes in Computer Science. Springer, 1994, pp. 238–254. DOI: 10.1007/3-540-58485-4_44

A Whole-Program Value Analysis for OCaml programs

- **Principle:**
  For every reachable sub-expression e of a program, compute:
  - A superset of the values that e may evaluate to, and
  - A superset of the exceptions e might raise
  - An approximation of the call stack where the exception was raised

  Expressions that are known to be unreachable are not analysed

  👉 Only the functions that are called are analysed

- **Principle:**
  For every reachable sub-expression e of a program, compute:
  - A superset of the values that e may evaluate to, and
  - A superset of the exceptions e might raise
  - An approximation of the call stack where the exception was raised

  Expressions that are known to be unreachable are not analysed

  ☞ Only the functions that are called are analysed
- **Technique:** based on the abstract interpretation of λ-calculus developed in

Benoît Montagu and Thomas P. Jensen. "Trace-Based Control-Flow Analysis". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021.* Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 482–496. DOI: 10.1145/3453483.3454057

We analyse programs *as if* they were untyped

A Whole-Program Value Analysis for OCaml programs

- **Principle:**
  For every reachable sub-expression e of a program, compute:
  - A superset of the values that e may evaluate to, and
  - A superset of the exceptions e might raise
  - An approximation of the call stack where the exception was raised

  Expressions that are known to be unreachable are not analysed

  ☞ Only the functions that are called are analysed
- **Technique:** based on the abstract interpretation of λ-calculus developed in

Benoît Montagu and Thomas P. Jensen. "Trace-Based Control-Flow Analysis". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021.* Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 482–496. DOI: `10.1145/3453483.3454057`

  We analyse programs *as if* they were untyped
- **Novelty:** an abstract domain to represent recursively defined sets of values

A Whole-Program Value Analysis for OCaml programs

- **Principle:**
  For every reachable sub-expression e of a program, compute:
  - A superset of the values that e may evaluate to, and
  - A superset of the exceptions e might raise
  - An approximation of the call stack where the exception was raised

  Expressions that are known to be unreachable are not analysed

  ☞ Only the functions that are called are analysed
- **Technique:** based on the abstract interpretation of λ-calculus developed in

Benoît Montagu and Thomas P. Jensen. "Trace-Based Control-Flow Analysis". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021.* Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 482–496. DOI: 10.1145/3453483.3454057

  We analyse programs *as if* they were untyped
- **Novelty:** an abstract domain to represent recursively defined sets of values
- **Implementation:** uses a dynamic fixpoint solver

What We Have Achieved So Far

- An abstract interpreter (big-step style) that supports:
  - ⊕ Higher-order programs
  - ⊕ Mutually-recursive functions
  - ⊕ Algebraic values, deep pattern matching
  - ⊕ Integers, strings, characters...
  - ⊕ Exceptions
  - ⊕ Modules and functors (first class, non-recursive)
  - ⊖ No mutable state yet
  - ⊖ No laziness
  - ⊖ No objects/classes
  - ⊖ No OCaml 5 features
- The analyser is *parameterised* over the abstract domain for values

What We Have Achieved So Far

- An abstract interpreter (big-step style) that supports:
  - ⊕ Higher-order programs
  - ⊕ Mutually-recursive functions
  - ⊕ Algebraic values, deep pattern matching
  - ⊕ Integers, strings, characters...
  - ⊕ Exceptions
  - ⊕ Modules and functors (first class, non-recursive)
  - ⊖ No mutable state yet
  - ⊖ No laziness
  - ⊖ No objects/classes
  - ⊖ No OCaml 5 features
- The analyser is *parameterised* over the abstract domain for values
- A forward analysis: it is not guided by user-written formulas
- The analysis is context- and flow-sensitive
- A non-relational analysis: no relations between values/variables are inferred

What We Have Achieved So Far

- An abstract interpreter (big-step style) that supports:
  - ⊕ Higher-order programs
  - ⊕ Mutually-recursive functions
  - ⊕ Algebraic values, deep pattern matching
  - ⊕ Integers, strings, characters...
  - ⊕ Exceptions
  - ⊕ Modules and functors (first class, non-recursive)
  - ⊖ No mutable state yet
  - ⊖ No laziness
  - ⊖ No objects/classes
  - ⊖ No OCaml 5 features
- The analyser is *parameterised* over the abstract domain for values
- A forward analysis: it is not guided by user-written formulas
- The analysis is context- and flow-sensitive
- A non-relational analysis: no relations between values/variables are inferred
- **Demo!** `list_filter` `map_merge` `mc91` `insert_sorted_list`

An Abstract Domain For Sets Of Values (simplified)

A finite representation for *recursively defined* sets of *untyped* values:

$$
\begin{array}{rcl}
v^\sharp \in \mathbb{V}^\sharp & = & \{ \ \mathsf{ints} = d \in \mathbb{Z}^\sharp; \\
& & \quad \mathsf{variants} = \{\mathsf{c}_1 \mapsto v^\sharp; \ldots; \mathsf{c}_n \mapsto v^\sharp\}; \\
& & \quad \mathsf{pairs} = (v^\sharp, v^\sharp); \\
& & \quad \mathsf{funs} = \{(\lambda^\ell \mathsf{x}.\, t) \mapsto [\mathsf{x}_1 \mapsto v^\sharp; \ldots; \mathsf{x}_n \mapsto v^\sharp]; \ldots\} \quad \} \\
& | & \top
\end{array}
$$

