Detection of Uncaught Exceptions in Functional Programs by Abstract Interpretation

Pierre Lermusiaux, Benoît Montagu

Detection of Uncaught Exceptions in Functional Programs by Abstract Interpretation*

Pierre Lermusiaux and Benoît Montagu

Inria, Campus universitaire de Beaulieu, Rennes, France
pierre.lermusiaux@inria.fr benoit.montagu@inria.fr

Abstract. Exception handling is a key feature in modern programming languages. Exceptions can be used to deal with errors, or as a means to control the flow of execution of a program. Since they might unexpectedly terminate a program, unhandled exceptions are a serious safety concern. We propose a static analysis to detect uncaught exceptions in functional programs, that is defined as an abstract interpreter. It computes a description of the values potentially returned by a program using a novel abstract domain, that can express inductively defined sets of values. Simultaneously, the analysis infers the possibly raised exceptions, by computing in the *abstract exception monad*. This abstract interpreter has been implemented as an effective static analyser for a large subset of OCaml programs, that supports mutable data types, the OCaml module system, and dynamically extensible data types such as the exception type. The analyser has been evaluated on several hundreds of OCaml programs.

Keywords: Static Analysis · Exceptions · Higher-Order Programs · Abstract Interpretation · Abstract Domain for Trees

1 Introduction

Programs that run in critical environments need to comply with strong safety guarantees. The minimal guarantee one expects for critical software is the *absence of runtime failures*. Sound static analyses can provide such guarantees statically, for every possible execution of a program, and in a fully automatic manner.

The static typing discipline found in the ML family of languages is such a static analysis technique, that brought strong safety guarantees to programs at a very low cost: *well-typed programs cannot "go wrong"* [48]. This soundness theorem for well-typed ML programs, however, does not preclude programs from abruptly ending with uncaught exceptions. Several analyses for ML-like languages have been developed to detect such undesirable behaviours, that were either leveraging type and effect systems [38,54], or that were based on variants of control-flow analyses or set constraints [68,67,14,15,66]. The recent success of *algebraic effects* and their introduction in popular languages such as OCaml [37] has renewed the interest in the static detection of uncaught exceptions and effects.

* This work was funded by the Salto grant, supported by Nomadic Labs and Inria.

Analysing uncaught exceptions in ML is a difficult problem, because data flow and control flow are interdependent. This is not only due to the first class nature of functions, but also due to the first class nature of exceptions themselves, *e.g.*, they can be taken as parameters, recorded in data structures or in mutable references. Furthermore, exceptions can carry any value as argument—including functions—and new exceptions can be dynamically generated at runtime.

In this paper, we propose a static analysis for a higher-order language, in which exceptions are first-class values. The analysis is based on the abstract interpretation framework [9]. It is a forward value analysis that infers which values any program point can compute, and which exceptions they might raise. For this purpose, we introduce a novel abstract domain that can represent recursively defined sets of values. We define a *widening* operator for this abstract domain, that is responsible for finding recursive generalisations of solutions.

Our analysis leverages this abstract domain to represent both possible values and exceptions, thanks to the *abstract exception monad*. This monad—that can also be used as an abstract domain—is an abstraction of the exception monad, that collects all values and exceptions.

We define our analysis as a big-step monadic interpreter, written in the *open recursive style*, that was emphasised in the "Abstracting Definitional Interpreter" approach [11]. Then, we obtain an effective analyser by applying a generic, dynamic fixpoint solver [6,63,59,24,12,30]. We prove that our analysis is sound, under the soundness assumption of the fixpoint solver.

We extend the analysis to handle a large subset of the OCaml language. In particular, it supports the dynamic creation of exceptions, mutable state, modules and functors. The analysis is so far limited to *sequential* programs that do not perform system calls, do not use the `Gc` or `Obj` modules, and do not employ recursive modules, general recursive definitions of values, objects, classes, arrays, or floats. We implemented an OCaml prototype for this analyser. It reports the possibly thrown exceptions and an over-approximation of the data they carry, along with an abstraction of the call trace that led to the program point where the exception was raised. We discuss some implementation choices, and evaluate the precision and performance of our analyser on 290 programs, that include examples from the literature and from the OCaml compiler's test suite.

2 Overview

Let us consider the classic example of the factorial function, as written below in a *continuation passing style*.

```
let rec fact_cont n i k =
  if i >= n then k i else
  fact_cont n (i + 1) (fun x -> k (x * i))
let fact n = fact_cont n 1 (fun x -> x)
let result = fact 5
```

The `fact_cont` function recursively calls itself with increasing values of its parameter `i`, until the value `n` is reached.

We are interested in finding which values (and exceptions) this program might return. To answer this question, we first need to find the possible continuations the function `fact_cont` can be called with, and, importantly, we need an abstract domain in which we can express this set, or an over-approximation thereof.

With the abstract domain that we introduce in §4, we can express such a set as the following abstract value:

$$\mu\alpha.\,\{\mathsf{funs} : \{(\lambda x.\,x) \mapsto \{\};\ (\lambda x.\,k\ (x * i)) \mapsto \{i \mapsto \{\mathsf{ints} : [1, +\infty]\}; k \mapsto \alpha\};\}\}$$

This abstract value represents a recursively-defined set—as indicated by the μ constructor—that is locally named α. This set is composed of function closures, that can be either the identity function, or the function $\lambda x.\,k\ (x * i)$, considered in an environment where the variable i is bound to an integer that is greater or equal to 1, and where the variable k is recursively bound to the local variable α, *i.e.*, to a value of the set we are defining.

Our abstract domain can also express structural invariants on data, such as the one for red-black trees [52], that forbids red nodes from having red children:

$$\mu\alpha.\left\{\mathsf{constructs} : \left\{\begin{array}{l} E : (); \\ R : \left(\begin{array}{l}\{\mathsf{constructs} : \{E : (), B : (\alpha, \{\mathsf{ints} : \top\}, \alpha)\}\}; \\ \{\mathsf{ints} : \top\}, \\ \{\mathsf{constructs} : \{E : (), B : (\alpha, \{\mathsf{ints} : \top\}, \alpha)\}\}\end{array}\right); \\ B : (\alpha, \{\mathsf{ints} : \top\}, \alpha)\end{array}\right\}\right\}$$

Our abstract domain bears a strong similarity with the theory of equi-recursive types [56], in the sense that *recursion* is a core aspect of our definition. However, it differs from recursive types, as function types are absent: sets of closures are used instead. Moreover, it is parameterised by a non-relational abstract domain used to represent integers values—which is not possible with *simple* type systems.

We leverage our abstract domain and define a static analysis for a call-by-value λ-calculus with pattern matching, exception handling, and first-class exceptions (§3). In this language, the order of evaluation is made explicit by let bindings, and pattern matching is *exhaustive* and *non-ambiguous* [8]. These requirements drastically simplify the semantics of programs and their analysis. The analysis is defined as an abstract interpreter that performs a forward value analysis (§5).

Based on this small abstract interpreter, we sketch (§6) several extensions that we implemented to obtain a static analyser for a subset of OCaml programs. The implementation uses an intermediate language that is close to the one of §3, into which we translated the OCaml typed abstract syntax tree. We evaluated the precision and performance of our analyser on 290 OCaml programs, written in a variety of styles (direct, CPS, monadic, *etc.*). We discuss these experimental results (§7), cover related work (§8), and finish with conclusive remarks (§9).

3 A λ-calculus With Exceptions

We introduce as an intermediate language a λ-calculus with pattern matching and exception handling. Its syntax resembles the monadic normal form, where the order of evaluation is made explicit with let bindings.

Definition 1. *Given $\mathcal{C}$ a set of constructor symbols, we give the following inductive definition of patterns p, q, and expressions t, u, r:*

$$\begin{aligned} p,\ q \in \mathbb{P} ::= {} & x \mid n \mid c(p_1, \ldots, p_k) \mid p_1 + p_2 \mid p \setminus q \\ t,\ u,\ r \in \mathbb{T} ::= {} & x \mid n \mid x_1\ op\ x_2 \mid c(x_1, \ldots, x_k) \\ & \mid \mu f.\, \lambda x.\, t \mid f\ y \mid \mathsf{let}\ x = t\ \mathsf{in}\ u \mid \mathsf{raise}\ x \\ & \mid \mathsf{match}\ t\ \mathsf{with}\ p_1 \Rightarrow u_1 \mid \cdots \mid p_n \Rightarrow u_n \\ & \mid \mathsf{dispatch}\ t\ \mathsf{with}\ \mathsf{val}\ x \Rightarrow u \mid \mathsf{exn}\ y \Rightarrow r \end{aligned}$$

where n is a constant integer, c is a constructor of $\mathcal{C}$, op is a binary operation on integers, and where the pattern q cannot contain any complement $p_1 \setminus p_2$.

We consider a pattern syntax and formalism inspired from [8]. The pattern disjunction $p+q$ matches any value matched by p or q, and the pattern complement $p \setminus q$ matches any value that is matched by p but not by q.

As in the OCaml typed AST, variables carry a type. We may write x_τ to denote that the variable x is of type τ. Patterns are linear, *i.e.*, sub-patterns of constructor patterns cannot share variables. All functions are recursive by default. If f does not occur in the expression t, then we write $\lambda x.\, t$ instead of $\mu f.\, \lambda x.\, t$.

The values of this language are integer constants, constructors applied to values, and function closures, that contain an environment of values:

$$\begin{aligned} v \in \mathbb{V} ::= {} & n \mid c(v_1, \ldots, v_k) \mid \langle E, \mu f.\, \lambda x.\, t \rangle \quad \text{where } \operatorname{dom} E = \operatorname{fv}(\mu f.\, \lambda x.\, t) \\ E \in \mathbb{E} ::= {} & [] \mid E, x \mapsto v \end{aligned}$$

Patterns induce a matching relation over values, that is described, with regard to a given environment E, by recursion on patterns:

$$\begin{array}{rcll} x & \prec\!\!\!\prec_E & v & \iff E(x) = v \\ c(p_1, \ldots, p_n) & \prec\!\!\!\prec_E & c(v_1, \ldots, v_n) & \iff \bigwedge_{i=1}^{n} p_i \prec\!\!\!\prec_E v_i \\ p + q & \prec\!\!\!\prec_E & v & \iff p \prec\!\!\!\prec_E v \ \vee\ q \prec\!\!\!\prec_E v \\ p \setminus q & \prec\!\!\!\prec_E & v & \iff p \prec\!\!\!\prec_E v \ \wedge\ q \not\prec\!\!\!\prec v \end{array}$$

We say that a pattern p matches a value v, denoted $p \prec\!\!\!\prec v$, iff there exists an environment E such that $p \prec\!\!\!\prec_E v$. In such case, we write $E\langle p \prec\!\!\!\prec v \rangle$ the smallest environment such that $p \prec\!\!\!\prec_{E\langle p \prec\!\!\!\prec v \rangle} v$.

Thanks to this pattern-matching formalism, we can focus on the class of programs where pattern matching is *exhaustive* and *non-ambiguous*, *i.e.*: In a term $\mathsf{match}\ t\ \mathsf{with}\ p_1 \Rightarrow u_1 \mid \cdots \mid p_n \Rightarrow u_n$ where $t : \tau$, we require that for any value $v : \tau$, there exists a unique $1 \leq i \leq n$ such that $p_i \prec\!\!\!\prec v$. The work presented in [8] shows how to *disambiguate* patterns, *i.e.*, how to make any pattern match non-ambiguous. We restrict ourselves to non-ambiguous patterns, because it simplifies both the dynamic semantics and the analysis of programs.

