Formalization of Coverage Checking in Agda

Satoshi Takimoto*, Sosuke Moriguchi, and Takuo Watanabe

Department of Computer Science, Institute of Science Tokyo, Tokyo, Japan,
takimoto@psg.comp.isct.ac.jp, chiguri@acm.org, takuo@acm.org

Abstract. Coverage checking is an essential static analysis for preventing runtime errors in languages with pattern matching. We present a work-in-progress formalization of Maranget's algorithm in Agda. The algorithm handles simple patterns but does not support complex patterns such as those involving GADTs or pattern guards. We have formally proven the algorithm's correctness and termination. Furthermore, our formalization is compatible with agda2hs, enabling extraction of a verified coverage checker implemented in Haskell. This work serves as a foundation for formalizing more efficient or expressive coverage checking algorithms.

Keywords: formal verification, pattern match, coverage check, agda

1 INTRODUCTION

Pattern matching is a fundamental feature in languages such as the ML family, Haskell, and Rust. It enables definitions to be specified on a case-by-case basis. This flexibility requires static analysis to ensure that pattern clauses satisfy two key properties: *exhaustiveness* and *non-redundancy*. Exhaustiveness means that all possible cases are covered by the pattern clauses, while non-redundancy means that there are no overlaps between clauses. Algorithms that verify these two properties are known as coverage checking algorithms. Coverage checking is crucial in languages with pattern matching, as it helps prevent runtime errors from unhandled cases and informs users when certain pattern clauses are unreachable.

Among the coverage checking algorithms available, we have formalized an algorithm proposed by Maranget [9], which serves as the foundation for Rust's coverage checker [10]. Although this algorithm does not handle complex patterns such as those involving GADTs or pattern guards, it is relatively simple and covers a wide range of use cases. The formalization is carried out in Agda [2] and is compatible with the extraction tool agda2hs [4], allowing us to generate readable Haskell programs. The formalization is available on GitHub[1].

In this paper, we present a simplified version of our formalization. The structure is as follows:

- Section 2 introduces the syntax for values and patterns, as well as the definition of pattern matching.

[1] GitHub repository: https://github.com/wasabi315/coverage-checking

– Section 3 formalizes the concept of usefulness and describes the usefulness checking algorithm. We begin with the basic algorithm and refine it into a verified version, proving both correctness and termination.
– Section 4 presents a verified exhaustiveness checking algorithm, building on the verified usefulness checking algorithm from the previous section.

Each section also includes small examples that build upon those from previous sections, creating running examples throughout the paper. Section 5 provides a brief overview of agda2hs and discusses the Haskell programs generated by extraction.

2 PATTERN MATCHING

2.1 Values

We begin by formalizing datatypes and signatures, and values. A signature maps each datatype name to its corresponding definition, specifying the number of constructors and the types of their arguments. In this formalization, we focus exclusively on datatypes, excluding primitive types such as integers.

```
record Signature : Type where
  field
    dataDef : (d : ℕ) → Dataty
    -- ^ d : Datatype name

data Ty : Type where
  dataty : (d : ℕ) → Ty
```

```
record Dataty : Type where
  field
    numCons : ℕ
    argsTy : (c : Fin numCons) → Tys
    -- ^ c : Constructor name

Tys = List Ty
```

Below is an example signature that defines a unit type and a list-like datatype.

```
-- type unit = Unit
unitDef : Dataty
unitDef .numCons  = 1
unitDef .argsTy 0F = []

-- type list = Nil | One unit | Cons unit list
listDef : Dataty
listDef .numCons  = 3
listDef .argsTy 0F = []
listDef .argsTy 1F = dataty 0 :: []
listDef .argsTy 2F = dataty 0 :: dataty 1 :: []

exampleSig : Signature
exampleSig .dataDef 0       = unitDef
exampleSig .dataDef (suc _) = listDef
```

We let t range over types and ts over lists of types. From here, we assume a single global signature is provided and refer directly to the dataDef field. We introduce

type aliases for datatype names and constructor names. We let d ranges over datatype names and c over constructor names.

```
NameData : Type                 NameCon : NameData → Type
NameData = ℕ                    NameCon d = Fin (dataDef d .numCons)
```

Values and (heterogeneous) value vectors are formalized with intrinsic typing.

```
data Value : Ty → Type where
  con : (c : NameCon d) (vs : Values (dataDef d .argsTy c))
      → Value (dataty d)

Values : Tys → Type
Values ts = All Value ts
```

The Values type is implemented using the All relation on lists, which asserts that every element in a given list satisfies a given predicate:

```
data All (P : A → Type) : (xs : List A) → Type where
  []   : All P []
  _::_ : P x → All P xs → All P (x :: xs)
```

We let u and v range over values, us and vs range over value vectors.