An Abstract Domain For Sets Of Values (simplified)

A finite representation for *recursively defined* sets of *untyped* values:

$$\begin{array}{rcl} v^\sharp \in \mathbb{V}^\sharp & = & \{\ \mathsf{ints} = d \in \mathbb{Z}^\sharp; \\ & & \quad \mathsf{variants} = \{\mathsf{c}_1 \mapsto v^\sharp; \dots; \mathsf{c}_n \mapsto v^\sharp\}; \\ & & \quad \mathsf{pairs} = (v^\sharp, v^\sharp); \\ & & \quad \mathsf{funs} = \{(\lambda^\ell \mathsf{x}.\, t) \mapsto [\mathsf{x}_1 \mapsto v^\sharp; \dots; \mathsf{x}_n \mapsto v^\sharp]; \dots\} \quad \} \\ & | & \top \\ & | & \alpha \quad | \quad \mu\alpha.\, v^\sharp \end{array}$$

- μ has the semantics of a least fixed point
- The widening operator detects some regularity and introduces the μs

An Abstract Domain For Sets Of Values (simplified)

A finite representation for *recursively defined* sets of *untyped* values:

$$
\begin{array}{rcl}
v^\sharp \in \mathbb{V}^\sharp & = & \{\ \mathsf{ints} = d \in \mathbb{Z}^\sharp; \\
& & \quad \mathsf{variants} = \{\mathsf{c}_1 \mapsto v^\sharp; \ldots; \mathsf{c}_n \mapsto v^\sharp\}; \\
& & \quad \mathsf{pairs} = (v^\sharp, v^\sharp); \\
& & \quad \mathsf{funs} = \{(\lambda^\ell \mathsf{x}.\, t) \mapsto [\mathsf{x}_1 \mapsto v^\sharp; \ldots; \mathsf{x}_n \mapsto v^\sharp]; \ldots\} \quad \} \\
& | & \top \\
& | & \alpha \quad | \quad \mu\alpha.\, v^\sharp
\end{array}
$$

- μ has the semantics of a least fixed point
- The widening operator detects some regularity and introduces the μs
- Example: Peano numbers

$$\mu\alpha.\{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \alpha\}\}$$

An Abstract Domain For Sets Of Values (simplified)

A finite representation for *recursively defined* sets of *untyped* values:

$$\begin{array}{rcl} v^\sharp \in \mathbb{V}^\sharp & = & \{\ \mathsf{ints} = d \in \mathbb{Z}^\sharp; \\ & & \quad \mathsf{variants} = \{\mathsf{c}_1 \mapsto v^\sharp; \ldots; \mathsf{c}_n \mapsto v^\sharp\}; \\ & & \quad \mathsf{pairs} = (v^\sharp, v^\sharp); \\ & & \quad \mathsf{funs} = \{(\lambda^\ell \mathsf{x}.\, t) \mapsto [\mathsf{x}_1 \mapsto v^\sharp; \ldots; \mathsf{x}_n \mapsto v^\sharp]; \ldots\} \quad \} \\ & | & \top \\ & | & \alpha \quad | \quad \mu\alpha.\, v^\sharp \end{array}$$

- μ has the semantics of a least fixed point
- The widening operator detects some regularity and introduces the μs
- Example: Peano numbers

$$\mu\alpha.\{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \alpha\}\}$$

- Example: A set of continuations (for CPSed factorial)

$$\mu\alpha.\left\{\mathsf{funs} = \left\{ \begin{array}{l} (\lambda^{\ell_1}\mathsf{x}.\,\mathsf{x}) \mapsto []; \\ (\lambda^{\ell_2}\mathsf{x}.\,\mathsf{k}\,(\mathsf{x} * \mathsf{n})) \mapsto [\mathsf{n} \mapsto \{\mathsf{ints} = [1, +\infty]\}; \mathsf{k} \mapsto \alpha]; \end{array} \right\}\right\}$$

Abstract Domain: Important Remarks

The design of the abstract domain draws inspiration from:

- Equi-recursive types + union types
- Type Graphs (analysis of Prolog programs)

Pascal Van Hentenryck, Agostino Cortesi and Baudouin Le Charlier. “Type Analysis of Prolog Using Type Graphs”. In: *The Journal of Logic Programming* 22.3 (Mar. 1995), pp. 179–209. DOI: `10.1016/0743-1066(94)00021-w`

- Tree grammars / Tree automata

Abstract Domain: Important Remarks

The design of the abstract domain draws inspiration from:

- Equi-recursive types + union types
- Type Graphs (analysis of Prolog programs)

Pascal Van Hentenryck, Agostino Cortesi and Baudouin Le Charlier. "Type Analysis of Prolog Using Type Graphs". In: *The Journal of Logic Programming* 22.3 (Mar. 1995), pp. 179–209. DOI: `10.1016/0743-1066(94)00021-w`

- Tree grammars / Tree automata

These abstract values admit two representations:

- **As graphs**
  ☞ Efficient algorithms for union, intersection, inclusion, emptiness test, widening, minimisation, …
- **As terms**, with bound variables
  ☞ Permits hash-consing/memoisation
  ☞ This is crucial to obtain decent performance ($\sim 10\times$ improvement!)

