# Validating Soundness and Completeness in Pattern-Match Coverage Analyzers

CYRIL MOSER, ETH Zurich, Switzerland
THODORIS SOTIROPOULOS, ETH Zurich, Switzerland
CHENGYU ZHANG, ETH Zurich, Switzerland
ZHENDONG SU, ETH Zurich, Switzerland

Pattern matching is a powerful mechanism for writing safe and expressive conditional logic. Once primarily associated with functional programming, it has become a common paradigm even in non-functional languages, such as Java. Languages that support pattern matching include specific analyzers, known as pattern-match coverage analyzers, to ensure its correct and efficient use by statically verifying properties such as exhaustiveness and redundancy. However, these analyzers can suffer from soundness and completeness issues, leading to false negatives (unsafe patterns mistakenly accepted) or false positives (valid patterns incorrectly rejected).

In this work, we present a systematic approach for validating soundness and completeness in pattern-match coverage analyzers. The approach consists of a novel generator for algebraic data types and pattern-matching statements, supporting features that increase the complexity of coverage analysis, such as generalized algebraic data types. To establish the test oracle without building a reference implementation from scratch, the approach generates both exhaustive and inexhaustive pattern-matching cases, either by construction or by encoding them as SMT formulas. The latter leads to a universal test oracle that cross-checks coverage analysis results against a constraint solver, exposing soundness and completeness bugs in case of inconsistencies.

We implement this approach in Ikaros, which we evaluate on three major compilers: Scala, Java, and Haskell. Despite pattern-match coverage analyzers being only a small part of these compilers, Ikaros has uncovered 16 bugs, of which 12 have been fixed. Notably, 7 instances were important soundness bugs that could lead to unexpected runtime errors. Additionally, Ikaros provides a scalable framework for extending it to any language with ML-like pattern matching.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Functional languages*; • **Theory of computation → Pattern matching**.

Additional Key Words and Phrases: pattern matching, testing, bug, Haskell, Scala, Java, exhaustiveness, redundancy, SMT solver

**ACM Reference Format:**
Cyril Moser, Thodoris Sotiropoulos, Chengyu Zhang, and Zhendong Su. 2025. Validating Soundness and Completeness in Pattern-Match Coverage Analyzers. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 393 (October 2025), 27 pages. https://doi.org/10.1145/3763171

## 1 Introduction

Pattern matching is a powerful and ubiquitous language feature for safe and elegant handling of control flow. It allows programmers to express conditions as patterns that match program values, guiding execution flow accordingly. Traditionally an integral feature of functional languages, such as OCaml and Haskell, pattern matching is now widely adopted in many modern languages, including Rust, Scala, and C#. Even Java, starting from JDK 22, has incorporated ML-like pattern

Authors' Contact Information: Cyril Moser, ETH Zurich, Zurich, Switzerland, cymoser@student.ethz.ch; Thodoris Sotiropoulos, ETH Zurich, Zurich, Switzerland, theodoros.sotiropoulos@inf.ethz.ch; Chengyu Zhang, ETH Zurich, Zurich, Switzerland, chengyu.zhang@inf.ethz.ch; Zhendong Su, ETH Zurich, Zurich, Switzerland, zhendong.su@inf.ethz.ch.

matching into its specification and implementation [Gosling et al. 2024]. This adoption reflects the growing importance of pattern matching in mainstream programming [Cheng and Parreaux 2024].

Two key properties of a pattern-matching expression are *exhaustiveness* and *non-redundancy*. Exhaustiveness ensures that a pattern-matching expression covers *all* possible values of the type of the expression we match against. Non-redundancy guarantees that no pattern is shadowed by a preceding one in the pattern-matching expression. These two properties provide numerous benefits for both program correctness and efficiency. On the one hand, they prevent programmers from unsafe usages of pattern matching, reducing the risk of runtime errors. On the other hand, the compiler leverages these properties to perform aggressive optimizations, such as dead code elimination, leading to more efficient code [Rust Team 2025].

To enforce exhaustiveness and non-redundancy, languages supporting pattern matching include dedicated analyzers in their implementation, known as *pattern-match coverage analyzers* (PMC analyzers for short), which verify exhaustiveness and non-redundancy at compile time. In case of violations, these analyzers issue a corresponding warning to developers. To illustrate how these warnings work in practice, consider the following Scala code and the question: Should the compiler warn developers about potential issues, and specifically, is the `match` expression on lines 4–6 exhaustive?

```scala
1  sealed trait A
2  case class CC_A[T](a: T) extends A
3  val x: CC_A[Int] = CC_A(10)
4  val res: Int = x match {
5    case CC_A(12) => 0
6  }
```

No, the `match` expression is not exhaustive, as the value of variable `x` (line 3) is *not* covered by the pattern `CC_A(12)` (line 5). The corresponding PMC analyzer should detect such issues and warn developers about the missing cases (e.g., `CC_A(_)`). However, PMC analyzers can produce incorrect results because of inherent limitations of their coverage checking algorithms [Graf et al. 2020; Karachalias et al. 2015] or unexpected implementation defects [Chaliasos et al. 2021]. In the latter case, these defects can lead to *unintentional* issues: (1) *soundness bugs*, where the analyzer falsely accepts erroneous pattern-matching statements, and (2) *completeness bugs*, where it incorrectly issues pattern-match coverage warnings, preventing well-written code from compiling.

In the above example program, the coverage analyzer of the Scala compiler (version < 3.7.0) exhibits a soundness bug, as it mistakenly marks the `match` expression as exhaustive, failing to report a corresponding warning to developers. Soundness and completeness bugs in pattern-match coverage analyzers lead to runtime errors or increase development effort by confusing developers with misleading reports. For example, running our Scala program leads to a runtime `MatchError` failure, which should have been caught at compile time.

Recent advancements in random program generation offer promising solutions for testing programming language implementations and program analysis tools [Chaliasos et al. 2022; Even-Mendoza et al. 2023; Frank et al. 2024; Pałka et al. 2011; Yang et al. 2011]. Nevertheless, none of the existing work focuses on finding soundness and completeness bugs in PMC analyzers due to two main challenges. First, they struggle to generate valid and interesting pattern-matching expressions because of limited support for pattern matching and type construction. In fact, most of existing program generators rarely include pattern matching in their resulting programs, making them unsuitable for testing PMC analyzers. Second, they lack a *test oracle* [Weyuker 1982], that is, a way to determine the expected behavior of the analyzer under test in response to a given input. Without such an oracle, there is no way to verify whether the compiler's exhaustiveness and redundancy checks are implemented correctly. For example, suppose we have already solved the first challenge and developed a generator that produces the Scala program shown above, *without knowing in advance whether its pattern-matching expression is exhaustive*. How can we tell whether the `scalac`'s warning about exhaustiveness is right or wrong without having to rebuild our own reference implementation of pattern-match coverage analysis?

**Approach:** To address these challenges, we introduce Ikaros, an effective tool for detecting bugs in pattern-match coverage analyzers. Ikaros achieves this by (1) efficiently generating diverse and complex pattern-matching expressions and (2) employing two approaches for constructing test oracles to identify coverage analysis bugs. Specifically, Ikaros features a generator for complex *algebraic data types* (ADTs), involving various features, such as polymorphism, and generalized algebraic data types (GADTs). Using these ADTs, Ikaros systematically produces pattern-matching statements using two distinct generation strategies. To establish the test oracle, Ikaros generates both exhaustive and inexhaustive patterns, either by construction or by encoding them as SMT formulas. The latter allows Ikaros to reason about exhaustiveness using a constraint solver, and then expose soundness and completeness bugs whenever the compiler's coverage analysis results are not aligned with the result of the constraint solver. Additionally, Ikaros employs an intermediate representation (IR) that enables seamless transformation of generated pattern-matching statements into multiple programming languages.

**Results:** Ikaros currently generates test programs for Scala, Java, and Haskell, but its approach is easily portable to any language with ML-like pattern matching. We evaluated Ikaros on the PMC analyzers integrated into these compilers. Despite these components being a small portion of the compiler codebase, they are prone to both soundness and completeness bugs. Ikaros has uncovered a total of 16 bugs, 12 of which have been fixed by developers. We also evaluated Ikaros's throughput and the types of bugs it detects, demonstrating its efficiency in uncovering diverse issues. Ikaros generates dozens of test programs per second, with most discovered bugs stemming from incorrect implementations of GADT reasoning [Graf et al. 2020].

**Contributions:** Our work makes the following contributions:

- A novel generator for algebraic data types and pattern-matching expressions.
- Two distinct strategies for automatically constructing test oracles that expose soundness and completeness bugs in pattern-match coverage analyzers.
- An openly-available and extensible implementation called Ikaros that is currently capable of producing programs in Scala, Java, and Haskell.
- An in-depth evaluation of Ikaros in terms of its bug-finding capability and performance. Using Ikaros, we have discovered 16 bugs in pattern-match coverage checkers integrated into popular languages, 12 of which have been fixed by developers.

## 2 Illustrative Examples and Background

To motivate our approach, we discuss two illustrative examples of bugs in real-world pattern-match coverage analyzers.

**Soundness bug in `scalac`:** Figure 1a presents a soundness bug in the PMC analyzer integrated into the compiler of Scala. Here, the code defines a generalized algebraic data type (GADT) containing three constructors. Two of them yield values of type `A<Int>` (lines 2, 3), while the third one returns values of type `A<Char>`. Later, the code constructs the object `CC_B(CC_A())`, which is assigned to a variable called `x` of type `A<Int>`. (line 6). In turn, the Scala program pattern-matches against this variable by considering two cases (lines 8–11).

Compiling the program with `scalac` (version < 3.6.0) produces *no* warnings. However, executing the compiled bytecode leads to a runtime exception, specifically a `MatchError`. This is because

```
 1  sealed trait A[T]
 2  case class CC_A() extends A[Int]
 3  case class CC_B[T](x: A[T]) extends A[Int]
 4  case class CC_C() extends A[Char]
 5
 6  val x: A[Int] = CC_B(CC_A())
 7
 8  val y: Int = match x {
 9    case CC_A() => 1
10    case CC_B(CC_B(_)) => 2
11  }
```

(a) A soundness bug in `scalac`.

```
 1  sealed interface A {}
 2  record CC_A() implements A {}
 3  record CC_B() implements A {}
 4
 5  sealed interface B {}
 6  record CC_C(A a, A b) implements B {}
 7
 8  B x = CC_C(CC_B(), CC_B());
 9  int y = switch(x) {
10    case CC_C(CC_A(), CC_A()) -> 1;
11    // case CC_C(CC_B(), CC_B()) -> 2;
12    case CC_C(CC_B(), _) -> 2;
13    case CC_C(_, CC_B()) -> 3;
14  }
```

(b) A completeness bug in `javac`.

Fig. 1. Illustrative bugs found by our approach in various pattern-match coverage analyzers.

the `match` expression on lines 8–11 is *unsafe*: the value of variable `x`, namely `CC_B(CC_A())`, is not covered by either pattern `CC_A()` (line 9) or pattern `CC_B(CC_B(_))` (line 10), violating the exhaustiveness property. The expected behavior is that the compiler should have issued a warning to alert programmers about the inexhaustive patterns. The issue was confirmed and subsequently fixed by the `scalac` developers, with the fix integrated into version 3.6.0 and later.

**Completeness bug in `javac`:** Pattern matching has been recently added to JDK 22 [Gosling et al. 2024]. Java programmers can encode algebraic data types through the combination of interfaces and record classes. Pattern-matching against ADTs in Java are then expressed through an enhanced version of the otherwise long-standing `switch` construct.

Figure 1b illustrates a Java program [1], where we define two ADTs. The first ADT called `A` consists of two constructors: `CC_A` and `CC_B` (lines 1–3), where none of them takes any parameters. The second ADT called `B` is implemented by a single constructor `CC_C` (line 6), which receives two parameters, both of type `A`. The code defines and initializes a variable of type `B`, and then performs pattern matching on it using three cases (lines 8–14). The are four possible values of type `B`:

- `CC_C(CC_A(), CC_A())`: It is covered by the first case in the `switch` expression (line 10).
- `CC_C(CC_A(), CC_B())`: It is covered by the third case in the `switch` expression (line 13).
- `CC_C(CC_B(), CC_A())`: It is covered by the second case of the `switch` expression (line 12).
- `CC_C(CC_B(), CC_B())`: It is covered by the second case of the `switch` expression (line 12).

