On Static Analysis for Expressive Pattern Matching

Mirco Dotta, Philippe Suter and Viktor Kuncak*

School of Computer and Communication Sciences, EPFL, Switzerland
{firstname.lastname}@epfl.ch

Abstract. Pattern matching is a widespread programming language construct that enables definitions of values by cases, generalizing if-then-else and case statements. The cases in a pattern matching expression should be exhaustive: when the value does not match any of the cases, the expression throws a run-time exception. Similarly, each pattern should be reachable, and, if possible, patterns should be disjoint to facilitate reasoning. Current compilers use simple analyses to check patterns. Such analyses ignore pattern guards, use static types to approximate possible expression values, and do not take into account properties of user-defined functions.

We present a design and implementation of a new analysis of pattern matching expressions. Our analysis detects a wider class of errors and reports fewer false alarms than previous approaches. It checks disjointness, reachability, and exhaustiveness of patterns by expressing these conditions as formulas and proving them using decision procedures and theorem provers. It achieves precision by propagating possible values through nested expressions and approximating pattern-matching guards with formulas. It supports user-defined "extractor" functions in patterns by relying on specifications of relationships between the domains of such functions. The result is the first analysis that enables verified, declarative pattern matching with guards in the presence of data abstraction. We have implemented our analysis and describe our experience in checking a range of pattern matching expressions in a subset of the Scala programming language.

1 Introduction

Pattern matching is a widely used conditional construct in programming languages. It specifies a computation as a sequence of tests on the shape of a given value and binds variables to value components if the test succeeds. Programming languages often support pattern matching on values of algebraic data types, which facilitates reasoning about programs while preserving their efficient execution. These constructs are among central features of modern functional

* This project was supported by the Swiss National Science Foundation Proposal #120433 "Precise and Scalable Analyses for Reliable Software"

programming languages [3, 21, 18, 2] and are often credited with increased programmer productivity (e.g., [22]). Moreover, recent research incorporates pattern matching into object-oriented programming languages [19,8,23,29], showing how to reconcile it with subtyping and data abstraction, and pointing out scenarios where it is more convenient than dynamic dispatch and its generalizations [20].

In this paper we describe a new static analysis for pattern matching. Its distinguishing features are

1. Verification of constraints on values (such as integers and objects) and not just tree-like constraints on shape. Our analysis takes into account pattern guards, conditionals, procedure preconditions and postconditions. The tracking of predicates on values differentiates our analysis from many approaches that focus on tree patterns including the work on statically checkable pattern abstractions [10];
2. Support for specification and verification of user-defined patterns. Following [8], this approach enables developers to provide multiple implementation-independent views of same data though extractor functions. In our system, however, developers can also specify the partitioning properties for such views. Our analysis checks reachability, exhaustiveness, and disjointness of patterns that use such views, as well as the fact that the partitioning properties are in fact satisfied by the bodies of extractor functions.

We also discuss our experience with a prototype implementation [5] of our pattern-matching analysis. To check the desired properties, our system first encodes them as logical formulas and then attempts to prove them using a system that combines decision procedures and theorem provers, which was originally developed for the Jahob verification system [16]. Our system accepts programs in a first-order functional subset of Scala [25, 1, 8] and checks exhaustiveness, disjointness, reachability of patterns, as well as any auxiliary specifications that the developer introduced to make the checking of patterns possible. In addition to Scala, we expect our techniques to be useful for other programming languages such as F# [28, 29].

Contributions. We summarize the contributions of this paper as follows:

- We describe an approach for checking patterns in the presence of guards, user-defined functions, data abstraction and information hiding.
- We show how to implement this approach by reducing pattern matching questions to validity of formulas.
- We present a prototype implementation of this approach and its evaluation on a series of examples that illustrate a range pattern-matching scenarios. Our results suggest that the approach is useful in improving software reliability and that the performance of the analysis is acceptable.

2 Overview of Our Approach

In this section we continue introducing features of our analysis through examples and discuss our approach to pattern specification and verification.