Related Work (1/2)

Pessaux & Leroy's effect type system:

- They infer recursive types, using unification
- They support arrow types, row variables for effects: enables modular analysis
- They do *not* infer abstract closures:
  Incurs a loss of information when using functions as first-class values
- Limited support for sets of integers: Int$[1:\mathtt{Pre};\ 3:\mathtt{Pre}]$ Int$[\top]$ Int$[\rho]$
  We support any abstract domain for integers (non-relational so far)

Related Work (2/2)

Control-Flow Analyses:

- They always avoid recursion in the abstract domain
- Recursion is obtained by means of indirections through an abstract heap

$$\left\{ \text{funs} = \left\{ \begin{array}{l} (\lambda^{\ell_1}\mathsf{x}.\,\mathsf{x}) \mapsto [\,]; \\ (\lambda^{\ell_2}\mathsf{x}.\,\mathsf{k}\,(\mathsf{x} * \mathsf{n})) \mapsto [\mathsf{n} \mapsto p_{\mathsf{n}}; \mathsf{k} \mapsto p_{\mathsf{k}}]; \end{array} \right\} \right\}$$

$$\text{where: } \hat{h}(p_{\mathsf{n}}) = \{\text{ints} = [1, +\infty]\}$$

$$\hat{h}(p_{\mathsf{k}}) = \left\{ \begin{array}{l} (\lambda^{\ell_1}\mathsf{x}.\,\mathsf{x}) \mapsto [\,]; \\ (\lambda^{\ell_2}\mathsf{x}.\,\mathsf{k}\,(\mathsf{x} * \mathsf{n})) \mapsto [\mathsf{n} \mapsto p_{\mathsf{n}}; \mathsf{k} \mapsto p_{\mathsf{k}}]; \end{array} \right\}$$

- Mimics the behaviour of a compiler: Values are allocated in the heap

Related Work (2/2)

Control-Flow Analyses:

- They always avoid recursion in the abstract domain
- Recursion is obtained by means of indirections through an abstract heap

$$\left\{ \mathsf{funs} = \left\{ \begin{array}{l} (\lambda^{\ell_1}\mathsf{x}.\,\mathsf{x}) \mapsto [\,]; \\ (\lambda^{\ell_2}\mathsf{x}.\,\mathsf{k}\,(\mathsf{x} * \mathsf{n})) \mapsto [\mathsf{n} \mapsto p_{\mathsf{n}}; \mathsf{k} \mapsto p_{\mathsf{k}}]; \end{array} \right\} \right\}$$

$$\text{where: } \hat{h}(p_{\mathsf{n}}) = \{\mathsf{ints} = [1, +\infty]\}$$

$$\hat{h}(p_{\mathsf{k}}) = \left\{ \begin{array}{l} (\lambda^{\ell_1}\mathsf{x}.\,\mathsf{x}) \mapsto [\,]; \\ (\lambda^{\ell_2}\mathsf{x}.\,\mathsf{k}\,(\mathsf{x} * \mathsf{n})) \mapsto [\mathsf{n} \mapsto p_{\mathsf{n}}; \mathsf{k} \mapsto p_{\mathsf{k}}]; \end{array} \right\}$$

- Mimics the behaviour of a compiler: Values are allocated in the heap
- In practice: inhibits sharing of equivalent abstract values
- There is a finite number of abstract pointer names:
  names are chosen based on a (finite) abstraction of the call stack
- The abstract heap is global:
  This prevents refining information when some control-flow branch is taken

Forward and Backward Analyses

Consider the following program: `if x < 42 then e1 else e2`

- To analyse `e1` with precision, we need to exploit the fact that `(x < 42)` evaluated to `true`
- This is done by running a **backward analysis** on the expression `(x < 42)`

Forward and Backward Analyses

Consider the following program: `if x < 42 then e1 else e2`

- To analyse `e1` with precision, we need to exploit the fact that `(x < 42)` evaluated to `true`
- This is done by running a **backward analysis** on the expression `(x < 42)`
- **Problem:** the condition is an arbitrary expression: it could be an application `if f x then e1 else e2`

☞ To obtain precise results, we need to know which closures `f` might evaluate to. For example, `f` could evaluate to `(fun x -> x < 42)`

Forward and Backward Analyses

Consider the following program: `if x < 42 then e1 else e2`

- To analyse `e1` with precision, we need to exploit the fact that `(x < 42)` evaluated to `true`
- This is done by running a **backward analysis** on the expression `(x < 42)`
- **Problem:** the condition is an arbitrary expression: it could be an application `if f x then e1 else e2`
- ☞ To obtain precise results, we need to know which closures `f` might evaluate to For example, `f` could evaluate to `(fun x -> x < 42)`

Forward analysis et backward analyses depend on each other!

