Reconciling Exhaustive Pattern Matching with Objects

Chinawat Isradisaikul Andrew C. Myers

Department of Computer Science
Cornell University
Ithaca, New York, United States
chinawat@cs.cornell.edu andru@cs.cornell.edu

Abstract

Pattern matching, an important feature of functional languages, is in conflict with data abstraction and extensibility, which are central to object-oriented languages. Modal abstraction offers an integration of deep pattern matching and convenient iteration abstractions into an object-oriented setting; however, because of data abstraction, it is challenging for a compiler to statically verify properties such as exhaustiveness. In this work, we extend modal abstraction in the JMatch language to support static, modular reasoning about exhaustiveness and redundancy. New matching specifications allow these properties to be checked using an SMT solver. We also introduce expressive pattern-matching constructs. Our evaluation shows that these new features enable more concise code and that the performance of checking exhaustiveness and redundancy is acceptable.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Patterns, Polymorphism; D.1.5 [*Programming Techniques*]: Object-oriented Programming; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Pre- and post-conditions, Mechanical verification

General Terms Design, Verification, Languages

Keywords JMatch; pattern matching; named constructor; equality constructor; matching specification; exhaustiveness; redundancy; modal abstraction; Java; data abstraction; subtyping

1. Introduction

Despite being an important feature of modern functional programming languages, pattern matching has not been adopted by most object-oriented languages. Data abstraction and extensibility, both primary goals of object-oriented languages, conflict with pattern matching. This work explores a language design for integrating pattern matching with object-oriented programming.

The following is a simple implementation of natural numbers in OCaml. The algebraic data type `nat`, with two constructors `Zero` and `Succ`, represents a natural number; the recursive `plus` function adds two naturals by matching them with one of three patterns.

```
type nat = Zero | Succ of nat

let rec plus m n =
  match (m, n) with
    (Zero, x)
  | (x, Zero) -> x
  | (Succ m', _) -> plus m' (Succ n)
```

This example illustrates two benefits of pattern matching in ML and other functional programming languages such as Haskell.

The first benefit is that patterns serve a dual role that enables algebraic reasoning and results in concise, intuitive code. A constructor such as `Succ` is also a pattern that matches the values produced by that constructor. Patterns can be nested to match complex values in a natural way, so a pattern like `Succ(Succ(n))` matches exactly the values constructed by expressions using the same syntax.

The second benefit is that pattern matching helps catch common programming errors. Patterns in a `match` expression can be checked to ensure that they are *exhaustive* and not *redundant*: that all possible values are matched by some pattern, and that every pattern can match some value. Without such checks, if the programmer forgot the first of the three cases above, the program could crash with an exception. With such checks, the compiler would warn that no cases match values of the form `(Zero, Succ _)`.

Relying on access to the concrete representation of data, however, makes the ML-style pattern matching inimical to data abstraction [30]. A value produced by one module can only be matched by patterns in another module if the second module knows the underlying representation of the value. Agreement on the concrete representation tightly couples the two modules in a way usually considered undesirable for large software systems. For example, we might initially implement natural numbers as above, then later want to change the representation to be an `int`. This change is not possible in ML without breaking client code.

To make pattern matching compatible with data abstraction, prior work has developed pattern-matching constructs that can be implemented by arbitrary code. Examples of this approach include views [30], extractors [5], and active patterns [29]. These mechanisms permit matching on deep patterns over abstract data, but sacrifice other benefits of algebraic pattern matching. There is no check that patterns are consistent with their corresponding constructors, so algebraic reasoning is weakened. Further, data abstraction interferes with checking exhaustiveness and redundancy.

The JMatch language [19] introduced another way to harmoniously integrate pattern matching into object-oriented languages, through *modal abstractions* that support multiple directions of computation. Modal abstractions allow a constructor and its pattern to be implemented by the same invertible computation, ensuring that they are inverses. Determining whether patterns are exhaustive or redundant, however, remained impossible under the data abstraction provided by JMatch. Furthermore, the added expressive power of patterns implemented by complex computations

PLDI'13, June 16–19, 2013, Seattle, Washington, USA.

```
1  class Nat {
2    private int value;
3    private Nat(int n) returns(n)
4      ( value = n )
5    public static Nat zero() returns()
6      ( result = Nat(0) )
7    public static Nat succ(Nat n) returns(n)
8      ( result = Nat(n.value + 1) )
9  }
10 ...
11 static Nat plus(Nat m, Nat n) {
12   switch (m, n) {
13     case (zero(), Nat x):
14     case (x, zero()):
15       return x;
16     case (succ(Nat k), _):
17       return plus(k, Nat.succ(n));
18   }
19 }
```

Figure 1. Natural numbers with data abstraction in JMatch.

means that programmers can accidentally omit patterns more easily than with algebraic data types.

The challenge for analysis of exhaustiveness and redundancy is to reason statically without violating data abstraction. The main contribution of this paper is, therefore, a way to extend modal abstractions with concise specifications that enable static reasoning about exhaustiveness and redundancy of pattern matching and, more generally, about the totality of computations.