2.1 Verifying Constraints on Values

```
sealed abstract class Tree {
  def contains(key: Int):Boolean = (this:Tree) match {
    case Empty() ⇒ false
    case Node(_ , value:Int, _ ) if value == key ⇒ true
    case Node(_ , value:Int, right:Tree) if value < key ⇒ right.contains(key)
    case Node(left:Tree, value:Int, _ ) if value > key ⇒ left.contains(key)
  }
}
class Empty extends Tree
class Node(val l: Tree, val v: Int, val r: Tree) extends Tree
```

Fig. 1. Binary search tree

The simple Scala example in Figure 1 shows one of the ways in which our analysis goes beyond the standard compiler warnings. The example introduces an algebraic data type Tree, which in an ML-like language would be given by a recursive type declaration `Tree = Leaf | Node of Tree * int * Tree`. The figure also defines the membership function contains, which uses binary search to test if the given key is stored in the tree. The function performs pattern matching on the tree denoted by the receiver object **this**. The function considers four cases: an empty tree, a node storing the desired key, a node storing a smaller key and a node storing a larger key. Each case is expressed using a **case** construct inside the **match** expression. It consists of the expression identifying the shape of the value and binding variables to its components, optionally followed by the **if** keyword and a boolean *guard* expression that imposes an additional condition for the case to apply. Note that, for a given tree, exactly one of the four cases in Figure 1 applies.

Pattern matching as a declarative construct. There are several desirable properties that a pattern matching expression should satisfy:

- **exhaustiveness:** for every value being matched, *at least one* of the cases should apply;
- **disjointness:** for every value being matched, ideally, *at most one* of the cases should apply;
- **reachability:** no case should be subsumed by previous ones, that is, no case should represent unreachable code.

Most current compilers attempt to check exhaustiveness and reachability for patterns expressed using solely structural constraints (such as Node(...)), but do not attempt any checks that also involve guards (such as value < key). Compilers typically do not emit any warnings for disjointness, although they may derive such information internally (again, ignoring guards) to optimize pattern matching code. Typically, the compiler conservatively requires the case with a guard to be covered by cases that have no guards, which results in false warning in examples such as Figure 1. Consequently, the declarative spirit of the pattern

matching construct is not ensured statically and programs risk the possibility of run-time errors or unintended meaning of the code. Programmers become encouraged to write patterns that rely on sequential evaluation order and insert catch-all cases that have no visible specification of when they apply, preventing reordering of patterns and increasing the risk of difficult-to-find semantic errors. The checks made possible by our analysis reestablish the declarative status of the pattern matching by enforcing desired pattern matching properties even in the presence of guards and other constraints on program values.

Handling guards using theorem provers and decision procedures. Guards are expressions of the underlying programming language and can contain, e.g., arithmetic operations, field dereference, and function calls. This generality makes questions about guards difficult and is perhaps the main reason why compilers ignore them. The analysis we present in this paper overcomes this limitation by encoding pattern matching questions as logical formulas, then using theorem provers and decision procedures to prove these formulas. Our approach is therefore related to, e.g., ESC/Java [11], but we implemented it in a simpler, first-order purely functional language fragment. We believe that the focus on functional subset is appropriate given that patterns are generally required to be side-effect-free in order to enable optimizations [7].

An important question when using theorem provers in verification is the behavior of the system on typical examples that arise in practice. As a step towards answering this question, we wrote or adapted several examples that demonstrate various cases where questions about patterns hold and do not hold (Section 5). We found that our system was able to quickly prove generated valid formulas by applying Jahob's theorem proving engine [16] to reduce formulas to the input to the CVC3 prover [14]. The experience with these examples suggests that our approach is effective in practice.

Tracking constraints. Because our analysis tracks not only types but also properties on values, it must take into account program context in which a pattern-matching expression occurs. Consider the example in Figure 2. Note that, when the inner **match** expression evaluates, key != value holds, because the guard of the outer pattern succeeded. Therefore, the two cases in the inner **match** expression are exhaustive, even though they would not be exhaustive in general. A similar situation would arise for a **match** expression occurring inside a standard **if** statement. Our system correctly handles such scenarios by generating

```
def contains(key: Int): Boolean = (this : Tree) match {
    case Node(_,value: Int, _) if key != value ⇒
      (this : Tree) match {
        case Node(left: Tree,value: Int, _) if key < value ⇒ left.contains(key)
        case Node(_,value:Int, right: Tree) if key > value ⇒ right.contains(key)
      }
    case Node(_,value: Int, _) if key == value ⇒ true
    case Empty() ⇒ false
}
```

Fig. 2. A version of binary search with nested patterns

formulas that maintain constraints on values at each program point and using these constraints when checking pattern properties.