- A problem in all interprocedural analyses
- Solution: use a **dynamic fixpoint solver**

Defining Static Analysers Using Dynamic Fixpoint Solvers

```
val fix: ((X.t -> Y.t) -> (X.t -> Y.t)) -> (X.t -> Y.t)
```

Computes a post-fixpoint of the functional passed as argument

Defining Static Analysers Using Dynamic Fixpoint Solvers

```
val fix: ((X.t -> Y.t) -> (X.t -> Y.t)) -> (X.t -> Y.t)
```

Computes a post-fixpoint of the functional passed as argument

☞ **Allows to define a big-step analyser using open recursion:**

```
fix @@ fun analyse (t, env) -> match t with
| Var x -> Env.get env x
| Lam (x, t) -> D.make_closure x t (Env.restrict env (fv (Lam (x, t))))
| App(t1, t2) ->
  let v2 = analyse (t2, env) in
  if D.is_bot v2 then D.bot else
  let v1 = analyse (t1, env) in
  D.joins (D.closures v1)
    (fun (x, t, env0) -> analyse (t, Env.add x v2 env0))
```

`fix` implements the iteration strategy of the analyser
and tracks dynamic dependencies to avoid unnecessary recomputations

Defining Static Analysers Using Dynamic Fixpoint Solvers

```
val fix: ((X.t -> Y.t) -> (X.t -> Y.t)) -> (X.t -> Y.t)
```

Computes a post-fixpoint of the functional passed as argument

- Idea pioneered by work on Prolog analysis

Pascal Van Hentenryck, Agostino Cortesi and Baudouin Le Charlier. "Type Analysis of Prolog Using Type Graphs". In: *The Journal of Logic Programming* 22.3 (Mar. 1995), pp. 179–209. DOI: 10.1016/0743-1066(94)00021-w

- Later re-emphasized (in a simpler setting)

David Darais et al. "Abstracting definitional interpreters (functional pearl)". In: *Proc. ACM Program. Lang.* 1.ICFP (2017), 12:1–12:25. DOI: 10.1145/3110256

- Actually used in a static analyser for C programs

Vesal Vojdani et al. "Static race detection for device drivers: the Goblint approach". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*. ACM, 2016, pp. 391–402. DOI: 10.1145/2970276.2970337

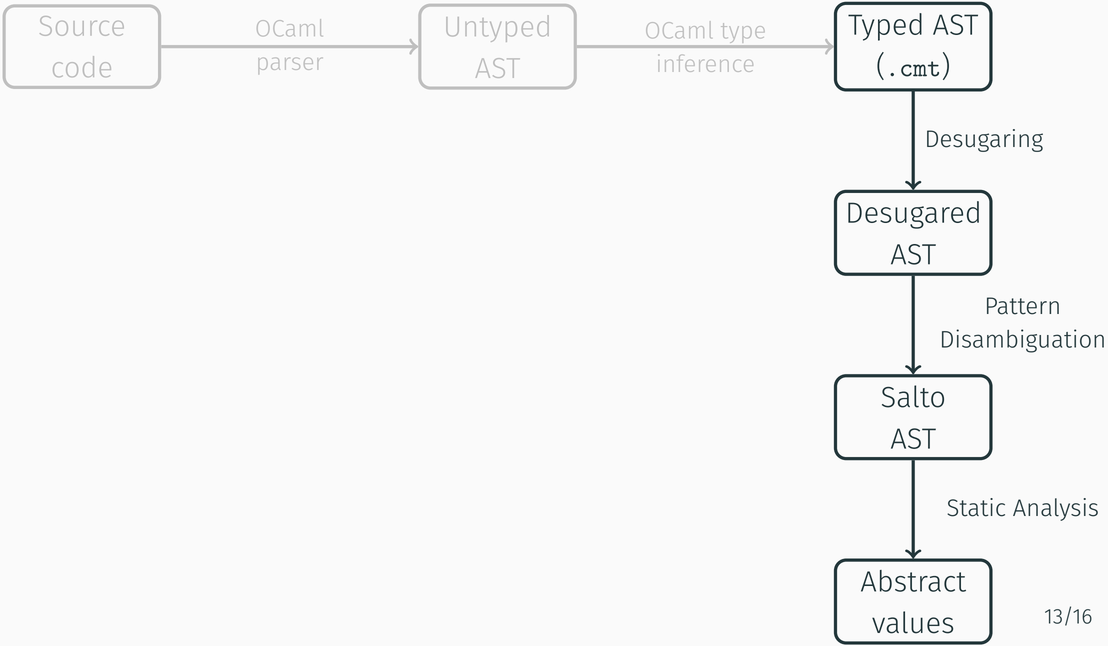
```
val fix: ((X.t -> Y.t) -> (X.t -> Y.t)) -> (X.t -> Y.t)
```