Object-oriented programming involves more than just data abstraction; subtyping and inheritance are key ingredients supporting extensibility. For extensibility, different implementations of (subtypes of) the same interface should support the same patterns without clients knowing which implementation has been used. We therefore introduce *named constructors* that can be used as patterns in this way. We also introduce two first-class *or-patterns* that generalize both data-type constructors and or-patterns in ML.

We proceed as follows. Section 2 reviews modal abstraction in JMatch. Section 3 introduces mechanisms that improve the expressive power of pattern matching and its integration with objects. Section 4 describes new static annotations that support reasoning about exhaustiveness and redundancy. The verification procedure is explained in Section 5. Section 6 describes our implementation of an extended version of JMatch. Using various code examples, we evaluate its expressiveness, analytic power, and efficiency in Section 7. Section 8 discusses related work, and Section 9 concludes.

2. Background

Some background will be helpful on JMatch [17, 19], an extension to Java 1.4 that supports pattern matching and iteration through modal abstraction.

2.1 Modal abstraction

Section 1 observed that in OCaml, natural numbers cannot support pattern matching while being represented internally with an `int`. Figure 1 shows how this can be done in JMatch. The key idea is that JMatch methods may declare multiple *modes* that correspond to different "directions" of evaluation, analogously to predicate mode declarations in the logic programming language Mercury [26]. In addition to the ordinary *forward mode*, which acts like a Java method, a JMatch method may also provide *backward modes* which, given a desired result, compute corresponding argument values. Backward modes support pattern matching. For example, the method `succ` may be used in the forward mode to compute the successor of a number. As indicated by the clause `returns(n)` on line 7, it also has a backward mode that computes the number `n` for which the value given in `result` is the successor.

This implementation of `Nat` *is* more complex than in OCaml because the abstract view that supports pattern matching must be related to its concrete representation as an `int`. The methods of `Nat` demonstrates that JMatch programs can define patterns that both preserve data abstraction, because the field `value` is private, and are also usable outside the module that defines them. Lines 11–19 show how the backward modes of these methods can be used to implement the method `plus` similarly to the earlier OCaml code.

In general, a JMatch method implements a relation over its arguments and its result. Each of its modes is a different way of exploring the relation. For example, the `succ` operation is a binary relation on `Nat`: a subset of `Nat` × `Nat`. In each mode, some of the arguments or the return value are *knowns* supplied by the caller, and the others are *unknowns* to be solved for.

Individual method modes may be implemented by imperative, Java-like code, but one single declarative-style implementation of multiple modes is often more concise. As in each of `Nat`'s methods in Figure 1, a declarative method implementation is a boolean formula placed inside parentheses, directly expressing the implemented relation. For example, the equation `result = Nat(n.value + 1)` at line 8 exactly captures the `succ` relation.[1] For each mode of a method, the compiler generates an imperative algorithm that, given values for knowns, finds values of all unknowns that satisfy the formula. Thus, the backward mode often comes nearly for free, unlike with related approaches such as extractors [5].

Not only user-defined abstractions but also built-in types such as primitive types support modal abstractions. For example, integer operations such as + and - can solve for either of their arguments, given a result to match against.

2.2 Iterative modes

Modes need not be functions; viewed as relations, they may be one-to-many or many-to-many. A mode is *iterative* when there may be more than one solution to the unknowns for given knowns; the keyword `iterates` is used in place of `returns` to indicate such a mode. For example, the `contains` method of the `Collection` class has the signature

```
boolean contains(Object x) iterates(x)
```

meaning that its backward mode can be used to iterate over all contained objects. Using iterative modes, the Java collections framework could be made 35% more concise by implementing its operations, including iterators, as modes of relatively few methods [17].

2.3 Semantics and solving

The semantics of JMatch is defined as a syntax-directed, type-preserving translation to Java$_{\texttt{yield}}$ [18], which extends Java with *coroutine methods* in which control is yielded to the caller via the `yield` statement, much as in languages like C#, Ruby, and, originally, CLU [16].

JMatch supports imperative Java code, the translation of which is relatively straightforward. The interesting parts of the translation involve the solving of boolean formulas and pattern expressions. JMatch considers a formula or pattern *solvable* when the compiler can generate an algorithm that either finds satisfying assignments to unknowns or determines that there are none. In the latter case, the formula or pattern is not *satisfiable* but is still solvable. A formula or pattern may also be satisfiable but not solvable if the

[1] Note that the operator = is an equality test, which unambiguously *subsumes* its usual Java role as imperative assignment.

```
interface Nat {
  constructor zero() returns();
  constructor succ(Nat n) returns(n);
  ...
}
```

Figure 2. Natural number interface with named constructors.

```
1  class ZNat implements Nat {
2    int val;
3    private ZNat(int n) returns(n)
4      ( val = n && n >= 0 )
5    constructor zero() returns()
6      ( val = 0 )
7    constructor succ(Nat n) returns(n)
8      ( val >= 1 && ZNat(val - 1) = n )
9    ...
10 }
11
12 class PZero implements Nat {
13   constructor zero() returns() ( true )
14   constructor succ(Nat n) returns(n) ( false )
15   ...
16 }
17 class PSucc implements Nat {
18   Nat pred;
19   constructor zero() returns() ( false )
20   constructor succ(Nat n) returns(n) ( pred = n )
21   ...
22 }
```

Figure 3. Three implementations of `Nat`.

compiler does not know how to generate an appropriate algorithm for determining satisfying assignments.