We present in Figure 1 a *call-by-value* big-step semantics for our language. We write $t \Downarrow_{\mathsf{val}} v$ to denote that the expression term t reduces to the value v, and we write $t \Downarrow_{\mathsf{exn}} v$ to denote that the reduction of t *raises* an exception evaluated as v. In this language, any value can be raised as an exception. The evaluation rules are mostly standard. We briefly explain the rules for match and dispatch.

$$\frac{}{E \vdash x \Downarrow_{\mathsf{val}} E(x)}\ \textsc{Var} \qquad \frac{}{E \vdash n \Downarrow_{\mathsf{val}} n}\ \textsc{Int} \qquad \frac{}{E \vdash x_1\ op\ x_2 \Downarrow_{\mathsf{val}} E(x_1)\ [\![op]\!]\ E(x_2)}\ \textsc{Op}$$

$$\frac{}{E \vdash \mathsf{raise}\ x \Downarrow_{\mathsf{exn}} E(x)}\ \textsc{Raise} \qquad \frac{}{E \vdash c(x_1, \ldots, x_k) \Downarrow_{\mathsf{val}} c(E(x_1), \ldots, E(x_k))}\ \textsc{Const}$$

$$\frac{E' = E|_{\mathrm{fv}(\mu f.\, \lambda x.\, t)}}{E \vdash \mu f.\, \lambda x.\, t \Downarrow_{\mathsf{val}} \langle E', \mu f.\, \lambda x.\, t \rangle}\ \textsc{Lam} \qquad \frac{E(y) = \langle E', \mu f.\, \lambda x.\, t \rangle \quad E', f \mapsto E(y), x \mapsto E(z) \vdash t \Downarrow_m v}{E \vdash y\ z \Downarrow_m v}\ \textsc{App}$$

$$\frac{E \vdash t_1 \Downarrow_{\mathsf{val}} v_1 \qquad E, x \mapsto v_1 \vdash t_2 \Downarrow_m v_2}{E \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \Downarrow_m v_2}\ \textsc{Let} \qquad \frac{E \vdash t_1 \Downarrow_{\mathsf{exn}} v}{E \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \Downarrow_{\mathsf{exn}} v}\ \textsc{LetRaise}$$

$$\frac{E \vdash t \Downarrow_{\mathsf{val}} v \qquad p_i \not\ll v \qquad E, E\langle p_i \ll v \rangle \vdash u_i \Downarrow_m v' \qquad 1 \le i \le n}{E \vdash \mathsf{match}\ t\ \mathsf{with}\ p_1 \Rightarrow u_1 \mid \cdots \mid p_n \Rightarrow u_n \Downarrow_m v'}\ \textsc{Match}$$

$$\frac{E \vdash t \Downarrow_{\mathsf{exn}} v}{E \vdash \mathsf{match}\ t\ \mathsf{with}\ p_1 \Rightarrow u_1 \mid \cdots \mid p_n \Rightarrow u_n \Downarrow_{\mathsf{exn}} v}\ \textsc{MatchRaise}$$

$$\frac{E \vdash t \Downarrow_m v \qquad E, x_m \mapsto v \vdash u_m \Downarrow_{m'} v'}{E \vdash \mathsf{dispatch}\ t\ \mathsf{with\ val}\ x_{\mathsf{val}} \Rightarrow u_{\mathsf{val}} \mid \mathsf{exn}\ x_{\mathsf{exn}} \Rightarrow u_{\mathsf{exn}} \Downarrow_{m'} v'}\ \textsc{Dispatch}$$

Fig. 1. Big-step semantics.

The non-ambiguous pattern-matching simplifies the semantics of the term $\mathsf{match}\ t\ \mathsf{with}\ p_1 \Rightarrow u_1 \mid \cdots \mid p_n \Rightarrow u_n$, as only one pattern can match the value of t, and thus only one branch is considered during the evaluation.

The rule Dispatch deals with exception handling: the evaluation of the term $\mathsf{dispatch}\ t\ \mathsf{with\ val}\ x_{\mathsf{val}} \Rightarrow u_{\mathsf{val}} \mid \mathsf{exn}\ x_{\mathsf{exn}} \Rightarrow u_{\mathsf{exn}}$ first evaluates t. If t reduces to a value, then the value branch u_{val} is evaluated. Otherwise, if t raises an exception, the exception branch u_{exn} is evaluated. In both cases, the value or the exception is added to the environment of the corresponding branch.

4 An Abstract Domain for Regular Sets of Values

In this section, we define an abstract domain that is able to represent inductively defined sets of values of our programming language. It is parameterised over a non-relational, numeric abstract domain $\mathbb{I}$, that provides a concretisation function $\gamma_{\mathbb{I}} : \mathbb{I} \to \wp(\mathbb{Z})$, a test for the abstract inclusion pre-order, and operations for union, intersection and widening, with the standard soundness conditions. For instance, the soundness of abstract union is stated: $\gamma_{\mathbb{I}}(\mathsf{I}_1) \cup \gamma_{\mathbb{I}}(\mathsf{I}_2) \subseteq \gamma_{\mathbb{I}}(\mathsf{I}_1 \sqcup_{\mathbb{I}} \mathsf{I}_2)$.

The definition of our abstract domain follows:

Definition 2 (Abstract values).

$$\begin{array}{rll}
\mathsf{A} \in \mathbb{A} ::= & \{\mathsf{ints} : \mathsf{I};\ \mathsf{constructs} : \mathsf{C};\ \mathsf{funs} : \mathsf{F}\} \mid \alpha \mid \mu\alpha.\mathsf{A} & \textit{(Abstract value)} \\
\mathsf{I} \in \mathbb{I} ::= & \textit{any numeric abstract domain} & \textit{(Abstract integers)} \\
\mathsf{C} ::= & \{\overline{c \mapsto (\mathsf{A}, \ldots, \mathsf{A})}\} \mid \top & \textit{(Abstract constructs)} \\
\mathsf{F} ::= & \{\overline{\mu f.\, \lambda x.\, t \mapsto \mathsf{E}}\} \mid \top & \textit{(Abstract closures)} \\
\mathsf{E} \in \mathbb{E} ::= & \{\overline{x \mapsto \mathsf{A}}\} & \textit{(Abstract environment)}
\end{array}$$

An abstract value, written A, describes which integers it denotes (in the field ints), *and* which values whose head is a constructor it denotes (in the field $\mathsf{constructs}$), *and* which function closures it denotes (in the field funs). The integer values are described by a numeric abstract domain that is taken as parameter.

The constructed values are described by a map whose keys are the possible head constructors of the values, and whose data are tuples of abstract values, that denote the possible values for all the arguments of that constructor. The constructed values might also be described by $\top$, which means that the head constructor could be any constructor, and the arguments may be any value.

Similarly, the possible function closures are described by a map that associates possible codes of the function to abstract environments. The environments map free variables of the corresponding function code to abstract values, denoting the possible concrete values of these variables. The closures might also be described by $\top$, to represent any closure made from any function code with any environment.

Finally, we can construct recursive sets of values through the use of variables α, that are introduced by the μ constructor of fixpoints.

The bottom value is $\{\mathsf{ints} : \bot;\ \mathsf{constructs} : \{\};\ \mathsf{funs} : \{\}\}$, and the top value is $\{\mathsf{ints} : \top;\ \mathsf{constructs} : \top;\ \mathsf{funs} : \top\}$. We may completely omit some of the fields (ints, $\mathsf{constructs}$ or funs) when they are associated with a bottom value.

This informal explanation is formalised in the concretisation function:

Definition 3 (Concretisation). *Assume Γ is a finite mapping from variables to abstract values. The concretisation $\gamma_\Gamma : \mathbb{A} \to \wp(\mathbb{V})$ is defined by $\gamma_\Gamma\{\mathsf{ints} : \mathsf{I};\ \mathsf{constructs} : \mathsf{C};\ \mathsf{funs} : \mathsf{F}\} = \gamma(\mathsf{I}) \cup \gamma_\Gamma(\mathsf{C}) \cup \gamma_\Gamma(\mathsf{F})$, where:*

$$
\begin{aligned}
\gamma_\Gamma(\alpha) &= \Gamma(\alpha) \\
\gamma_\Gamma(\mu\alpha.\mathsf{A}) &= \mathrm{lfp}_\subseteq(\lambda S.\gamma_{\Gamma,\alpha:S}(\mathsf{A})) \\
\gamma(\mathsf{I}) &= \gamma_{\mathbb{I}}(\mathsf{I}) \\
\gamma_\Gamma(\mathsf{C}) &= \begin{cases} \{c(v_1,\ldots,v_n) \mid c \in \mathcal{C} \wedge \forall 1 \le i \le n, v_i \in \mathbb{V}\} & \text{if } \mathsf{C} = \top \\ \left\{ c(v_1,\ldots,v_n) \,\middle|\, \begin{array}{l} (c \mapsto (\mathsf{A}_1,\ldots,\mathsf{A}_n)) \in \mathsf{C} \\ \wedge \forall 1 \le i \le n, v_i \in \gamma_\Gamma(\mathsf{A}_i) \end{array} \right\} & \text{otherwise} \end{cases} \\
\gamma_\Gamma(\mathsf{F}) &= \begin{cases} \{\langle E, \mu f.\, \lambda x.\, t\rangle \mid E \in \mathbb{E} \wedge t \in \mathbb{T}\} & \text{if } \mathsf{F} = \top \\ \{\langle E, \mu f.\, \lambda x.\, t\rangle \mid (\mu f.\, \lambda x.\, t \mapsto \mathsf{E}) \in \mathsf{F} \wedge E \in \gamma_\Gamma(\mathsf{E})\} & \text{otherwise} \end{cases} \\
\gamma_\Gamma(\mathsf{E}) &= \{E \mid \mathrm{dom}\, E = \mathrm{dom}\, \mathsf{E} \wedge \forall x \in \mathrm{dom}\, E, E(x) \in \gamma_\Gamma(\mathsf{E}(x))\}
\end{aligned}
$$

The definition is justified by the fact that the function $\lambda S.\gamma_{\Gamma,\alpha:S}(\mathsf{A})$ is monotonic, and thus has a least fixed point, thanks to the Knaster-Tarski theorem. This is formalised by the following lemma:

Lemma 1. *Consider the inclusion order $\subseteq$ on $\wp(S)$, and its pointwise extension on environments Γ. For any abstract value A, the function $\lambda\Gamma.\gamma_\Gamma(\mathsf{A})$ is monotonic.*

The fact that our abstract values may represent sets of values that might not all have the same types may seem surprising, since our goal is, ultimately, to analyse strongly typed programs. The crux of the explanation lies in the fact that our abstract domain can only represent *regular* sets of values. If we restricted our

abstract values so that they represent homogeneously typed values, it would be difficult to represent sets of values that are induced by a non-regular recursive type—like the type of finger trees [23]—or by generalised algebraic data types (GADTs). Indeed, one would need to find an over-approximation of such sets, and we would often approximate with the $\top$ abstract value. The ability to describe regular sets of values that may not have all the same type gives us more freedom, and allows to find more precise approximations. For instance, we can represent finger trees as a recursive set whose values are either trees or fingers, although trees and fingers have distinct types. In practice, the $\top$ value is never produced.

We write $\mathsf{A}_1[\alpha \leftarrow \mathsf{A}_2]$ to denote the capture avoiding substitution. We write $\gamma(\mathsf{A})$ for $\gamma_{[]}(\mathsf{A})$, *i.e.*, when the environment is empty.

The unwinding of fixpoints preserves the concretisation of abstract values.

Lemma 2 (Unwinding). $\gamma(\mu\alpha.\mathsf{A}) = \gamma(\mathsf{A}[\alpha \leftarrow \mu\alpha.\mathsf{A}])$

To define several operations on abstract values, we restrict them to *well-formed* values, using the standard contractiveness property for recursive types [16]:

Definition 4 (Contractiveness). *An abstract value* $\mathsf{A} = \mu\beta_1 \ldots \mu\beta_n.\mathsf{A}'$ *is* α*-contractive if* $n \geq 0$ *and* A' *does not start with* μ *and is not the variable* α*.*

Well-formedness requires that fixpoints must be contractive, that constructors are used with the correct arity, and that the environment in closures only define bindings for the free variables of the functions.

Definition 5 (Well-formedness). *An abstract value* A *is* well-formed *when the following conditions are satisfied:*

- *For any* $\mu\alpha.\mathsf{A}'$ *that occurs in* A*, the abstract value* A' *is* α*-contractive, and*
- *For any* $c \mapsto (\mathsf{A}_1, \ldots, \mathsf{A}_n)$ *that occurs in* A*, the arity of* c *is* n*, and*
- *For any* $\mu f.\, \lambda x.\, t \mapsto \mathsf{E}$ *that occurs in* A*,* $\operatorname{dom}\mathsf{E} = \operatorname{fv}(\mu f.\, \lambda x.\, t)$*.*

Well-formedness rules out the abstract value $\mu\alpha.\alpha$, whose concretisation is the empty set. Well-formedness is preserved by substitution, provided contractiveness for the substituted variable is satisfied. This ensures that unwinding fixpoints preserves well-formedness. In the rest of this article, we only consider closed, well-formed abstract values.