2.2 Patterns and Instance Relation

Patterns are also formalized with intrinsic typing. Following [9], we consider wildcard patterns, constructor patterns, and or-patterns. A pattern matrix is a list of pattern vectors, where each row corresponds to a pattern clause.

```
data Pattern : Ty → Type where
  _   : Pattern t
  con : (c : NameCon d) (ps : Patterns (dataDef d .argsTy c))
      → Pattern (dataty d)
  _|_ : (p q : Pattern t) → Pattern t

Patterns : Tys → Type                PatternMatrix : Tys → Type
Patterns ts = All Pattern ts         PatternMatrix ts = List (Patterns ts)
```

We let p, q, and r range over patterns, ps, qs, and rs over pattern vectors, and P and Q over pattern matrices.

To formalize pattern matching, we introduce a relation that specifies when a pattern matches a given value. This relation is called the *instance relation* in [9]. The definition is straightforward: a wildcard pattern matches any value of the same type; a constructor pattern matches values with the same constructor, provided each argument pattern matches the corresponding argument value; and an or-pattern matches a value if at least one of its branches matches. Note that

the instance relations are defined only when patterns and values have the same type. We define the instance relation for pattern matrices such that a value vector is related to a matrix if it matches at least one row of the matrix.

```
data _≺_ : Pattern t → Value t → Type where
  -≺    : - ≺ v
  con≺  : (is : ps ≺* vs) → con c ps ≺ con c vs
  |≺ˡ   : (i : p ≺ v) → (p | q) ≺ v
  |≺ʳ   : (i : q ≺ v) → (p | q) ≺ v

data _≺*_ : Patterns ts → Values ts → Type where
  []    : [] ≺* []
  _::_  : (i : p ≺ v) (is : ps ≺* vs) → (p :: ps) ≺* (v :: vs)

_⊀_ : Pattern t → Value t → Type
p ⊀ v = ¬ p ≺ v

_⊀*_ : Patterns ts → Values ts → Type
ps ⊀* vs = ¬ ps ≺* vs

_≺**_ _⊀**_ : PatternMatrix ts → Values ts → Type
P ≺** vs = Any (λ ps → ps ≺* vs) P
P ⊀** vs = ¬ P ≺** vs
```

Here, we use the Any relation on lists, which asserts that some element in a given list satisfies a given proposition:

```
data Any (P : A → Type) : List A → Type where
  here  : P x → Any P (x :: xs)
  there : Any P xs → Any P (x :: xs)
```

Building on the example signature, the following defines an instance relation for the example. Pattern synonyms provide convenient shorthand notation. The constructor con is overloaded for both values and patterns, allowing these pattern synonyms to be used in either context.

```
pattern unitTy = 0
pattern listTy  = 1

pattern unit       = con {d = unitTy} 0F []
pattern nil        = con {d = listTy}  0F []
pattern one x      = con {d = listTy}  1F (x :: [])
pattern cons x xs = con {d = listTy}  2F (x :: xs :: [])

_ : cons - (one -) ≺ cons unit (one unit)
_ = con≺ (-≺ :: con≺ (-≺ :: []) :: [])
```

2.3 Pattern matching

Pattern matching determines whether a given value is an instance of a specified pattern. We use the Dec datatype and its combinators to implement decision procedures for pattern matching. Here, _×_ denotes the product (pair) type, and _⊎_ denotes the sum (either) type.

```
Dec : Type → Type        yes : A → Dec A        no : ¬ A → Dec A

mapDec  : (A → B) → (B → A) → Dec A → Dec B
_×-dec_ : Dec A → Dec B → Dec (A × B)
_⊎-dec_ : Dec A → Dec B → Dec (A ⊎ B)
```