Computes a post-fixpoint of the functional passed as argument

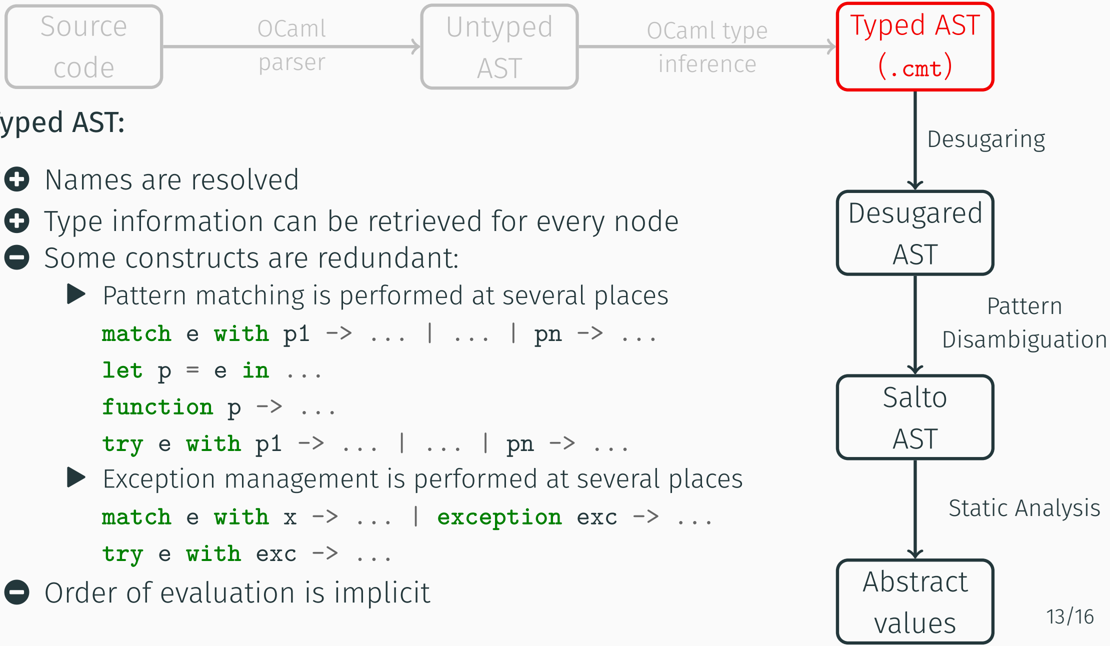
- You've heard about fixpoint solvers and static analysers at ICFP this week!

Sven Keidel, Sebastian Erdweg and Tobias Hombücher. "Combinator-Based Fixpoint Algorithms for Big-Step Abstract Interpreters". In: *Proceedings of the ACM on Programming Languages* 7.ICFP (Aug. 2023), pp. 955–981. DOI: 10.1145/3607863

High-Level Structure of the Analyser (Frontend)



High-Level Structure of the Analyser (Frontend)



Typed AST:

- ➕ Names are resolved
- ➕ Type information can be retrieved for every node
- ➖ Some constructs are redundant:
  - Pattern matching is performed at several places

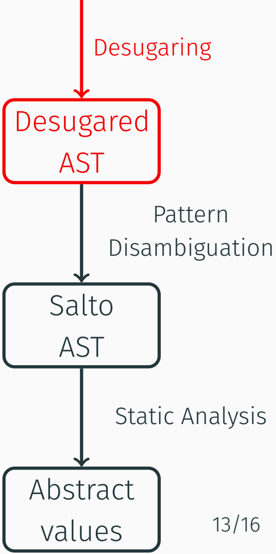
```
match e with p1 -> ... | ... | pn -> ...
let p = e in ...
function p -> ...
try e with p1 -> ... | ... | pn -> ...
```

  - Exception management is performed at several places

```
match e with x -> ... | exception exc -> ...
try e with exc -> ...
```

- ➖ Order of evaluation is implicit

High-Level Structure of the Analyser (Frontend)



Desugared AST:

- A **single** construct for pattern matching:

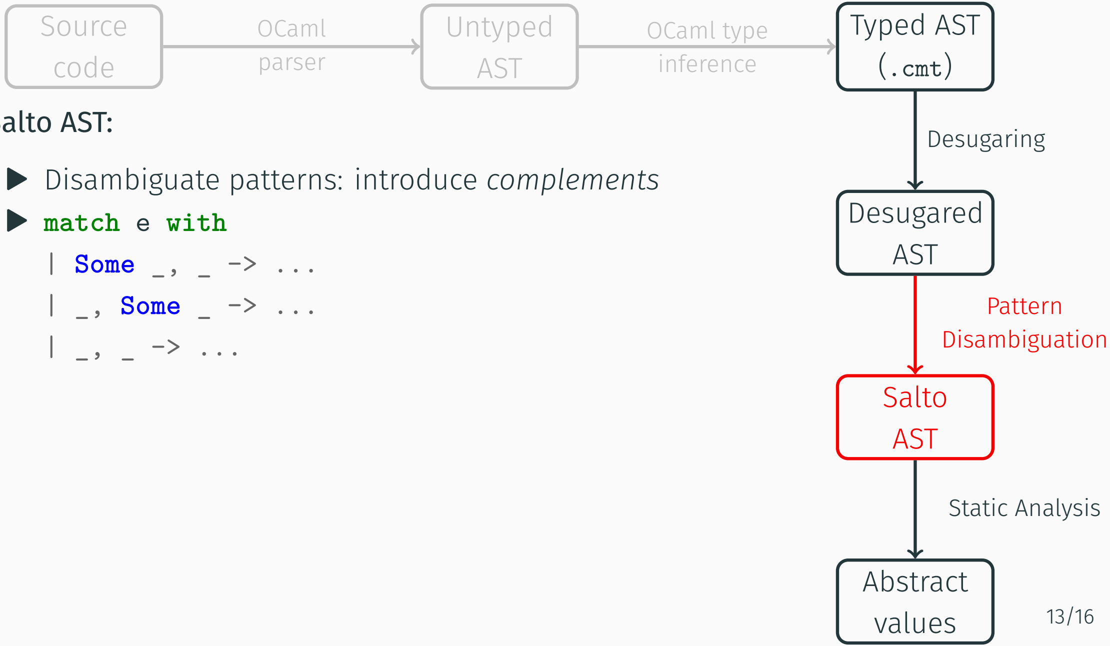
```
match e with
| p_1 -> ...
| ...
| p_n -> ...
```