2.2 Verified User-Defined Patterns

Our analysis supports data abstraction and user-defined patterns, viewing simple algebraic patterns as a particular case of user-defined patterns. The example in Figure 3 illustrates user-defined patterns in Scala based on extractor (unapply) functions [8, 7] and shows how to specify properties of such functions in our system.

Unapply functions. The example in Figure 3 introduces the list data type denoted by the Lst class, then defines both the usual decomposition with Cons pattern and the "backward" decomposition using the Snoc pattern.

```
/* domain Dom_Nill = Nill & Dom_Cons = Cons & Dom_Snoc = Cons &
            Dom_Cons \Un Dom_Nill = Lst & Dom_Cons \Int Dom_Nill = empty
            Dom_Snoc \Un Dom_Nill = Lst & Dom_Snoc \Int Dom_Nill = empty */
sealed abstract class Lst
class Cons(val head: Elem, val tail: Lst) extends Lst
class Nill extends Lst
class Elem
object Nill {
  def apply(): Nill = new Nill()
  def unapply(n: Nill): Boolean = true
}
object Cons {
  def apply(head: Elem, tail: Lst): Cons = new Cons(head,tail)
  def unapply(c: Cons): Option[(Elem,Lst)] = Some(Tuple2(c.head,c.tail))
}
object Snoc {
  def unapply(c: Cons): Option[(Lst,Elem)] = (c : Cons) match {
    case Cons(c : Elem, xs: Lst) ⇒ (xs : Lst) match {
      case Nill() ⇒ Some(Tuple2(Nill(),c))
      case Snoc(ys: Lst, y: Elem) ⇒ Some(Tuple2(Cons(c,ys),y))
    }
  }
}
...
(lst: Lst) match {
    case Nill() ⇒ lst
    case Cons(x: Elem, Nill()) ⇒ lst
    case Cons(x: Elem, Snoc(_,y: Elem)) ⇒ Cons(x,Cons(y,Nill())) }
```

Fig. 3. Two views of a list in our system

The standard decomposition uses Nill and Cons patterns. Cons(x,y) succeeds whenever the list is non-empty. When it succeeds, it binds x to the first element of the list and y to the rest of the list. In Scala, the programmer can define the Cons pattern as the unapply partial function of the Cons object, as in Figure 3 (we denote this function Cons.unapply). To represent the fact that Cons(x,y)

pattern match succeeds on value v and binds (x,y) to values (a,b), Scala uses the condition Cons.unapply(v)=Some(a,b). To represent that the pattern match fails, Scala uses Cons.unapply(v)=None. Each pattern therefore translates into unapply calls. The definition of the Cons.unapply function is a straightforward pair of field dereferences. The formal parameter of Cons.unapply has the type of Cons class, which means that a pattern match will fail on an object that is not a subtype of Cons. The simple Nill pattern consists solely of the type test on the Nill class. The **sealed** keyword implies that Cons and Nill are the only subclasses of Lst.

Domain constraints for specifying defined patterns. The bodies of unapply functions are arbitrary pure computations. How can a compiler check exhaustiveness, disjointness and reachability of such user-defined patterns? In Figure 3 example, for every non-null Lst object, exactly one of the two functions Nill.unapply, Cons.unapply return Some(...), but this need not be the case in general. The compiler clearly needs additional information on the behavior of unapply functions. However, exposing function bodies would violate the abstraction and prevent modular checking. Our system solves this tension by introducing **domain** specifications for unapply functions, expressed as special comments at the beginning of code in Figure 3. For each user-defined pattern P.unapply we introduce the set Dom_P of all values for which the pattern succeeds, that is, for which P.unapply returns Some(...). The **domain** declaration is a conjunction of set inclusion and equality constraints among such set expressions built from domain sets and the sets representing classes. The key piece of information that our system needs to check pattern matching on lists is that the two sets Dom_Nill, Dom_Cons form a partition of Lst, expressed in Figure 3 by the constraint that their intersection is empty and their union is Lst.

Our system uses domain constraints to check pattern matching expressions without the need to access function bodies. On the other hand, as the part of verifying data type implementation, our system ensures that the implementations of unapply methods satisfy these domain constraints. Our system therefore supports assume-guarantee reasoning with domains as a simple form of algebraic specifications [15]. Because they are simple and directly capture the properties needed to analyze pattern matching, we believe that our **domain** constraints are more appropriate for our system than the alternative of using specification variables (model fields) [16, 17].