With this API and a few basic inversion lemmas, we can implement decision procedures for the instance relation.

```
con≼⁻ : con c ps ≼ con c vs → ps ≼* vs

|≼   : (p ≼ v) ⊎ (q ≼ v) → (p | q) ≼ v
|≼⁻  : (p | q) ≼ v → (p ≼ v) ⊎ (q ≼ v)

::≼⁻  : (p :: ps) ≼* (v :: vs) → (p ≼ v) × (ps ≼* vs)

sameCon : con c ps ≼ con c' vs → c ≡ c'

_≼?_ : (p : Pattern t) (v : Value t) → Dec (p ≼ v)
–       ≼? v        = yes –≼
(p | q) ≼? v        = mapDec |≼ |≼⁻ ((p ≼? v) ⊎-dec (q ≼? v))
con c ps ≼? con c' vs with c ≟ c'
... | yes refl = mapDec con≼ con≼⁻ (ps ≼*? vs)
... | no neq = no (contraposition sameCon neq)

_≼*?_ : (ps : Patterns ts) (vs : Values ts) → Dec (ps ≼* vs)
[]       ≼*? []       = yes []
p :: ps ≼*? v :: vs = mapDec (uncurry _::_) ::≼⁻ ((p ≼? v) ×-dec (ps ≼*? vs))
```

In the constructor pattern case, we use Agda's with-abstraction, which enables pattern matching on the result of an intermediate computation.

The following are examples demonstrating the decision of instance relations.

```
_ :  (cons – (one –) ≼? cons unit (one unit))
  ≡ yes (con≼ (–≼ :: con≼ (–≼ :: []) :: []))
_ = refl

_ : (cons – (one –) ≼? cons unit (cons unit nil)) ≡ no _
_ = refl
```

3 USEFULNESS

In this section, we formalize an algorithm for checking *usefulness*, as introduced in [9]. A pattern vector ps is useful with respect to a pattern matrix P if and only if there exists a value vector that is an instance of ps but not an instance of any row in P.

```
Useful : (ps : Patterns ts) (P : PatternMatrix ts) → Type
Useful ps P = ∃[ vs ] (ps ≼* vs) × (P ⋠** vs)
```

Usefulness is an important concept because its negation characterizes redundant patterns. Moreover, exhaustiveness can be defined in terms of usefulness, as we will see in Section 4.

3.1 Core algorithm

In this subsection we describe an algorithm called $\mathcal{U}_{rec}$ for checking usefulness proposed in [9]. We require the terminating pragma to define the algorithm in Agda, as Agda cannot automatically verify its termination.

```
{-# TERMINATING #-}
isUseful : (ps : Patterns ts) (P : PatternMatrix ts) → Bool
```

The algorithm proceeds by examining the head (first element) of ps at each step and solves a smaller, equivalent usefulness checking problem. It terminates when ps becomes empty.

Base case If ps has length 0 and P has no columns, isUseful returns true if P has no rows (i.e., P is a 0×0 matrix):

```
isUseful [] [] = true                    isUseful [] (_ :: _) = false
```

Or-pattern case If the head is an or-pattern, we define:

```
isUseful ((r₁ | r₂) :: ps) P = isUseful (r₁ :: ps) P ∨ isUseful (r₂ :: ps) P
```

Constructor pattern case If the head is a constructor pattern, isUseful checks usefulness for the *specialized matrix*:

```
isUseful (con c rs :: ps) P = isUseful (rs ++ ps) (specialize c P)
```

The specialization operation filters out rows whose head pattern does not match the specified constructor. Formally, it is defined as follows, where –* : Patterns ts represents the vector of wildcard patterns.

```
specialize : (c : NameCon d) (P : PatternMatrix (dataty d :: ts))
           → PatternMatrix (dataDef d .argsTy c ++ ts)
```

```
specialize c P = concatMap (specialize' c) P

specialize' : (c : NameCon d) (ps : Patterns (dataty d :: ts))
            → PatternMatrix (dataDef d .argsTy c ++ ts)
specialize' c (–          :: ps) = (–* ++ ps) :: []
specialize' c (r₁ | r₂    :: ps) = specialize' c (r₁ :: ps) ++ specialize' c (r₂ :: ps)
specialize' c (con c' rs :: ps) with c ≟ c'
... | yes refl = (rs ++ ps) :: []
... | no neq = []
```

Wildcard pattern case If the head is a wildcard pattern, the result depends on whether the set of *root constructors* of P is complete (whether it contains all possible constructors for the type). Root constructors are those that appear at the top level of the first column of P.

```
rootConSet' : Pattern (dataty d) → Set (NameCon d)
rootConSet' –          = Set.empty
rootConSet' (con c _) = Set.singleton c
rootConSet' (r₁ | r₂)  = Set.union (rootConSet' r₁) (rootConSet' r₂)

rootConSet : PatternMatrix (dataty d :: ts) → Set (NameCon d)
rootConSet []         = Set.empty
rootConSet (ps :: P) = Set.union (rootConSet' (head ps)) (rootConSet P)
```

