A Generic Algorithm for Checking Exhaustivity of Pattern Matching (Short Paper)

Fengyun Liu

EPFL, Switzerland

fengyun.liu@epfl.ch

Abstract

Algebraic data types and pattern matching are key features of functional programming languages. Exhaustivity checking of pattern matching is a safety belt that defends against unmatched exceptions at runtime and boosts type safety. However, the presence of language features like inheritance, typecase, traits, GADTs, path-dependent types and union types makes the checking difficult and the algorithm complex. In this paper we propose a generic algorithm that decouples the checking algorithm from specific type theories. The decoupling makes the algorithm simple and enables easy customization for specific type systems.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Patterns

Keywords pattern matching, exhaustivity check, Scala

1. Introduction

One distinctive feature of functional programming languages, like Scala, OCaml, Haskell, is the availability of ADTs (algebraic data types) and the ability to deconstruct them using pattern matching. When a pattern match gets complex, it's easy to forget some cases. Such unhandled cases cause runtime exceptions and degrade type safety. To uphold type safety, most compilers do some exhaustivity check to issue warnings about missing cases. For example, the Scala compiler warns that the case `Some(Nil)` is not handled in the following code:

```
(Some(Nil): Option[List[Int]]) match {
  case Some(x::xs) => true
  case None => false
}
```

SCALA'16, October 30–31, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4648-1/16/10...$15.00
http://dx.doi.org/10.1145/2998392.2998401

Exhaustivity checking is a solved problem for ADTs [8]. However, in a language like Scala which is enriched with features like inheritance, typecase [1], traits [9], GADTs [4, 12], path-dependent types [2] and union types [3], exhaustivity checking becomes complex. Without good abstraction, each feature complicates the algorithm a little bit and the combination of features results in a monolithic algorithm that's impossible to maintain.

In this paper we introduce an abstraction called *space* which enables us to decouple knowledge about the underlying type system from the algorithm. The decoupling results in a simple and generic algorithm for exhaustivity checking that is easily customizable to specific type systems. Concretely, our contributions are as follows:

- We put forward a generic exhaustivity checking algorithm that can be easily extended to support concrete type systems without making changes to the core algorithm.
- We extend the generic algorithm for exhaustivity checking in Scala. The new algorithm is much simpler than the implementation in the standard Scala compiler, and it fixes 13 open issues about exhaustivity checking in Scala's issue tracker.

2. Algorithm

2.1 Idea

The basic idea of the algorithm is that types and patterns can be thought as *spaces* of values. The space of a type is the set of values that inhabit the type. The space of a pattern is the set of values that can be covered by the pattern. More concretely, *space* is inductively defined as follows:

1. $\mathscr{O}$ is an empty space.
2. $\mathscr{T}(T)$ is a type space of type T.
3. If $s_1, s_2, \cdots$ are spaces, then $s_1 \mid s_2 \mid$ ... is a union space.
4. If $s_1, s_2, \cdots, s_n$ are spaces and K is a constructor type, then $\mathscr{K}(K, s_1, s_2, ..., s_n)$ is a constructor space.

A *constructor type* refers the type of all values created using a constructor in an ADT definition. In Scala, all cases classes are constructor types. In Haskell or OCaml, programmers cannot explicitly use the constructor names of ADTs as types, but inside the checking algorithm we can treat them as types.

Now we can reformulate the problem of *exhaustivity checking* in terms of spaces:

> Is the space $\mathscr{T}(T)$, where T is the type of the value to be matched against, a subspace of the union of spaces covered by all the pattern clauses?

2.2 Design

With the formulation above, an exhaustivity checking algorithm only needs two definitions:

- $s_1 \preceq s_2$: whether space s_1 is a subspace of s_2
- $\mathscr{P}(p)$: projects a pattern p to a space.