Multiple views. Having seen how to define the standard view of the list, consider now the alternative "backward" list decomposition given by the Snoc pattern in Figure 3. The Snoc pattern splits a non-empty list into its initial prefix and its last element. Note that Snoc satisfies the same **domain** specification as Cons. In fact, the domains of both Snoc.unapply and Cons.unapply are equal to the Cons set, the set of all (non-null) objects of the Cons class.

Our system proves that all **domain** constraints in this example hold using a theorem prover. When verifying that the recursive Snoc body conforms to the specification Snoc = Cons, the system uses a built-in inductive rule. The system can then use these **domain** constraints to check the uses of patterns, for example,

the final expression in Figure 3. The system successfully detects that the three patterns in this expression are exhaustive, disjoint, and reachable.

Contracts and invariants for more complex properties. Our system can also check function preconditions and postconditions. It also checks invariants as part of the verification of constructors, because in our functional language class invariants can never be violated once they are established by the constructor. Figure 4 shows fragment from an example that we checked in our system and which was originally part of a class assignment.

```
/* domain Dom_Const = Const & Dom_Var = Var &
            Dom_Abstr = Abstr & Dom_App = App */
sealed abstract class Term
class Const extends Term
class Var(val index: Int) extends Term
class Abstr(val t: Term) extends Term
class App(val t1: Term, val t2: Term) extends Term
class Eval {
  def isValue(t: Term): Boolean = /* postcondition res ↔ t \in Abstr */
    t.isInstanceOf[Abstr]
  def step(t: Term): Term = (t : Term) match {
    case Var(_) ⇒ t
    case Const() ⇒ t
    case Abstr(t1: Term) ⇒ Abstr(this.step(t1))
    case App(v1: Abstr, t2: Term) if !this.isValue(t2) ⇒ App(v1, this.step(t2))
    case App(t1: Term, t2: Term) if !this.isValue(t1) ⇒ App(this.step(t1), t2)
    case App(Abstr(t1:Term),v2:Term) if this.isValue(v2) ⇒ this.subst(t1,1,v2)
  }
}
```

Fig. 4. A pattern matching expression in a lambda evaluator

The code Figure 4 shows an evaluation step function on lambda calculus terms represented in de Bruijn notation. To determine which lambda subterm to evaluate, the **match** expression in step uses a guard containing isValue function call. Our system proves these patterns with guards exhaustive, disjoint, and reachable. In this process it crucially relies on the postcondition of isValue. The system also checks that isValue implementation satisfies its postcondition. Already at the examples of this complexity we find that checking pattern properties manually is very unreliable (e.g., it was non-obvious to us that this pattern matching is exhaustive or disjoint). The confidence that our system gave us was valuable during program development and transformations such as reordering the cases. The postcondition in this example is admittedly simple. In general, we find that pattern matching properties are often of local nature, so in many cases we need few specifications beyond the **domain** constraints.

Recursive type refinement invariants. Type refinements [12] are a useful class of constraints that impose a recursive restriction on the subset of values of a given type. For example, a simplifying transformation in the implementation of our analysis system replaces each formula syntax tree of the form $(P \to Q)$

with the tree $((\neg P) \vee Q)$. Ideally, subsequent pattern matching on such transformed formula should not need to cover the case of implication ($\rightarrow$). Our system supports such examples by allowing developers to introduce of new sets of objects (denoting, for example, formula syntax trees without implication node), then use these sets in postconditions of constructors, patterns, and other functions. We have used this approach when specifying Propositional Logic example summarized in Section 5.

2.3 Related Work

User defined patterns and views have been proposed before [31, 26, 7, 8]. What distinguishes our analysis from previous ones is the simultaneous support for 1) data abstraction with information hiding (through **domain** constraints), and 2) analysis of guards that express constraints on values (such as linear arithmetic and uninterpreted function symbols).

Active patterns [29] are an extension to F# to allow pattern-matching over abstract data types, in a way very similar to unapply functions. Contrary to unapply functions, however, active patterns are categorized into several pattern classes, depending on the sets of values they match. Some of these ("total patterns") allow compile-time checks for completeness and redundancy, while for others ("partial" and "parameterized patterns"), no way of performing such checks is proposed.

