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/](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
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
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:

- 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 Analyses for Uncaught Exceptions

Two families of static analyses:

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

Static Analyses for Uncaught Exceptions

Two families of static analyses:

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


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

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
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 $\lambda$-calculus developed in


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 \( \lambda \)-calculus developed in

  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 $\lambda$-calculus developed in


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:

\[ \nu^\# \in \nu^\# = \{ \text{ints} = d \in \mathbb{Z}^\#; \]
\[ \text{variants} = \{ c_1 \mapsto \nu^\#; \ldots; c_n \mapsto \nu^\# \}; \]
\[ \text{pairs} = (\nu^\#, \nu^\#); \]
\[ \text{funs} = \{(\lambda x.t) \mapsto [x_1 \mapsto \nu^\#; \ldots; x_n \mapsto \nu^\#]; \ldots \} \}

\mid \top

\]
An Abstract Domain For Sets Of Values (simplified)

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

\[\nu^\# \in \mathbb{V}^\# = \{ \text{ints} = d \in \mathbb{Z}^\#; \]
\[\text{variants} = \{ c_1 \mapsto \nu^\#; \ldots; c_n \mapsto \nu^\# \}; \]
\[\text{pairs} = (\nu^\#, \nu^\#); \]
\[\text{funs} = \{ (\lambda^l x. t) \mapsto [x_1 \mapsto \nu^\#; \ldots; x_n \mapsto \nu^\#]; \ldots \} \} \]

\[\mid \top \]
\[\mid \alpha \mid \mu\alpha. \nu^\# \]

**\[\mu\] has the semantics of a least fixed point**

**Information:** The widening operator detects some regularity and introduces the \[\mu\]s
An Abstract Domain For Sets Of Values (simplified)

A finite representation for \textit{recursively defined} sets of \textit{untyped} values:

\[ v^\# \in \mathbb{V}^\# = \{ \text{ints} = d \in \mathbb{Z}^\#; \]
\[ \text{variants} = \{ c_1 \mapsto v^\#; \ldots; c_n \mapsto v^\# \}; \]
\[ \text{pairs} = (v^\#, v^\#); \]
\[ \text{funs} = \{ (\lambda x . t) \mapsto [x_1 \mapsto v^\#; \ldots; x_n \mapsto v^\#]; \ldots \} \}

\[ | \top \]
\[ | \alpha \quad | \mu \alpha . v^\# \]

\(\mu\) has the semantics of a least fixed point

\(\triangleright\) The widening operator detects some regularity and introduces the \(\mu s\)

\(\triangleright\) Example: Peano numbers

\[ \mu \alpha . \{ \text{variants} = \{ 0 \mapsto \cdot; S \mapsto \alpha \} \} \]
An Abstract Domain For Sets Of Values (simplified)

A finite representation for recursively defined sets of untyped values:

\[ v^\# \in \forall^\# = \{ \text{ints} = d \in \mathbb{Z}^\#; \]
\[ \text{variants} = \{ c_1 \mapsto v^\#; \ldots; c_n \mapsto v^\# \}; \]
\[ \text{pairs} = (v^\#, v^\#); \]
\[ \text{funs} = \{(\lambda^l x . t) \mapsto [x_1 \mapsto v^\#; \ldots; x_n \mapsto v^\#]; \ldots \} \} \]

\[ \mid T \]
\[ \mid \alpha \mid \mu \alpha . v^\# \]

- \( \mu \) has the semantics of a least fixed point
- The widening operator detects some regularity and introduces the \( \mu \)s

Example: Peano numbers

\[ \mu \alpha . \{ \text{variants} = \{ O \mapsto \cdot; S \mapsto \alpha \} \} \]

Example: A set of continuations (for CPSed factorial)

\[ \mu \alpha . \left\{ \text{funs} = \left\{ \begin{array}{l}
(\lambda^l x . x) \mapsto [];
(\lambda^l x . k (x \ast n)) \mapsto [n \mapsto \{ \text{ints} = [1, +\infty]; k \mapsto \alpha \}]; \end{array} \right\} \right\} \]
The design of the abstract domain draws inspiration from:
- Equi-recursive types + union types
- Type Graphs (analysis of Prolog programs)