Since all possible values of type `B` are covered by the `switch` expression, the `switch` expression is *exhaustive*. However, a bug in the implementation of `javac` causes the compiler to mistakenly flag it as inexhaustive. Consider replacing the second case (line 12) with the commented-out code (line 11). Since we are replacing a wildcard pattern (corresponding to the second parameter of `CC_C`) with a specific sub-pattern (`CC_B()`), we are not making the `switch` expression cover anything more. Interestingly, this commented-out modification changes the output of the compiler, as it no longer claims that the `switch` statement is inexhaustive, even though it covers less or equally as many cases as before. This completeness bug was found in JDK 22 and fixed in JDK 24 onwards.

**Challenges:** Identifying completeness and soundness bugs in PMC analyzers, such as the ones presented above, presents several challenges. First, these bugs arise from complex interactions of pattern-matching features. For example, triggering the `scalac` bug requires generating a GADT, while the `javac` bug depends on constructing a `switch` expression with deeply-nested patterns arranged in a specific order. Existing random program generation techniques are not designed to produce programs that exercise pattern-matching features. Furthermore, the search space of patterns

[1] For readability, we use _ to denote wildcard patterns, even though this is not valid Java syntax. In actual Java code, a pattern like `CC_A(_)` would be written as `CC_A(var x)`

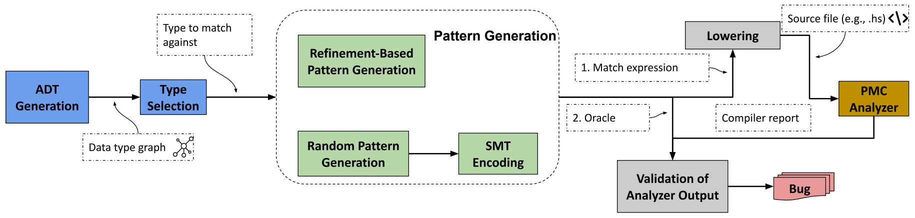


Fig. 2. The high-level overview of our approach for finding bugs in pattern-match coverage (PMC) analyzers.

grows exponentially with the structure of ADTs, making it difficult to identify the specific pattern combinations that trigger bugs. For example, as demonstrated in Figure 1b with the commented-out code, not all pattern combinations expose issues.

Second, even with a generator for pattern matching, we still need a test oracle [Weyuker 1982] to determine the expected behavior of PMC analyzers. For example, we need to automatically figure out that the `switch` expression in Figure 1b is exhaustive, and therefore, the expected behavior of `javac` is to accept the program. Doing so by building a reference implementation from scratch is impractical, as PMC analyzers are complex to implement [Graf et al. 2020], especially in the presence of features such as GADTs [Garrigue and Le Normand 2017]. Differential testing [McKeeman 1998] across languages is also problematic, as one language may support pattern-matching features that are not available in the other language, thus limiting generalizability.

## 3 Approach

Figure 2 outlines our approach for finding completeness and soundness issues in PMC analyzers, which we illustrate using the Scala program of Figure 1a.

- **Step 1. ADT Generation (Section 3.2.1):** The process begins with the generation of random algebraic data type declarations (ADTs) for pattern matching. These ADTs are encoded into a graph representation called *data type graph* (Section 3.2.2), which captures the type constraints imposed by constructors, particularly in the presence of generalized algebraic data types (GADTs). For example, the output of this step is the ADTs shown on lines 1–4 (Figure 1a).
- **Step 2. Type Selection (Section 3.3.1):** Once the ADTs are generated, our approach randomly selects and constructs a type $t$ from the pool of generated ADTs. This type is used to construct patterns that match against it. For example, for the program of Figure 1a, the approach randomly constructs type `A<Int>` (line 6).
- **Step 3. Pattern Generation (Section 3.3):** This step systematically generates diverse patterns that cover values of type $t$. The output of this step consists of two parts: (1) a pattern-matching expression that contains all generated patterns (Figure 1a, lines 8–11), and (2) a test oracle that describes the expected behavior of the PMC analyzer under test. For example, applying this step ultimately yields the Scala program in Figure 1a, where the `match` expression is known to be *inexhaustive*.

**Pattern generation strategies:** For *Step 3*, we introduce two distinct strategies for pattern generation: *refinement-based pattern generation (RefPG)*, and *random program generation (RngPG)*. RefPG produces new patterns by iteratively splitting a general pattern into multiple, more specific sub-patterns. The process ensures that these sub-patterns cover the *exact same* set of values as the original pattern. In contrast, RngPG considers all possible patterns derived from each constructor of type $t$ and combine a sample of these patterns in a random manner.

**Establishing the test oracle:** Our work focuses on defining a test oracle to verify the *exhaustiveness* of generated patterns. In RefPG, the test oracle is implicit, as patterns are constructed to be exhaustive or inexhaustive *by design*. However, in RngPG, exhaustiveness is not known during

$$
\begin{array}{rcl}
\langle P \in \mathit{Program} \rangle & ::= & \overline{d}; \overline{e} \\
\langle d \in \mathit{DataType} \rangle & ::= & \texttt{datatype}\ \mathcal{T}\texttt{<}\overline{\phi}\texttt{>} = \overline{k} \\
\langle e \in \mathit{Expr} \rangle & ::= & c \mid \texttt{var}\ x : t = e \\
 & \mid & \texttt{match}\ e\ \texttt{with}\ \overline{p \rightarrow e} \\
\langle p \in \mathit{Pattern} \rangle & ::= & c \mid \_ : t \mid C \mid C\overline{p} \\
\langle k \in \mathit{Constructor} \rangle & ::= & C : \overline{t} \rightarrow t \\
\langle C \in \mathit{ConstructorName} \rangle & ::= & \textit{the set of constructor names} \\
\langle x \in \mathit{VariableName} \rangle & ::= & \textit{the set of variable names}
\end{array}
$$

(a) Syntax

$$
\begin{array}{rcl}
\langle t \in \mathit{Type} \rangle & ::= & \phi \mid \top \mid \bot \\
 & & \texttt{Bool} \mid \texttt{Int} \mid \texttt{Char} \\
 & & \mathcal{T}\texttt{<}\overline{\phi}\texttt{>} \mid (\mathcal{T}\texttt{<}\overline{\phi}\texttt{>})\texttt{<}\overline{t}\texttt{>} \\
\langle \phi \in \mathit{TypeVariable} \rangle & ::= & \textit{the set of type variables} \\
\langle \mathcal{T} \in \mathit{TypeName} \rangle & ::= & \textit{the set of type names}
\end{array}
$$

(b) Types

Fig. 3. The syntax and the types of IR$_p$.

generation. To address this, patterns are encoded as SMT formulas, and an automated theorem prover determines their exhaustiveness based on the given constraints (*SMT encoding*).

**IR:** Our approach promotes generalizability. The generated pattern-matching expressions are written in an IR (Section 3.1). Each test program is lowered into a program in the target language under test (e.g., Scala, Haskell) and given as input to the corresponding PMC analyzer. Finally, our approach validates the analyzer output against the test oracle. If the compiler output is incompatible with the oracle (e.g., the analyzer reports inexhaustiveness while the oracle confirms exhaustiveness), our approach reports a potential bug in the PMC analyzer. In the subsequent sections, we describe each step our approach in detail.

## 3.1 Preliminary Definitions

First, we present some preliminary definitions and terminology used throughout the paper.

**IR:** Figure 3 presents the syntax and the type system of IR$_p$, a minimal intermediate representation designed to support ADTs and pattern matching. We use IR$_p$ to illustrate the fundamental concepts of our approach. Notably, IR$_p$ is intentionally minimal so that it has a direct correspondence with any programming language supporting ML-like pattern matching. The language supports GADTs, mutually recursive data types, and nested patterns. However, it does not offer support for more advanced features, such as guarded patterns, pattern synonyms, non-linear patterns, or strict types. In what follows, the notation $\overline{e}$ represents an ordered list of elements.

A program in IR$_p$ is a sequence of data type declarations followed by a sequence of expressions. Each data type declaration defines a new type and includes (1) a name $\mathcal{T}$, (2) a list of type variables $\overline{\phi}$ (if the type is polymorphic), and (3) a sequence of constructor definitions. A constructor, in turn, has a name $C$, takes a list of formal parameter types $\overline{t}$, and yields a return type that corresponds to the enclosing data type. Notably, the IR$_p$ supports generalized algebraic data types (GADTs), as each constructor can constrain the type parameters of the data type by specifying its own type arguments. The grammar of IR$_p$ also supports recursive data types.

IR$_p$ includes three types of expressions: constants, variable declarations, and pattern-matching expressions. A pattern-matching expression attempts to match a list of patterns $\overline{p}$ against a given expression $e$. A pattern can be a constant $c$, a wildcard ($\_ : t$) that matches *every* value of type $t$, a constructor name $C$, or a constructor name $C$ followed by list of sub-patterns $\overline{p}$.

The types in IR$_p$ include standard types (e.g., `Int`, top, and bottom types), type variables ($\phi$), and type constructors of the form $\mathcal{T}\texttt{<}\overline{\phi}\texttt{>}$, which are derived from (polymorphic) data type declarations. If the list of type parameters $\overline{\phi}$ is empty, the type is not considered polymorphic. Additionally, type constructors can be applied to a given list of types, denoted as $(\mathcal{T}\texttt{<}\overline{\phi}\texttt{>})\overline{t}$, where $\overline{t}$ represents the applied type arguments. For notational convenience, we use the shortcuts $\mathcal{T}$ for $\mathcal{T}\texttt{<}\emptyset\texttt{>}$, and $\mathcal{T}\texttt{<}\overline{t}\texttt{>}$

```
1  datatype A<T> =
2    CC_A → A<Int>
3    CC_B A<T> → A<Int>
4    CC_C → A<Char>
5
6  val x: A<Char> = ...
7  match x with
8    CC_C → 1
```

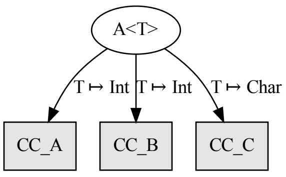


Fig. 4. An example data type declaration and its corresponding data type graph.

for $(\mathcal{T}\texttt{<}\overline{\phi}\texttt{>})\overline{t}$. In the example IR$_p$ programs shown in the following figures, we may use parentheses for nested patterns and represent the wildcard pattern as _ instead of _ : $t$ for better readability.
**Auxiliary definitions:** We use the auxiliary function *constructors* : *DataType* $\longrightarrow \mathcal{P}(\textit{Constuctors})$ that, given a data type definition $d$, returns the set of constructors that are part of $d$. Similarly, we define function *parameters* : *Constructor* $\longrightarrow \mathcal{P}(\textit{Type})$ that gives the set of formal parameter types of a given constructor.

Our language IR$_p$ supports the use of the type substitution operation. A type substitution $\sigma \in \Sigma = [\phi \mapsto t]$ is a mapping that replaces every occurrence of a type variable $\phi$ with a specific type $t$. The application of a type substitution $\sigma$ to type $t$ is given by $\sigma t$. For two type substitutions $\sigma_1, \sigma_2 \in \Sigma$, we have the subsumption relation $\sigma_1 \sqsubseteq \sigma_2$, which means that $\sigma_1$ is at least as general as $\sigma_2$: every type variable in $\sigma_1$ is mapped to exactly the *same* type as in $\sigma_2$. The empty substitution $\epsilon$ is always subsumed by any other substitution ($\forall \sigma \in \Sigma . \epsilon \sqsubseteq \sigma$).

We define the function *decompose* : *Type* $\longrightarrow \Sigma \times \textit{Type}$, which given a type $t$, returns a pair $\langle \sigma, t' \rangle$, where $\sigma$ is a type substitution and $t'$ a type such that applying $\sigma$ to $t'$ reconstructs the original type $t$ (i.e., $t = \sigma t'$). In essence, if the input type $t$ is a polymorphic instance (e.g., `List<Int>`), *decompose* returns the type constructor of $t$ (e.g., `List<T>`) along with the substitution that instantiates it (e.g., $[T \mapsto \texttt{Int}]$). If $t$ is non-polymorphic, the function returns an empty substitution $\epsilon$ and $t$ itself.