The generic algorithm leaves the function $\mathscr{P}$ to concrete implementations, as it depends on knowledge of the concrete type system. Generally, the following projection rules are observed:

$$\begin{array}{lcl}
\mathscr{P}(x : T) & = & \mathscr{T}(T) \\
\mathscr{P}(p_1 \mid p_2 \mid \cdots) & = & \mathscr{P}(p_1) \mid \mathscr{P}(p_2) \mid \cdots \\
\mathscr{P}(K(p_1, p_2, \cdots)) & = & \mathscr{K}(K, \mathscr{P}(p_1), \mathscr{P}(p_2), \cdots)
\end{array}$$

To define the subspace relation $\preceq$, the generic algorithm makes minimal assumptions about the type system:

- A constructor type cannot be a super type of other types.
- If every value that inhabits the type T_1 also inhabits the type T_2, then T_1 is a subtype of T_2.

The generic algorithm depends on the following definitions from concrete implementations:

- $T_1 <: T_2$: whether T_1 is a subtype of T_2
- $sig(K)$: get parameter types of the constructor type K
- $\mathscr{D}?(T)$: whether the type T is decomposable
- $\mathscr{D}(T)$: decompose the type T into a union of subspaces

The predicates and functions in the list above are all related to specific features of the type system. Deferring these definitions to the concrete implementation results in an elegant decoupling of the generic algorithm from the concrete type system.

The items in the list are self-evident except $\mathscr{D}?(T)$ and $\mathscr{D}(T)$, which deserve some explanation. The whole idea of pattern matching depends on the assumption that the values of a type can be *completely* (not necessarily disjointly) partitioned as values of some more specific types. Put it in the language of *spaces*, pattern matching assumes that the spaces of some types can be *completely* decomposed into sub-spaces. For example, an ADT can be decomposed to the spaces of its constructor types, an union type $S \mid T$ can be decomposed to the space of S and the space of T.

$$\boxed{a \ominus b}$$

$$\begin{array}{llcl}
1 & \mathscr{O} \ominus x & = & \mathscr{O} \\
2 & x \ominus \mathscr{O} & = & x \\
3 & \mathscr{T}(T_1) \ominus \mathscr{T}(T_2) & = & \mathscr{O} \text{ if } T_1 <: T_2 \\
4 & \mathscr{T}(K) \ominus \mathscr{K}(K, s_1, \cdots) & = & \begin{array}{l} \mathscr{K}(K, map\ \mathscr{T}\ sig(K)) \ominus \\ \mathscr{K}(K, s_1, \cdots) \end{array} \\
5 & (s_1 \mid s_2 \mid \cdots) \ominus x & = & s_1 \ominus x \mid s_2 \ominus x \mid \cdots \\
6 & x \ominus (s_1 \mid s_2 \mid \cdots) & = & x \ominus s_1 \ominus s_2 \ominus \cdots \\
7 & \mathscr{K}(K, s_1, \cdots) \ominus \mathscr{T}(T) & = & \mathscr{O} \text{ if } K <: T \\
8 & \begin{array}{l} \mathscr{K}(K, s_1, \cdots) \ominus \\ \mathscr{K}(K, w_1, \cdots) \end{array} & = & \left\{ \begin{array}{l} (1)\mathscr{O} \text{ if } \forall i.s_i \preceq w_i \\ (2)\mathscr{K}(K, s_1, \cdots) \\ \quad \text{if } \exists i.s_i \sqcap w_i \doteq \mathscr{O} \\ (3)(\mathscr{K}(K, s_1 \ominus w_1, s_2, \cdots) \mid \\ \quad \mathscr{K}(K, s_1, s_2 \ominus w_2, \cdots) \mid \\ \quad \cdots) \text{ otherwise} \end{array} \right. \\
9 & \mathscr{T}(T) \ominus x & = & \mathscr{D}(T) \ominus x \text{ if } \mathscr{D}?(T) \\
10 & x \ominus \mathscr{T}(T) & = & x \ominus \mathscr{D}(T) \text{ if } \mathscr{D}?(T) \\
11 & a \ominus b & = & a \text{ otherwise}
\end{array}$$