Predicate dispatch [9] was proposed as a unification of multiple dispatching mechanism, including functional-style pattern matching. [20] presents an implementation as an extension to Java, and demonstrates how it can be used to emulate a reasonably large subset of pattern-matching expressions. The system presented is general enough to allow mathematical expressions and comparisons in guards. Completeness and disjointness of the predicate guards are both required and verified and compile-time, using a predecessor of CVC3 that we also employ in our analysis. As in dynamic dispatch, pattern-matching can only be performed at the method level, so nested patterns are not supported. JMatch also introduces pattern matching into Java [19], but does not address completeness of disjointness of patterns.

Refinement types [13] enable specification of subsets of values of algebraic data types given by sets recognized by tree automata. A similar approach was used in [10] to introduce pattern abstractions that are similar to Scala's unapply functions that we use in our work. These abstractions however do not take guards into account, but could be extended in this direction [6].

[30] describes a translation of a purely functional language with pattern matching into Isabelle [24]. This approach could also enable checking pattern matching properties, given an automated approach for proving Isabelle formulas. Our system directly generates formulas specialized to checking pattern matching properties and we believe that this was important for being able to prove such formulas automatically.

3 Generating Formulas for Pattern Matching Properties

Our system checks the following properties:

- for each **match** expression it checks exhaustiveness, disjointness, and reachability;
- for each domain constraint it checks that the corresponding unapply functions satisfy it;
- for each function body, it checks that its postcondition holds, and that the preconditions and constructor invariants hold for each function call within the body.

Our system checks each of these properties by generating corresponding formulas and trying to prove them. We next describe the process of generating these formulas.

3.1 Exhaustiveness, Disjointness, and Reachability

Consider a matching expression with n patterns of the following form:

(s: T) **match** {
  case $p_1 \Rightarrow \ldots$
  case $p_2 \Rightarrow \ldots$
  ...
  case $p_n \Rightarrow \ldots$
}

For each pattern p_i, let $\xi(s, p_i)$ denote that the pattern p_i matches the scrutinee s. Then matching is

- exhaustive iff $\bigvee_{1 \le i \le n} \xi(s, p_i)$ holds;
- disjoint iff $\bigwedge_{1 \le i < j \le n} \neg(\xi(s, p_i) \wedge \xi(s, p_j))$ holds;
- reachable if $\bigvee_{1 \le i \le n} \big((\bigwedge_{1 \le j < i} \neg\xi(s, p_j)) \rightarrow \neg\xi(s, p_i) \big)$ does not hold.

Denote any of the above three properties by formula P. Such property P by itself typically does not contain enough information to be provable. What is missing is 1) global information about the program, denoted by G, and 2) local information about the conditions that hold at the entry of the pattern matching expression, denoted by L. The formula we generate is therefore of the form $(G \wedge L) \rightarrow P$. We check the validity of this formula for exhaustiveness and disjointness, and check its non-validity for reachability.

3.2 Translating Structural Patterns

Consider the case p_i of the form (**case** t **if** g)). The formula $\xi(s, p_i)$ consists of 1) the part describing t (the structural part of the pattern, and 2) the part describing the guard g. We next describe the translation of t.

There are three kinds of structural patterns:

1. **wildcard**, which is a placeholder imposing no constraints;
2. **variable pattern**, which binds a variable to a value if it has the specified type;
3. **class pattern**, which corresponds to an unapply call.

The system generates true as the constraint for wildcards, and generates simple set-membership atomic formula $x \in T$ for variable pattern $x : T$.

To translate a class pattern, the system encodes a call to unapply method. The system also assumes the postcondition of the unapply method. It handles nested patterns by introducing fresh variables to denote intermediate results during pattern matching computation. The generated formula has the form of an implication; the bindings for fresh variables appear on the left-hand side of the implication. When proving formula validity, these fresh variables are universally quantified, ensuring the desired meaning.

3.3 Translating Guards

The second part of $\xi(s, p_i)$ is the translation of guards, which relies on the translation of expressions in our functional programming language into formulas of our logic. Figure 5 shows this translation. It proceeds recursively on the structure of the expression. It replaces function calls with uninterpreted function symbol applications and assumes the postconditions of called functions.

As in the translation of nested patterns, the translation requires introducing fresh variables for subexpressions and binding them appropriately on the left-hand side of the implication. The auxiliary function Class denotes the class in which a method or field is declared.

3.4 Local Conditions

Our system builds formula L containing local information about values at a given program point by traversing syntax tree of function body and accumulating conditions in branching statements that lead to this program point. When translating patterns and guards it reuses the translation for generating $\xi(s, p_i)$ formulas.