Finally, when we say that a pattern $p \in$ *Pattern covers* a set of values $V$ denoted as $[\![p]\!] = V$, it means that if an expression evaluates to any value $v \in V$, the pattern $p$ successfully matches $v$.
**Example:** An example data type declaration is shown in Figure 4. Here, the code defines a GADT named `A` that involves three constructors. The first two constructors return a value of type `A` instantiated by type `Int` (lines 2–3), while the last constructor returns a value of type `A<Char>` (line 4). The `match` expression contains a single pattern (`CC_C`) that matches against a value of type `A<Char>` (lines 6–8). The `match` expression is exhaustive, as the missing constructors `CC_A` and `CC_B` yield values of type `A<Int>`, and not `A<Char>`.

## 3.2 Random Generation of ADTs and Data Type Graph

*3.2.1 Random Generation of ADTs.* As shown in Figure 2, our approach begins with ADT generation. This is a key element because it provides us with the types that we later match against (Section 3.3). Generating our own ADTs is straightforward: we create data types randomly. Our approach employs a configuration $c$ to steer the data types into a direction where more complex types are possible. Among other things, this configuration $c$ controls (1) the number of the generated data types per program, (2) the number of constructors per data type, (3) the number of parameters each constructor takes, (4) the balance between polymorphic ADTs and GADTs, or (5) whether the constructor's parameter is itself a data type, enabling nested patterns. All generated ADTs conform to the grammar of IR$_p$ (Figure 3) under the specified configuration $c$. Any well-formed ADT from IR$_p$, including GADTs, mutually recursive types, and uses of built-in types (`Int`, `Bool`), can be produced by our random generator.

*3.2.2 Data Type Graph.* In the presence of GADTs, there are type constraints that influence which constructors can produce values of specific types. These constraints arise because each constructor can restrict how the type parameters of the data type are instantiated (Figure 4). To precisely capture the type constraints of GADTs, we represent each generated data type declaration (Section 3.2.1) through a graph called *data type graph*, which encodes the constructors of a data type along with the type constraints imposed by each constructor. This graph representation is inspired by similar approaches used to model type constraints in previous work [Koppel et al. 2022; Sotiropoulos et al. 2024]. However, unlike prior work, which primarily focuses on capturing the dependencies of polymorphic APIs in software libraries, our approach specifically models type constraints within data types. Overall, data type graphs serve two main purposes (formally defined later):

- *Constructor lookup for a given type.* The graph allows us to easily identify constructors associated with a specific type $t$. Specifically, given a type $t$, we query the graph to efficiently determine which constructors produce values of $t$ by considering only those whose type constraints are compatible with the constraints imposed by $t$.
- *Identification of inhabited types.* The graph helps determine which specific types can be used in pattern-matching expressions (see Section 3.3.1) without yielding empty matches. By leveraging the graph structure, we can construct valid type instantiations that correspond to existing constructors. This prevents us from considering types known to have no constructors, therefore reducing unnecessary computation.

Formally, we define a data type graph as a directed graph $G = (N, E)$. A node $n \in N$ is either a type $t \in \mathit{Type}$ or a constructor $k \in \mathit{Constructor}$. The edges in the graph are labelled as follows: $E \subseteq N \times N \times \Sigma$, where $\Sigma$ is the set of substitutions that impose constraints on the type parameters of the data type. An edge $t \xrightarrow{\sigma} k$ indicates that the constructor $k$ yields a value of type $t$ under the constraint $\sigma \in \Sigma$. The graph is straightforwardly built on-the-fly while randomly generating ADTs as discussed in Section 3.2.1. Specifically, given a program $P \in \mathit{Program}$ written in $\mathrm{IR}_p$:

- For each data type definition $d$ in $P$, iterate over its constructors given by $k \in \mathit{Constructors(d)}$ and add $k$ to the graph. Then, examine the return type $t$ of $k$.
- Decompose the return type $t$ of the constructor $k$ as $\mathit{decompose}(t) = \langle \sigma, t' \rangle$, where $\sigma$ represents the type substitution and $t'$ the base type before applying $\sigma$. Then, add edge $t' \xrightarrow{\sigma} k$ to the graph.

**Example:** Figure 4 presents a program that defines an ADT and its data type graph. In the graph, the type `A<T>` is represented by a circle node, which is connected to its constructors (represented as box nodes). Each edge connecting `A<T>` to a constructor is labeled according to the type substitution that represents the constraints imposed by the constructor on the type parameter of `A<T>`. Specifically, the edges for constructors `CC_A` and `CC_B` are labeled with their respective type constraints (i.e., $[\mathtt{T} \mapsto \mathtt{Int}]$). On the other hand, the constructor `CC_C` constrains `A<T>` with the constraint $[\mathtt{T} \mapsto \mathtt{Char}]$.
**Operations:** We next define two operations on data type graphs.

*Definition 3.1 (Constructor lookup).* Let $\mathit{lookup} : G \times \mathit{Type} \longrightarrow \mathcal{P}(\mathit{Constructor})$ be a function that, given a data type graph $G$ and a type $t$, returns all constructors that produce values of type $t$. It is formally defined as:

$$\mathit{lookup}(G, t) = \{k | t' \xrightarrow{\sigma'} k \in G, \mathit{decompose}(t) = \langle \sigma, t' \rangle, \sigma' \sqsubseteq \sigma\}$$

The function *lookup* takes a type $t$ and decomposes it into $\langle \sigma, t' \rangle$. Then, it examines all edges that originate from node $t'$ denoted as $t' \xrightarrow{\sigma'} k$, and checks whether the type substitution $\sigma'$ associated with the edge (imposed by the constructor $k$) is compatible with the substitution imposed by the given type $t$. If compatibility holds, the constructor $k$ is considered to yield values of type $t$.

For example, consider again the data type graph $G$ shown in Figure 4. Applying $lookup(G,$ `A<Char>`$)$ results in {`CC_C`}. This follows from the decomposition of `A<Char>` into $\langle\sigma,$ `A<T>`$\rangle$, where $\sigma = [$`T` $\mapsto$ `Char`$]$. Since the edge `A<T>` $\xrightarrow{\sigma}$ `CC_C` exists in the graph $G$, the constructor `CC_C` is added to the resulting set. On the other hand, the other two constructors are not selected because their incoming edges are associated with a type substitution $\sigma' = [$`T` $\mapsto$ `Int`$]$, which is incompatible with $\sigma$ ($\sigma' \not\sqsubseteq \sigma$), that is, $\sigma'($`T`$) \neq \sigma($`T`$)$.

*Definition 3.2 (Inhabited types).* We define the function $types : G \longrightarrow \mathcal{P}(Type)$, which takes a data type graph $G$ and returns the set of *inhabited* types, that is, types for which there exists at least one constructor that produces values of those types. It is formally defined as follows, where the function *leafs* returns all leaf nodes in a given graph:

$$types(G) = \{\sigma t | t \xrightarrow{\sigma} k \in G, k \in leafs(G)\}$$

Back to the example of Figure 4, applying function *types* to the graph yields the set of inhabited types {`A<Int>`, `A<Char>`}.

**Remark on inhabited types:** Definitions 3.1 and 3.2 are intentionally relaxed: Definition 3.1 treats a constructor as inhabiting a given type if it produces a value of that type. In turn, Definition 3.2 considers a type inhabited if it has at least one such constructor. Importantly, these definitions do not require the constructor's arguments to be inhabited. To illustrate this, consider the following code:

```
1  datatype A<T> =
2    CC_A → A<Int>
3  datatype B<T> =
4    CC_B A<Char> → A<T>
5  var x: B<Int> = ...
6  match x with
7    CC_B _ → 1
```

We match against the type `B<Int>` (line 6), which has a single constructor `CC_B` (line 4). However, this constructor requires an argument of type `A<Char>`, which is uninhabited. As a result, it is not possible to create a valid value of type `B<Int>`.

While the type `B<Int>` is technically uninhabited in practice, our definitions still treat them as inhabited. Why? One might argue that removing the pattern `CC_B _` (line 7) is safe, since no concrete value can match it. However, many languages (including those we evaluate in Section 5) support bottom values, which are special values that inhabit all types, such as `null` in Java and Scala or `undefined` in Haskell. As a result, the type `B<Int>` can be inhabited by a value like `CC_B(null)`, which in turn matches the pattern `CC_B _`. Removing the pattern would therefore cause the PMC analyzer in those languages to (correctly) report a missing case.

In contrast, languages without bottom values, especially those with strict types and strict constructor fields (e.g., using the `!` annotations in Haskell or the `-XStrictData` option in the Glasgow Haskell Compiler), would require a stricter notion of Definitions 3.1 and 3.2. In such languages, a type is inhabited only if all of its constructor arguments are also inhabited. Our approach does *not* currently assume and exercise these stricter semantics, but it could be extended to accommodate them (see Section 5 for further discussion).

## 3.3 Generation of Patterns

Having generated a set of ADTs and the corresponding data type graph (Section 3.2), the next step is to generate interesting and diverse pattern-matching expressions derived from these ADTs.

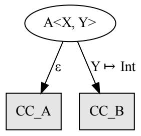
*3.3.1 Selecting Type to Match Against.* Before proceeding with the step of pattern generation, our approach first selects a type to match against (*type selection*, Figure 2). Selecting a good type to match against is crucial, because if the selected type does not have any constructors, it is impossible to generate meaningful patterns. For this reason, we *intentionally* exclude cases involving empty matches, as we consider them less important for our testing goals. While we could relax this restriction to allow matches over types with no constructors, such cases typically result in trivially exhaustive matches with no real pattern-matching logic.

To disregard empty matches, we employ Definition 3.2, which allows us to match against only inhabited types. Specifically, our approach applies the function *types* to the data type graph $G$ (Definition 3.2), and obtains the set of inhabited types $T$. In turn, we randomly select a type $t$ from $T$. Since there might be constructors that do not fully constrain the type parameters of a data type (especially if the data type is not a GADT), the selected type $t$ might not be fully instantiated. To address this, we proceed as follows:

- **Step 1**: Decompose the selected type $t$ as $\mathit{decompose}(t) = \langle \sigma_1, t' \rangle$.
- **Step 2**: Instantiate unconstrained type parameters: For each type parameter in type $t'$ that is *not* already mapped by substitution $\sigma_1$, we generate a new substitution $\sigma_2$ by assigning it a *random* type from the typing context, without enforcing any constraints, such as requiring the selected type to be inhabited. The typing context includes standard built-in types in $\text{IR}_p$, such as `Bool`, `Int`, `Char` (Figure 3), and all data types generated in the program. This typing context therefore includes both regular types and type constructors. If a type constructor is randomly chosen from the context, it is recursively instantiated with random types. To avoid infinite recursion (e.g., `A<A<A<...>>>`), we impose a bound on type nesting depth. Once this bound is reached, only the non-polymorphic types in the context are considered for instantiation (e.g., `Int`).
- **Step 3**: Compute the final type to match against: The selected type to match against is then given by applying the type constructor $t'$ to the combined substitution, that is, $(\sigma_1 \cup \sigma_2)t'$.

This procedure ensures that we always select a *fully* instantiated type, even when some of its type parameters are unconstrained by its constructors.
**Example:** Assume the following ADT and its corresponding data type graph $G$:

```
datatype A<X, Y> =
  CC_A → A<X, Y>
  CC_B → A<X, Int>
```



In this scenario, Definition 3.2 gives us two inhabited types given by $\mathit{types}(G) = \{$`A<X, Y>`, `A<X, Int>`$\}$. Suppose the type $t$ = `A<X, Int>` is selected for pattern matching. We observe that $t$ is not fully instantiated, as the type parameter `X` remains unconstrained by the corresponding constructor `CC_B`. Therefore, we proceed with the three steps described above:

- **Step 1**: We decompose $t$ to get its partial substitution $\sigma_1$ and its type constructor $t'$. That is, $\mathit{decompose}(t) = \langle [Y \mapsto$ `Int`$, t' \rangle$, where $t'$ = `A<X, Y>`.
- **Step 2**: We instantiate the type parameter `X`, which is not constrained by $\sigma_1$ using a random type from the context $\{$`Bool`, `Int`, `Char`, `A<X, Y>`$\}$. Assuming that the type `Bool` is randomly selected, this yields the substitution $\sigma_2 = [X \mapsto$ `Bool`$]$.