For any abstract value A, we can retrieve the subset of integer values (respectively, constructed values, or function closures) by unwinding the top-level μs if there are any, and eventually getting the ints field (respectively, $\mathsf{constructs}$, or funs). This is formalised in the following definition for projection on integers:

Definition 6 (Projection on integers). *The projection on integers of a well-formed abstract value* A*, written* $\mathsf{A}.\mathsf{ints}$*, is defined as follows:*

$$\{\mathsf{ints} : \mathsf{I};\ \mathsf{constructs} : \mathsf{C};\ \mathsf{funs} : \mathsf{F}\}.\mathsf{ints} = \mathsf{I}$$
$$(\mu\alpha.\mathsf{A}).\mathsf{ints} = (\mathsf{A}[\alpha \leftarrow \mu\alpha.\mathsf{A}]).\mathsf{ints}$$

The definition for projection is well founded, thanks to the contractiveness of μs: only a finite number of unwindings is necessary. The projections $\mathsf{A}.\mathsf{constructs}$ and $\mathsf{A}.\mathsf{funs}$ are defined in a similar way. Projection on integers is sound, as it over-approximates the set of integers an abstract value contains:

Lemma 3 (Soundness of projection on integers). $\gamma(\mathsf{A}) \cap \mathbb{Z} \subseteq \gamma(\mathsf{A.ints})$

Projections for constructors and closures enjoy similar soundness properties.

4.1 Inclusion, Union and Intersection

Following the methodology employed in the context of recursive subtyping, we define the inclusion relation between abstract values as a *co-inductive* relation.

Definition 7 (Abstract inclusion). *The inclusion between abstract values, written* $\mathsf{A}_1 \sqsubseteq \mathsf{A}_2$ *is defined as a co-inductive relation by the following rules:*

$$\frac{\mathsf{A}_1[\alpha \leftarrow \mu\alpha.\mathsf{A}_1] \sqsubseteq \mathsf{A}_2}{\mu\alpha.\mathsf{A}_1 \sqsubseteq \mathsf{A}_2} \qquad \frac{\mathsf{A}_1 \neq \mu\beta.\mathsf{A}'_1 \qquad \mathsf{A}_1 \sqsubseteq \mathsf{A}_2[\alpha \leftarrow \mu\alpha.\mathsf{A}_2]}{\mathsf{A}_1 \sqsubseteq \mu\alpha.\mathsf{A}_2}$$

$$\frac{\mathsf{I}_1 \sqsubseteq_{\mathsf{I}} \mathsf{I}_2 \qquad \mathsf{C}_1 \sqsubseteq_{\mathsf{C}} \mathsf{C}_2 \qquad \mathsf{F}_1 \sqsubseteq_{\mathsf{F}} \mathsf{F}_2}{\{\mathsf{ints} : \mathsf{I}_1;\ \mathsf{constructs} : \mathsf{C}_1;\ \mathsf{funs} : \mathsf{F}_1\} \sqsubseteq \{\mathsf{ints} : \mathsf{I}_2;\ \mathsf{constructs} : \mathsf{C}_2;\ \mathsf{funs} : \mathsf{F}_2\}}$$

$$\frac{}{\mathsf{C}_1 \sqsubseteq_{\mathsf{C}} \top} \qquad \frac{}{\mathsf{F}_1 \sqsubseteq_{\mathsf{F}} \top}$$

$$\frac{\begin{array}{c}\forall (c \mapsto (\mathsf{A}_{1,1}, \ldots, \mathsf{A}_{1,n})) \in \mathsf{C}_1, \\ \exists (c \mapsto (\mathsf{A}_{2,1}, \ldots, \mathsf{A}_{2,n})) \in \mathsf{C}_2, \\ \forall 1 \leq i \leq n,\ \mathsf{A}_{1,i} \sqsubseteq \mathsf{A}_{2,i}\end{array}}{\mathsf{C}_1 \sqsubseteq_{\mathsf{C}} \mathsf{C}_2} \qquad \frac{\begin{array}{c}\forall (\mu f.\, \lambda x.\, t \mapsto \mathsf{E}_1) \in \mathsf{F}_1, \\ \exists (\mu f.\, \lambda x.\, t \mapsto \mathsf{E}_2) \in \mathsf{F}_2, \\ \forall x \in \operatorname{dom} \mathsf{E}_1,\ \mathsf{E}_1(x) \sqsubseteq \mathsf{E}_2(x)\end{array}}{\mathsf{F}_1 \sqsubseteq_{\mathsf{F}} \mathsf{F}_2}$$

In this definition, the relation $\sqsubseteq_{\mathsf{I}}$ is provided by the abstract domain on integers. The inclusion relation unfolds fixpoints when necessary, and otherwise compares each field (integers, constructed values, closures) separately, by treating the finite maps for constructed values and closures as *disjunctions*, *i.e.*, by using the standard Hoare ordering. In practice, the inclusion test is implemented by transforming abstract values into graphs that resemble tree automata: each graph node corresponds to a sub-term of an abstract value, and μ-nodes create cycles. Then, it suffices to check whether one automaton simulates the other [1,31,16].

Lemma 4 (Inclusion is a pre-order). *The inclusion between closed, well-formed abstract values is a pre-order, i.e., a reflexive and transitive relation.*

The definitions for abstract union and intersection are defined in the companion research report [34] in a similar way, as co-inductive relations that unwind fixpoints when needed.

The abstract operations enjoy the expected soundness properties:

Lemma 5 (Soundness of abstract operations). *For any closed, well-formed abstract values* A_1 *and* A_2*:*

- $\mathsf{A}_1 \sqsubseteq \mathsf{A}_2$ *implies* $\gamma(\mathsf{A}_1) \subseteq \gamma(\mathsf{A}_2)$*, and*
- $\gamma(\mathsf{A}_1) \cup \gamma(\mathsf{A}_2) \subseteq \gamma(\mathsf{A}_1 \sqcup \mathsf{A}_2)$*, and*
- $\gamma(\mathsf{A}_1) \cap \gamma(\mathsf{A}_2) \subseteq \gamma(\mathsf{A}_1 \sqcap \mathsf{A}_2)$*.*

The proof of Lemma 5 crucially relies on Lemma 2, that proves that unwinding a recursive value preserves its concretisation.

Union and intersection are implemented by translating the values into graphs, on which union and intersection are easily computed. Then, we transform them back into trees with μ nodes. Our implementation exploits the *locally nameless* representation [5], where bound variables are encoded as de Bruijn indices. We leverage this canonical representation by *hash-consing* values and *memoising* the operations [13]. This has proved essential to obtain acceptable performance.

4.2 Widening

The widening, written $\mathsf{A}_1 \nabla \mathsf{A}_2$, is a binary operator on abstract values that over-approximates the union of abstract values, and is used to approximate the Kleene fixpoint iterations. The role of the widening is central in abstract interpretation, as it serves two purposes. Firstly, the widening must find generalisations of abstract values, in order to find invariants. This part impacts the *precision* of the analysis, and relies on heuristics. Secondly, it must ensure the *termination* of the analysis, by enforcing a stability property: every widening chain must reach a limit in finite time. This part impacts the *performance* of the analyser.

In our abstract domain, the widening operator is responsible for finding *regularities* in abstract values and for creating μ nodes. A similar idea was used in the analysis of Prolog programs using *type graphs* [22], that are trees that contain cycles. Our widening draws inspiration from type graphs.

We now give the informal procedure to compute the widening of two abstract values A_1 and A_2. It operates in two phases. The first phase proceeds as follows:

1. Compute the union A_{12} of A_1 and A_2 where the widening of the numeric abstract domain is used, instead of the standard union. This ensures that the numeric parts of abstract values won't grow indefinitely.
2. Compute $\mathsf{A}_{\mathsf{new}}$, which is a *minimised* version of A_{12}. Minimisation is performed by an algorithm on tree automata, that produces a semantically equivalent abstract value, and whose size is smaller.
3. Compare the $\mathsf{A}_{\mathsf{new}}$ and A_1 (viewed as trees):
   - If the height of $\mathsf{A}_{\mathsf{new}}$ is not greater than the height of A_1, return $\mathsf{A}_{\mathsf{new}}$;
   - If, for each construct and each code of closures, the maximal number of occurrences in each tree path of $\mathsf{A}_{\mathsf{new}}$ is less than those occurrences in A_1, or a user-provided threshold, return $\mathsf{A}_{\mathsf{new}}$;
   - Otherwise, go to the *shrinking phase*.

Steps 2 and 3 allow the size of abstract values to grow enough, before a shrinking phase starts. In practice, this is important to find precise invariants.

The *shrinking phase*, which takes inspiration from the widening operation of *type graphs*, tries to *shrink* $\mathsf{A}_{\mathsf{new}}$, by introducing μ nodes at appropriate positions to "fold the abstract value on itself". It proceeds as follows:

1. Find *clashes* between A_1 and $\mathsf{A}_{\mathsf{new}}$, *i.e.*, nodes that are reachable through the same path (possibly unwinding μ nodes) in the two trees, and such that:
   - Either, the two nodes have different sets of head constructors or codes of functions: this means that the two nodes might differ *semantically*.

   - Or, the two nodes have different *depths* in the two trees: this means that some path was followed through a μ-unwinding.
2. If no clash is found, then return $\mathsf{A}_{\mathsf{new}}$.
3. If a clash is found, then we try to create a cycle in $\mathsf{A}_{\mathsf{new}}$ by merging the clashing node with one of its ancestors:
   - We search for the closest ancestor of the clashing node that is *semantically larger* in the sense of the pre-order. If there is such an ancestor, then we merge it with the clashing node, thus creating a cycle.
   - If no such ancestor exists, we search for the closest ancestor that has *at least the same head constructors and function codes* as the clashing node, then we merge it with the clashing node too.
   - If no such ancestor exists, then we return $\mathsf{A}_{\mathsf{new}}$ unchanged, which allows the abstract values to grow.

   We repeat this operation until no clashing node remains, or until a maximal number of iterations is reached. In the latter case, we *truncate* $\mathsf{A}_{\mathsf{new}}$, *i.e.*, we replace some nodes with $\top$, so that it has the same height as A_1.

In practice, we could not find any case where the final truncation is needed. We have observed that our widening operator finds precise generalisations in practice.

5 An Abstract Interpreter to Detect Uncaught Exceptions

To design our abstract interpreter, we took inspiration from the "Abstracting Definitional Interpreter" approach [11]. This methodology prescribes to derive an abstract interpreter from a concrete big-step interpreter that computes in a monad, that is a parameter of the interpreter. Furthermore, the methodology fosters the use of the *open recursive style*: the interpreter should be a function that takes as extra parameter the function that was intended to be called recursively.

The first aspect—being parameterised by a monad—is motivated by the fact that one could use a monad that computes over *abstractions* of values. In §5.1, we present a monad that is an abstraction of the exception monad. It is also an abstract domain, and is therefore well suited to define an abstract interpreter.

The second aspect—using open recursive style—permits the use of dynamic fixpoint solvers [59,63,12,24,6,30]. Such solvers compute post-fixpoints, *i.e.*, over-approximations of solutions of systems of equations over abstract values, for which the set of equations might be discovered dynamically, while solving the equations. New equations can be discovered, for instance, when the control flow of a program depends on its data flow. This is the case of higher-order programs, as the function that can be called at a given call site can possibly result from a computation. We present in §5.2 our abstract interpreter as a function that computes in the abstract exception monad, and is defined in open recursive style.

5.1 The Abstract Exception Monad

A big-step interpreter for a programming language with exceptions can be defined in an elegant manner using the *exception monad*, which we briefly recall. In the

exception monad, a computation is either a success value, or an exception that carries some value—typically of type exception—from the object language.

$$\mathsf{type}\ \mathsf{m}\,\beta = \mathsf{Success}\,\beta \mid \mathsf{Exception}\,\mathbb{V}$$

$$\begin{array}{ll} \mathbf{return} :: \beta \to \mathsf{m}\,\beta & \gg\!= :: \mathsf{m}\,\beta_1 \to (\beta_1 \to \mathsf{m}\,\beta_2) \to \mathsf{m}\,\beta_2 \\ \mathbf{return}\,x = \mathsf{Success}\,x & (\mathsf{Success}\,x) \gg\!= f = f\,x \\ & (\mathsf{Exception}\,e) \gg\!= f = \mathsf{Exception}\,e \end{array}$$

In this monad, the **raise** function expresses the action of throwing an exception, while the **dispatch** function, corresponds to the dispatch construct of our prototype language (§3), and expresses the action of catching an exception.

$$\begin{array}{ll} \mathbf{raise} :: \mathbb{V} \to \mathsf{m}\,\mathbb{A} & \mathbf{dispatch} :: \mathsf{m}\,\beta_1 \to (\beta_1 \to \mathsf{m}\,\beta_2) \to (\mathbb{V} \to \mathsf{m}\,\beta_2) \to \mathsf{m}\,\beta_2 \\ \mathbf{raise}\,e = \mathsf{Exception}\,e & \mathbf{dispatch}\,(\mathsf{Success}\,x)\,f\,g = f\,x \\ & \mathbf{dispatch}\,(\mathsf{Exception}\,e)\,f\,g = g\,e \end{array}$$

The **raise** function simply injects its argument into the exception case, whereas the **dispatch** function takes two continuations, to handle, respectively, the success case, and the exception case, by performing a case analysis on the monadic value.