Figure 1. Definition of space subtraction

$$\boxed{a \sqcap b}$$

$$\begin{array}{llcl}
1 & \mathscr{O} \sqcap x & = & \mathscr{O} \\
2 & x \sqcap \mathscr{O} & = & \mathscr{O} \\
3 & \mathscr{T}(T_1) \sqcap \mathscr{T}(T_2) & = & \mathscr{T}(T_1) \text{ if } T_1 <: T_2 \\
4 & \mathscr{T}(T_1) \sqcap \mathscr{T}(T_2) & = & \mathscr{T}(T_2) \text{ if } T_2 <: T_1 \\
5 & \mathscr{T}(T) \sqcap \mathscr{K}(K, s_1, \cdots) & = & \mathscr{K}(K, s_1, \cdots) \text{ if } K <: T \\
6 & (s_1 \mid s_2 \mid \cdots) \sqcap x & = & s_1 \sqcap x \mid s_2 \sqcap x \mid \cdots \\
7 & x \sqcap (s_1 \mid s_2 \mid \cdots) & = & x \sqcap s_1 \mid x \sqcap s_2 \mid \cdots \\
8 & \mathscr{K}(K, s_1, \cdots) \sqcap \mathscr{T}(T) & = & \mathscr{K}(K, s_1, \cdots) \text{ if } K <: T \\
9 & \begin{array}{l} \mathscr{K}(K, s_1, \cdots) \sqcap \\ \mathscr{K}(K, w_1, \cdots) \end{array} & = & \mathscr{K}(K, s_1 \sqcap w_1, s_2 \sqcap w_2, \cdots) \\
10 & \mathscr{T}(T) \sqcap x & = & \mathscr{D}(T) \sqcap x \text{ if } \mathscr{D}?(T) \\
11 & x \sqcap \mathscr{T}(T) & = & x \sqcap \mathscr{D}(T) \text{ if } \mathscr{D}?(T) \\
12 & a \sqcap b & = & \mathscr{O} \text{ otherwise}
\end{array}$$

Figure 2. Definition of space intersection

2.3 Algorithm

Given the symbols introduced in the previous section, the problem of *exhaustivity checking* can be reformulated as follows:

> Patterns $p_1, p_2, \cdots$ are exhaustive for type T iff $\mathscr{T}(T) \preceq \mathscr{P}(p_1) \mid \mathscr{P}(p_2) \mid \cdots$

The generic algorithm only needs to define the subspace relation ($\preceq$), which can be defined in terms of space subtraction ($\ominus$).

DEFINITION (Subspace). $s_1 \preceq s_2$ if and only if $s_1 \ominus s_2 \doteq \mathscr{O}$

The equality relation ($\doteq$) used in the definition implies there should be a theory about equality between spaces. We don't bother to define a general equality relation here, as it's only used to compare with an empty space, for which the following simple rules suffice:

$$\begin{array}{rcll}
\mathscr{O} & \doteq & \mathscr{O} & \\
\mathscr{T}(T) & \doteq & \mathscr{O} & \text{if } \mathscr{D}?(T) \wedge \mathscr{D}(T) \doteq \mathscr{O} \\
s_1 \mid s_2 \mid \cdots & \doteq & \mathscr{O} & \text{if } \forall i.s_i \doteq \mathscr{O} \\
\mathscr{K}(K, s_1, s_2, \cdots) & \doteq & \mathscr{O} & \text{if } \exists i.s_i \doteq \mathscr{O}
\end{array}$$

The definition of space subtraction is shown in Figure 1. It depends on space intersection ($\sqcap$), the latter is defined in Figure 2. Most of the definitions are straight-forward. The most complex one is the subtraction of two constructor spaces (rows 8 in Figure 1). Given two constructor spaces, $a = \mathscr{K}(K, s_1, \cdots)$ and $b = \mathscr{K}(K, w_1, \cdots)$:

1. If for all i, we have $s_i \preceq w_i$, then each value in a is also a value in b, thus $a \ominus b = \mathscr{O}$.
2. In contrast, if there exists i such that $s_i \sqcap w_i \doteq \mathscr{O}$, then it's impossible for any value of a to be in b, thus $a \ominus b = a$.
3. What cases are not matched in the following code?

```
val x, y: Option[Int] = ...

(x, y) match {
  case (None, Some(_)) => true
}
```

The algorithm will warn that `(Option, None)` and `(Some(_), Option)` are not covered. It's easy to see that `(Some(_), None)` is included in both of the counterexamples. However, as this duplication doesn't harm correctness and it's the simplest rule we can arrive at, we're happy to keep it.