- A **single** construct for exception handling:

```
dispatch e with
| val x -> ...
| exception exc -> ...
```

- Evaluation order made explicit using local `lets`, when possible (close to a “monadic normal form”)

High-Level Structure of the Analyser (Frontend)



Salto AST:

- Disambiguate patterns: introduce *complements*
-
```
match e with
| Some _, _ -> ...
| _, Some _ -> ...
| _, _ -> ...
```

High-Level Structure of the Analyser (Frontend)

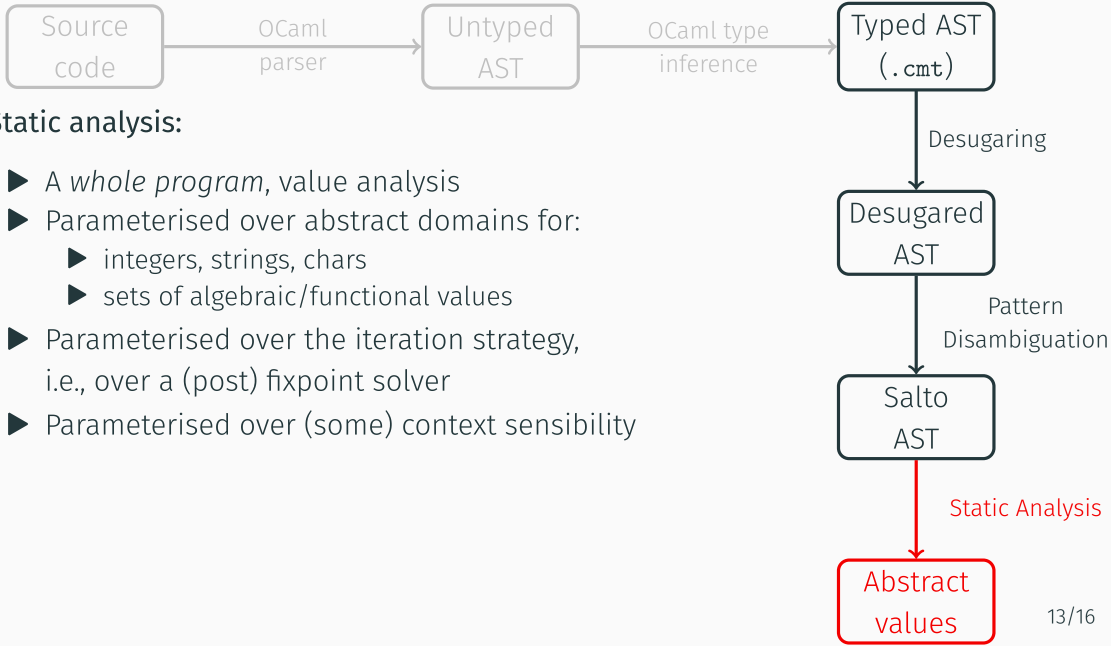
Source code → OCaml parser → Untyped AST → OCaml type inference → Typed AST (`.cmt`)

Typed AST (`.cmt`) → Desugaring → Desugared AST → Pattern Disambiguation → Salto AST → Static Analysis → Abstract values

Salto AST:

- Disambiguate patterns: introduce *complements*
- ```
  match e with
  | Some _, _ -> ...
  | _, Some _ -> ...
  | _, _ -> ...
  match e with
  | Some _, _ -> ...
  | (_ \ Some _), Some _ -> ...
  | (_ \ Some _), (_ \ Some _) -> ...
  ```
  ```

High-Level Structure of the Analyser (Frontend)

Source code → OCaml parser → Untyped AST → OCaml type inference → Typed AST (`.cmt`) → Desugaring → Desugared AST → Pattern Disambiguation → Salto AST → Static Analysis → Abstract values

Salto AST:

- Disambiguate patterns: introduce *complements*
- ```
  match e with
  | Some _, _ -> ...
  | _, Some _ -> ...
  | _, _ -> ...
  ```
  ```
  match e with
  | Some _, _ -> ...
  | (_ \ Some _), Some _ -> ...
  | (_ \ Some _), (_ \ Some _) -> ...
  ```
- This is valuable information for *any* static analyser
- Allows to analyse the branches of a `match` independently
- Important for *extensible* data-types (*e.g.*, exceptions)
  ```