We can easily define a monad that mimics the behaviour of the exception monad, with the difference that it deals with abstractions of sets of (possibly exceptional) values, instead of mere exceptional values. The construction is based on the observation that $\wp(\mathsf{m}\,\beta)$ is isomorphic to $\wp(\beta) \times \wp(\mathbb{V})$, that can itself be abstracted into $\wp(\beta) \times \wp(\mathbb{A})$ by using our abstract domain for sets of values. Thus, we define the abstract exception monad, written $\mathsf{m}^\sharp\,\beta$, as follows:

$$\mathsf{type}\ \mathsf{m}^\sharp\,\beta = \beta \times \mathbb{A}$$

$$\begin{array}{ll} \mathbf{return}^\sharp :: \beta \to \mathsf{m}^\sharp\,\beta & \gg\!=^\sharp :: \mathsf{m}^\sharp\,\beta_1 \to (\beta_1 \to \mathsf{m}^\sharp\,\beta_2) \to \mathsf{m}^\sharp\,\beta_2 \\ \mathbf{return}^\sharp B = (B, \bot) & (B, \mathsf{A}) \gg\!=^\sharp f = \text{let } (B', \mathsf{A}') = f\,B \text{ in } (B', \mathsf{A} \sqcup \mathsf{A}') \end{array}$$

The $\mathbf{return}^\sharp$ operation records its argument as the set of possible values, and asserts that no exception is returned: the set of possible exceptions is $\bot$. The $\gg\!=^\sharp$ operation retrieves the value part of its monadic argument and passes it to the continuation. The final value is composed of the value part that was produced by the continuation, and of the union of the exceptions that might have been raised by the monadic value and by the evaluation of the continuation. The functions $\mathbf{return}^\sharp$ and $\gg\!=^\sharp$ satisfy the *monad laws* if $(\bot, \sqcup)$ is a monoid.

The fact that $\mathsf{m}^\sharp\,\beta$ is a monad does not suffice to use it in an abstract interpreter, though. We also need $\mathsf{m}^\sharp\,\beta$ to be an abstract domain, *i.e.*, one must decide when two monadic values are included in each other, and how to compute abstract unions, intersections, and widening.

Interestingly, the monad $\mathsf{m}^\sharp\,\beta$ acts as an abstract domain as soon as β is an abstract domain: this is the standard cartesian product of abstract domains, where operations are defined pointwise. In practice, we only need to consider the instance $\mathsf{m}^\sharp\,\mathbb{A}$, *i.e.*, the domain of exceptional abstract values.

The remaining pieces that are needed to use $\mathsf{m}^\sharp\,\beta$ in an abstract interpreter are the abstract versions of **raise** and **dispatch**. They are defined as follows:

$$\begin{array}{ll}\mathbf{raise}^\sharp :: \mathbb{A} \to \mathsf{m}^\sharp\,\mathbb{A} & \mathbf{dispatch}^\sharp\ : \mathsf{m}^\sharp\,\beta \to (\beta \to \mathsf{m}^\sharp\,\mathbb{A}) \to (\mathbb{A} \to \mathsf{m}^\sharp\,\mathbb{A}) \to \mathsf{m}^\sharp\,\mathbb{A} \\ \mathbf{raise}^\sharp \mathsf{A} = (\bot, \mathsf{A}) & \mathbf{dispatch}^\sharp\ (B, \mathsf{A})\ F\ G = F\,B \sqcup G\,\mathsf{A}\end{array}$$

The $\mathbf{raise}^\sharp$ operation raises a set of possible exceptions, by recording the abstract value for exceptions in the set of possibly returned exceptions, and by returning the bottom value, since it can never return any value. It is the dual of $\mathbf{return}^\sharp$.

The $\mathbf{dispatch}^\sharp$ function executes the value continuation on the set of possible values, and executes the exception continuation on the set of possible exceptions, and then returns their abstract union in the domain of exceptional values.

We can easily show that the abstract operations compute over-approximations of their counterpart in the exception monad. Assume the type β is equipped with a concretisation function $\gamma_\beta : \beta \to \wp(\mathbb{B})$ for some set $\mathbb{B}$. Then, we define the concretisation for the abstract monad:

$$\begin{array}{l}\gamma_{\mathsf{m}^\sharp\,\beta} : \mathsf{m}^\sharp\,\beta \to \wp(\mathsf{m}\,\mathbb{B}) \\ \gamma_{\mathsf{m}^\sharp\,\beta}(B, \mathsf{A}) = \{\mathsf{Success}\,b \mid b \in \gamma_\beta(B)\} \cup \{\mathsf{Exception}\,v \mid v \in \gamma(\mathsf{A})\}\end{array}$$

The concretisation specifies that the first component of monadic values form the success values, and that the second component describe possible exceptions.

The soundness results for the abstract operations show that they compute over-approximations of their concrete counterparts:

Lemma 6. *The following inclusions are satisfied:*

- $\{\mathbf{return} b \mid b \in \gamma_\beta(B)\} \subseteq \gamma_{\mathsf{m}^\sharp\,\beta}(\mathbf{return}^\sharp B)$
- $\{m \gg\!= f \mid m \in \gamma_{\mathsf{m}^\sharp\,\beta_1}(M), f \in \gamma_{\beta_1 \to \mathsf{m}^\sharp\,\beta_2}(F)\} \subseteq \gamma_{\mathsf{m}^\sharp\,\beta_2}(M \gg\!=^\sharp F)$
- $\{\mathbf{raise} v \mid v \in \gamma(\mathsf{A})\} \subseteq \gamma_{\mathsf{m}^\sharp\,\mathbb{A}}(\mathbf{raise}^\sharp \mathsf{A})$
- $\left\{ \mathbf{dispatch}\,m\,f\,g \;\middle|\; \begin{array}{l} m \in \gamma_{\mathsf{m}^\sharp\,\beta_1}(M), \\ f \in \gamma_{\beta_1 \to \mathsf{m}^\sharp\,\beta_2}(F), \\ g \in \gamma_{\mathbb{V} \to \mathsf{m}^\sharp\,\beta_2}(G) \end{array} \right\} \subseteq \gamma_{\mathsf{m}^\sharp\,\beta_2}(\mathbf{dispatch}^\sharp M\,F\,G)$

where $\gamma_{\beta_1 \to \beta_2}(F) = \{f \mid \forall X, \forall x \in \gamma_{\beta_1}(X), f\,x \in \gamma_{\beta_2}(F\,X)\}$.

5.2 A Monadic Abstract Interpreter in Open Recursive Style

In this section, we describe our whole-program static analyser. It infers an over-approximation of the values that a program might compute, and the exceptions that it might raise, with the possible values they carry. Although it analyses programs that can deal with first-class functions, it is *not* defined as a control-flow analyser [60], but rather as an abstract interpreter that performs a value analysis. The insight is the following: since functions are first-class citizens in the language, a value analysis also infers an approximation of the control flow. A value analysis will indeed compute which functions may be called at every call site.

Our analyser follows the *open recursive style*, and has the following type:

$$(\mathbb{T} \to \mathbb{E} \to \mathsf{m}^\sharp\,\mathbb{A}) \to (\mathbb{T} \to \mathbb{E} \to \mathsf{m}^\sharp\,\mathbb{A})$$

$$
\begin{aligned}
&\text{Assuming } \mathsf{eval} :: \mathbb{T} \to \mathbb{E} \to \mathsf{m}^\sharp\,\mathbb{A}, \quad \text{we define } [\![\cdot]\!]^{\mathsf{eval}}_{\cdot} :: \mathbb{T} \to \mathbb{E} \to \mathsf{m}^\sharp\,\mathbb{A}\\
[\![x]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \mathbf{return}^\sharp\,\mathsf{E}(x)\\
[\![c(x_1,\ldots,x_n)]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \begin{cases} \mathbf{return}^\sharp \bot \text{ if } \mathsf{E}(x_i) = \bot \text{ for some } 1 \le i \le n \\ \mathbf{return}^\sharp \{\mathsf{constructs} : \{c \mapsto (\mathsf{E}(x_1),\ldots,\mathsf{E}(x_n))\}\} \\ \quad \text{otherwise} \end{cases}\\
[\![n]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \mathbf{return}^\sharp \{\mathsf{ints} : \{n\}\}\\
[\![x_1\ op\ x_2]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \mathbf{return}^\sharp \{\mathsf{ints} : \mathsf{E}(x_1).\mathsf{ints}\ [\![op]\!]\ \mathsf{E}(x_2).\mathsf{ints}\}\\
[\![\mu f.\,\lambda x.\,t]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \mathbf{return}^\sharp \{\mathsf{funs} : \{\mu f.\,\lambda x.\,t \mapsto \mathsf{E}|_{\mathrm{fv}(\mu f.\,\lambda x.\,t)}\}\}\\
[\![x\ y]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \text{if } \mathsf{E}(y) = \bot \text{ then } \mathbf{return}^\sharp \bot \text{ else}\\
&\quad \bigsqcup_{(\mu f.\,\lambda x.\,t \mapsto \mathsf{E}') \in \mathsf{E}(x).\mathsf{funs}} \mathsf{eval}\ t\ \mathsf{E}''\\
&\quad \text{where } \mathsf{E}'' = \mathsf{E}', f \mapsto F, x \mapsto \mathsf{E}(y)\\
&\quad \text{and } F = \{\mathsf{funs} : \{\mu f.\,\lambda x.t \mapsto \mathsf{E}'\}\}\\
[\![\mathsf{let}\ x = t\ \mathsf{in}\ u]\!]^{\mathsf{eval}}_{\mathsf{E}} &= [\![t]\!]^{\mathsf{eval}}_{\mathsf{E}} \gg\!=^\sharp \lambda v.\, \text{if } v = \bot \text{ then } \mathbf{return}^\sharp \bot \text{ else}\\
&\quad [\![u]\!]^{\mathsf{eval}}_{\mathsf{E},x:v}\\
[\![\mathsf{match}\ t\ \mathsf{with}\ p_1 \Rightarrow t_1 \mid \cdots \mid p_n \Rightarrow t_n]\!]^{\mathsf{eval}}_{\mathsf{E}} &= [\![t]\!]^{\mathsf{eval}}_{\mathsf{E}} \gg\!=^\sharp \lambda v.\text{if } v = \bot \text{ then } \mathbf{return}^\sharp \bot \text{ else}\\
&\quad \bigsqcup_{1 \le i \le n} (p_i \prec\!\!<^\sharp v) \gg\!=^\sharp \lambda \mathsf{E}'.\\
&\qquad \text{if } \mathsf{E}' = \bot \text{ then } \mathbf{return}^\sharp \bot \text{ else } [\![t_i]\!]^{\mathsf{eval}}_{\mathsf{E},\mathsf{E}'}\\
[\![\mathsf{raise}\ x]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \mathbf{raise}^\sharp \mathsf{E}(x)\\
[\![\mathsf{dispatch}\ u\ \mathsf{with}\ \mathsf{val}\ x \Rightarrow t \mid \mathsf{exn}\ y \Rightarrow r]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \mathbf{dispatch}^\sharp\ [\![t]\!]^{\mathsf{eval}}_{\mathsf{E}}\\
&\qquad (\lambda v.\, \text{if } v = \bot \text{ then } \mathbf{return}^\sharp \bot \text{ else } [\![u]\!]^{\mathsf{eval}}_{\mathsf{E},x:v})\\
&\qquad (\lambda e.\, \text{if } e = \bot \text{ then } \mathbf{return}^\sharp \bot \text{ else } [\![r]\!]^{\mathsf{eval}}_{\mathsf{E},y:e})
\end{aligned}
$$

Fig. 2. Definition of the abstract interpreter.

It takes as a parameter an analyser, that represents the information that has been discovered so far on the program, and produces an analyser as output, that exploits the input analyser to produce more analysis results, that are possibly less precise. The role of the fixpoint solver is to find a post-fixpoint of this functional. Similar approaches—leveraging fixpoint solvers to define static analysers—have been successfully used in other work on static analysis [22,64,50,4].

Our abstract interpreter is defined in Figure 2, where $[\![t]\!]^{\mathsf{eval}}_{\mathsf{E}}$ denotes the abstract value of type $\mathsf{m}^\sharp\,\mathbb{A}$ obtained by analysing the program t under the abstract environment E, and using the analysis function eval for recursive calls. Importantly, the analyser does not call eval for *every* recursive call. Instead, eval is only used when the analyser cannot be called on a strict sub-term. In practice, this means that eval is only used to analyse function calls. In every other place, we have the guarantee that the analysis is demanded on a strict sub-term, and a standard recursive call is performed. This strategy saves time in practice, as it lightens the burden of the fixpoint solver, that only needs to find post-fixpoints for function calls rather than for every program point.