2.4 Discussion

Theoretically, it is a little surprising that $\preceq$ is defined in terms of $\ominus$. Is it possible to define $\preceq$ independently? We tried but failed. The problem we encounter is how to define the following case: $\mathscr{K}(K, w_1, \cdots, w_m) \preceq s_1 \mid \cdots \mid s_n$.

In the case above, the constructor space might be covered jointly by multiple components of the union space. We can either resort to subtraction as we do in this paper, or try to fuse all constructor spaces s_i of the shape $\mathscr{K}(K, \_, \_, \cdots)$ into a single constructor space. Then the problem reduces to the case of comparing two single spaces, which can be handled easily.

However, the latter approach is not always possible. For example, there's no simple way to fuse `(Some(_), Some(_))` and `(None, None)`. The naive approach to fuse it as `(Option, Option)` is obviously incorrect.

Another argument for defining $\preceq$ in terms of $\ominus$ is that subtraction is needed anyway in order to produce friendly counterexamples in the warning message.

The generic algorithm makes the assumption that *a constructor type cannot be a super type of any other types*. Without this assumption, it is unclear how to define following cases correctly:

$$
\begin{array}{lcll}
\mathscr{T}(T) \ominus \mathscr{K}(K, s_1, \cdots) & = & ? & \text{if } T <: K \\
\mathscr{K}(K, s_1, \cdots) \ominus \mathscr{T}(T) & = & ? & \text{if } T <: K \\
\mathscr{T}(T) \sqcap \mathscr{K}(K, s_1, \cdots) & = & ? & \text{if } T <: K \\
\mathscr{K}(K, s_1, \cdots) \sqcap \mathscr{T}(T) & = & ? & \text{if } T <: K
\end{array}
$$

Luckily, in most programming languages this is not a problem. In Scala, case classes can be inherited. However, it's an anti-pattern and rarely used in practice.

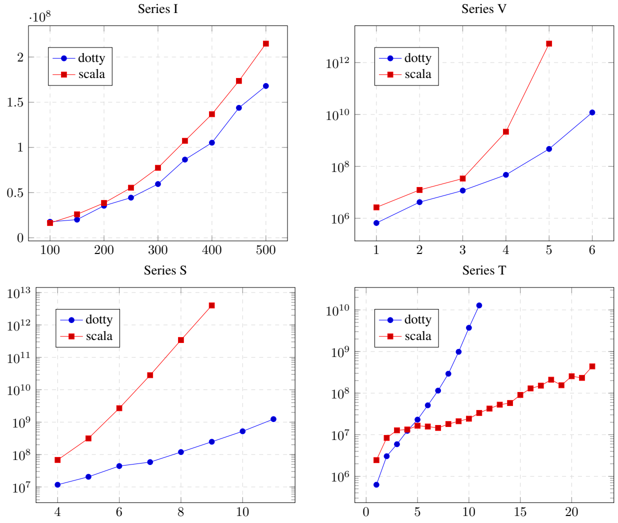

Figure 3. Performance comparison (time unit: ns)

3. Evaluation

3.1 Correctness

We have not formally proved correctness of the algorithm. However, we tested the implementation on Dotty. The new algorithm passes all tests migrated from the Scala compiler and fixes 13 open issues in Scala's issue tracker[1].

In principle, these issues could also be fixed in the current Scala compiler. However, the complexity of the implementation incurs higher cost to fix them, which explains that some issues stayed for a long time on the issue tracker.

3.2 Maintainability

The new algorithm takes 442LOC. The algorithm in the current Scala compiler amounts to 1369LOC. We have extended the approach to introduce a concept called *point* in order to support constants, Java enumerations and stable identifiers in patterns. The extension is straight-forward and trivial to implement.

3.3 Performance

The general problem of pattern matching is NP-Complete [10]. Thus we expect the worst-case performance of the generic algorithm to be exponential. This is caused by the subtraction of two constructor spaces which could result in a proliferation of spaces. However, in practice the algorithm performs well without performance issues.

We compared the performance of the new algorithm and the algorithm in current Scala compiler based on series I, S, V and T from Maranget's paper [8]. Figure 3 shows that the new algorithm performs better in series I, S and V. Series T suggests potential optimizations to the new algorithm,

[1] https://github.com/lampepfl/dotty/pull/1364

which needs further investigation. The benchmarks are easy to reproduce following the instructions here[2].