If the set of root constructors is complete, the algorithm checks whether –* ++ ps is useful with respect to P specialized by some constructor. Otherwise, isUseful checks usefulness with respect to the *default matrix*.

```
isUseful {ts = dataty d :: ts} (– :: ps) P =
  let conSet     = rootConSet P in
  let missConSet = Set.difference Set.universal conSet in
  if Set.null missConSet
    then any (λ c → isUseful (–* ++ ps) (specialize c P)) Set.universal
    else isUseful ps (default P)
```

The default matrix operation removes all rows whose head is a constructor pattern.

```
default : (P : PatternMatrix (dataty d :: ts)) → PatternMatrix ts
default P = concatMap default' P

default' : (ps : Patterns (dataty d :: ts)) → PatternMatrix ts
default' (–          :: ps) = ps :: []
default' (con _ _ :: ps) = []
default' (r₁ | r₂    :: ps) = default' (r₁ :: ps) ++ default' (r₂ :: ps)
```

Using isUseful, we can verify that the pattern vector ps is useful with respect to P, but not with respect to Q.

```
ps = one – :: one – :: []

P = (nil    :: –      :: []) ::
    (–      :: nil    :: []) :: []

Q = (nil        :: –          :: []) ::
    (–          :: nil        :: []) ::
    (one –      :: –          :: []) ::
    (–          :: one –      :: []) ::
    (cons – –   :: –          :: []) ::
    (–          :: cons – –   :: []) :: []

_ : isUseful ps P ≡ true
_ = refl

_ : isUseful ps Q ≡ false
_ = refl
```

3.2 Correctness

With the "raw" core algorithm explained, our next goal is to implement the verified version (without relying on the terminating pragma):

```
useful? : (ps : Patterns ts) (P : PatternMatrix ts) → Dec (Useful ps P)
```

This function not only decides whether ps is useful with respect to P, but also provides a proof of usefulness or uselessness. In particular, in the yes case, we obtain a value vector that witnesses usefulness. If we only care about the boolean result, we can easily recover a boolean-returning algorithm by checking whether the result is yes or no and discarding the accompanying proof. In the implementation, we show that each step of the algorithm reduces the usefulness checking problem to an equivalent one. Termination is discussed in Section 3.3.

We first establish several properties of specialize and default.

Properties of specialization specialize preserves the instance relation, and the converse also holds.

```
specialize-preserve-⊴  : P ⊴** (con c us :: vs) → specialize c P ⊴** (us ++ vs)
specialize-preserve-⊴⁻ : specialize c P ⊴** (us ++ vs) → P ⊴** (con c us :: vs)
```

This property can be proved by induction on P.

Properties of default matrix If the default matrix of P has an instance relation with vs, then P also has an instance relation with vs with any value prepended.

```
default-preserve-⊴⁻ : default P ⊴** vs → P ⊴** (v :: vs)
```

The converse holds if the constructor of the head value is not in the set of root constructors of P. Note that any value is allowed if the root constructor set is empty.

```
_∈**_ _∉**_ : NameCon d → PatternMatrix (dataty d :: ts) → Type
```

```
default-preserve-⪯ : c ∉** P → P ⪯** (con c us :: vs) → default P ⪯** vs
```

The proof proceeds by induction on P, similarly to the proof for specialize.

At this point, we can prove that each step of the algorithm preserves usefulness.

Base case We first establish the usefulness properties for the base case. The empty pattern vector is useful with respect to the empty matrix, as witnessed by the empty value vector. On the other hand, the empty pattern vector is not useful with respect to a matrix containing one or more empty rows.

```
baseOkCase : Useful [] []
baseOkCase = [] , [] , λ ()

baseBadCase : ¬ Useful [] (ps :: P)
baseBadCase {ps = []} ([] , _ , ni) = ni (here [])
```

