Pattern Coverage

Brian Goetz and Gavin Bierman

March 24, 2022

1 Coverage

A `switch` expression must be exhaustive on its target; this means that when switching over an expression of type T, for any operand expression, some case should match, and the compiler will want to statically type-check this exhaustiveness. (Other pattern-aware constructs, such as pattern assignment or statement switches using patterns, may have similar requirements.) We express this by saying that the set of `case` patterns in a `switch` must *cover* the target type of the switch.

We define a relation, written $\Delta \vdash P$ **covers** T, to denote that a set of patterns P covers the (expressible) type T. Coverage is used in static type checking to assess whether the patterns in a construct such as `switch` covers the target type.

Coverage is a relationship between a *set* of patterns and a type; if we want to talk about an individual pattern covering a type, we can do so by evaluating whether the singleton set containing that patterns covers the desired type.

A set containing only the total type pattern on T, or an `any` pattern, covers T:

$$\text{T-Any}\ \frac{}{\Delta \vdash \{\ \texttt{any}\ \}\ \textbf{covers}\ T}$$

$$\text{T-Type}\ \frac{}{\Delta \vdash \{\ T\ \texttt{t}\ \}\ \textbf{covers}\ T}$$

If a set of patterns covers a type, adding more patterns to the set does affect coverage (coverage is contravariant with subsetting), and if a set of patterns covers a type, it covers all the subtypes of that type (coverage is covariant with subtyping):

$$\text{T-Subset}\ \frac{\Delta \vdash P\ \textbf{covers}\ T \qquad P \subseteq Q}{\Delta \vdash Q\ \textbf{covers}\ T}$$

$$\text{T-Subtype}\ \frac{\Delta \vdash P\ \textbf{covers}\ T \qquad T' <: T}{\Delta \vdash P\ \textbf{covers}\ T'}$$

2 Sealing

A *sealed* class is one that restricts extension to an enumerated set of classes. Sealing offers another source of exhaustiveness information; if class C is sealed to permit only A and B, then if a set of patterns covers each of A and B, we know it also covers C, without requiring an explicit total pattern on C or a default case. This enables better type checking, because a switch that intends to explicitly cover all cases, and does not provide a default, will get help from the compiler to validate the assumption of coverage, and will be alerted if someone later redefines C to have additional permitted subtypes.

It is possible that for a given parameterization of a sealed class, one or more of its permitted subtypes may not be applicable. For example, given:

```
sealed interface Node<T>
    permits StringNode, IntNode, PlusNode { }

record StringNode(String s) implements Node<String> { }
record IntNode(String s) implements Node<Integer> { }
record PlusNode<T>(T a, T b) implements Node<T> { }
```

For a `switch` with a target type of `Node<Double>`, the patterns for `StringNode` and `IntNode` are not going to be applicable, because a `Node<Double>` cannot be either of these types. In this case, we need not consider these impossible subclasses when computing exhaustiveness. We appeal to *cast conversion* to exclude such implausible patterns. We write **castable**(T, U) to indicate that T is castable to U without an unchecked conversion.

A set of patterns covers an instantiation of a sealed class if it covers all of the permitted subtypes of that class. We use the generic information to exclude implausible subtype patterns; once this is done, for purposes of computing coverage, we need not consider the parameterizations, and can work with wildcard types.

$$\text{T-Sealed}\ \frac{\begin{array}{c}\texttt{abstract sealed class C<}\overline{\texttt{X}}\texttt{> permits } N_1,\ \ldots,\ N_n \\ \forall_i\ \ \Delta \vdash P \textbf{ covers } N_i\texttt{<?>}\ \ \vee\ \neg\ \textbf{castable}(\texttt{C<}\overline{\texttt{A}}\texttt{>}, N_i\texttt{<?>})\end{array}}{\Delta \vdash P \textbf{ covers } \texttt{C<}\overline{\texttt{A}}\texttt{>}}$$

2.1 Coverage on records

We now extend coverage to *record patterns*. Given a record declaration

$$\texttt{record } R\texttt{(}T_1\ \texttt{t}_1\texttt{,}\ldots\texttt{,}T_n\ \texttt{t}_n\texttt{)}$$

a record pattern is denoted $R(p_1,\ \ldots,\ p_n)$, where each p_i is applicable to T_i.