3.4 Limitations

Guards in pattern clauses pose a theoretical difficulty, thus are not handled by the algorithm. For example, in the following example the arithmetic in the guard is too complex for the compiler:

```
(x, y) match {
  case (a, b) if a*a + 2*a*b + b*b >= 0 => true
}
```

It's no surprise that no existing compilers can properly handle guards, as the guard can be any possible code that may be undecidable. Scala supports a language feature called *extractors* [5]. Theoretically, they are as complex as guards, thus cannot be handled properly.

The algorithm assumes the type of each constructor parameter is independent, thus it cannot handle type constraints that relate different constructor parameters. This can be illustrated by the following example:

```
sealed trait Expr[T]
case class IntExpr(x: Int) extends Expr[Int]
case class BooleanExpr(b: Boolean) extends
    Expr[Boolean]

def foo[T](x: Expr[T], y: Expr[T]) = (x, y)
    match {
  case (IntExpr(_), IntExpr(_)) => true
  case (BooleanExpr(_), BooleanExpr(_)) => false
}
```

It's obvious that the pattern match in the above is exhaustive, but the algorithm will complain that it is non-exhaustive.

Note that the algorithm implemented in current Scala compiler also faces the same limitations.

4. Related Work

Maranget proposes an elegant algorithm for checking the exhaustivity of ADTs [8], which inspired the algorithm in this paper. The two algorithms are the same in spirit, though Maranget's algorithm is presented with pattern matrices based on the concept of *useful clause*. Maranget's algorithm assumes a type system with only ADTs, thus it may perform an optimization which is unavailable to us. Our algorithm abstracts the type system away, thus is more generic.

In OCaml, the GADT implementation extended the original checking algorithm by eliminating the ill-typed uncovered cases [6]. This is also a possible approach to improve our algorithm to better handle GADTs in the future.

Xi takes a two-step approach for eliminating dead code for GADT pattern matching [11]: first add all the missing patterns using simple pattern checking techniques, and then remove redundant clauses by checking when typing constraints are un-satisfiable.

Karachalias introduces a new algorithm for checking GADTs based on type constraint solving [7]. This algorithm depends on a type constraint solver as an oracle.

5. Conclusion

In this paper we presented a generic algorithm for checking exhaustivity of pattern matching. The decoupling of knowledge about the type system from the algorithm makes the algorithm simple and extensible for potentially any type systems.

Acknowledgments

We thank Dmitry Petrashko and Guillaume Martres for helping us implement the algorithm in Dotty. We thank Sandro Stucki, Nada Amin and Nicolas Stucki for their helpful feedback on the draft of this paper.

References

[1] M. Abadi, L. Cardelli, B. C. Pierce, D. Rémy, and R. Taylor. Dynamic typing in polymorphic languages. *Journal of functional programming*, 5(1):111–130, 1995.

[2] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. The essence of dependent object types. In *A List of Successes That Can Change the World*, pages 249–272. Springer, 2016.

[3] F. Barbanera, M. Dezaniciancaglini, and U. Deliguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[4] J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[5] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *European Conference on Object-Oriented Programming*, pages 273–298. Springer, 2007.

[6] J. Garrigue and J. Normand. Adding GADTs to ocaml: The direct approach. In *Workshop on ML*, 2011.

[7] G. Karachalias, T. Schrijvers, D. Vytiniotis, and S. P. Jones. GADTs meet their match. In *International Conference on Functional Programming, ICFP*, volume 15, 2015.

[8] L. Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(03):387–421, 2007.

[9] M. Odersky and M. Zenger. Scalable component abstractions. In *ACM Sigplan Notices*, volume 40, pages 41–57. ACM, 2005.

[10] R. Sekar, R. Ramesh, and I. Ramakrishnan. Adaptive pattern matching. In *International Colloquium on Automata, Languages, and Programming*, pages 247–260. Springer, 1992.

[11] H. Xi. Dependently typed pattern matching. *Journal of universal computer science*, 9(8):851–872, 2003.

[12] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *ACM SIGPLAN Notices*, volume 38, pages 224–235. ACM, 2003.

[2] https://github.com/liufengyun/bench-patmat