As an extension to Java, JMatch allows side effects, although its new features encourage a declarative programming style. With side effects, programmers need to reason about the order in which computations occur. The JMatch solver therefore solves formulas in a well-defined order that is left-to-right as much as possible.

3. Pattern-matching extensions

We extend JMatch, adding new pattern-matching constructs to better support object-oriented programming and data abstraction and to increase expressive power in other ways.

3.1 Named constructors

In JMatch, pattern matching using procedures is successful only if the value being matched is either their result or one of their arguments. Therefore, a JMatch procedure can successfully match on its own receiver object (`this`) only if the procedure is a constructor or happens to return its receiver object as the result. Since a constructor belongs to a particular class, code using a constructor pattern is tightly coupled to that particular implementation. This tight coupling interferes with extensibility and code reuse.

To support implementation-oblivious pattern matching, we extend JMatch with *named constructors* that can pattern-match an object whose run-time class is unknown. Named constructors have an explicit name different from that of their class, and they can be declared in interfaces.

For example, Figure 2 shows a `Nat` interface exposing two named constructors, `zero` and `succ`. Figure 3 shows two partial implementations of `Nat`. The first (`ZNat`) corresponds to the implementation of Figure 1. The second is analogous to the OCaml

```
1  class ZNat { ...
2    constructor equals(Nat n)
3      ( zero() && n.zero() |
4          succ(Nat y) && n.succ(y) )
5  }
6
7  class PZero { ...
8    constructor equals(Nat n)
9      ( n.zero() )
10 }
11 class PSucc { ...
12   constructor equals(Nat n)
13     ( n.succ(pred) )
14 }
```

Figure 4. Equality constructors.

version and consists of two classes: `PZero`, representing zero, and `PSucc`, representing the successor of its field `pred` at line 18.

Unlike ordinary constructors, a named constructor like `zero` can be applied to an object `x` of type `Nat` or of any subtype of `Nat`. It acts as a boolean predicate in this style of invocation. For example, `ZNat(0).zero()` evaluates to `true` because its implementation tests the equation `val = 0`. A named constructor for type T may be invoked without an explicit receiver object when it is used to pattern-match a value of type T. In this case, the receiver object is the value being matched. Finally, named constructors can be invoked to construct new objects of their class, as in the expression `ZNat.zero()`. In the forward mode, the fields of `this` are in scope as unknowns to be solved for either directly in the formula or via another constructor. For example, `val` in the equation `val = 0` at line 6 is solved directly by assigning zero to it.

3.2 Equality constructors

As written, the implementations of `Nat` in Figure 3 are incomplete. The problem is that the forward mode of `succ` in `ZNat` promises to construct a `ZNat` from an arbitrary `Nat` predecessor `n`. If `n` is not a `ZNat`, the equality test at line 8 between `ZNat(val - 1)` and `n` will fail. We fix this by adding an operation to `Nat` that allows solving for equality between objects of different classes:

```
constructor equals(Nat n);
```

Because `equals` is used to pattern-match on the receiver object, it becomes a special named constructor—an *equality constructor*—rather than an ordinary boolean method as in Java. If defined, `equals` is used for solving equality in addition to JMatch's default strategy of direct assignment. The code of `equals` for the classes implementing `Nat` is given in Figure 4.

Using `equals`, the equality `ZNat(val - 1) = n` is solved for non-`ZNat` objects `n` by invoking `ZNat.equals`, defined at lines 2–4. This method tests whether `n` is zero or the successor of some number. If the former, it returns `ZNat.zero`; if the latter, it invokes `ZNat.succ` recursively to retrieve the predecessor of `n`, which is bound to `y` by the constructor invocation `n.succ(y)`. Operationally, `ZNat`'s `equals` and `succ` interoperate to find successive predecessors until either zero or a `ZNat` representation (as in `PSucc.succ(ZNat(3))`, which is legal!) is encountered. Once `equals` converts `n` to a `ZNat` object, `succ` matches the internal representation of this `ZNat` object with `val - 1`, solving for `val`, which internally represents the desired successor.