Or-pattern case A pattern vector headed by an or-pattern is useful if and only if at least one of its branch patterns is useful. The witnessing value vector is preserved through this step.

```
orCase : Useful (r₁ :: ps) P ⊎ Useful (r₂ :: ps) P → Useful ((r₁ | r₂) :: ps) P
orCase (inj₁ (vs , i :: is , nis)) = vs , (|⪯ˡ i) :: is , nis
orCase (inj₂ (vs , i :: is , nis)) = vs , (|⪯ʳ i) :: is , nis

orCaseInv : Useful ((r₁ | r₂) :: ps) P → Useful (r₁ :: ps) P ⊎ Useful (r₂ :: ps) P
orCaseInv (vs , (|⪯ˡ i) :: is , nis) = inj₁ (vs , i :: is , nis)
orCaseInv (vs , (|⪯ʳ i) :: is , nis) = inj₂ (vs , i :: is , nis)
```

Constructor pattern case A pattern vector headed by a constructor pattern is useful if and only if its specialized form is useful. This follows from the fact that specialize preserves the instance relation. Observe that the appropriate segment of the witnessing value vector is extracted, wrapped with the constructor, and then prepended to the rest of the witness in conCase, and that the reverse operation is performed in conCaseInv.

```
split : (ps : Patterns ts) {qs : Patterns ts'}
  → (ps ++ qs) ⪯* vs
  → ∃[ vs₁ ] ∃[ vs₂ ] (vs₁ ++ vs₂ ≡ vs) × (ps ⪯* vs₁) × (qs ⪯* vs₂)


conCase :   Useful (rs ++ ps) (specialize c P)
          → Useful (con c rs :: ps) P
conCase (vs , is , nis) with split rs is
... | vs₁ , vs₂ , refl , is₁ , is₂ =
    con c vs₁ :: vs₂ , con⪯ is₁ :: is₂ , contraposition specialize-preserve-⪯ nis
```

```
conCaseInv :   Useful (con c rs :: ps) P
             → Useful (rs ++ ps) (specialize c P)
conCaseInv (con c us :: vs , con≼ js :: is , nis) =
  us ++ vs , js ++ is , contraposition specialize-preserve-≼⁻ nis
```

Wildcard pattern case A pattern vector headed by a wildcard pattern is useful if and only if there exists a constructor such that replacing the head with a vector of wildcards of appropriate length yields a pattern vector that is useful with respect to the matrix specialized by that constructor. The proof is very similar to the constructor pattern case.

```
wildCompCase :   ∃[ c ] Useful (–* ++ ps) (specialize c P)
               → Useful {dataty d :: _} (– :: ps) P
wildCompCase (c , (vs , is , nis)) with split {dataDef d .argsTy c} –* is
... | vs₁ , vs₂ , refl , is₁ , is₂ =
    con c vs₁ :: vs₂ , –≼ :: is₂ , contraposition specialize-preserve-≼ nis

wildCompCaseInv :   Useful {dataty d :: _} (– :: ps) P
                  → ∃[ c ] Useful (–* ++ ps) (specialize c P)
wildCompCaseInv (con c us :: vs , _ :: is , nis) =
  c , (us ++ vs , –≼* ++ is , contraposition specialize-preserve-≼⁻ nis)
```

A pattern vector headed by a wildcard pattern is useful with respect to a pattern matrix if there exists a constructor that does not appear in the root constructor set of P, and the tail patterns are useful with respect to the default matrix. Here, we must assume that every type is non-empty (the non-empty axiom); if a type is uninhabited, no value matches the wildcard pattern for that type. This is a restriction of the algorithm. Given the non-empty axiom, we can always construct a value for any specified constructor, which allows us to provide the required witnessing value. The converse direction holds regardless of the root constructor set of P.

```
inhab : (ax : (t : Ty) → Value t) → (c : NameCon d) → Value (dataty d)
```

```
wildMissCase :   (ax : (t : Ty) → Value t)
               → ∃[ c ] c ∉** P → Useful ps (default P) → Useful (– :: ps) P
wildMissCase ax (c , miss) (vs , is , nis) =
  inhab ax c :: vs , –≼ :: is , contraposition (default-preserve-≼ miss) nis

wildMissCaseInv : Useful (– :: ps) P → Useful ps (default P)
wildMissCaseInv (v :: vs , i :: is , nis) =
  vs , is , contraposition default-preserve-≼⁻ nis
```

3.3 Termination

We now prove termination of the algorithm. To do this, we use Bove's method for formalizing general recursion [3]. In this approach, we define a special-purpose accessibility predicate that captures the recursion pattern of the computation in question. The computation itself is then defined by structural recursion on this predicate, which is provided as an additional parameter. Separately, we prove that the predicate holds for all inputs and supply it to the algorithm. This method allows us to clearly separate the computational content from the termination proof.