3.5 Axioms

Our system builds formula G containing global information about the program as a conjunction of the following kinds of formulas.

Subtyping. Our system represents information about Scala types using sets of objects. Type membership is encoded as set membership. Immediate subclasses of a given class are represented as subsets, which are disjoint if the class is abstract and whose union is the superclass if the class is sealed.

Class fields. Our system models class fields as uninterpreted function symbols mapping objects into values of appropriate type. We use universally quantified axioms to encode the types of fields.

```
def expr2formula(e: Expression): (Formula,Alias) = e match {
    case se.isInstanceOf[A] ⇒
        val (f_se,se_obj) = expr2formula(se)
        return (f_se ∧ (fresh_v = (se_obj ∈ A)) , fresh_v)

    case new A(arg) ⇒
        val (f_arg,res_arg) = expr2formula(arg)

        val form = (fresh_A ∈ A) ∧ invariant(A) ∧
                   f_arg ∧ A_getField_arg(fresh_A) = res_arg
        return (form , fresh_A)

     case se.m(arg) ⇒
       val (f_se,se_obj) = expr2formula(se)
       val (f_arg,res_arg) = expr2formula(arg)
       var form = f_se ∧ f_arg

       form = form ∧
                [res ↦ fresh_res, this ↦ se_obj, arg ↦ res_arg] postcondition(m) ∧
                fresh_res = Class(m)_m(se_obj,res_arg)
       return (left , fresh_res)

     case se.v ⇒
       val (f_se,se_obj) = expr2formula(se)
       val form = f_se ∧ (fresh_v = Class(v)_getField_v(se_obj))
       return (form , fresh_v)

     case e1 binop e2 ⇒
       (f_e1,res_e1) = expr2formula(e1)
       (f_e2,res_e2) = expr2formula(e2)
       return (f_e1 ∧ f_e2 ∧ (fresh_res = (res_e1 binop res_e2)) , fresh_res)

     case L: Literal ⇒ return (fresh_L = L , fresh_L)
     case C: Constant ⇒ return (fresh_C = C , fresh_C)
     case x : Identifier ⇒ return (fresh_X = x , fresh_X)
}
```

Fig. 5. Translating expressions into formulas

Extractors. Our system also introduces axioms that encode information about return types of extractors.

Domains. The system assumes the **domain** conditions as part of global axioms. These conditions are key for proving properties of pattern matching in our approach.

Constraint. The developer can introduce additional axioms into our system using the **constraint** declaration. We currently use this feature to state axioms that implicitly define sets that correspond to type refinements [12].

3.6 Checking Domain Constraints

To check a domain constraint C, our system computes a formula $A_P(x)$ characterizing Dom_P for each P.unapply function.

In most cases the system then emits a set comprehension of the form Dom_P $= \{x.A_P(x)\}$ for each variable Dom_P occurring in C and attempts to prove C. This approach leaves the problem of unfolding the definitions of Dom_P to the theorem prover, which is beneficial when one simple constructor calls another one. In the case of recursive P.unapply function, however, our system explicitly unfolds the definition of Dom_P to be able to assume the domain constraint for recursive invocations of P.unapply.

Once it proves a domain constraint conjunct, the system assumes it when proving subsequent conjuncts (as well as when proving pattern matching properties).

3.7 Checking Postconditions, Preconditions, and Invariants

To express validity of preconditions, postconditions and invariants, our system generates verification conditions using standard techniques. This task is simplified by the fact that our language is purely functional and contains no loops. The system inlines function contracts at function call sites, retaining modular analysis approach and avoiding inference of invariants over statically unbounded paths.

4 Proving Formula Validity

Proving validity of generated formulas is a crucial step in our analysis. Fortunately, we were able to rely on previously developed infrastructure and tools for this step. We have used formDecider, a theorem proving engine originally developed for the Jahob verification system [16]. In our particular application the task of this engine was to eliminate set operations and comprehensions using quantifiers, then invoke the CVC3 prover [14] which performed the main part of the reasoning task. We found this approach to be effective, as discussed in the next section.