3.3 Other extensions

A complete overview of the existing patterns in JMatch can be found in Section 2.2 of the JMatch technical report [18]. We extend the language with additional operators and a new pattern that increase expressive power:

```
1  public Expr CPS(Expr e) returns(e) (
2    Var k = freshVar("k", e) &&
3    (e, result) =
4      (Var(_),
5         Lambda(k, Apply(k, e)))
6    | (Lambda(Var vl, Expr body),
7         Lambda(k,
8           Apply(k, Lambda(vl,
9             Lambda(k, Apply(CPS(body), k))))))
10   | (Apply(Expr fn, Expr arg),
11        Lambda(k, Apply(CPS(fn),
12          Lambda(f, Apply(CPS(arg),
13            Lambda(Var("v") as Var va,
14              Apply(Apply(f, va), k))))))
15          where Var f = freshVar("f", arg))
16 )
```

Figure 5. Invertible CPS conversion.

- JMatch already has a pattern conjunction operator called `as`, which generalizes ML's pattern operator of the same name by requiring two arbitrary patterns to match the same value. We add a pattern *disjunction* operator, `#`, that combines two patterns into a single pattern that matches either or both of the two, and solves for the same unknowns. For example, the formula `int x = y-1 # y+1` (which should be read as `int x = (y-1 # y+1)`) generates the two solutions `x = y-1` and `x = y+1` when solving for `x`, and `y = x+1` and `y = x-1` when solving for `y`. Unlike Icon's alternation expression [9], a match is attempted against all alternatives even if one of them fails.
- We also add a *disjoint* disjunction operator, `|`, that behaves like `#` except that the patterns must be disjoint. A pattern constructed with this operator produces at most one solution when a value is matched against it, unlike `#`. The number of solutions is important because the pattern-matching statements require that there be only a single solution. The compiler verifies that patterns combined via `|` are disjoint. The formula `x = 1 | 2` would therefore be legal but `x = y-1 | y+1` would not if used to solve for `y`.
- A *tuple pattern*, written $(p_1, \ldots, p_n)$, may be used to match multiple values at once. Tuples are not first-class values; uses of tuple patterns are equivalent to, but often more concise than, a set of equations expressed over the tuple components.

These new constructs add expressiveness. For example, the JMatch 1.1.6 release [18] includes an example of invertible conversion to continuation-passing style (CPS). The same two computations, CPS conversion and its inverse, are both expressed even more concisely in Figure 5 using the new pattern operators. In this code, the use of tuples enables the translation rules to be expressed essentially as inference rules. The pattern (p `where` f) on line 15 refines pattern p to succeed only when formula f is also satisfiable. The use of `|` ensures that `CPS` is one-to-one, though not total in its backward mode. Without `|`, the JMatch compiler would be unable to conclude that the three cases are disjoint and would raise the error that `CPS` is not one-to-one.

4. Static annotations for exhaustiveness reasoning

Several pattern-matching forms in JMatch can benefit from verification of exhaustiveness. As we saw in Figure 1, `switch` statements are one such form. Whether `switch (`e`) {`$\overrightarrow{\texttt{case } p_i\texttt{: } s_i}$`}` is exhaustive corresponds to (roughly) whether

$$\bigvee_{i=1}^{n} e = p_i \tag{1}$$

```
Nat n;
...
switch (n) {
  case succ(Nat p): ...
  case succ(succ(Nat pp)):  ...
  case zero(): ...
}
```

Figure 6. Redundant switch statement.

is a tautology. A second such form is the JMatch statement `cond` `{`$\overrightarrow{(f_i)\ \{s_i\}}$`}`, which executes the first statement s_i such that its corresponding formula f_i is true. For exhaustiveness, at least one such formula must be true. A third pattern-matching form is `let` f, which is equivalent to `cond {(`f`) {}}` except that variable bindings made in f are in scope for the remainder of the statement's block. The declaration `int x = 2` is in fact syntactic sugar for `let int x = 2`. Therefore, the formula f in a `let` statement should always be satisfiable.

In principle, exhaustiveness checking seems simple. Reasoning about exhaustiveness while preserving data abstraction, however, is challenging because the client code performing pattern matching is oblivious to the concrete representation (e.g., private fields) of objects. For example, given the code in Figure 6, the compiler does not know the implementation of `succ` and `zero` with which `n` will be matched. Even if it did know, using this knowledge would violate modularity, coupling correctness of this code to implementation choices internal to `Nat`. Moreover, given a value of type `Nat`, the compiler may not assume that `succ` and `zero` are the only ways to construct the value; there could be another constructor defined in `Nat` that could produce the same value. As a result, the compiler does not have enough information about the patterns to show that disjunction (1) is a tautology.

To enable the compiler to reason modularly about exhaustiveness, we must expose enough information to the client about the relation implemented by a method without exposing implementation details. Supplied with this information for the code in Figure 6, the compiler should be able to determine that all values of type `Nat` will be matched by some case. If that were not true (e.g., if the first case were omitted), the compiler should issue a warning. Also, in the code as written, the second case is redundant because anything matching `succ(succ(Nat pp))` must have matched `succ(Nat p)`. Redundant code often indicates errors in programmer's reasoning; the compiler ought to report this too. At the same time, the exposed information should let the compiler know that `zero` and `succ` are indeed disjoint and conclude that the third case and the first two are not redundant. Without such information, the compiler could generate a false redundancy warning.