The accessibility predicate for the usefulness checking algorithm is defined as follows:

```
data UsefulAcc : (ps : Patterns ts) (P : PatternMatrix ts) → Type where
  done     :  UsefulAcc [] P
  orStep   :  UsefulAcc (r₁ :: ps) P
           → UsefulAcc (r₂ :: ps) P
           → UsefulAcc ((r₁ | r₂) :: ps) P
  conStep  :  {c : NameCon d} {rs : Patterns (dataDef d .argsTy c)}
           → UsefulAcc (rs ++ ps) (specialize c P)
           → UsefulAcc {dataty d :: ts} (con c rs :: ps) P
  wildStep :  (∀ c → c ∈** P → UsefulAcc (–* ++ ps) (specialize c P))
           → (∃[ c ] c ∉** P → UsefulAcc ps (default P))
           → UsefulAcc {dataty d :: ts} (– :: ps) P
```

Note that this predicate simply mirrors the recursive structure of the algorithm. At this point, we are very close to obtaining the verified usefulness checking algorithm we set out to construct. Compared to the raw algorithm presented in Section 3.1, the following version differs in that it uses the usefulness properties established earlier to guarantee correctness, and is parametrized by the accessibility predicate to ensure termination.

```
completeOrMissing : (P : PatternMatrix (dataty d :: ts))
  → (∀ c → c ∈** P) ⊎ (∃[ c ] c ∉** P)


useful?' : (ax : (t : Ty) → Value t) (ps : Patterns ts) (P : PatternMatrix ts)
  → UsefulAcc ps P
  → Dec (Useful ps P)
useful?' _ [] []          done = yes baseOkCase
useful?' _ [] (ps :: P) done = no baseBadCase
useful?' ax (r₁ | r₂ :: ps) P (orStep acc acc') =
  mapDec orCase orCaseInv
    (useful?' ax (r₁ :: ps) P acc ⊎-dec useful?' ax (r₂ :: ps) P acc')
useful?' ax (con c rs :: ps) P (conStep acc) =
  mapDec conCase conCaseInv
    (useful?' ax (rs ++ ps) (specialize c P) acc)
```

```
useful?' {ts = dataty d :: _} ax (– :: ps) P (wildStep acc acc')
  with completeOrMissing P
... | inj₁ comp = mapDec wildCompCase wildCompCaseInv
      (anyFin? λ c → useful?' ax (–* ++ ps) (specialize c P) (acc c (comp c)))
... | inj₂ miss = mapDec (wildMissCase ax miss) wildMissCaseInv
      (useful?' ax ps (default P) (acc' miss))
```

The only remaining task is to prove that the accessibility predicate holds for all possible inputs:

```
∀UsefulAcc : (ps : Patterns ts) (P : PatternMatrix ts) → UsefulAcc ps P
```

To prove this, we need to define a suitable measure on the inputs. Finding such a measure is tricky and challenging because:

- The measure must strictly decrease in cases where the wildcard at the head is replaced by multiple wildcards. This occurs in the first branch of the wildcard pattern case and when specialize encounters a wildcard pattern at the first column.
- The measure must also strictly decrease when the first column of P contains or-patterns, which causes specialize and default to duplicate the remaining columns.

Thus, several standard measures including the total size of patterns in the matrix fail. The measure we managed to found is the following lexicographic measure:

1. The combined size of ps and P, where size is measured by expanding all or-patterns in both and not counting wildcard patterns.
2. The length of the pattern vector ps.

The second component is necessary because the first component may not strictly decrease in the second branch of the wildcard pattern case. The second component guarantees strict reduction in this scenario, since the wildcard pattern at the head is removed. Using this measure, we can prove that ∀UsefulAcc holds by well-founded induction, and thus we can finally implement useful?:

```
useful? : (ax : (t : Ty) → Value t) (ps : Patterns ts) (P : PatternMatrix ts)
  → Dec (Useful ps P)
useful? ax ps P = useful?' ax ps P (∀UsefulAcc ps P)
```

We can verify that ps (from the isUseful example) is indeed useful with respect to P, as witnessed by one unit :: one unit :: []. However, it is not useful with respect to Q:

```
_ : useful? nonEmpty ps P ≡ yes (one unit :: one unit :: [] , _ , _)
_ = refl

_ : useful? nonEmpty ps Q ≡ no _
_ = refl
```

In this example, we only have the unit type and the unit list type, which allows us to define nonEmpty.