5 Evaluation

This section summarizes our experience with a prototype system [5] that implements our analysis. Our system checks pattern matching properties (exhaustiveness, disjointness, reachability), the correctness of implementation of the corresponding domain constraints, as well as preconditions, postconditions, and invariants that we found useful when enforcing pattern matching properties. As the table in Figure 6 shows, we used our system to verify the examples mentioned in the introduction, as well as additional benchmarks. For each example the first column shows the number of lines of code, the total number of function applications, and the total number of unapply function calls (generated from pattern matching expressions). We briefly discuss the properties of these examples (the full source code of the examples and the prototype implementation are in the public svn repository [5]).

Test (*lines*/*calls*/*unapply*)		Pattern Matching exhaustive	disjoint	reachable	Spec Verification pre/post/inv	domain
Binary Search	*gen*	0.12s	0.12s	0.16s	0.06s	0.03s
Tree (*40*/*19*/*3*)	*prove*	0.18s	0.16s	*4.41s	0.17s	0.22s
ConsSnoc	*gen*	1.12s	0.12s	0.13s	0.05s	0.05s
(*60*/*25*/*8*)	*prove*	0.12s	0.27s	*6.43s	0.16s	0.48s
Lambda Evaluator	*gen*	0.14s	0.16s	0.15s	0.08s	0.03s
(*160*/*72*/*18*)	*prover*	0.23s	0.95s	*21.9s	1.20s	0.18s
Leftist Heap	*gen*	0.12s	0.13s	0.13s	0.07s	0.02s
(*115*/*55*/*11*)	*prover*	0.14s	0.18s	*11.9s	0.66s	0.08s
Propositional Logic	*gen*	0.13s	0.13s	0.12s	0.09s	0.05s
(*125*/*64*/*10*)	*prover*	0.25s	1.86s	*9.48s	3.35s	0.53s
ScalacTypers	*gen*	0.12s	0.12s	0.12s	0.05s	0.02s
(*80*/*13*/*2*)	*prover*	0.08s	*5.90s	*10.5s	0.13s	0.01s
Verification Condition	*gen*	0.21s	0.23s	0.22s	0.68s	0.36s
Generator (*440*/*270*/*49*)	*prover*	2.84s	44.9s	*69.2s	40.5s	12.9s

Fig. 6. Benchmarks

Binary Search Tree is the example from Figure 1 that illustrates the effectiveness of our system on patterns with guards.

ConsSnoc is the dual view of the linked list with Cons and Snoc patterns, as shown in Figure 3.

Lambda Evaluator is a complete implementation of call-by-value semantics for untyped lambda calculus, in small-step style. Its fragment was shown in Figure 4.

Leftist Heap is the implementation of a working leftist heap that follows [27].

Propositional Logic defines abstract syntax tree for propositional formulas, defines simplification that recursively transforms away nodes of certain type and then manipulates such simplified nodes. This example illustrates type refinements in our system.

ScalacTypers is a simple code fragment inspired by the type checker in Scala compiler implementation that searches for a particular subtree in the syntax tree. It illustrates the application of user-defined patterns to simulate a generalization of *OR-patterns*.

Verification Condition Generator is a verification condition generator for a simple imperative language, written as a class assignment. It contains 49 unapply calls and illustrates the scalability of our prototype analyzer.

5.1 Experience with the analysis system

Table 6 summarizes the results of running our analyzer. We ran the tests on a 2 quad-core Xeon 2.66Ghz with 16GB of RAM, running Debian GNU/Linux (64bit version). However, our system uses no parallelization and we observed similar behavior on standard desktop machines. We used CVC3 prover version 1.2.1. The running times in the table are in seconds. For each example we show the time taken to generate the formulas (*gen*) and the time taken in the theorem prover (*prove*). The star to the left of a running time entry means that the prover exceeded the timeout. We used one second as the prover timeout; the prover proved all valid formulas in our examples within this time limit.

Note that all desirable conditions that our system checks reduce to formula validity, except for reachability. Reachability reduces to the satisfiability of a formula representing a path reaching a pattern-matching case. Consequently, when a program has all desired properties, we expect the prover to succeed in all cases except for reachability where we expect it to time-out or detect that the formula is satisfiable.

The examples in Figure 6 satisfy all expected properties for pattern matching, as well as the domain specifications, preconditions, postconditions, and invariants. The exception is ScalacTypers, which contains pattern matching expressions that are not disjoint and intentionally rely on sequential order of pattern evaluation. In addition to the correct versions of examples in Figure 6 we also applied our prototype to versions of these and other examples that contain errors in each of the kinds of properties our system checks. For such cases the system typically timeouts (except for reachability where unreachable cases were identified quickly).