To support static verification of exhaustiveness and other properties, three new kinds of concise and intuitive specifications provide the missing information: *class invariants*, *matches clauses*, and *ensures clauses*. As an orthogonal benefit, all of these specifications can exploit the new pattern operator `|` to prove patterns disjoint. We now explore these new features in more detail.

4.1 Class and interface invariants

One way to provide the information needed to determine exhaustiveness is as a *class* or *interface invariant*. For example, we can express that all instances of `Nat` match either `zero()` or `succ()` by adding the following invariant to the `Nat` interface, using `|` to assert that the two patterns are disjoint:

```
1  class ZNat implements Nat {
2    int val;
3    private invariant(val >= 0);
4    private ZNat(int n) matches(n >= 0) returns(n)
5      ( val = n && n >= 0 )
6    ...
7  }
```

Figure 7. Private invariant and `matches` clause.

```
interface Nat {
  invariant(this = zero() | succ(_));
  ...
}
```

This example invariant shows how to obtain the exhaustiveness analysis provided by algebraic data types, while preserving data abstraction and allowing extensibility. New implementations of the `Nat` interface do not alter this invariant.

Class and interface invariants can be thought of as a kind of boolean-valued method whose value is always asserted to be true and whose implementation is visible to callers. Invariants may be given visibility modifiers (`public`, `protected`, or `private`). To maintain modularity, an invariant may only mention methods and fields that are at least as visible as the invariant itself.

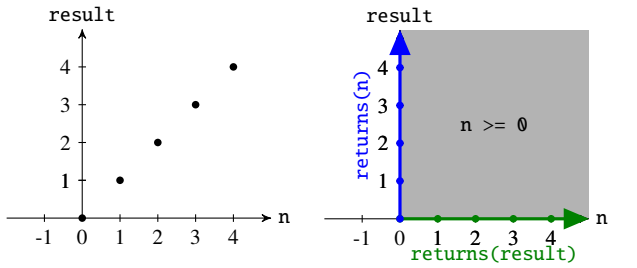
Invariants not publicly visible may be useful for verifying the implementation of a class, such as the totality of the implementation of its methods. For instance, in the `ZNat` code of Figure 3, the field `val` cannot be negative. We can add a private invariant asserting this constraint, as in Figure 7. This invariant supports successful verification of the backward mode of the implementation of the constructor `ZNat()`, which should be total among all `ZNat` values. The invariant plus the first conjunct imply the second conjunct, `n >= 0`. The private invariant also helps verify both modes of `succ()`.

4.2 Matches clauses

One impediment to checking exhaustiveness is that a method mode may implement partial functions: on some inputs, its body might be unsatisfiable, in which case the method will fail rather than returning values for its unknowns. We extend the JMatch language with a way to specify when a method will successfully produce a result. This "matching precondition" is analogous to a precondition, but rather than specifying when a method call is legal, it specifies when pattern matching is guaranteed to succeed. The specification is conservative in that matching could succeed even when the condition does not hold.

For example, consider the constructor `ZNat()` in Figure 3. Any `ZNat` object must have a representation as a nonnegative integer. The corresponding matching precondition for the forward mode is `n >= 0`, meaning that for any nonnegative `n`, there exists a `ZNat` object matching that `n`. This matching precondition implies the constructor body, allowing successful verification of the forward mode. The backward mode of `ZNat()`, on the other hand, is total, corresponding to the matching precondition `true`.

Asking the programmer to specify matching preconditions for each mode would be verbose and repetitive, since different modes may share knowns (i.e., inputs). Our insight is that the programmer can write a single condition that captures when matching will succeed for the entire relation implemented by a method. We call this condition the *matches clause* for the method. Methods having no `matches` clause defaults to `matches(false)`, meaning that matching is not guaranteed to succeed for any input. The JMatch 2.0 compiler must extract the matching precondition for each mode from the consolidated `matches` clause. The extraction is described formally in Section 4.3; the rest of this section illustrates how extraction works for `ZNat()`.



(a) actual relation (b) matching preconditions

Figure 8. The `ZNat` relation.

The `matches` clause for `ZNat()` is shown in Figure 7. Figure 8(a) shows the actual relation implemented by `ZNat`; Figure 8(b) shows the `matches` clause, describing the relation consisting of integral points in the shaded region. This relation can be viewed as an approximation to the true `ZNat` relation. Informally, the extraction obtains the matching precondition by projecting this relation onto the axis corresponding to an appropriate mode, obtaining matching preconditions shown as thick arrows. For the forward mode (`returns(result)`), the relation is protected onto the `n` axis, obtaining `n >= 0`. For the backward mode (`returns(n)`), it is projected onto the `result` axis, obtaining `true`.