4 EXHAUSTIVENESS

The concept of exhaustiveness for a pattern matrix can be derived from usefulness. A pattern matrix is exhaustive if, for every value vector, there exists a row in the matrix that matches it. Conversely, a pattern matrix is non-exhaustive if there exists a value vector that does not match any row in the matrix.

```
Exhaustive NonExhaustive : (P : PatternMatrix ts) → Type
Exhaustive    P = ∀ vs → Any (λ ps → ps ≼* vs) P
NonExhaustive P = ∃[ vs ] All (λ ps → ps ⋠* vs) P
```

In fact, this is equivalent to checking whether the vector of wildcard patterns is useful with respect to the given pattern matrix:

```
Exhaustive' NonExhaustive' : (P : PatternMatrix ts) → Type
NonExhaustive' P = Useful –* P
Exhaustive'    P = ¬ NonExhaustive' P

convNonExhaustive    : NonExhaustive' P → NonExhaustive P
convNonExhaustiveInv : NonExhaustive P → NonExhaustive' P
convExhaustive       : Exhaustive' P → Exhaustive P
convExhaustiveInv    : Exhaustive P → Exhaustive' P
```

This equivalence allows us to define an exhaustiveness checking algorithm based on the verified usefulness checking algorithm:

```
exhaustive? : (ax : (t : Ty) → Value t) (P : PatternMatrix ts)
  → Exhaustive P ⊎ NonExhaustive P
exhaustive? ax P with useful? ax –* P
... | yes h = inj₂ (convNonExhaustive h)
... | no h  = inj₁ (convExhaustive h)
```

We can verify that the example pattern matrix Q is exhaustive, while P is not. This is witnessed by the counterexample one unit :: one unit :: [].

```
_ : exhaustive? nonEmpty Q ≡ inj₁ (the (∀ vs → Q ≼** vs) _)
_ = refl

_ : exhaustive? nonEmpty P ≡ inj₂ (one unit :: one unit :: [] , _)
_ = refl
```

5 EXTRACTION

The formalization becomes more practical when it can be executed. To generate working Haskell code, we adapted our Agda formalization to be compatible with agda2hs [4], a tool that extracts readable Haskell programs from Agda code.

The agda2hs tool uses the erasure annotation `@0` in Agda to determine which parts of the program should be preserved in Haskell and which parts are only needed for proofs in Agda and can be erased. For example, the finite number datatype with erasure-annotated indices on the left translates to the Haskell datatype on the right:

```
data Fin : @0 ℕ → Type where
  Zero : ∀ {@0 n} → Fin (suc n)
  Suc : ∀ {@0 n} → Fin n → Fin (suc n)
```

```
data Fin
  = Zero
  | Suc Fin
```

Agda ensures that computations do not depend on these erased arguments.

We carefully annotated our formalization with erasure, for example on the indices of values and patterns. As a result, our formalization compiles to the following Haskell code:

```
1  data Ty = TyData Name
2  type Tys = [Ty]
3
4  data Value = VCon Name Values
5  data Values = VNil | VCons Value Values
6
7  data Pattern = PWild | PCon Name Patterns | POr Pattern Pattern
8  data Patterns = PNil | PCons Pattern Patterns
9
10 newtype Useful = MkUseful{witness :: Values}
11
12 decUseful :: Signature -> (Ty -> Value) -> Tys -> [Patterns] ->
       Patterns -> DecP Useful
```

Notice that the syntax translates to standard Haskell datatypes. Also, observe that the resulting usefulness checking function has a clear type signature, and only the witnessing value vectors are returned from the function.

6 RELATED WORK

Although coverage checking is a well-studied problem, there has been little work on formalizing coverage checking algorithms within theorem provers. The only related work we found is a concurrent effort by Cohen [5][2]. Cohen developed a Rocq mechanization of pattern matching compilation to decision trees and first-order ADT axiomatization. Cohen's approach compiles a pattern matrix using an algorithm based on Maranget [8, 9], making it very similar to our coverage checking algorithm. In fact, pattern compilation and coverage checking are closely related; compilation to decision trees inherently provides exhaustiveness checking, as a non-exhaustive pattern matrix cannot be compiled into a decision tree. However, Cohen's formalization only proves that compilation fails if the pattern matrix is non-exhaustive, because this direction is sufficient for their purpose. It does not prove the converse, i.e., the pattern matrix is non-exhaustive if compilation fails, so it only provides a weak version of exhaustiveness checking.