To analyse a variable, we return the abstract value found in the environment.

To analyse a construct, we retrieve the abstract values for every argument, and return the corresponding abstract value for that constructor, or $\bot$ if some of the argument was $\bot$, because of the *eager* semantics.

The analysis of an integer returns this integer injected in the integer domain. The analysis of binary operations on integers retrieves the integer parts of the abstract values for the two arguments, and returns the result of the transfer function from the integer domain for that binary operation.

The analysis of a function mimics the concrete semantics: it returns an abstract closure composed of the code of the function and its abstract environment.

The analysis of function calls is more interesting. If the abstract value for the argument is $\bot$, then we return $\bot$, because evaluation is eager. Otherwise, we retrieve all the possible closures for the value at the call position, and analyse their bodies by extending their environments with the abstract value for the argument, and with the abstract closure itself (we are dealing with recursive functions). The final result is the union—at the level of the abstract monad—of the analyses of all the possible function bodies. Because the bodies of the functions that are analysed are not strict sub-terms of the original term $x\,y$, we perform an external recursive call to the analyser, by using the eval parameter.

The analysis of let bindings chains the analyses of its two parts, and, because evaluation is eager, checks for emptiness before analysing the second sub-term.

The pattern matching construct is analysed by first analysing the scrutinee, and then analysing each branch of the match *independently*. For each branch, we retrieve the environment produced by matching the abstract value with the pattern (written $p \prec^\sharp v$), and then we analyse the code of that branch if the matching was possible. Then, we take the union—at the level of the abstract monad—of the analysis results from each branch. Notably, the exceptions that any branch might raise are reported in the final result. The definition for matching abstract values against patterns is available in the companion research report [34].

Analysing the raise construct is easy: a call to the $\mathbf{raise}^\sharp$ function suffices. Finally, the analysis of $\mathsf{dispatch}$ amounts to calling the $\mathbf{dispatch}^\sharp$ function from the abstract monad, on the analysis of the scrutinee, and on two continuations, that will analyse the codes of the two branches, if they are given non-$\bot$ arguments.

5.3 Soundness of the Abstract Interpreter

We show that the abstract interpreter of Figure 2 is *sound*, in the sense that it computes an over-approximation of the behaviour of programs.

Definition 8 (Behaviour of programs). *Let S be a set of evaluation environments:* $\mathrm{EVAL}_S\, t = \bigcup_{E \in S} \{\mathsf{Success}\, v \mid E \vdash t \Downarrow_{\mathsf{val}} v\} \cup \{\mathsf{Exception}\, e \mid E \vdash t \Downarrow_{\mathsf{exn}} e\}$

The behaviour of a program t as a function EVAL that takes a set of evaluation environments as input, and produces a set of values with a tag that indicates whether it results from normal or from exceptional evaluation.

Then, the soundness of the abstract interpreter follows:

Theorem 1 (Soundness). *Assume* eval *is a post-fixpoint, i.e.,* $[\![t]\!]^{\mathsf{eval}}_{\mathsf{E}} \sqsubseteq \mathsf{eval}\, t\, \mathsf{E}$ *for every* t *and* E. *Then,* $\mathrm{EVAL}_{\gamma(\mathsf{E})}\, t \subseteq \gamma_{\mathsf{m}\,\mathbb{A}}([\![t]\!]^{\mathsf{eval}}_{\mathsf{E}})$.

Proof. We have to show that for every $E \in \gamma(\mathsf{E})$, $m \in \{\mathsf{val}, \mathsf{exn}\}$ and $v \in \mathbb{V}$, if $E \vdash t \Downarrow_m v$, then $r \in \gamma_{\mathsf{m}\,\mathbb{A}}([\![t]\!]^{\mathsf{eval}}_{\mathsf{E}})$, where $r = \mathsf{Success}\, v$ when $m = \mathsf{val}$, and $r = \mathsf{Exception}\, v$ when $m = \mathsf{exn}$. The proof proceeds by induction on the evaluation judgement, generalising over m and E. The only interesting case is the one for function application, which exploits the induction hypothesis, the post-fixpoint property of eval and the soundness of abstract inclusion $\sqsubseteq$. All other cases result from the soundness of the abstract operations and from induction hypotheses. □

The soundness theorem assumes that eval is a post-fixpoint, *i.e.*, $[\![t]\!]^{\mathsf{eval}}_{\mathsf{E}} \sqsubseteq \mathsf{eval}\, t\, \mathsf{E}$. This property is ensured by the soundness of the fixpoint solver, that always returns a post-fixpoint. The function eval is, indeed, the result of the fixpoint solver called on the function $\lambda\mathsf{eval}.\lambda t.\lambda\mathsf{E}.[\![t]\!]^{\mathsf{eval}}_{\mathsf{E}}$.

6 An Abstract Interpreter for OCaml Programs

Based on the abstract interpreter of §5, we implemented a static analyser for OCaml programs (version 4.14), that returns a map from top-level identifiers of the program to their abstract values. Our prototype and its test suite (see §7) are available as a companion artefact [35].

We have implemented several optimisations, that are crucial to obtain decent performance. For example, nodes of the analysed AST are indexed by program points using unique integers as identifiers. This enables efficient comparison of sub-terms and allows using efficient data structures like Patricia trees [53]. Moreover—this is of paramount importance for performance—we perform *hash-consing* of abstract values and *memoise* the operations on these abstract values.

We present in the next sections some key implementation details that we needed to analyse OCaml programs.

6.1 Refinements With Respect to the Formal Presentation

The abstract interpreter we implemented follows the structure we have presented in §5.2, but implements three more refinements, that we purposely elided to follow the presentation more easily. A thorough presentation of these refinements would go beyond the scope of the current paper.

Context sensitivity. Our analyser is *context sensitive*: we implemented a form of *call site* sensitivity, that is akin to an abstraction of the call stack. Following [50], we retain full sensitivity until the list of call sites becomes *maximal*, *i.e.*, when a program point appears more than once in that list, which may indicate a recursive call to some function. In addition, we always remember the last call site. In practice, the list of call sites is an additional parameter to the abstract interpreter. Following [50] again, we use this list of call sites to decide when widening on the environments should be performed: it is performed only when eval is called on a *maximal* list of call sites. The same list of call sites is also used to derive dynamic exception names and abstract pointers (see §6.4 and §6.5).

Flow sensitivity. Our abstract interpreter is able to exploit information that is learned when a branch in a match is taken, or when branching on an arithmetic test. For example, in the program match (x, y) with (None, _) $\Rightarrow x$ | _ $\Rightarrow t$, our analyser is able to refine the possible environments, by taking into account that $x =$ None in the first branch, and that this first branch necessarily returns the value None. This is done by performing a *backward analysis* of the scrutinee (x, y). This backward analysis infers an over-approximation of the environment, knowing that the scrutinee successfully matched against the pattern (None, _).

Dynamic partitioning. Finally, we have employed a form of *dynamic partitioning* to avoid conflating some analyses results, that could degrade precision. Based on a notion of *similarity* on the shapes of abstract values found in environments, we decide whether to conflate contexts or not. The technique is inspired by the *silhouettes* used in shape analysis [39].

6.2 Transformation of Typed OCaml ASTs

The actual language that our interpreter takes as input is more complex than the one we presented in §3, but undoubtedly simpler than the OCaml AST. The main differences between our intermediate language and the OCaml AST, is that we deal with only one construct for pattern matching, and only one construct for exception handling, and that those two constructs implement orthogonal features in our language. This is in contrast with OCaml's `try t1 with p -> t2` and `match t with p1 -> t1 | exception p2 -> t2`, that conflate pattern matching with exception handling. The transformation into our two constructs is mostly straightforward, and greatly simplifies the job of the static analyser.

Our intermediate language makes the evaluation order explicit using let bindings. While the evaluation order in OCaml is generally unspecified, we did our best to mimic the choices that the OCaml compiler makes.

We added specific application nodes for OCaml primitives. To ensure they are called with the correct arity, we inserted λ-abstractions when they were partially applied, or additional application nodes when they were given more arguments than expected. We also handled specifically the short-circuiting primitives on boolean expressions `&&` and `||`, as they change the evaluation order.

We kept the n-ary application nodes of the OCaml AST (instead of the binary applications from §3), as this is important for the semantics of labelled/optional function arguments. Nevertheless, the transformation from the OCaml AST into our intermediate language needed a lot of care and effort. In particular, missing labelled arguments required the insertion of λ-abstractions, which can be particularly subtle when interacting with optional arguments.

6.3 Pattern Disambiguation

The last major difference between OCaml and our intermediate language is the exhaustive and non-ambiguous requirements on pattern matching. These

properties not only simplify the semantics of our intermediate language, but also facilitate the analysis of programs. Indeed, each branch of the pattern-matching can be analysed *independently* of the other ones, whereas in OCaml, branches must be considered in order, until one pattern matches the inspected value. The OCaml type-checker still provides warnings to verify the *utility* of each branch and the *exhaustiveness* of the overall pattern matching.

Enforcing exhaustive and non-ambiguous pattern matchings in OCaml would require to use of cumbersome patterns, and, furthermore, it is not always possible to write such patterns in OCaml. It is, indeed, allowed to match on values whose types may have an infinity of constructors, *e.g.*, arrays, strings, or extensible variant types (see §6.4 for details). To reach these requirements, we extend the language of patterns with a complement $p \setminus q$ [8]. A value v matches a pattern $p \setminus q$ if and only if it matches p but not q. In an ordered pattern matching $\mathsf{match}\ t\ \mathsf{with}\ p_1 \Rightarrow u_1 \mid \cdots \mid p_n \Rightarrow u_n$, we can express that the value v of the term t matches the i^{th} pattern, unambiguously. It suffices to add that v does not match any of the preceding patterns p_j with $j < i$, *i.e.*, v matches $p_i \setminus (\Sigma p_j) \preccurlyeq v$.

The method presented in [8] shows how to solve the disambiguation problem [32]. It relies on the notion of pattern semantics $[\![p]\!]$ that is the set of values matched by a pattern: $[\![p]\!] = \{v \in \mathbb{V} \mid p \preccurlyeq v\}$. The idea is to reduce any pattern p into a purely disjunctive pattern q, *i.e.*, a pattern containing no complements $\setminus$, while preserving its semantics : $[\![p]\!] = [\![q]\!]$. The reduction relies on rewriting rules that correspond to algebraic laws of set theory: a constructor c behaves like a labelled cartesian product, the disjunction $+$ like set union, and the complement $\setminus$ like set difference. Note that the pattern language proposed in §3 conflates the different forms of OCaml constructors (constructor variant, polymorphic variant, records, arrays and tuples) as they behave similarly *w.r.t.* to their semantics.

In order to *fully* reduce a pattern, the method also relies on the observation that a variable x_τ of a variant type τ must be matched by a value whose head is a constructor of the type τ. Therefore, the semantics of this variable x_τ can be described as the union of semantics of all constructor instances of τ: $[\![x_\tau]\!] = \bigcup_{c \in \mathcal{C}_\tau} [\![c(z_1, \ldots, z_n)]\!]$, where $\mathcal{C}_\tau$ is the finite set of constructors of co-domain τ. Similarly, the utility [40] approach, implemented in the OCaml compiler, relies on the ability to enumerate all the constructors of a type to provide a non-ambiguous description of the useful patterns. For types that may not be finitely described, the semantic approach can still be used to partially reduce the *complements* [7]. We keep *anti-patterns*—patterns of the form $x \setminus q$ where q contains no complements—when there exists a value v such that $x \setminus q \preccurlyeq v$.

Finally, to guarantee the exhaustiveness of pattern matching, it suffices to add a rule $z \setminus (p_1 + \cdots + p_n) \Rightarrow \mathsf{raise\ Match\_failure}$ when necessary. Again, generating such a non-ambiguous rule, for data types that may not be finitely described, is only possible thanks to pattern complements.