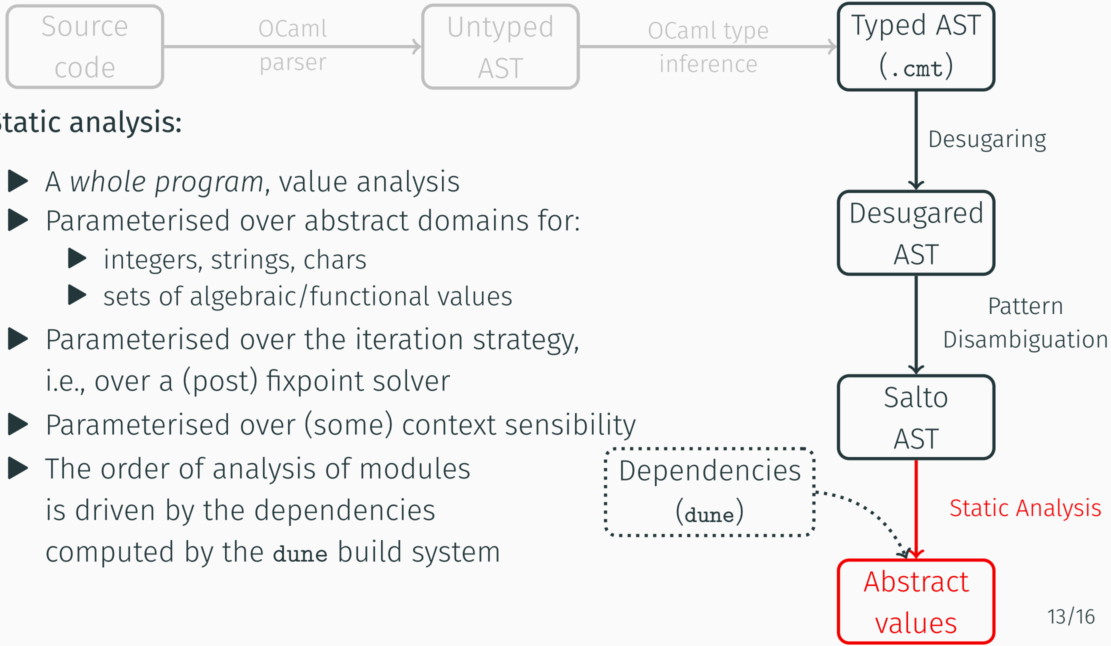
High-Level Structure of the Analyser (Frontend)



Static analysis:

- A *whole program*, value analysis
- Parameterised over abstract domains for:
  - integers, strings, chars
  - sets of algebraic/functional values
- Parameterised over the iteration strategy, i.e., over a (post) fixpoint solver
- Parameterised over (some) context sensibility

High-Level Structure of the Analyser (Frontend)



Static analysis:

- A *whole program*, value analysis
- Parameterised over abstract domains for:
  - integers, strings, chars
  - sets of algebraic/functional values
- Parameterised over the iteration strategy, i.e., over a (post) fixpoint solver
- Parameterised over (some) context sensibility
- The order of analysis of modules is driven by the dependencies computed by the `dune` build system

State of the Implementation

Code component	Code size
AST transformations	~ 3000 LoC
Abstract domain for values	~ 3500 LoC
Core of the abstract interpreter	~ 4300 LoC
Fixpoint engine	~ 500 LoC

State of the Implementation

Code component	Code size
AST transformations	~ 3000 LoC
Abstract domain for values	~ 3500 LoC
Core of the abstract interpreter	~ 4300 LoC
Fixpoint engine	~ 500 LoC

267 test programs (≤ 200 LoC), featuring:

- Higher-order, direct style programs
- Church encodings
- CPS programs
- Defunctionalised programs
- Monadic programs
- Non-regular types, GADTs

Analysis times range from 200 ms to 2 mn

Long Term Challenges

- Relational analysis (especially: input/output relations)

Benoît Montagu and Thomas P. Jensen. "Stable Relations and Abstract Interpretation of Higher-order Programs". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 119:1–119:30. DOI: `10.1145/3409001`

- Expressive and efficient relational domains for sets of trees are still an open problem
- Low-level representation of data (`Obj` module)
- Algebraic effects (one-shot continuations)
- Multicore
- Signals
- Scalability of the analysis

Conclusion

Salto: Static Analyses for Trustworthy OCaml

- A work in progress!
- An abstract interpreter for OCaml programs that detects uncaught exceptions
- Features an abstract domain for inductively defined sets of values
- Implemented using a dynamic fixpoint solver

`raise Questions`

https://salto.gitlabpages.inria.fr/

B. Montagu + P. Lermusiaux + T. Genet + T. Jensen

The Road Ahead (1)

Support more features of OCaml:

- Support mutable state
  - References and mutable data-types
  - Arrays
  - External state provided by the OS (e.g., file descriptors)
- Detect arithmetic overflows/underflows
- Detect problematic cases of pattern matching on mutable data
- Cyclic values, e.g.: `let rec l = 1 :: l`
- The `lazy` construct
- Objects, classes, recursive modules...

The Road Ahead (2)

Refine the analysis:

- Incorporate a narrowing phase to the fixpoint solver
- Exploit the types inferred by the OCaml compiler (reduced product)
- Specific abstract domains for strings, bytes, sets, maps, hash-tables...