[2] Cohen's paper appeared after the submission of the present work.

In contrast, we proved both directions by formalizing coverage checking as a decision procedure. Moreover, our algorithm is witness-producing, which is useful for displaying coverage errors to programmers.

Other than Cohen's work we could not find mechanization of full-fledged coverage checker, even in verified compilers. CakeML is a verified compiler for a substantial subset of StandardML [7]. According to Cohen's paper, CakeML implements a simple and restricted exhaustiveness checker. However, in our own testing, we were unable to observe it functioning as described. Agda Core is a work-in-progress core language for Agda, implemented in Agda [1]. It features a derivation-producing type checker compatible with agda2hs, from which we have learned techniques for utilizing agda2hs. Currently, Agda Core appears to lack a verified coverage checker. Our formalization of a coverage checker may not be directly applicable to Agda Core, as our algorithm only handles simple patterns and does not account for dependent types. However, we believe that this work serves as a foundation for formalizing more practical coverage checking algorithms.

7 CONCLUSIONS

In this paper, we described an Agda formalization of a coverage checking algorithm by Maranget [9]. The final, verified version of the usefulness checking algorithm not only determines usefulness but also provides a proof of usefulness, including the witnessing value vector. Our formalization is compatible with agda2hs, allowing it to compile to readable, working Haskell programs.

Our formalization is still in progress, and there are several future directions we intend to pursue. First, we plan to implement an exhaustiveness checking algorithm that reports all missing cases, not just a single witnessing value, as OCaml and Rust's exhaustiveness checkers do. This can be achieved by adjusting the definition of usefulness so that it consists of a list of witnessing pattern vectors, a proof that all witnessing pattern vectors are subsumed by the useful pattern vector in question, and a proof that all witnessing pattern vectors are disjoint from all rows in the pattern matrix. Second, we plan to implement practical extensions of the algorithm. One extension we would like to implement is proposed in Maranget's paper, which can identify which particular branches of or-patterns are useless. Another is an optimization used in Rust, where exhaustiveness of pattern clauses and redundancy of each branch can be computed simultaneously, rather than invoking the original algorithm for each checking problem. Third, it would be interesting to formalize more expressive algorithms, such as the one used in Haskell [6].

Acknowledgements

The authors would like to thank the anonymous reviewers for their feedback on the draft of this paper. This work was supported by JST SPRING, Japan

Grant Number JPMJSP2180, and in part by JSPS KAKENHI Grant Numbers JP22K11967 and JP24K14892.

References

1. Agda Core Contributors: Agda core. https://github.com/jespercockx/agda-core (2025)
2. Agda Development Team: Agda 2.7.0 documentation. https://agda.readthedocs.io/en/v2.7.0/ (2024)
3. Bove, A., Capretta, V.: Modelling general recursion in type theory. Math. Struct. Comput. Sci. 15(4), 671–708 (Aug 2005)
4. Cockx, J., Melkonian, O., Escot, L., Chapman, J., Norell, U.: Reasonable agda is correct haskell: writing verified haskell using agda2hs. In: Proceedings of the 15th ACM SIGPLAN International Haskell Symposium. ACM, New York, NY, USA (Sep 2022)
5. Cohen, J.M.: A Mechanized First-Order Theory of Algebraic Data Types with Pattern Matching. In: Forster, Y., Keller, C. (eds.) 16th International Conference on Interactive Theorem Proving (ITP 2025). Leibniz International Proceedings in Informatics (LIPIcs), vol. 352, pp. 5:1–5:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2025), https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2025.5
6. Graf, S., Peyton Jones, S., Scott, R.G.: Lower your guards: a compositional pattern-match coverage checker. Proc. ACM Program. Lang. 4(ICFP), 1–30 (Aug 2020)
7. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. SIGPLAN Not. 49(1), 179–191 (Jan 2014)
8. Le Fessant, F., Maranget, L.: Optimizing pattern matching. In: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming. ACM, New York, NY, USA (Oct 2001)
9. Maranget, L.: Warnings for pattern matching. J. Funct. Programming 17(3), 387–421 (May 2007)
10. Rust Development Team: rustc_pattern_analysis::usefulness - rust. https://doc.rust-lang.org/nightly/nightly-rustc/rustc_pattern_analysis/usefulness/index.html (2025)