4.3 Extracting matching precondition from matches clause

In general, the body of a method implements some relation B and the `matches` clause specifies another relation M. Suppose that the method is a relation over a set of variables $\{\vec{x}\}$. For each mode $\mathcal{M}$ of the method, this set is partitioned into disjoint sets of knowns (inputs) $\{\vec{k}\}$ and unknowns (outputs) $\{\vec{u}\}$. We can then view the relations M and B as predicates over knowns and unknowns, $M(\vec{k},\vec{u})$ and $B(\vec{k},\vec{u})$, respectively. Given $\vec{k}$, the precise condition in which the body guarantees success is therefore $\exists\vec{u}.\ B(\vec{k},\vec{u})$. We call this formula the *precise matching precondition*.

For brevity, we define a function $\pi_{\mathcal{M}}$ that constructs the precise matching precondition for mode $\mathcal{M}$ by projecting an arbitary predicate B onto the knowns:

$$(\pi_{\mathcal{M}} B)(\vec{k}) \stackrel{\Delta}{\iff} \exists\vec{u}.\ B(\vec{k},\vec{u})$$

Given $\mathcal{M}$ and B, $\pi_{\mathcal{M}} B$ is a predicate on $\vec{k}$ that holds when $\vec{k}$ provides some way to satisfy B in mode $\mathcal{M}$ and hence to successfully pattern-match.

To preserve abstraction, reasoning about exhaustiveness must be done using the `matches` clause M, not B. Intuitively, if $M \Rightarrow B$, the body will be satisfiable whenever the `matches` clause holds; however, to require this implication would be unnecessarily restrictive. In the case of `ZNat` relation, for example, `n >= 0` does *not* imply the actual relation, and the only `matches` clause that does so while preserving abstraction is `false`. A more useful correctness condition is $M \Rightarrow \bigwedge_{\mathcal{M}'} \pi_{\mathcal{M}'} B$, where $\mathcal{M}'$ is ranges over declared modes. In other words, satisfiability of the `matches` clause only needs to imply satisfiability of the body in all the modes actually available. This can be visualized as expanding B (in the case of `ZNat`, the dots in Figure 8(a)) along all the dimensions corresponding to the supported modes (obtaining, in this case, the shaded area in Figure 8(b)).

It is easy to show that the formula $M \Rightarrow \bigwedge_{\mathcal{M}'} \pi_{\mathcal{M}'} B$ is equivalent to $\pi_{\mathcal{M}} M \Rightarrow \pi_{\mathcal{M}} B$ for each declared mode $\mathcal{M}$. This suggests

that given the mode $\mathcal{M} = (\{\vec{k}\}, \{\vec{u}\})$ and the inputs $\vec{k}$, we should verify exhaustiveness by using $(\pi_{\mathcal{M}} M)(\vec{k})$ as the matching precondition. Unfortunately, the existential quantifiers in this formula make it ill-suited to automated reasoning. Instead, we construct a weakening of $\pi_{\mathcal{M}} M$ that does not mention existential quantifiers. Let us denote this weakened predicate on $\vec{k}$ as $Extract_{\mathcal{M}} M$.

The construction of $Extract_{\mathcal{M}} M$ proceeds as follows. We first convert the `matches` clause into negation normal form (NNF) so that the formula uses only positive logical operators over atomic formulas. We then use a variant of the usual JMatch algorithm for generating solutions to a formula. The first step is to reorder the atoms so that as many unknowns as possible can be solved from left to right. After this reordering, atoms that do not mention unknowns are left unchanged, as are atoms in which all unknowns are solvable in the left-to-right order. Atoms mentioning any unsolvable unknowns are dropped; that is, they are replaced with `true`. Any remaining occurrences of unknowns can be thought of as existentially quantified, but because each remaining unknown is solvable, it represents a solution expressed entirely in terms of knowns.

For instance, in the `ZNat` example, the atomic formula `n >= 0` is dropped in the backward mode because `n` is unsolvable, leaving only `true`. As another example, consider extracting a matching precondition for `x > 0 && y >= 0 && x+1 = y`, where `x` is unknown and `y` known. The formula is first reordered to allow solving for `x`, yielding `y >= 0 && x+1 = y && x > 0`. The first atom is left unchanged because it only mentions `y`. The second is also kept because it solves fore `x`, allowing the third atom to be retained as well. Because `x` is solved by the value `y-1`, the extracted precondition is `y >= 0 && (y-1)+1 = y && (y-1) > 0`, which is equivalent to `y > 1`. In general, `x` might be solved by a user-defined method. Section 5 explains how atoms containing such unknowns are handled.

Dropping unsolvable atoms is a heuristic, but it seems effective because such atoms are typically satisfiable for all possible values of the knowns. In general, however, dropped atoms might not be satisfiable, in which case $Extract_{\mathcal{M}} M$ may not be conservative. For example, if the `matches` clause were instead `y >= 0 && x < y && x > 0`, dropping the atoms `x < y` and `x > 0` would result in the extracted precondition `y >= 0`. The precise matching precondition $(\pi_{\mathcal{M}} M)(\mathtt{y})$ is rather `y >= 2`, since there is no satisfying assignment for `x` when $\mathtt{y} < 2$.