6.4 Dynamic Exceptions

The exception type in OCaml is an *extensible variant type*: it can be *dynamically* extended with new variant constructors. This means that new exception con-

$$t ::= \ldots \mid \mathsf{let\ exception}\ \overline{c}\ \mathsf{of}\ \tau_1 * \cdots * \tau_n\ \mathsf{in}\ t \mid \mathsf{let\ exception}\ \overline{b} = \overline{c}\ \mathsf{in}\ t$$
$$v ::= \ldots \mid d \mid d(v_1, \ldots, v_k)$$

$$\frac{E(\overline{c}) = d}{S; E \vdash \overline{c}(x_1, \ldots, x_k) \Downarrow_{\mathsf{val}} S; d(E(x_1), \ldots, E(x_n))}\ \textsc{DynamicConstruct}$$

$$\frac{S \uplus \{d\}; E, \overline{c} \mapsto d \vdash t \Downarrow_m S'; v}{S; E \vdash \mathsf{let\ exception}\ \overline{c}\ \mathsf{of}\ \tau_1 * \cdots * \tau_n\ \mathsf{in}\ t \Downarrow_m S'; v}\ \textsc{LetException}$$

$$\frac{S; E, \overline{b} \mapsto d \vdash t \Downarrow_m S'; v \qquad E(\overline{c}) = d}{S; E \vdash \mathsf{let\ exception}\ \overline{b} = \overline{c}\ \mathsf{in}\ t \Downarrow_m S'; v}\ \textsc{RebindException}$$

$$\mathsf{A} ::= \{\ldots;\ \mathsf{names} : \mathsf{V}\} \mid \alpha \mid \mu\alpha.\mathsf{A} \qquad \text{(Abstract value)}$$
$$\mathsf{V} ::= \{\overline{(c, \delta)}\} \qquad \text{(Abstract names)}$$

$$[\![\mathsf{let\ exception}\ \overline{c}\ \mathsf{of}\ \tau_1 * \cdots * \tau_n\ \mathsf{in}\ t]\!]^{\mathsf{eval}}_{\mathsf{E}} = [\![t]\!]^{\mathsf{eval}}_{\mathsf{E}, \overline{c} \mapsto \{\mathsf{names} = (c, \delta)\}}$$
$$[\![\mathsf{let\ exception}\ \overline{b} = \overline{c}\ \mathsf{in}\ t]\!]^{\mathsf{eval}}_{\mathsf{E}} = [\![t]\!]^{\mathsf{eval}}_{\mathsf{E}, \overline{b} \mapsto \mathsf{E}(\overline{c})}$$

Fig. 3. Changes to support dynamic exception naming (excerpts).

structors are dynamically generated during the execution of programs. Although this section focuses on the exception type, the techniques we present apply to any extensible variant type as well.

To model the dynamic behaviour of type extension, we introduce dynamic constructors, written $\overline{c}$, that, unlike static constructors c, are dynamically associated to a variant name d during the evaluation. We update the language of §3 and its semantics to support these dynamic constructors (Figure 3).

The $\mathsf{let\ exception}\ \overline{c}\ \mathsf{of}\ \tau_1 * \cdots * \tau_n\ \mathsf{in}\ t$ construct defines the new exception constructor $\overline{c}$, that is dynamically bound to a fresh variant name in the sub-term t. The *exception alias* construct $\mathsf{let\ exception}\ \overline{b} = \overline{c}\ \mathsf{in}\ t$ defines the exception constructor $\overline{b}$, that is bound in the sub-term t to the variant name of $\overline{c}$. Constructed values can now have a dynamic variant name as their head constructor.

To account for the generative aspect of dynamic constructors, the evaluation rules now carry an execution state S, that contains the set of the already generated variant names. These are akin to the *time-stamps* from the CFA literature [25,44], that are used to allocate data in memory locations. In the analysis, we use an over-approximation δ of the list of call sites—that we used already in §6.1 to control the widening strategy—to give abstract names (c, δ) to dynamic constructors.

Finally, as the variant name of an exception constructor is resolved dynamically, the pattern matching relation depends on the evaluation environment E: $\overline{c}(p_1, \ldots, p_n) \preccurlyeq d(v_1, \ldots, v_n)$ if and only if $E(\overline{c}) = d$, and $p_i \preccurlyeq v_i$ for all $i \in [1, n]$.

As the exception type is extensible, a *finite* number of constructor patterns never forms an exhaustive set of patterns for the exception type. Therefore, the utility approach on pattern matching [40] used in OCaml for exhaustiveness checking cannot provide an exhaustive list of *non-ambiguous* counter-examples: that list is not known statically. In contrast, the disambiguation approach from §6.3 is particularly well suited to such types, by leveraging *anti-patterns* [7]. Moreover,

$$
\begin{array}{lll}
t ::= \ldots \mid \{f_1 = x_1; \ldots; f_n = x_n\} \mid x.f \mid x.f \leftarrow y & & \\
v ::= \ldots \mid \ell & & \text{(Heap locations)} \\
S ::= \{\ell_1 \mapsto r_1;\ \ldots;\ \ell_n \mapsto r_n\} & & \text{(Memory heaps)} \\
r ::= \{f_1 \mapsto v_1;\ \ldots;\ f_n \mapsto v_n\} & & \text{(Record blocks)}
\end{array}
$$

$$
\frac{\ell \notin \operatorname{dom} S \qquad S' = S, \ell \mapsto \{f_1 \mapsto E(x_1); \ldots; f_n \mapsto E(x_n)\}}{S; E \vdash \{f_1 = x_1; \ldots; f_n = x_n\} \Downarrow_{\mathsf{val}} S'; \ell} \textsc{Alloc}
$$

$$
\frac{E(x) = \ell \qquad S(\ell) = \{f_1 \mapsto v_1; \ldots; f_n \mapsto v_n\} \qquad 1 \le i \le n}{S; E \vdash x.f_i \Downarrow_{\mathsf{val}} S; v_i} \textsc{GetField}
$$

$$
\frac{\begin{array}{c} E(x) = \ell \qquad S(\ell) = \{f_1 \mapsto v_1; \ldots; f_n \mapsto v_n\} \\ 1 \le i \le n \qquad S' = S, \ell \mapsto \{f_1 \mapsto v_1; \ldots; f_i \mapsto E(y); \ldots; f_n \mapsto v_n\} \end{array}}{S; E \vdash x.f_i \leftarrow y \Downarrow_{\mathsf{val}} S'; ()} \textsc{SetField}
$$

$$
\begin{array}{lll}
\mathsf{A} ::= \{\ \ldots;\ \mathsf{locs} : \{\ell_1^\sharp, \ldots, \ell_n^\sharp\}\} & & \text{(Abstract locations in abstract values)} \\
h^\sharp ::= \{\ell_1^\sharp \mapsto r_1^\sharp;\ \ldots;\ \ell_n^\sharp \mapsto r_n^\sharp\} & & \text{(Abstract heaps)} \\
r^\sharp ::= \{f_1 \mapsto \mathsf{A}_1;\ \ldots;\ f_n \mapsto \mathsf{A}_n\} & & \text{(Abstract record blocks)}
\end{array}
$$

Fig. 4. Changes to support mutable records (excerpts).

the equality of two exception constructors $\overline{b}$ and $\overline{c}$ of the same arity can only be resolved dynamically. Therefore, there is no way to statically prove, or disprove, the utility of a pattern $\overline{b}(q_1, \ldots, q_n)$ against a pattern $\overline{c}(p_1, \ldots, p_n)$. On the other hand, in our pattern formalism, we can simply write $\overline{b}(q_1, \ldots, q_n) \setminus \overline{c}(p_1, \ldots, p_n)$ to guarantee the non-ambiguity between the two.

6.5 Mutable Records and Global State

OCaml supports *mutable* records. While immutable records can be modelled in the programming language of §3 in the form of *constructs*—an immutable record is a variant with a single case—mutable records require extending the semantics with a global memory heap S (Figure 4).

Heaps are maps from memory locations ℓ to record blocks. Record blocks are structured memory blocks, that contain values for all the registered fields of the record. The standard notion of *reference* can be modelled as a mutable record with a single field. This is exactly how the type of references is defined in OCaml.

We adapt the big-step semantics in a standard way, so that it takes a heap as input and returns an updated heap as output. The evaluation rules for record creation, access, and update, either query or modify the memory heap as expected.

OCaml features pattern matching on mutable records. We adapt the rules for pattern matching, so that matching on a mutable record first queries the memory heap to retrieve the values for the fields of the record, before matching continues.

To analyse programs that involve mutable records, we add a new field to abstract values, that contains the possible *abstract locations* $\ell^\sharp$ a value might be equal to. Abstract locations denote sets of concrete locations. Similarly to the

dynamic extension constructors of §6.4, fresh abstract locations are chosen by following a naming scheme that is based on the abstract call stack.

The abstract interpreter is easily adapted to support global state, by lifting the abstract exception monad to the state monad, where states are abstract heaps. Abstract heaps map abstract locations to abstract record blocks, that themselves map record fields to abstract values. The operations on abstract heaps and the transfer functions on records are standard, and elided from the presentation.

6.6 Modules and Functors

The OCaml language includes an expressive module system [36], that supports hierarchical structures, higher-order functors, and first-class modules. In this section, we give the reader the main insights for the analysis of OCaml modules.

First, we consider an *untyped* semantics of modules, *i.e.*, we do not propagate type information. In particular, we do not take type abstraction boundaries into account. We carefully to keep track of module *coercions*, however: signature ascriptions may have, indeed, a computational content, as they can remove some module fields. Coercions are automatically applied at functor applications to "reshape" the functor argument. Coercions distribute on functors, contravariantly on their formal arguments, and covariantly on their results.

Embracing further the untyped nature of our approach, we made the choice of having a *single class of values*, that comprises both values from the core language and values for module structures and functor closures. This simplifies both the concrete semantics (for example, transfers from the module language to the core language and back are *no-ops*), and the design of the abstract domain. As we sketched in the previous sections, it suffices to add new fields to abstract values to describe the possible structures and functor closures.

We represent structures as unordered records, *i.e.*, maps from field names to values. Functor closures hold the functor code, an environment, and coercions for the argument and the result, that shall be applied when the functor is called.

Importantly, the support of dynamic exceptions (§6.4) was *required* to support functors, since an exception might be declared in a functor's body: this leads to the creation of a fresh exception every time this functor is instantiated.

The analysis functions for the core language and the module language, of types $\mathbb{T} \to \mathbb{E} \to \mathbb{A}$ and $\mathbb{M} \to \mathbb{E} \to \mathbb{A}$, are mutually recursive. Still, the approach of using a fixpoint solver to define our abstract interpreter remains applicable. The two functions can be transformed into a single function of type $(\mathbb{T}+\mathbb{M}) \to \mathbb{E} \to \mathbb{A}$, then given to the solver, and split back into two functions. Our untyped approach was again crucial, as we could keep a single type of abstract values, and a single type of abstract environments, which made the previous transformation possible.

7 Experiments

We tested our prototype analyser for OCaml programs on 290 programs, that range from small, manually written programs, to larger examples extracted

Table 1. Experiments: size of the programs, analysis time (with minimisation disabled, and enabled). They are sorted by program decreasing size.

Program	Size (LoC)	Analysis (w/o minim.)	Analysis (w/ minim.)
`heintze_mcallester_1000`	4002	0.2 s	0.2 s
`boyer`	1292	26 m	57 m
`kb`	552	1.2 s	1.4 s
`map_merge`	152	4.5 s	5.6 s
`sliding_window`	122	4.4 s	5.8 s
`skolemize`	82	38 m	2.9 s
`negative_normal_form`	64	40 m	4.6 s
`red_black_trees2`	64	0.5 s	1.0 s
`church`	20	0.05 s	0.07 s
`sieve`	19	0.01 s	0.01 s
`tak_cps`	8	0.04 s	0.04 s
`tak`	4	$<$ 0.01 s	0.01 s
`mc91_cps`	4	$<$ 0.01 s	$<$ 0.01 s
`mc91`	2	$<$ 0.01 s	$<$ 0.01 s

from the literature or from the OCaml compiler's test suite. The test programs include some classic functions such as the factorial program from §2, Takeuchi's function, McCarthy's 91 function, fixpoint combinators, programs that compute over church numerals, transformations of abstract syntax trees for arithmetic expressions or logical formulas, and the algorithm for Knuth-Bendix completion of rewriting systems. The test suite covers a large array of coding styles, *e.g.*, direct style, continuation-passing style, monadic style, or imperative style, and exhibits different language features, *e.g.*, assertions, exception-based control flow, GADTs and non-regular types, polymorphic recursion, second-order polymorphism, *etc.*

We present in Table 1, a selection of the test results on some key examples. The complete test results are reproducible via the companion artefact [35]. The experimental results are encouraging, both in terms of performance and precision.

In terms of precision, our analyser infers the best achievable abstract values on several programs: For McCarthy's 91 function `mc91`, the result is shown to be greater than 91; for the skolemisation of logical formulas `skolemize`, the analyser correctly infers the form of returned terms, *i.e.*, they cannot contain existential quantifiers. For other programs, the analyser only infers an over-approximation: for the `red_black_tree` program, it correctly infers the general shape of trees, but cannot infer the structural invariant that no red node has red children.