- Tree grammars / Tree automata
The design of the abstract domain draws inspiration from:

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


- 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 (∼10× improvement!)
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: $\text{Int}[1:\text{Pre}; 3:\text{Pre}] \quad \text{Int}[\top] \quad \text{Int}[\rho]$
  We support any abstract domain for integers (non-relational so far)
Control-Flow Analyses:

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

\[
\begin{align*}
\text{funs} &= \left\{ \begin{array}{l}
(\lambda^1 x. x) \mapsto []; \\
(\lambda^2 x. k (x \star n)) \mapsto [n \mapsto p_n; k \mapsto p_k];
\end{array} \right\}
\end{align*}
\]

where:
\[
\begin{align*}
\hat{h}(p_n) &= \{ \text{ints} = [1, +\infty] \} \\
\hat{h}(p_k) &= \left\{ \begin{array}{l}
(\lambda^1 x. x) \mapsto []; \\
(\lambda^2 x. k (x \star n)) \mapsto [n \mapsto p_n; k \mapsto p_k];
\end{array} \right\}
\end{align*}
\]

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

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

\[
\begin{align*}
\text{funs} &= \{ \\
&(\lambda \ell_1 x. x) \mapsto []; \\
&(\lambda \ell_2 x. k (x \ast n)) \mapsto [n \mapsto p_n; k \mapsto p_k]; \\
\} \\
\end{align*}
\]

where: \( \hat{h}(p_n) = \{\text{ints} = [1, +\infty]\} \)

\[\hat{h}(p_k) = \{ \\
&(\lambda \ell_1 x. x) \mapsto []; \\
&(\lambda \ell_2 x. k (x \ast n)) \mapsto [n \mapsto p_n; k \mapsto p_k]; \\
\} \]

- 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
Consider the following program: \[ \text{if } x < 42 \text{ then } e_1 \text{ else } e_2 \]

- To analyse \( e_1 \) with precision, we need to exploit the fact that \((x < 42)\) evaluated to \text{true}\n- This is done by running a \textbf{backward analysis} on the expression \((x < 42)\)
Consider the following program:

\[
\text{if } x < 42 \text{ then } e_1 \text{ else } e_2
\]

- To analyse \(e_1\) with precision, we need to exploit the fact that \((x < 42)\) evaluated to \text{true}.
- This is done by running a \textbf{backward analysis} on the expression \((x < 42)\).
- \textbf{Problem}: the condition is an arbitrary expression: it could be an application

\[
\text{if } f \ x \text{ then } e_1 \text{ else } e_2
\]

- To obtain precise results, we need to know which closures \(f\) might evaluate to.
  - For example, \(f\) could evaluate to \((\text{fun } x \rightarrow x < 42)\)
Consider the following program: \[ \text{if } x < 42 \text{ then } e1 \text{ else } e2 \]

- To analyse \( e1 \) with precision, we need to exploit the fact that \((x < 42)\) evaluated to \text{true}

- This is done by running a \textit{backward analysis} on the expression \((x < 42)\)

- \textbf{Problem}: the condition is an arbitrary expression: it could be an application \[ \text{if } f \; x \text{ then } e1 \text{ else } e2 \]

- To obtain precise results, we need to know which closures \( f \) might evaluate to

  For example, \( f \) could evaluate to \((\text{fun } x \rightarrow x < 42)\)

\textbf{Forward analysis et backward analyses depend on each other!}

- A problem in all interprocedural analyses

- Solution: use a \textit{dynamic fixpoint solver}
val fix: ((X.t -> Y.t) -> (X.t -> Y.t)) -> (X.t -> Y.t)
Computes a post-fixpoint of the functional passed as argument
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:

```ocaml
define 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

```olang
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

- Later re-emphasized (in a simpler setting)

- Actually used in a static analyser for C programs
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!