- **Step 3**: We apply the type constructor $t' =$ `A<X, Y>` to the combined substitution $\sigma_1 \cup \sigma_2$ and obtain a fully instantiated type, which is guaranteed to be inhabited that is, $[X \mapsto \texttt{Bool}, Y \mapsto \texttt{Int}]\texttt{A<X, Y>} = \texttt{A<Bool, Int>}$.

Beyond selecting a type to match against, a pattern-matching expression requires a set of patterns that attempt to cover values of this type. To do so, our approach proceeds with *pattern generation* (Figure 2), where it leverages two different generation strategies namely, *refinement-based pattern generation (RefPG)* and *random pattern generation (RngPG).*

*3.3.2 Refinement-Based Pattern Generation.* The high-level idea of RefPG is that the generation process starts with a general pattern $p$ (e.g., the wildcard pattern) and iteratively generates new patterns by decomposing the original pattern into a list of complex, more specific sub-patterns. These sub-patterns together *fully* cover the same values as $p$, ensuring that any value covered by the original pattern $p$ is also covered by these sub-patterns. To formalize the idea, we introduce the concept of *pattern refinement.*

*Definition 3.3 (Pattern refinement).* Let $p \in \mathit{Pattern}$ be a pattern, and let $V$ be a set of values covered by $p$, denoted as $[\![p]\!] = V$. Now, consider $A$, an ordered list of patterns $A = \langle p_1, p_2, \ldots, p_n \rangle$. We say that $A$ refines pattern $p$ written as $p \Rightarrow A$, iff

- the union of all refined patterns $A$ *fully* covers $V$: $\bigcup_{i=1}^{n} [\![p_i]\!] = V$
- the patterns in $A$ are *mutually exclusive* (no overlap between patterns in $A$): $[\![p_i]\!] \cap [\![p_j]\!] = \emptyset$ for $1 \leq i < j \leq n$.

The definition above ensures two key properties. First, the refined patterns $A$ are a *safe* replacement of $p$, meaning that the patterns in $A$ do not miss any values covered by $p$. The second property is that the patterns in $A$ are *disjoint*. This means that no pattern in $A$ is redundant or unreachable, that is, patterns later in the list capture new values not covered by earlier patterns.

**Realization of pattern refinement:** Now, we explain how we realize the concept of pattern refinement in the context of our IR$_p$. We provide a concrete implementation of the function *refine* that takes a pattern $p \in \mathit{Pattern}$ and a data type graph $G$, and produces a list of patterns $A$ that refines $p$, that is $p \Rightarrow A$, as follows.

$$\begin{aligned}
\mathit{refine}(p, G) &= \langle p \rangle && \text{if } p = C \vee p = c \\
\mathit{refine}(\_ : t, G) &= \langle C\ \overline{p} \mid (C : \overline{t_1} \rightarrow t_2) \in \mathit{lookup}(G, t), \overline{p} = \overline{\_ : t_1} \rangle \\
\mathit{refine}(C\ \overline{p}, G) &= \langle C\ \overline{p'} \mid \overline{p'} \in \mathit{refine}(\overline{p}, G) \rangle
\end{aligned}$$

The implementation of *refine* works as follows. If the given pattern is a constant or a constructor with no parameters (sub-patterns), refinement is not possible. In this case, *refine* returns a singleton list that contains the pattern given as input. If the pattern to be refined is a wildcard of type $t$, then *refine* retrieves all the available constructors that yield values of type $t$ using the *lookup* function (recall Definition 3.1). For each constructor given by *lookup*, a corresponding pattern is created. If the constructor has parameters, wildcards are used as sub-patterns to ensure that all values generated by the constructor are covered. Finally, if the input pattern is $C\ \overline{p}$, we recursively apply *refine* to the sub-pattern $\overline{p}$.

Theorem 3.4 (Correctness of refinement). *For any pattern $p \in \mathit{Pattern}$ and data type graph $G$, $\mathit{refine}(p, G) = A$ such that $p \Rightarrow A$. This means that the list $A$ returned by refine satisfy the two properties stated in Definition 3.3.*

Proof. The proof follows straightforwardly by induction on the structure of patterns in IR$_p$. □

```
datatype A<T> =
  CC_A → A<Int>
  CC_B A<T> → A<Int>
  CC_C → A<Char>


let x: A<Int> = ...
match x with
  _ → 1
```

(a) Initial pattern

```
datatype A<T> =
  CC_A → A<Int>
  CC_B A<T> → A<Int>
  CC_C → A<Char>

let x: A<Int> = ...
match x with
  CC_A → 1
  CC_B(_) → 2
```

(b) Refined patterns

```
datatype A<T> =
  CC_A → A<Int>
  CC_B A<T> → A<Int>
  CC_C → A<Char>

let x: A<Int> = ...
match x with
  CC_A → 1
  CC_B(CC_A) → 2
  CC_B(CC_B(_)) → 3
  CC_B(CC_C) → 4
```

(c) Final refined patterns

```
datatype A<T> =
  CC_A → A<Int>
  CC_B A<T> → A<Int>
  CC_C → A<Char>

let x: A<Int> = ...
match x with
  CC_A → 1
  CC_B(CC_B(_)) → 3
```

(d) Inexhaustive patterns

Fig. 5. The steps of refinement-based pattern generation. The process starts with the most general pattern (wildcard) that covers all possible values of the type we match against (Figure 5a). The general patterns are iteratively refined to yield to multiple, more specific patterns. Figure 5c gives the set of the generated patterns after two iterations. Figure 5d presents inexhaustive patterns after removing patterns from Figure 5c.

**Algorithm 1:** Algorithm of refinement-based pattern generation.

```
1  fun refine_gen(G, t) =
2      $P_{old} \leftarrow \emptyset$
3      $P_{new} \leftarrow \langle \_ : t \rangle$
4      while $P_{old} \neq P_{new} \land$ not stop_gen() do
5          $P_{old} \leftarrow P_{new}$
6          for $p \in P_{old}$ do
7              if flip_coin() then continue
8              $A \leftarrow refine(p, G)$
9              $P_{new} \leftarrow$ replace $p$ with $A$ in $P_{new}$
10     return $P_{new}$
```

**Algorithm of pattern generation based on pattern-refinement:** After establishing the concept of pattern refinement, we now present the algorithm for generating patterns based on this notion. Refinement-based pattern generation (RefPG) is summarized in Algorithm 1, which we will illustrate through an example.

Consider again the ADTs shown in Figure 4 (lines 1–4). Assume that we want to generate patterns that match values of type $t$ = `A<Int>`. Algorithm 1 takes as input (1) the type we match against ($t$), and (2) the data type graph $G$, which encodes the ADTs of the program (Algorithm 1, line 1). The first step of RefPG is to produce the most general pattern that captures *every possible value* of the given type $t$ (line 3). This stands for the wildcard pattern as shown in Figure 5a. Every pattern generated by the algorithm (including the wildcard pattern, line 3) is stored in a list called $P_{new}$.

The algorithm iterates over each generated pattern in the list $p \in P_{new}$, and decides whether to refine it further or keep it as is (lines 6–7). This decision is made through a random procedure, implemented in the function *flip_coin()*. If *flip_coin()* decides to refine $p$, the algorithm executes lines 8 and 9, where it applies the *refine* function to $p$ and replaces it with the result of *refine*. For example, the wildcard pattern of Figure 5a is refined and replaced with two disjoint patterns, as shown in Figure 5b. The refinement process continues until either no further refinement is possible ($P_{old} = P_{new}$) or *stop_gen()* determines that the pattern generation should stop based on user-provided criteria (e.g., maximum number of generated patterns).

Now, assume the algorithm performs one more refinement step (iteration). In this step, it attempts to refine the patterns produced in the previous iteration, namely `CC_A`, `CC_B _` (Figure 5b). The pattern `CC_A` cannot be further refined because it corresponds to a constructor that takes no parameters. However, the pattern `CC_B _` can be further refined, and it is replaced by three additional patterns, as shown in Figure 5c. If *stop_gen()* decides to terminate at this point, the final set of generated patterns is as illustrated in Figure 5c.

*3.3.3 Random Pattern Generation.* Beyond refinement-based pattern generation (Section 3.3.2), we also introduce *random pattern generation (RngPG)*. The core idea of RngPG is straightforward: given a type $t$ and a data type graph $G$, RngPG generates patterns that cover values of $t$ and combine them in a random fashion, though not necessarily exhaustively. Unlike RefPG, which follows a structured refinement process to produce disjoint and exhaustive patterns, RngPG does not enforce any particular structure. Instead, it *freely* combines patterns in an arbitrary manner. As a result, RngPG can generate pattern sets with highly varied and unpredictable compositions, such as those shown in Figure 1b.

**Algorithm:** The algorithm of RngPG is straightforward: for each constructor $k \in$ *lookup(G, t)* (Definition 3.1), RngPG generates a set $A_k$ that contains *all* possible patterns derived from the constructor $k$, up to a specified depth. It then *randomly* selects a subset from each $A_k$ and combines them into a pattern-matching expression.

**Example:** To illustrate RngPG, consider again the ADTs from Figure 4. Suppose that we aim to generate patterns that cover values of type `A<Int>`. According to Figure 4, there are two constructors that produce values of this type: `CC_A` and `CC_B`. For each constructor, RngPG constructs a set that includes all possible patterns derived from it up to a specified depth. For the sake of the example, we limit the depth to two.

- The set for constructor `CC_A` is $A_a$ = {`CC_A`}.
- The set for constructor `CC_B` is $A_b$ = {`CC_B(_)`, `CC_B(CC_A)`, `CC_B(CC_B(_))`, `CC_B(CC_C)`}.

The last step of RngPG is to take a *random* sample of each set $A_k$ and combine these samples into a pattern-matching expression. Suppose RngPG randomly picks the following samples: {`CC_A`} from $A_a$, and {`CC_B(CC_B(_))`} from $A_b$. These selected patterns are then combined to form the program shown in Figure 5d.

## 3.4 Test Oracle

Having presented our approach for producing test programs that contain pattern-matching features via the generation of random ADTs (Section 3.2) and `match` expressions (Section 3.3), the next step is to establish a test oracle that reliably predicts the expected behavior of the PMC analyzer under test. A PMC analyzer statically verifies two properties: *exhaustiveness* and *non-redundancy*. Exhaustiveness ensures that a pattern-matching expression covers all possible values of the matched type, preventing runtime errors due to missing cases. Non-redundancy guarantees that no pattern is unnecessary or shadowed by a preceding one, eliminating unreachable cases. In this work, our focus is to establish a test oracle associated with the *exhaustiveness* property, that is a reliable mechanism to automatically determine whether a `match` expression generated by RefPG or RngPG is truly exhaustive or not. We focus on exhaustiveness checking because bugs there can pose a more severe risk. Our test oracle can be easily further extended to redundancy checking.

We formally define exhaustiveness in pattern matching. In the following, we use function $type : Expr \longrightarrow Type$ that gives the type $t$ of an expression $e$.

*Definition 3.5 (Exhuastiveness).* Let a pattern-matching expression $M = \langle e, \overline{p} \rangle$ that attempts to match a list of patterns $\overline{p}$ against an expression $e$. Let $[\![t]\!]$ denote the set of all possible values of type $t$. We say that $M$ is *exhaustive* if

$$\forall v \in [\![type(e)]\!],\ \exists p \in \overline{p},\ v \in [\![p]\!]$$

and M is *inexhaustive* if

$$\exists v \in [\![ type(e) ]\!],\ \forall p \in \overline{p},\ v \notin [\![ p ]\!]$$

Based on Definition 3.5, we now illustrate how to build the test oracle for each of the pattern generation strategies introduced in Section 3.3.

*3.4.1 Test Oracle for Refinement-Based Pattern Generation.* The algorithm of RefPG begins the refinement process with the most general pattern (Algorithm 1, line 3; see also Figure 5a), that is the wildcard pattern. Let us now use the symbol $p\_$ to denote this wildcard pattern, which exhaustively captures *all possible values* of the type we match against. Formally, this means: $[\![ t ]\!] = [\![ p\_ ]\!]$. Therefore, a pattern-matching expression $M = \langle e, \overline{p} \rangle$, where $e$ is an expression of type $t$ and $\overline{p}$ is the singleton list $\langle p\_ \rangle$, is always exhaustive.