For a record with a single component of type T, and a set of patterns that cover T, we can construct a set of patterns that cover R simply by wrapping each pattern in a record pattern:

$$\textsc{T-RecBase}\ \frac{\begin{array}{c}\Delta \vdash P \textbf{ covers } T \\ \texttt{record R(}T\ \texttt{t)}\end{array}}{\Delta \vdash \{\ R(p)\ :\ p \in P\ \} \textbf{ covers } R}$$

For records with more than one component, we can define coverage by induction on the number of components. For a record R with n components, for $n > 1$, we define its *tail record*, denoted R^t, as the record with the trailing $n-1$ components of R.

If we denote the set of all patterns on type T as $\mathbb{P}[T]$, we can define two functions for decomposing a record pattern, *head* and *tail*:

$$head \quad : \quad \mathbb{P}[R] \to \mathbb{P}[T_1] \tag{1}$$
$$tail \quad : \quad \mathbb{P}[R] \to \mathbb{P}[R^t] \tag{2}$$

These functions have the obvious definition; for a record pattern $R(p_1,\ \ldots,\ p_n)$, the head pattern is p_1 and the tail pattern is $R^t(p_2,\ \ldots,\ p_n)$.

In the absence of sealing information, a covering set of patterns must have some pattern that by itself covers the target. Record patterns are no different; we could require that to cover a switch on R, there must be some pattern $R(p_1,\ \ldots,\ p_n)$ where each of the p_i covers the correponding component. But instead, we'll define coverage so that we can more easily define how sealing affects record patterns.

We will do this by taking the record patterns on R whose head pattern covers the first component of R – and then taking the tail patterns of these patterns, and asking whether the corresponding set of tail patterns also cover R^t:

$$\textsc{T-RecInduct}\ \frac{\begin{array}{c}\texttt{record } R\texttt{<}\overline{\texttt{X}}\texttt{>(}T_1\ \texttt{t}_1\texttt{,}\ \ldots\texttt{,}T_n\ \texttt{t}_n\texttt{)} \\ Q \subseteq \mathbb{P}[R] \\ \Delta \vdash \{\ head(q)\ :\ q \in Q\ \} \textbf{ covers } T_1[\overline{\texttt{X}} := \overline{\texttt{A}}] \\ \Delta \vdash \{\ tail(q)\ :\ q \in Q,\ \ \Delta \vdash head(q) \textbf{ covers } T_1\ \} \textbf{ covers } R^t[\overline{\texttt{X}} := \overline{\texttt{A}}]\end{array}}{\Delta \vdash Q \textbf{ covers } R\texttt{<}\overline{\texttt{A}}\texttt{>}}$$

So far, this is just a fancy way to ask whether there is a single record pattern on R whose subpatterns cover all the components of R. But this becomes much more useful when one or more of the component types of R are sealed. To define coverage when one of the record components is sealed, we must first define another projection on records. For a record R, the first-component-projection to X, denoted $R^{h=X}$, is the record whose components are identical to that of R, except the type of the first component is X.

$$\text{T-RecSealed}\ \frac{\begin{array}{c}\texttt{abstract sealed class C<}\overline{\texttt{X}}\texttt{> permits } N_1,\ \ldots,\ N_n \\ \texttt{record } R\texttt{(}C\texttt{<}\overline{\texttt{A}}\texttt{>},\ T_2,\ \ldots,\ T_n\texttt{)} \\ Q \subseteq \mathbb{P}[R] \\ \forall_i\ \ \Delta \vdash Q\ \textbf{covers}\ R^{h=N_i[\overline{\texttt{X}}:=\overline{\texttt{A}}]}\end{array}}{\Delta \vdash Q\ \textbf{covers}\ R}$$

Here is a simple example of how these rules work together.

```
sealed interface C permits A, B { }
final class A extends C { }
final class B extends C { }

record R(C x, C y) { }

switch (r) {
  case R(C x, A y): ...
  case R(A x, B y): ...
  case R(B x, B y): ...
}
```

The switch in the example covers R. We can first apply T-RecSealed, which says we can individually look at A and B in the first position. We then apply T-RecInduct twice. For A, both $A\ x$ and $C\ x$ cover A, sowe take the set of corresponding tail patterns (the first and second cases), and ask whether the set $\{R^t(A\ y),\ R^t(B\ y)\}$ covers the tail record. We then repeat the same for B, and take the tail patterns from the first and third cases, and ask the same question. We can answer both of these questions by appealing to T-RecBase. The important thing is that we may use the tail patterns from *different* subsets of the original for each permitted subtype.