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
High-Level Structure of the Analyser (Frontend)

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

**Typed AST:**

- Names are resolved
- Type information can be retrieved for every node
- Some constructs are redundant:
  - Pattern matching is performed at several places
    ```ocaml
    match e with p1 -> ... | ... | pn -> ...
    let p = e in ...
    function p -> ...
    try e with p1 -> ... | ... | pn -> ...
    ```
  - Exception management is performed at several places
    ```ocaml
    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:
  ```ocaml
memo e with
  | p_1 -> ...
  | ...
  | p_n -> ...
  ```

- A **single** construct for exception handling:
  ```ocaml
dispatch e with
  | val x -> ...
  | exception exc -> ...
  ```

- Evaluation order made explicit using local **let**s, when possible (close to a “monadic normal form”)
High-Level Structure of the Analyser (Frontend)

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

Salto AST:

- Disambiguate patterns: introduce *complements*
  - match e with
    - | 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)

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

Salto AST:

- Disambiguate patterns: introduce *complements*

```ocaml
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)

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

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)

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

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

<table>
<thead>
<tr>
<th>Code component</th>
<th>Code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST transformations</td>
<td>~ 3000 LoC</td>
</tr>
<tr>
<td>Abstract domain for values</td>
<td>~ 3500 LoC</td>
</tr>
<tr>
<td>Core of the abstract interpreter</td>
<td>~ 4300 LoC</td>
</tr>
<tr>
<td>Fixpoint engine</td>
<td>~ 500 LoC</td>
</tr>
</tbody>
</table>

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
## State of the Implementation

<table>
<thead>
<tr>
<th>Code component</th>
<th>Code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST transformations</td>
<td>~ 3000 LoC</td>
</tr>
<tr>
<td>Abstract domain for values</td>
<td>~ 3500 LoC</td>
</tr>
<tr>
<td>Core of the abstract interpreter</td>
<td>~ 4300 LoC</td>
</tr>
<tr>
<td>Fixpoint engine</td>
<td>~ 500 LoC</td>
</tr>
</tbody>
</table>

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)

- 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
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
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 1 = 1 :: 1`
- The `lazy` construct
- Objects, classes, recursive modules...
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

\[ \mu \alpha. \{ \text{variants} = \{ O \mapsto \cdot; S \mapsto \alpha \} \}\]
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

\[ \mu \alpha.\{\text{variants} = \{O \mapsto \cdot; S \mapsto \alpha\}\} \]

\[ \equiv \{\text{variants} = \{O \mapsto \cdot; S \mapsto \mu \alpha.\{\text{variants} = \{O \mapsto \cdot; S \mapsto \alpha\}\}\}\} \]  (external unfolding)
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

\[\mu \alpha. \{\text{variants} = \{O \mapsto \cdot; S \mapsto \alpha\}\}\]

\[\equiv \{\text{variants} = \{O \mapsto \cdot; S \mapsto \mu \alpha. \{\text{variants} = \{O \mapsto \cdot; S \mapsto \alpha\}\}\}\}\] (external unfolding)

\[\equiv \mu \alpha. \{\text{variants} = \{O \mapsto \cdot; S \mapsto \{\text{variants} = \{O \mapsto \cdot; S \mapsto \alpha\}\}\}\}\] (internal unfolding)
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.

\[
\mu \alpha.\{\text{variants} = \{O \mapsto \cdot; S \mapsto \alpha\}\}
\]

\[
\equiv \{\text{variants} = \{O \mapsto \cdot; S \mapsto \mu \alpha.\{\text{variants} = \{O \mapsto \cdot; S \mapsto \alpha\}\}\}\}\]  
\text{(external unfolding)}
\]

\[
\equiv \mu \alpha.\{\text{variants} = \{O \mapsto \cdot; S \mapsto \{\text{variants} = \{O \mapsto \cdot; S \mapsto \alpha\}\}\}\}\]  
\text{(internal unfolding)}
\]

- 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**

- **Control-flow analysis**

- **Control-flow analysis using widening**
Related Static Analyses: Related Work

- **Analysis of Prolog with type graphs**
  

- **Analysis of logic programs with tree grammars**
  

- **Graph-based representations for sets of trees**
  
Related Static Analyses: Related Work

- A relational abstract domain for trees with numeric data

- Equality constrained tree automata (ECTAs)
Some uses of fixpoint solver for static analyses:

▶ **Analysis of Prolog Programs**


▶ **Approach followed by Interproc**


▶ **Approach followed by the Goblint static analyser**

[https://goblint.in.tum.de/home](https://goblint.in.tum.de/home)