Theorem 3.6 (Exhaustiveness of refined pattern matching). *Let the most general pattern $p\_$, and a list of patterns $A$ such that $p\_ \Rightarrow A$. The pattern-matching expression $M_A = \langle e, A \rangle$ is exhaustive.*

Proof. By Theorem 3.4, any list of patterns $A = \langle p_1, p_2, \ldots, p_n \rangle$ that refines $p\_$ satisfies the two properties stated in Definition 3.3. This ensures:

$$\bigcup_{i=1}^{n} [\![ p_i ]\!] = [\![ p\_ ]\!].$$

Thus, we have

$$\forall v \in [\![ p\_ ]\!],\ \exists p \in A,\ v \in [\![ p ]\!].$$

Since we know that $p\_$ covers all values of the matched type $t = type(e)$, it follows that $[\![ t ]\!] = [\![ p\_ ]\!]$. Consequently, we know that

$$\forall v \in [\![ t ]\!],\ \exists p \in A,\ v \in [\![ p ]\!].$$

Therefore, for the pattern-matching expression $M_A = \langle e, A \rangle$, every possible value of type $t$ is matched by at least one pattern in $A$. Thus, $M_A$ is exhaustive by construction. □

**Example:** By Theorem 3.6, a pattern-matching expression produced by RefPG is exhaustive *by construction*. For example, the pattern-matching expression in Figure 5c, which is the result of RefPG after two iterations, is exhaustive. This is because all the refined patterns originate from the wildcard pattern shown in Figure 5a, which matches all possible values of the type. If a PMC analyzer reports the program of Figure 5c as inexhaustive, it suggests a *completeness bug* in the PMC analyzer.

**Finding soundness bugs using RefPG:** To validate soundness in a PMC analyzer, we need to construct inexhaustive pattern-matching expressions. How can we generate inexhaustive pattern-matching expressions using RefPG, given that it only produces exhaustive ones? The answer is: we derive inexhaustive pattern-matching expressions by removing *at least one* pattern from the generated set. Because RefPG ensures exhaustiveness by construction, this removal guarantees that the resulting `match` expression is inexhaustive. This allows us to systematically test whether the PMC analyzer correctly detects the missing cases.

Theorem 3.7 (Inexhaustiveness of removed pattern). *Let the most general pattern $p\_$, and a list of patterns $A$ such that $p\_ \Rightarrow A$. Removing any pattern in $A$ yields a new pattern list $\mathbb{A}$. The pattern-matching expression $M_{\mathbb{A}} = \langle e, \mathbb{A} \rangle$ is inexhaustive.*

Proof. By Theorem 3.4, any list of patterns $A = \langle p_1, p_2, \ldots, p_n \rangle$ that refines $p\_$ satisfies the two properties stated in Definition 3.3. Specifically, these properties ensure that the patterns in $A$ are *mutually exclusive* and *exhaustive*, i.e.,

$$[\![p_i]\!] \cap [\![p_j]\!] = \emptyset, \text{ for all } 1 \leq i < j \leq n, \text{ and } \bigcup_{i=1}^{n} [\![p_i]\!] = [\![p\_]\!].$$

Consequently, for any pattern $p \in A$, if a value $v$ belongs to $[\![p]\!]$, then $v$ does not belong to $[\![p']\!]$ for any other pattern $p' \in A \setminus \{p\}$. That is,

$$v \in [\![p]\!] \implies v \notin \bigcup_{p' \in A \setminus \{p\}} [\![p']\!].$$

Now, consider removing a pattern $p$ from $A$, yielding a new pattern list $\mathbb{A} = A \setminus \{p\}$. Under the relaxed notion of inhabited types (Definitions 3.1 and 3.2), which considers types inhabited even when the constructor arguments can only be initialized via bottom values, there must exist some value $v$ in $[\![p\_]\!]$ that is not covered by any pattern in $\mathbb{A}$, i.e.,

$$\exists v \in [\![p\_]\!], \ v \notin \bigcup_{p' \in \mathbb{A}} [\![p']\!].$$

Since $p\_$ is the most general pattern covering all values of type $t$, it follows that $[\![t]\!] = [\![p\_]\!]$. Thus, we conclude that

$$\exists v \in [\![t]\!], \ \forall p' \in \mathbb{A}, \ v \notin [\![p']\!].$$

This establishes that the pattern-matching expression $M_{\mathbb{A}} = \langle e, \mathbb{A} \rangle$, where $\overline{p} = \mathbb{A} = A \setminus \{p\}$, is *inexhaustive*. □

**Example:** By Theorem 3.7, removing any pattern from a `match` expression generated by RefPG leads to an inexhaustive expression. For example, consider the exhaustive `match` expression shown in Figure 5c, which is the outcome of RefPG after two iterations. Removing at least one pattern from this `match` expression, namely `CC_B(CC_A)` and `CC_B(CC_C)`, ensures that the resulting expression in Figure 5d is inexhaustive. If a PMC analyzer reports that this expression is exhaustive, it suggests a *soundness issue* in the analyzer. Notably, the inexhaustive IR$_p$ program of Figure 5d uncovered a soundness bug in `scalac` (see the Scala syntax in Figure 1a).
**Finding bugs in redundancy checking:** Although our focus is on establishing a test oracle for exhaustiveness, the disjointness property of RefPG (Definition 3.3) also enables us to uncover bugs in the redundancy checking functionality of PMC analyzers. Specifically, RefPG guarantees that all patterns in a refined list $A$ are pairwise disjoint, meaning no pattern is redundant *by construction.* As a result, if a PMC analyzer reports redundant patterns in a `match` expression generated by RefPG, it indicates a completeness bug in the analyzer's redundancy checker.

*3.4.2 Test Oracle for Random Pattern Generation.* In Section 3.4.1, we showed that RefPG guarantees both exhaustive and inexhaustive patterns by construction. However, this guarantee does not hold for RngPG. Because of the random nature of RngPG, patterns are generated without a structured process to ensure exhaustiveness. One way to establish a test oracle for patterns generated by RngPG is differential testing [McKeeman 1998] by comparing the results of PMC analyzers across different languages, such as Scala and Java. However, this approach is impractical and not general, since each language may support unique pattern-matching features, not available in others. As a result, mismatches could stem from language differences rather than actual bugs. Developing our own reference PMC analyzer is also impractical as it would be time-consuming and susceptible to its own implementation errors.

```
1 (declare-sort Int_ 0)
2 (declare-sort Char_ 0)
3 (declare-datatypes ((A 1))
4   ((par (T) (
5      (CC_A)
6      (CC_B (CC_B_a (A T)))
7      (CC_C)))))
```

(a) Declarations of types

```
1 (define-funs-rec
2   ((valid ((x (A Int_))) Bool)
3    (valid ((x (A Char_))) Bool))
4   ((or ((_ is (CC_A () (A Int_))) x)
5        ((_ is (CC_B ((A Int_)) (A Int_))) x))
6    ((_ is (CC_C () (A Char_))) x)))
7 )
```

(b) Constraining applicable constructors

```
1 (declare-fun x () (A Char_))
2 (assert ((_ valid 0) x))
3 (assert
4   (not
5     ((_ is (CC_C ()
6         (A Char_))) x)))
7 (check-sat)
```

(c) Pattern encoding

Fig. 6. Steps of SMT encoding for the exhaustiveness of a pattern-matching expression. The encoding begins with the declarations of the algebraic data types used in pattern matching (Figure 6a). Next, we constrain the constructors to their respective return types to ensure valid term generation (Figure 6b). Finally, we encode the pattern-matching expression as an SMT assertion, enabling automated exhaustiveness checks (Figure 6c).

To address the oracle challenge for arbitrary pattern-matching expressions produced by RngPG, we encode the exhaustiveness property as an SMT formula and use an off-the-shelf solver to verify it. If the solver proves exhaustiveness but the PMC analyzer reports the `match` expression as inexhaustive, this reveals a completeness bug in the PMC analyzer. Conversely, if the solver proves inexhaustiveness but the analyzer accepts the `match` expression, it indicates a soundness bug in the PMC analyzer. This approach gives us a *general*, *language-agnostic* test oracle "for free" by leveraging the power of constraint solving.

Our test oracle *assumes* the correctness of the SMT solver, although solvers themselves can have bugs [Mansur et al. 2020; Winterer and Su 2024; Winterer et al. 2020a,b]. In our evaluation (Section 5), we encountered only one such issue where a specific formula caused the solver to crash with a segmentation fault. We reported this bug to the solver developers, who promptly fixed it. We now describe how pattern-matching expressions are encoded into SMT.

*Definition 3.8 (SMT Encoding of the Exhaustiveness of a Pattern-Matching Expression).* Given a pattern-matching expression $M = \langle e, \overline{p} \rangle$ and an SMT formula $\varphi(x)$, where $x$ is the free variable in $\varphi$, we say that $\varphi$ is an *SMT encoding* of the exhaustiveness of $M$ if: $\varphi$ is *unsatisfiable* implies that $M$ is exhaustive, and $\varphi$ is *satisfiable* implies that $M$ is inexhaustive. Formally, this means:

$$\exists x, \varphi(x) \iff \exists v \in [\![ type(e) ]\!],\ \forall p \in \overline{p},\ v \notin [\![ p ]\!].$$

According to Definition 3.8, encoding the exhaustiveness of a pattern-matching expression into an SMT formula requires: (1) constraining the variable $x$ to values within $[\![ type(e) ]\!]$ to ensure it conforms to the matched type, and (2) translating the pattern set $\overline{p}$ into assertions that represent the matching logic. We detail the steps of our SMT encoding through the example of Figure 4.

**Step 1: Declaration of ADTs:** The SMT-LIB language includes a dedicated theory for algebraic data types. This makes encoding ADT declarations from our intermediate representation (IR) into SMT-LIB relatively straightforward. For example, given the data type declarations shown in Figure 4, we can encode them into the SMT-LIB format as illustrated in Figure 6a. First, we declare two sorts representing the basic types `Int` (named `Int_`, line 1), `Char` (named `Char_`, line 2). Then, we declare the polymorphic ADT `A` that takes one type parameter `T` (lines 3 and 4). The data type is initialized by three constructors `CC_A`, `CC_B`, and `CC_C` (lines 5–7), where the constructor `CC_B` takes one parameter named `CC_B_a` whose type is `A<T>` (line 6).

**Step 2: Encoding GADTs:** In SMT-LIB, GADTs are not natively supported. As a result, constructors in SMT-LIB do not constrain the type parameters of their data types. For example, the constructor `CC_A` in Figure 6a (line 5) could be incorrectly used to create a value of type `A<Char>`, even if such a construction would be invalid in the original program shown in Figure 4.

To encode GADTs in SMT-LIB, we explicitly constrain which data type instantiations each constructor is allowed to produce values for. For example, in Figure 6b, we introduce two overloaded

`valid` functions to constrain values of `A<Int>` and `A<Char>`, respectively. The first function, which operates on an input $x$ of type `A<Int>` (line 2), ensures that $x$ can only stem from `CC_A` or `CC_B` (lines 4–5). Similarly, the second `valid` function, which applies to inputs of type `A<Char>`, restricts $x$ to `CC_C` (line 6). This encoding eliminates ill-formed terms at the SMT level, preventing spurious counterexamples when checking the exhaustiveness of pattern-matching expressions (see below).
**Step 3: Encoding patterns:** The final step in the SMT encoding process is to transform the patterns in a given pattern-matching expression into an assertion. Consider the set of patterns $\overline{p}$ in Figure 4, which includes only a single pattern $p$ corresponding to `CC_C`. This pattern attempts to match an expression of type `A<Char>`.

Figure 6c presents the final query we make to the solver. We first introduce a fresh variable $x$ of type `A<Char>` (line 1). Since SMT-LIB does not inherently handle GADTs, we explicitly filter out inapplicable constructors using the `valid` function (Figure 6b; Figure 6c, line 2). This ensures that $x$ can be initialized only by constructors that yield values of type `A<Char>`. To encode the pattern-matching process, we introduce constraints that prevent $x$ from being one of the patterns in $\overline{p}$ that appear in the given `match` expression (lines 3–6 in Figure 6c). In our example, the `match` expression of Figure 4 has a single pattern. Therefore, the constraints on Figure 6c (lines 4–5) ensure that the variable $x$ is *not* `CC_C`. Any value of $x$ satisfying the constraints of Figure 6c represents an *uncovered* case. If such a value exists (the solver returns SAT), it confirms that the pattern-matching expression is inexhaustive; otherwise, the expression is exhaustive.
**Handling the wildcard pattern:** When a `match` expression includes a wildcard pattern, our SMT-LIB encoding expands the assertions to exclude all constructors of the matched type. For example, if the `match` expression in Figure 4 used a wildcard instead of `CC_C` (line 8), the final SMT query would assert: $\neg(x\ \mathit{is}\ \texttt{CC\_A} \lor x\ \mathit{is}\ \texttt{CC\_B} \lor x\ \mathit{is}\ \texttt{CC\_C})$.