Minimisation Examples

- Minimisation is important to reduce memory consumption
- And also helps avoid some unnecessary computations thanks to memoisation
- **Example:** Peano numbers admit several equivalent representations

$$\mu\alpha.\{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \alpha\}\}$$

Minimisation Examples

- Minimisation is important to reduce memory consumption
- And also helps avoid some unnecessary computations thanks to memoisation
- **Example:** Peano numbers admit several equivalent representations

$$
\begin{aligned}
& \mu\alpha.\{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \alpha\}\} \\
\equiv \quad & \{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \mu\alpha.\{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \alpha\}\}\}\} \quad \text{(external unfolding)}
\end{aligned}
$$

Minimisation Examples

- Minimisation is important to reduce memory consumption
- And also helps avoid some unnecessary computations thanks to memoisation
- **Example:** Peano numbers admit several equivalent representations

$$
\begin{aligned}
& \mu\alpha.\{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \alpha\}\} \\
\equiv\ & \{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \mu\alpha.\{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \alpha\}\}\}\} && \text{(external unfolding)} \\
\equiv\ & \mu\alpha.\{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \alpha\}\}\}\} && \text{(internal unfolding)}
\end{aligned}
$$

Minimisation Examples

- Minimisation is important to reduce memory consumption
- And also helps avoid some unnecessary computations thanks to memoisation
- **Example:** Peano numbers admit several equivalent representations

$$
\begin{aligned}
& \mu\alpha.\{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \alpha\}\} \\
\equiv \quad & \{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \mu\alpha.\{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \alpha\}\}\}\} \quad && \text{(external unfolding)} \\
\equiv \quad & \mu\alpha.\{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \{\mathsf{variants} = \{\mathsf{O} \mapsto \cdot; \mathsf{S} \mapsto \alpha\}\}\}\} \quad && \text{(internal unfolding)}
\end{aligned}
$$

- Our minimisation algorithm canonises these three abstract values into the first one

From a Research Prototype To an Actual Tool

- Improve error reporting and UI (LSP server?)
- Incremental changes of code
- “Explainable Abstract Interpretation”
- Produce examples of “bad” inputs
- Requires a lot of testing, engineering, time, and love!

Related Static Analyses: Related Work

- Type-based analysis of exceptions

Xavier Leroy and François Pessaux. "Type-Based Analysis of Uncaught Exceptions". In: *ACM Trans. Program. Lang. Syst.* 22.2 (2000), pp. 340–377. DOI: 10.1145/349214.349230

- Control-flow analysis

Olin Shivers. "The Semantics of Scheme Control-Flow Analysis". In: *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. PEPM '91. New York, NY, USA: Association for Computing Machinery, 1991, pp. 190–198. ISBN: 0897914333. DOI: 10.1145/115865.115884

- Control-flow analysis using widening

Benoît Montagu and Thomas P. Jensen. "Trace-Based Control-Flow Analysis". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 482–496. DOI: 10.1145/3453483.3454057

Related Static Analyses: Related Work

- Analysis of Prolog with type graphs

Pascal Van Hentenryck, Agostino Cortesi and Baudouin Le Charlier. "Type Analysis of Prolog Using Type Graphs". In: *The Journal of Logic Programming* 22.3 (Mar. 1995), pp. 179–209. DOI: 10.1016/0743-1066(94)00021-w

- Analysis of logic programs with tree grammars

Patrick Cousot and Radhia Cousot. "Formal Language, Grammar and Set-constraint-based Program Analysis by Abstract Interpretation". In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture - FPCA '95*. ACM Press, 1995. DOI: 10.1145/224164.224199

- Graph-based representations for sets of trees

Laurent Mauborgne. "Representation of Sets of Trees for Abstract Interpretation". PhD thesis. École Polytechnique, Nov. 1999. URL: https://www.di.ens.fr/~mauborgn/publi/t.pdf

Related Static Analyses: Related Work

- A relational abstract domain for trees with numeric data

Matthieu Journault, Antoine Miné and Abdelraouf Ouadjaout. "An Abstract Domain for Trees with Numeric Relations". In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Ed. by Luís Caires. Vol. 11423. Lecture Notes in Computer Science. Springer, 2019, pp. 724–751. DOI: `10.1007/978-3-030-17184-1_26`

- Equality constrainted tree automata (ECTAs)

James Koppel et al. "Searching entangled program spaces". In: *Proceedings of the ACM on Programming Languages* 6.ICFP (Aug. 2022), pp. 23–51. DOI: `10.1145/3547622`

Fixpoint Solvers: Related Work

Some uses of fixpoint solver for static analyses:

- Analysis of Prolog Programs

Baudouin L. Charlier and Pascal Van Hentenryck. *A Universal Top-Down Fixpoint Algorithm*. Tech. rep. USA, 1992. URL: `ftp://ftp.cs.brown.edu/pub/techreports/92/cs92-25.pdf`

- Approach followed by Interproc

Bertrand Jeannet. “Some Experience on the Software Engineering of Abstract Interpretation Tools”. In: *Electronic Notes in Theoretical Computer Science* 267.2 (Oct. 2010), pp. 29–42. DOI: `10.1016/j.entcs.2010.09.016`

- Approach followed by the Goblint static analyser

`https://goblint.in.tum.de/home`