$Extract_{\mathcal{M}} M$ can be used not only for analyzing exhaustiveness, but also for verifying that the method body implements its extracted precondition in mode $\mathcal{M}$. That is, when the method body is implemented as a formula, the compiler verifies in each mode $\mathcal{M}$ that for all inputs $\vec{k}$, $(Extract_{\mathcal{M}} M)(\vec{k}) \Rightarrow (\pi_{\mathcal{M}} B)(\vec{k})$. This ensures soundness for exhaustiveness analysis done using $Extract_{\mathcal{M}} M$. Verification is done using an SMT solver, as described in Section 6. For imperative method implementations, this verification is left to the programmer, though existing program logics might be used to obtain a verifiable logical interpretation in many cases.

4.4 Opaquely refining matches

In general, we may want to support modes in which precondition extraction fails because the `matches` clause does not or cannot capture the relationship among the arguments. For example, consider adding to `ZNat()` a *predicate mode* `returns()`, in which there are no unknowns. In this mode, the `matches` clause `n >= 0` does not correctly capture the matching precondition, yet the existing implementation is correct. To support such modes, `matches` clauses may be refined using the special opaque predicate `notall`. During precondition extraction, an atom `notall`$(\overrightarrow{x_i})$ is treated as unsolvable if any of the variables x_i is unknown, and is therefore dropped; if all of the variables are known or already solved, however, the predicate is treated as `false`.

Thus, to support a predicate mode for `ZNat()`, the predicate `notall(result, n)` is conjoined with `n >= 0` to indicate that pattern matching is not guaranteed to succeed when both `result` and `n` are known. This `notall` predicate corresponds to the refinement that converts the gray area in Figure 8(b) into just the black dots in Figure 8(a). The opaque `notall` is needed because this refinement cannot be characterized abstractly.

4.5 Ensures clauses

Matches clauses are a kind of multimodal precondition. To improve the precision of verification and exhaustiveness reasoning in JMatch, we add the *ensures clause*, a multimodal postcondition whose syntax resembles previous, unimodal postcondition specifications (e.g., [25, 27]). The `ensures` clause for a method is an abstraction of the relation implemented by the method, expressed in terms that client code can understand; that is, it only mentions names a legal caller could name, similar to the specifications proposed by Leavens and Müller [14].

Unlike the `matches` clause, the `ensures` clause must define an overapproximation (a superset) of the implemented relation. Thus, in any context where a method call is known to have succeeded, the `ensures` clause can be assumed to hold with respect to the values supplied as knowns and the values returned as unknowns.

Because the clauses for both `matches` and `ensures` are often identical, the syntax `matches ensures(`f`)` may be used as a shorthand for `matches(`f`) ensures(`f`)`.

5. Checking exhaustiveness and totality

JMatch 2.0 must show the exhaustiveness of various pattern-matching statements (`switch`, `cond`, and `let`). Similar verification is required for methods with a `matches` or `ensures` clause, since they promise to succeed in each mode when the extracted matching precondition is true, and since the postcondition must hold if the methods succeed. In addition, both arms of `|` must be verified as disjoint. In each case, the analysis constructs quantifier-free formulas that can be satisfied only if some cases are not handled by the appropriate patterns or formulas.

Section 4 described verification informally while pretending that formulas can be verified directly, e.g., by an SMT solver. This is not true in general, because formulas may contain user-defined predicates that must be treated abstractly.

To aid in constructing formulas to be verified the SMT solver, we introduce an intermediate representation language $\mathbb{F}$ that is similar to the language of quantifier-free logical formulas. Each formula is transformed to an $\mathbb{F}$ formula using function $\mathcal{V}_{\mathcal{F}}$ defined inductively on the syntax of formulas. $\mathcal{V}_{\mathcal{F}}$ takes a formula f to be transformed, along with the set U of unknowns to be solved in f, and an additional $\mathbb{F}$ formula F that represents the rest of the constraint. $\mathcal{V}_{\mathcal{F}}[\![f]\!]\ U\ F$ holds if (1) f is satisfiable and (2) F also holds under any solution to the unknowns in U satisfying f.

Because space is limited, the definitions of the translation are elided here; they are available in the technical report [11].

The Z3 theorem prover [3] is used to find a model satisfying these $\mathbb{F}$ formulas. This model can be used to construct a counterexample to explain the failure of exhaustiveness or totality to the user. The verification done by Z3 does not affect the dynamic semantics of JMatch 2.0; it only affects warnings given to the programmer.

In the remainder of this section, let μf be the set of variables declared in f.

5.1 Verifying exhaustiveness

Every `switch` and `if` statement can be written as a `cond` statement. Thus, verification of `switch` and `if` reduces to that of `cond`.