## 4 Implementation Details and Discussion

We have implemented the techniques from Figure 2 in a tool called Ikaros, including approximately 10k lines of Rust code. The current implementation generates programs that exercise the PMC analyzers integrated into three languages: Scala, Java, and Haskell. Ikaros offers a minimal command-line interface, where users select the target language and choose a pattern-generation strategy (RefPG or RngPG). Ikaros also comes with a configuration that controls the generation process of ADTs and patterns. This configuration influences the complexity of the generated programs by disabling or enabling certain features (e.g., GADTs) or specifying parameters, such as the maximum pattern depth or maximum number of constructors per data type. Finally, Ikaros employs the Z3 constraint solver [de Moura and Bjørner 2008] to obtain the oracle for RngPG.
**Generalizability:** Ikaros is extensible to any language that supports ML-like pattern matching. Adding support for a new language requires implementing a Rust trait that lowers the in-memory representation (IR) of the generated program into a source file written in the target language. This extension process is straightforward: for example, it took us a couple of hours to add Haskell support, which involved writing approximately 600 lines of Rust code.

When adding a new language, Ikaros also allows implementing language-specific typing features. Beyond types derived from the generated ADTs, Ikaros supports built-in types native to the target language. For example, our implementation includes basic numeric (e.g., `Int`) and string types for all supported languages, while for Haskell and Scala, Ikaros additionally supports tuple types and patterns against tuples.
**Completeness of Ikaros:** Our current implementation is not complete, as the formalism of $\text{IR}_p$ excludes several features along two dimensions that impact the reasoning of PMC analyzers: (1) pattern-matching features and (2) typing features.

*Pattern-matching features.* Ikaros does not yet support some advanced features, such as guarded patterns or pattern synonyms, which are available in all the languages considered in this work. Guarded patterns include arbitrary conditions that PMC analyzers must reason about to determine exhaustiveness and redundancy. Both pattern generation techniques (Section 3.3) can be extended to accommodate this feature. In RefPG, we could randomly associate patterns with arbitrary boolean conditions, ensuring that they still cover values in the program when the condition does not hold. For example, the exhaustive pattern in Figure 5c could be rewritten as:

```
let y: Int = 2
match x with =
  CC_A if y > 1 → 1
  CC_A → 1
  CC_B(CC_A) → 2
  CC_B(CC_B(_)) → 3
  CC_B(CC_C) → 4
```

Our SMT encoding that provides the test oracle for the RngPG method can also be extended to handle guarded patterns. This would involve translating the pattern conditions into logical formulas and using an SMT solver to determine pattern exhaustiveness. Similar SMT encodings have been explored in the domain of refinement types [Vazou et al. 2014, 2017].

*Typing features.* In addition to pattern-matching features, Ikaros does not support many typing features that influence the behavior of PMC analyzers. These include union types and declaration-site variance in Scala, bounded polymorphism in Scala and Java, strict types and constructor fields in Haskell, or existential types across all the studied languages. For example, consider the following Haskell program, which declares an ADT with strict fields via the `!` annotation (line 2).

```
 1  data A b where
 2    CC_A :: !(B Char) → A b
 3    CC_B :: A b
 4  data B b where
 5    CC_C :: B Int
 6
 7  x :: A Int
 8  x = ...
 9  y = case x of
10    CC_A _ → 1
11    CC_B → 2
```

Ikaros does not currently generate ADTs that use the `!` annotation. Even if it did, doing so would compromise our existing test oracles. This is because the `!` annotation enforces strict evaluation, which affects the PMC analyzer's reasoning. In this example, the pattern `CC_A _` is considered unreachable, as it requires an argument of the uninhabited type `B Char`. However, our current oracle would incorrectly treat the pattern as reachable, leading to false positives.

As previously discussed, Ikaros provides an extensible framework for introducing new types and pattern features; we have already extended it for tuples in Scala and Haskell. To support strict annotations, we would need to (1) strengthen Definitions 3.1 and 3.2 to account for constructor arguments, and (2) update the SMT encoding accordingly. We leave this extension as future work.

**Limitations of refinement-based pattern generation:** While RefPG systematically refines and generates new patterns, it comes with certain limitations. One key challenge is the exponential growth of the search space when refining cases into all possible sub-patterns. This can quickly cause the size of the pattern-matching statement to explode. In our evaluation (Section 5.1), we limit the maximum refinement depth.

Another limitation lies in the restrictive structure of the generated patterns. This is because the resulting patterns often exhibit similar structures and properties. For example, RefPG ensures that patterns never overlap and always form a cross-product of all possible combinations at each refinement step. This means that every program generated by RefPG could have been produced by a fully random approach, such as RngPG, but the reverse is not true.

**Limitations of random pattern generation:** The main limitation of RngPG lies is that it is biased toward generating inexhaustive patterns (Section 5.3). This bias occurs because deeper patterns dominate the initial pool of possible patterns per constructor, which makes them more likely to be selected. However, deeply nested patterns also require exponentially more cases to achieve exhaustiveness.

## 5 Evaluation

We evaluate Ikaros based on the following research questions.

**RQ1** Is Ikaros effective in finding completeness and soundness issues in pattern-match coverage analyzers (Section 5.2)?
**RQ2** What are the characteristics of the test programs generated by Ikaros (Section 5.3)?
**RQ3** What is the performance of Ikaros (Section 5.4)?

### 5.1 Experimental Setup

**Targets:** We used Ikaros to validate the correctness of PMC analyzers integrated into three compilers, namely, the compiler of Scala 3 (`scalac`), the compiler of Java (`javac`), and the Glasgow Haskell compiler (`ghc`). Since our practical focus is to discover new, previously unknown bugs, we examined the latest, most stable release of each compiler.

**Opportunistic bug-finding experiment:** Our opportunistic bug-finding experiment ran over nine months, during which we tested compilers and developed Ikaros in parallel. To facilitate testing, we set up six automated `cron` jobs, each targeting a specific compiler (`scalac`, `javac`, or `ghc`) and a specific pattern generation strategy (RefPG or RngPG). For example, one `cron` job executed Ikaros to generate Haskell programs and validate `ghc` using RngPG, while another did the same using RefPG. Each `cron` job ran for 24 hours and was repeated every two days. Note that we did not run Ikaros continuously for nine months, because we were actively developing the tool in parallel. For example, we first developed RefPG and after three months we developed RngPG and its solver-based oracle (Section 3.4).

After each run, we manually analyzed the alerts produced by Ikaros to determine whether they indicated new bugs or were duplicates of previously known issues. To facilitate this process, we developed a test-case reduction method (see below).

**Input reduction:** The programs generated by Ikaros can grow exponentially in size, especially when the structure of the generated ADTs is complex. This poses challenges when bug-triggering programs are too large. Reporting such large programs to developers is impractical and counter-productive [Regehr et al. 2012], as they are difficult for developers to debug and analyze. Existing test-case reducers, such as C-Reduce [Regehr et al. 2012], are ineffective for minimizing ADTs and pattern-matching expressions. The key issue is the lack of an oracle to verify whether a reduced program still triggers the same bug in the PMC analyzer.

To address this, we developed our own test-case reducer tailored for programs with ADTs and pattern-matching expressions. Our reducer iteratively simplifies ADTs by removing constructor parameters, constructors, and entire ADTs. Each modification is propagated to dependent patterns, e.g., removing a constructor parameter also removes patterns referencing that parameter. For example, consider the patterns `CC_A(1)` and `CC_A(_)` in a pattern-matching expression, both

Table 1. (a) Status of the reported bugs in `scalac`, `javac`, and `ghc`, (b) number of bugs that lead to false positives (FP) and false negatives (FN) in exhaustiveness checks, false positives (FP) in redundancy checks, or compilation performance issues (c) bugs revealed by each pattern generation strategy (RefPG or RngPG) during a 12-hour run by Ikaros.

| **Status** | **scalac** | **javac** | **ghc** | **Total** |
|---|---|---|---|---|
| Unconfirmed | 0 | 1 | 0 | 1 |
| Confirmed | 2 | 0 | 0 | 2 |
| Fixed | 10 | 2 | 0 | 12 |
| Won't fix | 0 | 1 | 0 | 1 |
| Total | **12** | **4** | **0** | **16** |

(a)

| **Symptom** | **scalac** | **javac** | **ghc** | **Total** |
|---|---|---|---|---|
| Exhaustiveness FP | 0 | 4 | 0 | 4 |
| Exhaustiveness FN | 7 | 0 | 0 | 7 |
| Redundancy FP | 4 | 0 | 0 | 4 |
| Performance | 1 | 0 | 0 | 1 |

(b)

| **Pat Gen** | **scalac** | **javac** | **ghc** | **Total** |
|---|---|---|---|---|
| RefPG | 36 | 0 | 0 | 36 |
| RngPG | 3,642 | 86 | 0 | 3,728 |

(c)

derived from the constructor `CC_A Int`. If the `Int` parameter is removed from the constructor, these patterns are reduced to a single `CC_A`. Our test-case reducer checks whether after each update, the updated program still triggers the bug. If not, it reverts the modification and proceeds with other modifications. Our test-case reducer leverages *hierarchical delta debugging* to group updates based on their depth in the AST [Misherghi and Su 2006].

During our testing efforts, we employed our test-case reducer to minimize and simplify every bug-triggering program produced by Ikaros. The reducer enabled us to easily identify duplicate bugs reported by Ikaros and, on average, reduced program size by 70%.

**Configuration:** It is important to maintain a balance between bug-finding capability and performance. Producing ADTs and patterns that are too complex can lead to programs with hundreds of patterns, often causing compilers to crash due to out-of-memory errors. To avoid this, we configured Ikaros with controlled limits, including a maximum of 2 ADTs per program, 3 constructors per data type, 2 type variables, and 5 levels of pattern depth in our evaluation. Our preliminary experiments confirmed that these settings do not produce pattern-matching expressions that are overcomplicated.

**Hardware:** All experiments were conducted on a Linux server (Ubuntu 20.04 LTS) with AMD EPYC 7742 64-core CPUs and 256GB RAM.

## 5.2 RQ1: Bug-Finding Results

Table 1a outlines the results from our opportunistic bug-finding experiment. Overall, Ikaros has uncovered 16, previously unknown bugs, of which 12 have been fixed. Most of the bugs were discovered and reported in the PMC analyzer of `scalac` (12), while we also uncovered 4 unique bugs in `javac`'s new PMC analyzer. Despite our efforts, Ikaros did not find any new bugs in `ghc`, which seems to be the most reliable among the studied compilers given the recent improvements of its PMC analyzer [Graf et al. 2020]. The discovery and reporting of 16 bugs is already a strong indication of Ikaros's effectiveness, especially if we consider that PMC analyzers represent only a small portion of the compiler codebase. The high number of detected issues suggests that these components can be challenging to implement correctly given their relatively small size.

A factor that delayed the discovery of new bugs was that some bugs obscured others. In many cases, we had to wait for developers to fix previously reported bugs before we could uncover and report new ones. For example, in one `scalac` case, we reported a bug that developers fixed after a month. However, once the fix was integrated into `scalac`, Ikaros generated another program that triggered an issue with a similar root cause, suggesting that the fix was incomplete or ineffective.