The `map_merge` example calls the `Map.Make` functor of finite maps from the standard library, builds several maps, and calls the `merge` function on those maps, that merges the maps. The `merge` function has the following signature:

```
val merge: (key -> 'a option -> 'b option -> 'c option) ->
           'a M.t -> 'b M.t -> 'c M.t
```

Its first argument specifies what should be done when a key/value pair is found in one of the maps, or in both. This argument is *never* called for keys that are absent in both maps, *i.e.*, the case where the second and third arguments are both equal to `None` is *unreachable*. OCaml programmers often write `assert false` in the corresponding pattern matching branch. The analyser infers that the `Assertion_failure` exception is never raised, which means that this branch cannot be reached. The analyser cannot show, however, that every assertion present in `Map.Make` is satisfied: in the re-balancing function for pseudo-balanced trees, assertion failures are reported, because the analyser cannot infer that the heights that are recorded in the trees are strictly positive.

In terms of performance, most examples, and even some large programs, are analysed in a couple of seconds, or in less than a second. In contrast, some examples like `boyer` need approximately one hour for the analysis to terminate. `boyer` is a tautology checker, that is run on a large formula (its definition takes about 1000 lines). This formula, of mutable type, requires the creation of several hundreds of abstract pointers, which makes abstract operations on abstract heaps very costly. If we reduce context sensitivity to "the *last* call site", fewer abstract pointers are created, and the analysis completes in 31 s. This suggests that context sensitivity choices for naming abstract pointers need further investigation.

Our experiments show that the minimisation of abstract values during widening and unions (§4.2) may impact performance positively or negatively. For instance, for AST transformations like `skolemize` and `negative_normal_form`, minimisation decreases the analysis time from about 45 m down to a few seconds. For `boyer`, however, minimisation incurs a heavy cost, as it doubles the analysis time. Further investigations are needed to reduce the cost of minimisation.

8 Related Work

The static detection of uncaught exceptions for ML programs has been the topic of many related work. We only discuss a selection of them, and some results on static analysis of functional programs that are also relevant to the current work.

Set Constraints. Several static analyses for functional programs were based *set constraints* [21]. The principle is to transform a program into a constraint, that features unions, intersections, negations, and a form of conditional constraint. Then, the constraint is simplified and given to a solver, from which the analysis result is obtained. Fähndrich and his coauthors built a exception analysis tool that infers types and effects for SML programs [14,15] using the BANE constraint analysis engine, using a mix of set constraints and type constraints.

Type and Effect Systems. Pessaux and Leroy have developed `ocamlexc` [38,54,55], a tool that detects uncaught exceptions in OCaml programs. They use a *type and effect* system to analyse programs modularly. Their analyser extends unification-based type inference, and makes use of *row variables* [57] and polymorphism to produce precise types for functions. They type variants

structurally using equi-recursive types. Recursion may also occur through the effect annotations on arrow types. They also describe an algorithm to improve the accuracy of their analysis, that uses *polymorphic recursion* for row variables. The programming language Koka [33] also leverages row variables to type algebraic effects. Recently, de Vilhena and Pottier [62] devised a type system based on row variables for a language that supports the dynamic creation of algebraic effects.

Control-Flow Analyses. An important family of analyses for higher-order programs are control-flow analyses (CFA) [60,65,51,45,19]. The goal of CFA is to determine which functions might be called at a call site, and on which arguments. CFA can be expressed as instances of abstract interpretation [46,44,47,50]. CFA can easily be extended to analyse exceptions. Yi developed an abstract interpreter that detects uncaught exceptions in SML [68,67,66]. It implements an analysis that is close to a 0-CFA analysis extended to support exceptions.

Abstract Domains in CFA. Most previous work on CFA share a common representation for abstract values: Although they need to represent some inductively defined sets, they refrain from using a native device to express fixpoints, such as our μ constructor. Instead, cyclic definitions are encoded using *indirections through abstract pointers*, that point to an *abstract heap*. For example, the inductive set of continuations from §2 is expressed as follows in CFA domains:

$$\begin{array}{ll} \{\mathsf{funs} : \{(\lambda x.\, x) \mapsto \{\};\ (\lambda x.\, k\ (x * i)) \mapsto \{i \mapsto p_i; k \mapsto p_k\}\}\} \\ \text{where:} & \hat{h}(p_i) = \{\mathsf{ints} : [1, +\infty]\} \\ & \hat{h}(p_k) = \{(\lambda x.\, x) \mapsto \{\};\ (\lambda x.\, k\ (x * i)) \mapsto \{i \mapsto p_i; k \mapsto p_k\}\} \end{array}$$

In this abstract value, the closures' environments contain the pointers p_i and p_k, that are defined in the abstract heap $\hat{h}$. This abstract heap contains a cycle, since p_k is used in the definition of the abstract value pointed by p_k. This is in contrast to our approach, where we make use of μ nodes to introduce cycles directly, without referring to a heap. We only use the abstract heap for mutable data. In CFA domains, all data (constructs, closures, *etc.*) are "abstractly allocated" in the abstract heap, regardless of whether they are mutable or not.

A benefit of the approach with heap indirections is that abstract values have a bounded height, and cycles need no special treatment: The equality of abstract pointers is used to compute on abstract values. While this makes the operations of CFA abstract domains easy to define, using pointer names limits drastically the detection of semantically equivalent values. We argue that our approach allows to detect more semantics inclusions, therefore decreasing the number of iterations of the analysers, at the cost of more complex abstract domain operations.

Tree Grammars. Several analyses for functional languages have been defined using *tree grammars*. For example, Reynolds [58] defined an analysis for pure first-order LISP using *data sets*, *i.e.*, tree grammars that denote the possible outputs of function symbols. Extended tree grammars, *i.e.*, grammars with *selectors* of the form $X \to Y.hd$, have been used by Jones and his coauthors to analyse full LISP

[28], and, later, strict and lazy λ-calculi [26,27]. From a λ-term, they produce tree grammars with selectors, that denote the possible inputs and outputs of function symbols. Selectors can then be eliminated in order to simplify the grammars. *Deterministic* tree grammars have been identified as an abstract domain to recast analyses based on set constraints into the abstract interpretation framework [10].

Tree Automata. Generalising string automata, tree automata are an established formalism to represent sets of trees. They have been used to define static analysers for term-rewriting systems (TRSs) [3] and higher-order programs [20]. They have been extended to *lattice tree automata* to support arbitrary non-relational abstract domains at their leaves [17,18], and improve the performance of analysers for TRSs. Recently, tree automata were combined with relational numeric abstract domains [29], to express *relations* between scalar data contained in trees. Recent work report on the design of relational domains for algebraic data types [2,61].

Cyclic Abstract Domains. Type graphs [22] are a form of deterministic tree grammars, that are represented as cyclic graphs with *no sharing*, *i.e.*, trees with cycles. They have been used to analyse Prolog programs. We used a similar graph-based representation as an intermediate form to compute union, intersection and widening. We use, however, a term-based representation with binders as our main representation, as it allows easy and efficient hash-consing and memoisation [13]. Our widening operator (§4.2) is inspired by the one from type graphs.

Mauborgne [42,43,41] studied graph-based abstract domains for sets of trees, and defined ways to have minimal, canonical representations of such abstract values. Using Mauborgne's structures natively could improve our analyser's performance, as we could avoid translating back and forth from terms to graphs.

Finally, *recursive types* [56] were a strong inspiration for the abstract domain of §4. Recursive types have been thoroughly studied in the context of subtyping [16,31,1], where polynomial algorithms have been devised to decide inclusion. They proceed by translating types into variants of tree automata, that can also deal with the contravariance of arrow types.

Fixpoint Solvers. To the best of our knowledge, Le Charlier and Hentenryck [6] were the first to exploit a dynamic fixpoint solvers to define static analysers. They used the *top-down solver* to analyse Prolog programs. The same approach has been followed for the Goblint static analyser for C programs [64,59], and for the analysis of WebAssembly programs [4]. Recent work introduced combinators to define dynamic fixpoint solvers in a modular manner [30]. Several dynamic fixpoint solvers have been successfully formally verified [24,63].

9 Conclusive Remarks and Future Work

We have introduced a λ-calculus that features pattern matching primitives and exception handling, in which exceptions are first-class citizens. We have presented a static analysis for this language, in the form of a monadic abstract interpreter,

that can be used as an effective static analyser. This analyser detects uncaught exceptions, and provides a description of the values that a program may return. The abstract interpreter relies on a generic abstract domain, that is parameterised over a domain for scalars, and that can represent *regular sets* of values of our programming language. This is achieved by a fixpoint constructor in the syntax of abstract values, that denotes an inductive set of values.

The abstract interpreter is defined in an *open recursive style*, where the recursive knot is tied by calling a *dynamic fixpoint solver*. Importantly, the analyser does not call the solver for every recursive call: it performs standard recursive calls on strict sub-terms, but calls the solver to analyse function calls.

Based on this approach, we implemented a static analyser for OCaml programs. We presented some extensions of our formalism to support several core features of OCaml, including dynamic generation of exceptions, mutable records, the module system. Our analyser starts with transforming the OCaml typed AST into a simpler language where evaluation order is explicit. This transformation required a lot of care and demanded a substantial implementation effort. One key aspect of this transformation is the disambiguation of pattern matching, as we chose to work with an *exhaustive and non-ambiguous* pattern matching primitive in order to simplify the analysis of programs.

Our experiments on 290 OCaml programs show some encouraging results, both in terms of performance and precision. Still, some improvements are needed for the analysis to be applicable to larger code bases. In particular, the minimisation of abstract values requires some more study and fine tuning: while it plays a crucial role to analyse some examples in a reasonable time, it can also severely undermine the analyser's performance in some other cases.

At the moment, the analyser can deal with whole programs only. To analyse libraries more modularly, we plan to experiment with generating abstract values that over-approximate the inputs of a library's function, based on their types. In the near future, we also plan to extend the analyser with OCaml features that are yet to be supported (*e.g.*, arrays, laziness, floats, objects, recursive modules, interactions with the operating system, *etc.*), most of which will require substantial formalisation and implementation efforts. Recently introduced features, such as algebraic effects and one-shot continuations, are also on our agenda, and are likely to raise interesting challenges.

Finally, we hope that our abstract interpreter can be extended to perform other kinds of static analyses for OCaml programs, such as a purity analysis, or the detection of whether the behaviour of a program might depend on the *order of evaluation*. We would also like our implementation to serve as a basis for experimenting with recent relational domains for trees and scalars [29,61,2], and with relational analyses of functional programs [49].

Data-Availability Statement. The companion artefact [35] is hosted on the Zenodo platform and referenced by the DOI `10.5281/zenodo.10457925`.