Overall, we found that the system was very useful in increasing our confidence in the correctness of code that uses pattern matching with guards, and its performance was acceptable. This is particularly the case for exhaustiveness checks that prove the absence of run-time errors (or user-defined exceptions that the user would be forced to insert given a less precise analysis).

6 Conclusions

The support for checking user-defined patterns brings the benefits of pattern matching checks to reusable and maintainable code that admits representation change. The support for pattern matching guards enables verification of more declarative pattern matching expressions (such as disjoint patterns with guards) and provides implicit documentation for invariants on values that programs manipulate (such as pattern matching that is exhaustive due to invariants on the data type). We have presented an analysis that supports both of these features and shown that it is effective in a range of pattern matching code fragments.

Our results were possible thanks for the state-of-the-art theorem provers. Imperative program verification has driven the advance of such theorem provers for a long time [4]. At the same time, programming language researchers have identified functional programming as a style that aids correctness of data-manipulating

programs. We believe that it is time to revisit applications of modern provers to functional programs. We find pattern matching to be a good starting point for such applications due to its wide-spread use and visible limitations of current analyses. Overcoming these limitations is a productive way to introduce precise program analysis and verification techniques into verification of functional programs.

In the future we expect to investigate the integration of our prototype checker into Scala compiler, including interactions of our analysis with Scala's sophisticated type system.

Acknowledgements. We thank Martin Odersky and Burak Emir for useful discussions on Scala and pattern matching.

References

1. The Scala programing language. http://www.scala-lang.org. Last accessed January 2008.
2. Haskell 98 language and libraries: The revised report, December 2002.
3. R. M. Burstall, D. B. MacQueen, and D. Sannella. Hope: An experimental applicative language. In *LISP Conference*, pages 136–143, 1980.
4. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
5. M. Dotta. Pattern matching checker prototype. http://scalapatterns.googlecode.com/svn/matcheck/trunk/, january 2008.
6. J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007. CMU-CS-07-129.
7. B. Emir, Q. Ma, and M. Odersky. Translation Correctness for First-Order Object-Oriented Pattern Matching. Technical report, 2007.
8. B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *ECOOP*, 2007.
9. M. Ernst, C. Kaplan, and C. Chambers. Predicate Dispatching: A Unified Theory of Dispatch. *ECOOP'98–object-oriented Programming: 12th European Conference, Brussels, Belgium, July 20-24, 1998: Proceedings*, 1998.
10. M. Fähndrich and J. Boyland. Statically checkable pattern abstractions. *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 75–84, 1997.
11. C. Flanagan, K. R. M. Leino, M. Lilibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2002.
12. T. Freeman and F. Pfenning. Refinement types for ML. In *Proc. ACM PLDI*, 1991.
13. T. Freeman and F. Pfenning. Refinement types for ML. *ACM SIGPLAN Notices*, 26(6):268–277, 1991.
14. Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *CADE*, 2007.
15. J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
16. V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

17. K. R. M. Leino and P. Müller. A verification methodology for model fields. In *ESOP'06*, 2006.
18. X. Leroy. *The Objective Caml system, release 3.08*, July 2004.
19. J. Liu and A. Myers. JMatch: Iterable abstract pattern matching for Java. *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 110–127, 2003.
20. T. D. Millstein. Practical predicate dispatch. In *OOPSLA*, pages 345–364, 2004.
21. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Mass., 1997.
22. Y. M. Minsky. Caml trading. In *POPL*. ACM, 2008.
23. P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *CC*, pages 61–76, 2003.
24. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
25. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.
26. C. Okasaki. Views for Standard ML. *1998 ACM SIGPLAN Workshop on Standard ML*, pages 14–23.
27. C. Okasaki. Functional data structures. In *Advanced Functional Programming*, pages 131–158, 1996.
28. R. Pickering. *Foundations of F#*. Apress, 2007.
29. D. Syme, G. Neverov, and J. Margetson. Extensible pattern matching via a lightweight language extension. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 29–40, New York, NY, USA, 2007. ACM.
30. S. Thompson. A Logic for Miranda, Revisited. *Formal Aspects of Computing*, (7), March 1995.
31. P. Wadler and S. Munchnik. Views:A Way for Pattern Matching to Cohabit with Data Abstraction. *Proceedings, 14th Symposium on Principles of Programming Languages*, pages 307–312, 1987.