**Reproducible bug reports:** When reporting a bug, we provided developers with executable programs that demonstrated its impact. For soundness issues (false negatives in exhaustiveness checks), we included a program that constructed a value not covered by existing patterns, leading to a runtime `MatchError` (Figure 1a). For completeness issues in redundancy checks (discovered by RefPG, Section 3.3.2), we built programs where the compiler incorrectly flagged a reachable

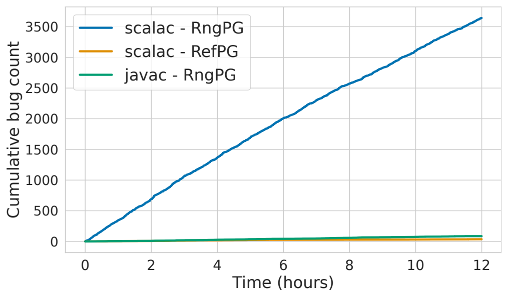


Fig. 7. Cumulative bug count over time between different pattern generation strategies.

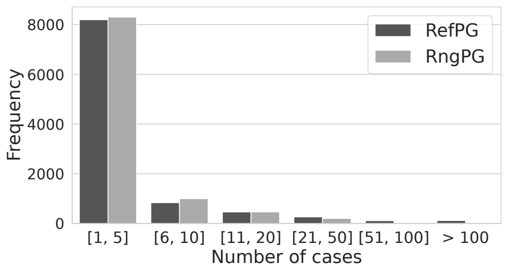


Fig. 8. Complexity of the generated programs per method measured by the number of `match` cases.

case as unreachable. Running these programs demonstrated that the supposedly unreachable case could, in fact, be executed. The most challenging cases were completeness issues in exhaustiveness checks, where the compiler mistakenly classified an exhaustive pattern as inexhaustive. In such cases, constructing a convincing bug report was difficult, as we had to enumerate all possible values of a data type, like the way we did for the example of Figure 1b. This task became challenging in certain cases due to exponential growth of values. This is the reason why for one case, it was hard to convince `javac` developers, and our report was marked as “won’t fix”.

**Comparison of bug-finding capability:** To compare the bug-finding capability of the two strategies, RefPG and RngPG, we conducted a controlled comparative analysis by running both for 12 hours on each compiler and measuring the number of bug triggers detected by each strategy. Table 1c presents the results of this comparison. After 12 hours, RngPG found 3,728 bug triggers, whereas RefPG identified only 36. Notably, RngPG not only uncovered significantly more bugs than RefPG but also detected bugs in the `javac` compiler, where RefPG failed to find any.

For completeness, Figure 7 presents the cumulative bug count over time, comparing RefPG and RngPG. The results clearly show that RngPG detects bugs significantly faster in pattern-match coverage checkers. In total, RngPG finds the first `scalac` bug within five seconds and the first `javac` bug in 389 seconds. In contrast, RefPG takes roughly 30 minutes to find its first `scalac` bug and fails to detect any bugs in `javac`.

## 5.3 RQ2: Bug and Test-Case Characteristics

**Types of bugs:** Table 1b presents the types of bugs discovered by Ikaros. The detected bugs have a balanced spread of implications and manifestations. Most of them (7 out of 16) correspond to soundness issues in the exhaustiveness checks of PMC analyzers. These are the most critical bugs, as they lead to false negatives, causing the checker to incorrectly classify inexhaustive patterns as safe. Notably, all such cases were found in `scalac`.

`scalac` also suffers from false positives (4 in total) in its redundancy checks. These bugs cause the compiler to mistakenly mark reachable cases as unreachable. This type of error is particularly “sneaky”, as it misleads programmers into trusting the compiler’s report and removing a case that is actually necessary. This can potentially result in inexhaustive and unsafe patterns.

Ikaros has also detected 4 cases where the PMC analyzers produce false positives when checking pattern exhaustiveness. All of them were found in `javac`. As a by-product of our testing efforts, Ikaros also uncovered a `scalac` bug that caused a compilation performance issue [Chaliasos et al. 2021]. Specifically, after fixing some of our reported bugs, `scalac` (version 3.6.0) began to hang indefinitely in a particular pattern-matching expression consisting of 174 cases. Interestingly, older

Table 2. (a) Language features that appear in the minimized, bug-triggering test cases of the bugs discovered by IKAROS, (b) summary statistics about the characteristics of the generated programs.

| ID | Lang | ADT | GADT | Poly. ADT | Constant | null |
|---|---|---|---|---|---|---|
| 1 | Scala | ○ | ● | ○ | ○ | ○ |
| 2 | Scala | ○ | ● | ○ | ○ | ○ |
| 3 | Scala | ○ | ● | ○ | ○ | ○ |
| 4 | Scala | ○ | ○ | ● | ○ | ○ |
| 5 | Scala | ○ | ● | ○ | ○ | ● |
| 6 | Scala | ○ | ● | ○ | ○ | ● |
| 7 | Scala | ○ | ● | ○ | ○ | ● |
| 8 | Scala | ○ | ● | ○ | ○ | ● |
| 9 | Scala | ○ | ● | ○ | ○ | ● |
| 10 | Scala | ○ | ○ | ● | ● | ○ |
| 11 | Scala | ○ | ● | ○ | ○ | ○ |
| 12 | Scala | ○ | ● | ○ | ● | ● |
| 13 | Java | ● | ○ | ○ | ○ | ○ |
| 14 | Java | ● | ○ | ○ | ○ | ○ |
| 15 | Java | ● | ○ | ○ | ○ | ○ |
| 16 | Java | ● | ○ | ○ | ○ | ○ |

(a)

| Description | 5% | Mean | Median | 95% | Histogram |
|---|---|---|---|---|---|
| Type declarations | 2 | 4 | 4 | 8 | |
| Polymorphic types | 0 | 3 | 2 | 6 | |
| Constructors | 1 | 3 | 3 | 6 | |
| GADTs | 0 | 2 | 2 | 5 | |
| Constructor parameters | 0 | 2 | 2 | 3 | |
| Patterns | 1 | 8 | 2 | 16 | |

(b)

versions of the compiler (< 3.6.0) were able to compile the given program within seven seconds. This performance regression issue was later resolved in more recent versions of `scalac`.

**Characteristics of test cases:** Table 2b provides descriptive statistics on the programs generated by IKAROS. On average, with the specified configuration (see Section 5.1), each test case contains four types, three of which are polymorphic. The corresponding constructors have an average of two parameters, while the generated pattern-matching expressions consist of eight cases, on average.

We also analyzed the language features that contributed to the 16 bugs discovered by IKAROS. To do so, we manually examined the minimized bug-inducing test cases. As shown in Table 2a, out of the 16 bugs found by IKAROS, 12 bugs involved polymorphic ADTs, and notably, 10 of them were GADTs. This indicates the complexity of handling GADTs correctly, which aligns with the insights of previous work [Graf et al. 2020; Karachalias et al. 2015].

Another recurring pattern in `scalac` bugs involved constructors with uninhabited parameters, i.e., parameters that can only be instantiated with bottom values like `null`. Half of the `scalac` bugs were triggered by test cases with such constructors (see Figures 9b and Figures 9d for concrete examples). This suggests that uninhabited types require a more principled handling in `scalac`, as exemplified by the discussion about Figure 9d. Four bugs were triggered by non-polymorphic ADTs, and interestingly, all of them were found in `javac`. The problematic test cases contained combinations of (recursive) data types, which made it difficult for the compiler to reason about the exhaustiveness of `switch` expressions. As a result, the compiler incorrectly reported false positives (see Figures 1b and 9c). Finally, two `scalac` bugs were triggered through the combination of constant patterns (e.g., an integer constant) and polymorphic ADTs (Figures 9a and 9b), showing that primitive types can also complicate coverage reasoning.

**Comparison of the complexity:** To compare the characteristics of programs generated by RefPG and RngPG, we used the number of patterns at each generated program as a metric. Since both methods influence only the complexity of `match` expressions (without affecting ADT generation), this serves as a direct comparison of their impact. Figure 8 shows that RefPG generates programs with slightly more patterns than RngPG, and it is the only method that produces programs with over 50 patterns. However, despite RngPG generating fewer patterns on average, it proves to be more effective at finding bugs (see Section 5.2). This suggests that diversity in generated programs is more important than complexity for triggering bugs.

**Exhaustiveness trends in RngPG:** : We measured the distribution of exhaustiveness results when using RngPG. Although RngPG selects and combines patterns randomly, it is inherently biased toward generating inexhaustive pattern sets. Out of 10k programs generated with RngPG, 70.4%

Table 3. Average performance of Ikaros at different stages, measured per program.

| | Generation Time | | Compilation Time | | SMT Solving Time | |
|---|---|---|---|---|---|---|
| | RefPG | RngPG | RefPG | RngPG | without timeout | with timeout |
| `scalac` | 747µs | 243µs | 3167ms | 3078.9ms | 9.6ms | 43.5ms |
| `javac` | 495µs | 324µs | 598.9ms | 1027.9ms | 9.4ms | 47.5ms |
| `ghc` | 395µs | 174µs | 111.5ms | 97.2ms | 8ms | 40.4ms |

were inexhaustive (the solver returned SAT), 25.2% were exhaustive (UNSAT), and 4.4% resulted in UNKNOWN or timeout responses from the solver. In contrast, RefPG allows for precise and uniform control over whether the generated patterns are exhaustive or inexhaustive.

## 5.4 RQ3: Performance

To measure performance, we used Ikaros to generate 10k programs for each target language; 5k using RefPG and 5k using RngPG. Table 3 presents the average running time for each stage of test case generation. The generation time includes both the time Ikaros takes to generate a type context and the time required to generate a pattern-matching statement. Notably, the results show that the generation time for both RefPG and RngPG is negligible. Naturally, compiling the generated programs adds some overhead.

The RefPG strategy consists of only two stages (*generation time* and *compilation time*) since test oracles are "baked into" the generator. In contrast, RngPG requires additional time to invoke the SMT solver for the oracle. To prevent SMT solving from slowing down the testing process, we imposed a 500ms timeout, discarding any generated program that exceeds this limit. We compare the average solving time with and without the timeout. The results show that without a timeout, the solving time is 5× longer, though still relatively minor. In practice, the timeout can be adjusted to balance throughput and accuracy.

In general, Ikaros achieves a high throughput, with the majority of testing time spent on program compilation, which is unavoidable. The generation time is minimal, and while SMT solving adds some overhead, it remains acceptable. Due to the better diversity of RngPG, even with SMT solving, RngPG still identifies more bug triggers than RefPG, as shown in Section 5.2.

## 5.5 Examples of Bug-Triggering Programs

**Figure 9a:** This program exposes a soundness bug in `scalac`'s exhaustiveness checks. The code defines an ADT with a single polymorphic constructor that takes a parameter of type `T` (lines 1–2). Later, a `match` expression attempts to pattern-match against `CC_A(10)`, which has type `CC_A[Int]`. However, the only pattern in the `match` expression is `CC_A(12)`, which does *not* cover `CC_A(10)`. This leads to a runtime `MatchError`, because `scalac` incorrectly marked the pattern as exhaustive. We reported this issue to developers, who immediately fixed the bug.
**Figure 9b:** This program reveals another soundness bug in `scalac`'s exhaustiveness checks. It defines a GADT `A[T]` with a single constructor that returns values of type `A[Char]` and takes two parameters: an `Int` and an `A[Int]` (line 2). The `match` expression includes a single pattern for `CC_A`, using the constant pattern `10` for the first constructor parameter and a wildcard for the second one. However, this pattern is clearly not exhaustive, as it fails to match cases like `CC_A(12, null)`, resulting in runtime `MatchError`. Notably, this bug arises only when the second parameter of the constructor `CC_A` is an uninhabited type that can be initialized solely with `null` (Figure 2). Moreover, the fix for Figure 9a does not resolve this issue, as the bug of Figure 9b is triggered only when a primitive type and a GADT coexist as constructor parameters (line 2).

```
sealed trait A
case class CC_A[T](a: T) extends A

val x: CC_A[Int] = CC_A(10)
val res: Int = x match {
  case CC_A(12) => 0
}
```

(a) A soundness bug in scalac (exhaustiveness).

```
sealed trait A[T]
case class CC_A(a: Int, b: A[Int])
      extends A[Char]

val x: A[Char] = CC_A(10, null)
val res: Int = x match {
  case CC_A(0, _) => 0
}
```

(b) A soundness bug in scalac (exhaustiveness).

```
sealed interface I {}
record A(I a, I b) implements I {}
record B(A a, I b) implements I {}

I x = B(null, null);
Integer res = switch (x) {
  case B(A(A(_, _), _), A(_, B(_, _))) -> 1;
  case B(A(B(_, _), _), _) -> 1;
  case B(_, B(_, _)) -> 2
  case B(_, A(_, A(_, _))) -> 3
  case A(_, _) -> 4
};
```

(c) A completeness bug in javac (exhaustiveness).

```
1   sealed trait A[T]
2   case class CC_A[T](a: B[T]) extends A[T]
3   case class CC_B[T]() extends A[T]
4   sealed trait B[T]
5   case class CC_C() extends B[Int]
6
7   val x: A[CC_C] = CC_A(null)
8   val res: Int = x match{
9     case CC_A(_) => 0
10    case CC_B() => 1
11  }
12  print(res) // prints 0
```

(d) A completeness bug in scalac (redundancy).