References

1. Amadio, R.M., Cardelli, L.: Subtyping recursive types p. 575–631 (9 1993). https://doi.org/10.1145/155183.155231
2. Bautista, S., Jensen, T., Montagu, B.: Lifting Numeric Relational Domains to Algebraic Data Types. In: Singh, G., Urban, C. (eds.) Static Analysis. pp. 104–134. Springer Nature Switzerland, Cham (2022)
3. Boichut, Y., Genet, T., Jensen, T., Roux, L.L.: Rewriting Approximations for Fast Prototyping of Static Analyzers. In: Lecture Notes in Computer Science, pp. 48–62. Springer Berlin Heidelberg (2007). https://doi.org/10.1007/978-3-540-73449-9_6
4. Brandl, K., Erdweg, S., Keidel, S., Hansen, N.: Modular Abstract Definitional Interpreters for WebAssembly. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.ECOOP.2023.5
5. Charguéraud, A.: The Locally Nameless Representation. Journal of Automated Reasoning **49**(3), 363–408 (May 2011). https://doi.org/10.1007/s10817-011-9225-2
6. Charlier, B.L., Van Hentenryck, P.: A Universal Top-Down Fixpoint Algorithm. Tech. rep., USA (1992), ftp://ftp.cs.brown.edu/pub/techreports/92/cs92-25.pdf
7. Cirstea, H., Lermusiaux, P., Moreau, P.E.: Static analysis of pattern-free properties. In: Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming, PPDP 2021. pp. 9:1–9:13. ACM (sep 2021). https://doi.org/10.1145/3479394.3479404
8. Cirstea, H., Moreau, P.: Generic Encodings of Constructor Rewriting Systems. In: Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming, PPDP 2019. pp. 8:1–8:12. ACM (oct 2019). https://doi.org/10.1145/3354166.3354173
9. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proceedings of the 4$^{\text{th}}$ ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. POPL '77, Association for Computing Machinery, New York, NY, USA (1977). https://doi.org/10.1145/512950.512973
10. Cousot, P., Cousot, R.: 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). https://doi.org/10.1145/224164.224199
11. Darais, D., Labich, N., Nguyen, P.C., Horn, D.V.: Abstracting definitional interpreters (functional pearl). Proc. ACM Program. Lang. **1**(ICFP), 12:1–12:25 (2017). https://doi.org/10.1145/3110256
12. Fecht, C., Seidl, H.: A Faster Solver for General Systems of Equations. Sci. Comput. Program. **35**(2), 137–161 (1999). https://doi.org/10.1016/S0167-6423(99)00009-X
13. Filliâtre, J.C., Conchon, S.: Type-safe modular hash-consing. In: Proceedings of the 2006 workshop on ML. ACM (sep 2006). https://doi.org/10.1145/1159876.1159880
14. Fähndrich, M., Aiken, A.: Tracking down exceptions in Standard ML programs. techreport 98-996, University of California at Berkeley, Computer Science Division (1998)
15. Fähndrich, M., Forster, J.S., Aiken, A., Cu, J.: Tracking down Exceptions in Standard ML Programs. techreport UCB/CSD-98-996, University of California, Computer Science Division (EECS), Berkeley, California 94720 (Feb 1998), https://theory.stanford.edu/~aiken/publications/papers/tr98.pdf
16. Gapeyev, V., Levin, M.Y., Pierce, B.C.: Recursive Subtyping Revealed. Journal of Functional Programming **12**(6), 511–548 (2002). https://doi.org/10.1017/S0956796802004318

17. Genet, T., Gall, T.L., Legay, A., Murat, V.: A Completion Algorithm for Lattice Tree Automata. In: Konstantinidis, S. (ed.) Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7982, pp. 134–145. Springer (2013). https://doi.org/10.1007/978-3-642-39274-0_13
18. Genet, T., Le Gall, T., Legay, A., Murat, V.: Tree Regular Model Checking for Lattice-Based Automata. In: CIAA - 18th International Conference on Implementation and Application of Automata. LNCS, vol. 7982. Springer, Halifax, Canada (Jul 2013)
19. Gilray, T., Lyde, S., Adams, M.D., Might, M., Horn, D.V.: Pushdown control-flow analysis for free. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 691–704. ACM (2016). https://doi.org/10.1145/2837614.2837631
20. Haudebourg, T., Genet, T., Jensen, T.P.: Regular language type inference with term rewriting. Proc. ACM Program. Lang. **4**(ICFP), 112:1–112:29 (2020). https://doi.org/10.1145/3408994
21. Heintze, N., Jaffar, J.: Set constraints and set-based analysis. In: Lecture Notes in Computer Science, pp. 281–298. Springer Berlin Heidelberg (1994). https://doi.org/10.1007/3-540-58601-6_107
22. Hentenryck, P.V., Cortesi, A., Charlier, B.L.: Type analysis of Prolog using type graphs. The Journal of Logic Programming **22**(3), 179–209 (Mar 1995). https://doi.org/10.1016/0743-1066(94)00021-w
23. Hinze, R., Paterson, R.: Finger trees: a simple general-purpose data structure. Journal of Functional Programming **16**(02), 197 (nov 2005). https://doi.org/10.1017/s0956796805005769
24. Hofmann, M., Karbyshev, A., Seidl, H.: Verifying a Local Generic Solver in Coq. In: Cousot, R., Martel, M. (eds.) Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6337, pp. 340–355. Springer (2010). https://doi.org/10.1007/978-3-642-15769-1_21
25. Horn, D.V., Might, M.: Abstracting Abstract Machines. In: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming - ICFP '10. ACM Press (2010). https://doi.org/10.1145/1863543.1863553
26. Jones, N.D.: Flow Analysis of Lambda Expressions. DAIMI Report Series **10**(128) (Jan 1981). https://doi.org/10.7146/dpb.v10i128.7404
27. Jones, N.D., Andersen, N.: Flow analysis of lazy higher-order functional programs. Theoretical Computer Science **375**(1-3), 120–136 (May 2007). https://doi.org/10.1016/j.tcs.2006.12.030
28. Jones, N.D., Muchnick, S.S.: Flow Analysis and Optimization of LISP-like Structures. In: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '79. ACM Press (1979). https://doi.org/10.1145/567752.567776
29. Journault, M., Miné, A., Ouadjaout, A.: An Abstract Domain for Trees with Numeric Relations. In: Caires, L. (ed.) 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. Lecture Notes in Computer Science, vol. 11423, pp. 724–751. Springer (2019). https://doi.org/10.1007/978-3-030-17184-1_26

30. Keidel, S., Erdweg, S., Hombücher, T.: Combinator-Based Fixpoint Algorithms for Big-Step Abstract Interpreters. Proceedings of the ACM on Programming Languages **7**(ICFP), 955–981 (aug 2023). https://doi.org/10.1145/3607863
31. Kozen, D., Palsberg, J., Schwartzbach, M.I.: Efficient Recursive Subtyping. Mathematical Structures in Computer Science **5**(1), 113–125 (Mar 1995). https://doi.org/10.1017/s0960129500000657
32. Krauss, A.: Pattern minimization problems over recursive data types. In: Hook, J., Thiemann, P. (eds.) Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP '08. pp. 267–274. ACM (sep 2008). https://doi.org/10.1145/1411204.1411242
33. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. ACM (jan 2017). https://doi.org/10.1145/3009837.3009872
34. Lermusiaux, P., Montagu, B.: Detection of Uncaught Exceptions in Functional Programs by Abstract Interpretation (Extended Version). Research report, Inria (Jan 2024), https://inria.hal.science/hal-04410771
35. Lermusiaux, P., Montagu, B.: Detection of Uncaught Exceptions in Functional Programs by Abstract Interpretation: Software Artefact (Jan 2024). https://doi.org/10.5281/zenodo.10457925
36. Leroy, X.: A Modular Module System. Journal of Functional Programming **10**(3), 269–303 (2000). https://doi.org/10.1017/S0956796800003683
37. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml System, Documentation and User's Manual – Release 5.1. INRIA (Nov 2023), https://v2.ocaml.org/releases/5.1/htmlman/index.html
38. Leroy, X., Pessaux, F.: Type-based analysis of uncaught exceptions. ACM Transactions on Programming Languages and Systems **22**(2), 340–377 (2000). https://doi.org/10.1145/349214.349230
39. Li, H., Berenger, F., Chang, B.E., Rival, X.: Semantic-directed clumping of disjunctive abstract states. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 32–45. ACM (2017). https://doi.org/10.1145/3009837.3009881
40. Maranget, L.: Warnings for pattern matching. Journal of Functional Programming **17**(3), 387–421 (2007). https://doi.org/10.1017/S0956796807006223
41. Mauborgne, L.: Representation of Sets of Trees for Abstract Interpretation. phdthesis, École Polytechnique (Nov 1999), https://www.di.ens.fr/~mauborgn/publi/t.pdf
42. Mauborgne, L.: An Incremental Unique Representation for Regular Trees. Nordic Journal of Computing **7**(4), 290–311 (Dec 2000), https://software.imdea.org/~mauborgn/publi/njc7.pdf
43. Mauborgne, L.: Improving the representation of infinite trees to deal with sets of trees. In: Smolka, G. (ed.) Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1782, pp. 275–289. Springer (2000). https://doi.org/10.1007/3-540-46425-5_18
44. Midtgaard, J.: Control-flow analysis of functional programs. ACM Computing Surveys **44**(3), 10:1–10:33 (2012). https://doi.org/10.1145/2187671.2187672
45. Midtgaard, J., Jensen, T.: A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In: Static Analysis, pp. 347–362. Springer Berlin Heidelberg (2008). https://doi.org/10.1007/978-3-540-69166-2_23

46. Midtgaard, J., Jensen, T.P.: Control-Flow Analysis of Function Calls and Returns by Abstract Interpretation. In: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming - ICFP '09. ACM Press (2009). https://doi.org/10.1145/1596550.1596592
47. Midtgaard, J., Jensen, T.P.: Control-Flow Analysis of Function Calls and Returns by Abstract Interpretation. Information and Computation **211**, 49–76 (Feb 2012). https://doi.org/10.1016/j.ic.2011.11.005
48. Milner, R.: A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences **17**(3), 348–375 (dec 1978). https://doi.org/10.1016/0022-0000(78)90014-4
49. Montagu, B., Jensen, T.P.: Stable Relations and Abstract Interpretation of Higher-order Programs. Proc. ACM Program. Lang. **4**(ICFP), 119:1–119:30 (2020). https://doi.org/10.1145/3409001
50. Montagu, B., Jensen, T.P.: Trace-Based Control-Flow Analysis. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211. pp. 482–496. ACM (2021). https://doi.org/10.1145/3453483.3454057
51. Nielson, F., Nielson, H.R.: Interprocedural Control Flow Analysis. In: Programming Languages and Systems, pp. 20–39. Springer Berlin Heidelberg (1999). https://doi.org/10.1007/3-540-49099-x_3
52. Okasaki, C.: Red-Black Trees in a Functional Setting. J. Funct. Program. **9**(4), 471–477 (1999), http://journals.cambridge.org/action/displayAbstract?aid=44273
53. Okasaki, C., Gill, A.: Fast Mergeable Integer Maps. In: Morriset, G. (ed.) Proceedings of the 1998 ACM SIGPLAN workshop on ML. pp. 77–86 (Sep 1998)
54. Pessaux, F., Leroy, X.: Type-based analysis of uncaught exceptions. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM (jan 1999). https://doi.org/10.1145/292540.292565
55. Pessaux, F.: Détection statique d'exceptions non rattrapées en Objective Caml. Ph.D. thesis, Université Paris 6 (1999), http://www.theses.fr/1999PA066398
56. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, Mass (2002)
57. Rémy, D.: Type checking records and variants in a natural extension of ML. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89. ACM Press (1989). https://doi.org/10.1145/75277.75284
58. Reynolds, J.C.: Automatic Computation of Data Set Definitions. Information Processing **68**, 456–461 (1969)
59. Seidl, H., Vogler, R.: Three Improvements to the Top-Down Solver. In: Sabel, D., Thiemann, P. (eds.) Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018. pp. 21:1–21:14. ACM (2018). https://doi.org/10.1145/3236950.3236967
60. Shivers, O.: The Semantics of Scheme Control-Flow Analysis. In: Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 190–198. PEPM '91, Association for Computing Machinery, New York, NY, USA (1991). https://doi.org/10.1145/115865.115884
61. Valnet, M., Monat, R., Miné, A.: Analyse statique de valeurs par interprétation abstraite de programmes fonctionnels manipulant des types algébriques récursifs. In: JFLA 2023-34èmes Journées Francophones des Langages Applicatifs. pp. 210–241 (2023)

62. de Vilhena, P.E., Pottier, F.: A Type System for Effect Handlers and Dynamic Labels. In: Programming Languages and Systems, pp. 225–252. Springer Nature Switzerland (2023). https://doi.org/10.1007/978-3-031-30044-8_9
63. de Vilhena, P.E., Pottier, F., Jourdan, J.: Spy game: verifying a local generic solver in Iris. Proc. ACM Program. Lang. **4**(POPL), 33:1–33:28 (2020). https://doi.org/10.1145/3371101
64. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: the goblint approach. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016. pp. 391–402. ACM (2016). https://doi.org/10.1145/2970276.2970337
65. Wright, A.K., Jagannathan, S.: Polymorphic Splitting: An Effective Polyvariant Flow Analysis. ACM Trans. Program. Lang. Syst. **20**(1), 166–207 (1998). https://doi.org/10.1145/271510.271523
66. Yi, K.: Compile-time Detection of Uncaught Exceptions in Standard ML Programs. In: Charlier, B.L. (ed.) Static Analysis, First International Static Analysis Symposium, SAS'94, Namur, Belgium, September 28-30, 1994, Proceedings. Lecture Notes in Computer Science, vol. 864, pp. 238–254. Springer (1994). https://doi.org/10.1007/3-540-58485-4_{4}{4}
67. Yi, K.: An Abstract Interpretation for Estimating Uncaught Exceptions in Standard ML Programs. Sci. Comput. Program. **31**(1), 147–173 (1998). https://doi.org/10.1016/S0167-6423(96)00044-5
68. Yi, K., Ryu, S.: A cost-effective estimation of uncaught exceptions in Standard ML programs. Theoretical Computer Science **277**(1-2), 185–217 (2002). https://doi.org/10.1016/S0304-3975(00)00317-0