Fig. 9. Sample tests that trigger soundness and completeness bugs in pattern-match coverage analyzers.

**Figure 9c:** Figure 9c presents a completeness bug in javac's exhaustiveness checks. The Java program defines an ADT with recursive structures and multiple constructor parameters. The corresponding switch expression is quite complex, as it involves multiple and deeply nested patterns. Despite its complexity, the switch statement is provably exhaustive. javac incorrectly rejects the program with a compile-time error, claiming that the switch statement is not exhaustive. This bug affects both older (e.g., JDK 22) and newer (e.g., JDK 24) versions of the compiler. This bug highlights the difficulty of reporting completeness issues in exhaustiveness checks, as proving exhaustiveness requires enumerating all possible ADT values and demonstrating their full coverage.
**Figure 9d:** This code reveals a completeness bug in scalac's redundancy checks. It defines two data types, A[T] and B[T], where A[T] (it is not a GADT) has two constructors: CC_A and CC_B (lines 2 and 3). The match expression operates on values of type A[CC_C], using two patterns: one general pattern covering all CC_A values and another specifically for CC_B (lines 9 and 10). However, during compilation, scalac incorrectly warns that the first pattern (case CC_A(_), line 9) is unreachable and redundant, even though it is clearly reachable. Running the program prints 0 (line 12), proving that the pattern is matched and reachable. Interestingly, if the programmer follows scalac's suggestion and removes the "unreachable" case, the compiler contradicts itself, as it issues a new warning: *"Match may not be exhaustive: it would fail on pattern case CC_A(_)"*. This bug was found using RefPG, as it creates patterns with no unreachable cases by construction. The root cause appears to lie in how scalac reasons about constructors with uninhabited parameters (lines 2 and 9), which reveals an inconsistency in its reasoning, leading to misleading results.

## 6 Related Work

**Compiler testing:** Our work is closely related to the field of compiler testing [Chen et al. 2020], which involves the development of random program generators to validate the implementation of optimizing compilers or type checkers. Pałka et al. [2011] design a program generator built on QuickCheck [Claessen and Hughes 2000], capable of producing well-typed lambda terms based on the typing rules of a typed lambda calculus. This generator has successfully found new optimization bugs in GHC, particularly in the implementation of the strictness analyzer. Fetscher et al. [2015] generalizes this approach by deriving a random program generator from a PLT Redex specification [Felleisen et al. 2009]. Another extension involves producing well-typed lambda terms that maximize function parameter usage [Frank et al. 2024]. Poperty-based testing and QuickCheck have been also explored in other domains, such as functional languages for the blockchain [Hoang et al. 2022] or even object-oriented languages like Java [da Silva Feitosa et al. 2019].

Moving to imperative languages, Dewey et al. [2015] encode the typing rules of Rust into a constraint logic programming (CLP) problem which enables the generation of either well-typed or ill-typed Rust programs [Dewey et al. 2015]. CLP-based program generation has been effective in uncovering precision and soundness bugs in Rust's type checker. Targeting the type checkers of JVM languages (e.g., Java, Scala), HEPHAESTUS [Chaliasos et al. 2022] produces random programs that rely on complex typing features, such as parametric polymorphism, higher-order programming. Its extension, THALIA [Sotiropoulos et al. 2024] synthesizes client programs that invoke methods from complex API definitions in software libraries to stress-test type checkers.

The goal of this prior work is to uncover general bugs in type checkers and optimizing compilers, rather than in PMC analyzers. To our knowledge, our work is the first to focus on PMC analyzers.

**Pattern-match coverage checking:** PMC analyzers are integral to those languages that support ML-like pattern matching. As languages evolve and introduce complex features, such as GADTs [Xi et al. 2003], guarded patterns, or typed holes, these analyzers often struggle to provide accurate or precise warnings to developers [Graf et al. 2020; Karachalias et al. 2015]. This has led to extensive research into new algorithms and approaches for more effective detection of inexhaustive and redundant patterns [Garrigue and Normand 2011; Graf et al. 2020; Karachalias et al. 2015; Yuan et al. 2023]. PMC analyzers typically rely on constraint-solving techniques, such as type constraint solving for GADT-like reasoning [Karachalias et al. 2015] or SMT-based constraint solving for guarded patterns [Graf et al. 2020; Isradisaikul and Myers 2013]. Krishnaswami [2009] presents a framework for building PMC analyzers that are correct by construction. This work gives a logical foundation for ML-like pattern matching by interpreting patterns as proof terms in a focused sequent calculus via the Curry-Howard correspondence. This ultimately enables coverage checking and pattern compilation that are correct by construction.

Our work is orthogonal to this prior work. It could be used to validate the correctness of existing and future PMC analyzers [Cheng and Parreaux 2024], and assessing their limitations.

## 7 Conclusion

We have presented a systematic approach for identifying soundness and completeness issues in real-world pattern-match coverage analyzers. The approach features a novel generator for pattern-matching constructs that produces complex ADTs (e.g., GADTs), and on top of them, diverse combinations of (deeply-nested) patterns. To address the test oracle challenge, our approach leverages two strategies: (1) generating pattern-matching expressions that exhibit specifc properties (exhaustiveness) by construction, and (2) encoding programs into SMT formulas to verify these properties using an off-the-shelf constraint solver. When a pattern-match coverage analyzer produces results that contradict the expected properties, our approach reports a bug.

We have implemented our approach in IKAROS, an extensible framework that can be adapted to any language supporting ML-like pattern matching. Our evaluation focused on pattern-match coverage analyzers in Scala, Java, and Haskell, where IKAROS uncovered 16 bugs, 12 of which have already been fixed. Notably, many of these issues stem from unsound GADT reasoning, highlighting the challenges of implementing sound pattern-match coverage analysis.

## Data-Availability Statement

The research artifact [Moser et al. 2025] is available at Zenodo under the GNU General Public License v3.0. It provides the scripts, the data, and the results presented in this work. Ikaros is also available as an open-source software under the same license at https://github.com/CyrilFMoser/Ikaros.

## References

Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-Typed Programs Can Go Wrong: A Study of Typing-Related Bugs in JVM Compilers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 123 (Oct. 2021), 30 pages. doi:10.1145/3485500

Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding Typing Compiler Bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 183–198. doi:10.1145/3519939.3523427

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (Feb. 2020), 36 pages. doi:10.1145/3363562

Luyu Cheng and Lionel Parreaux. 2024. The Ultimate Conditional Syntax. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 306 (Oct. 2024), 30 pages. doi:10.1145/3689746

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279. doi:10.1145/357766.351266

Samuel da Silva Feitosa, Rodrigo Geraldo Ribeiro, and Andre Rauber Du Bois. 2019. Generating Random Well-Typed Featherweight Java Programs Using QuickCheck. *Electronic Notes in Theoretical Computer Science* 342 (2019), 3–20. doi:10.1016/j.entcs.2019.04.002 The proceedings of CLEI 2018, the XLIV Latin American Computing Conference.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 482–493. doi:10.1109/ASE.2015.65

Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. doi:10.1145/3597926.3598130

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex* (1st ed.). The MIT Press.

Burke Fetscher, Koen Claessen, Michał Pałka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 383–405.

Justin Frank, Benjamin Quiring, and Leonidas Lampropoulos. 2024. Generating Well-Typed Terms That Are Not "Useless". *Proc. ACM Program. Lang.* 8, POPL, Article 77 (Jan. 2024), 22 pages. doi:10.1145/3632919

Jacques Garrigue and Jacques Le Normand. 2017. GADTs and Exhaustiveness: Looking for the Impossible. *arXiv e-prints*, Article arXiv:1702.02281 (Feb. 2017), arXiv:1702.02281 pages. arXiv:1702.02281 [cs.PL] doi:10.48550/arXiv.1702.02281

Jacques Garrigue and JL Normand. 2011. Adding GADTs to OCaml: the direct approach. In *Workshop on ML*. 27.

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2024. The Java Language Specification: Java SE 22 Edition. https://docs.oracle.com/javase/specs/jls/se22/jls22.pdf.

Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. 2020. Lower your guards: a compositional pattern-match coverage checker. *Proc. ACM Program. Lang.* 4, ICFP, Article 107 (Aug. 2020), 30 pages. doi:10.1145/3408989

Tram Hoang, Anton Trunov, Leonidas Lampropoulos, and Ilya Sergey. 2022. Random testing of a higher-order blockchain language (experience report). *Proc. ACM Program. Lang.* 6, ICFP, Article 122 (Aug. 2022), 16 pages. doi:10.1145/3547653

Chinawat Isradisaikul and Andrew C. Myers. 2013. Reconciling exhaustive pattern matching with objects. *SIGPLAN Not.* 48, 6 (June 2013), 343–354. doi:10.1145/2499370.2462194

Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2015. GADTs meet their match: pattern-matching warnings that account for GADTs, guards, and laziness. *SIGPLAN Not.* 50, 9 (Aug. 2015), 424–436. doi:10.1145/2858949.2784748

James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. 2022. Searching Entangled Program Spaces. *Proc. ACM Program. Lang.* 6, ICFP, Article 91 (aug 2022), 29 pages. doi:10.1145/3547622

Neelakantan R. Krishnaswami. 2009. Focusing on pattern matching. 44, 1 (Jan. 2009), 366–378. doi:10.1145/1594834.1480927

Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 701–712. doi:10.1145/3368089.3409763

William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107. http://dblp.uni-trier.de/db/journals/dtj/dtj10.html#McKeeman98

Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 142–151. doi:10.1145/1134285.1134307

Cyril Moser, Thodoris Sotiropoulos, Chengyu Zhang, and Zhendong Su. 2025. *Artifact for "Validating Soundness and Completeness in Pattern-Match Coverage Analyzers"*. doi:10.5281/zenodo.16909625

Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST '11)*. Association for Computing Machinery, New York, NY, USA, 91–97. doi:10.1145/1982595.1982615

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. doi:10.1145/2254064.2254104

Rust Team. 2025. Pattern and Exhaustiveness Checking. https://rustc-dev-guide.rust-lang.org/pat-exhaustive-checking.html. Online accessed; 19-03-2025.

Thodoris Sotiropoulos, Stefanos Chaliasos, and Zhendong Su. 2024. API-Driven Program Synthesis for Testing Static Typing Implementations. *Proc. ACM Program. Lang.* 8, POPL, Article 62 (Jan. 2024), 32 pages. doi:10.1145/3632904

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. *SIGPLAN Not.* 49, 9 (Aug. 2014), 269–282. doi:10.1145/2692915.2628161

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement reflection: complete verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article 53 (Dec. 2017), 31 pages. doi:10.1145/3158141

Elaine J. Weyuker. 1982. On Testing Non-Testable Programs. *Comput. J.* 25, 4 (11 1982), 465–470. arXiv:https://academic.oup.com/comjnl/article-pdf/25/4/465/1045637/25-4-465.pdf doi:10.1093/comjnl/25.4.465

Dominik Winterer and Zhendong Su. 2024. Validating SMT Solvers for Correctness and Performance via Grammar-Based Enumeration. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 355 (Oct. 2024), 24 pages. doi:10.1145/3689795

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020a. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 193 (Nov. 2020), 25 pages. doi:10.1145/3428261

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020b. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 718–730. doi:10.1145/3385412.3385985

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. *SIGPLAN Not.* 38, 1 (Jan. 2003), 224–235. doi:10.1145/640128.604150

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532

Yongwei Yuan, Scott Guest, Eric Griffis, Hannah Potter, David Moon, and Cyrus Omar. 2023. Live Pattern Matching with Typed Holes. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 96 (April 2023), 27 pages. doi:10.1145/3586048

Received 2025-03-25; accepted 2025-08-12