To verify a cond statement `cond { (f_i) {s_i} else s }`, we begin by asserting the invariants of all the known variables in the context. We then proceed case by case. The algorithm first checks whether f_i yields a solution to its unknowns. If not, the compiler issues a warning that this arm is redundant. In either case, the assertion is updated to rule out patterns matched up to the current arm. The `else` arm, if present, is equivalent to `true`.

Finally, the `cond` statement is exhaustive if the final assertion is unsatisfiable. If not, a counterexample is generated from a satisfying assignment, and a nonexhaustive warning is reported.

A `cond` statement can be used to refine patterns in the same way as a `where` pattern. Since both `switch` and `if` are syntactic sugar for `cond`, so can they. Let I be the invariant prior to the verification of a conditional case `(f) {s}`. We verify s with the stronger invariant $I' \triangleq I \wedge (\mathcal{V}_{\mathcal{F}}[\![f]\!]\ \mu f\ \mathsf{true})$.

To verify `let f`, we check whether the negation of f is satisfiable. If so, a warning that the `let` statement may not always be total is reported to the programmer.

5.2 Verifying matching specifications

As described in Section 4.3, the bodies of methods are checked against the `matches` clause of the method to ensure that the body succeeds whenever the `matches` clause is true. Recall that this entails verifying the proposition $\mathit{Extract}_{\mathcal{M}} M \Rightarrow \pi_{\mathcal{M}} B$.

One complication is that the `matches` clause M of a method may refer to other methods. These method references may solve for unknown variables in M. In turn, these unknowns may be further referenced by other atoms in M, imposing additional matching preconditions.

The `matches` and `ensures` clauses of the referenced methods are used to resolve this complication. The `matches` clause imposes additional matching precondition to M, and the `ensures` clause constrains the values of unknowns that may be referenced later in M.

In the following example, the `matches` clause of method `bar` refers to `foo`:

```
int foo(int x)
  matches(x > 2) ensures(result >= x);
int bar(int y)
  matches(y > 0 && result = foo(y) && result < 4);
```

Now, suppose we want to extract `bar`'s matching precondition for the forward mode, i.e., when `y` is known. The reordering and atom-dropping procedure does not alter the `matches` clause. This means `bar(y)` succeeds if `y > 0`, `foo(y)` returns a result, and `foo(y) < 4`. The invocation of `foo` in `bar`'s `matches` clause succeeds if `y > 2`, and `foo`'s `ensures` clause says `result` $\geq$ `y`. Therefore, `bar(y)` is guaranteed to succeed if `y` $> 0 \wedge$ `y` $> 2 \wedge$ `result` $\geq$ `y` $\wedge$ `result` < 4, which is equivalent to `y` $= 3$.

We now give the formal translation for $\mathit{Extract}_{\mathcal{M}} M$, where $\mathcal{M} = (\{\vec{k}\}, \{\vec{u}\})$. If $\hat{M}$ is the result of reordering and dropping atoms in M, and $\{\vec{\hat{u}}\} \subseteq \{\vec{u}\}$ is the set of unknowns remaining in $\hat{M}$, then we have

$$\mathit{Extract}_{\mathcal{M}} M \triangleq \mathcal{V}_{\mathcal{F}}[\![\hat{M}]\!]\ \left(\{\vec{\hat{u}}\} \cup \mu\hat{M}\right)\ \mathsf{true}$$

Similarly, the precise matching precondition is defined as

$$\pi_{\mathcal{M}} B \triangleq \mathcal{V}_{\mathcal{F}}[\![B]\!]\ \left(\{\vec{u}\} \cup \mu B\right)\ \mathsf{true}$$

With the above definitions, we are ready to formally define the verification conditions for JMatch methods. To verify a method

$$T_r\ \ \texttt{foo}(\overrightarrow{T_i\ x_i})\ \texttt{matches}(M)\ \texttt{ensures}(E)$$

in mode $\mathcal{M}$ with body B, we prove these two assertions:

$$\mathit{Extract}_{\mathcal{M}} M \Rightarrow \pi_{\mathcal{M}} B \tag{2}$$

$$\pi_{\mathcal{M}} B \Rightarrow \mathcal{V}_{\mathcal{F}}[\![E]\!]\ \mu E\ \mathsf{true} \tag{3}$$

Assertion (2) says that if the extracted matching precondition for $\mathcal{M}$ holds, then B succeeds in generating a solution to all of its unknowns, which can be part of the arguments or declared in B itself. Assertion (3) says that if B succeeds in generating a solution, then the postcondition of the method holds.

For a method declared in an interface or declared abstract, and for each mode $\mathcal{M}$ declared in that method, the assertion

$$\mathit{Extract}_{\mathcal{M}} M \Rightarrow \mathit{Extract}_{\mathcal{M}} E$$

is proven instead. Since the `matches` clause must specify an under-approximation of the (unimplemented) relation and the `ensures` clause an overapproximation, this assertion says that by transitivity, if the matching precondition holds, then the postcondition should hold as well.

Failure to prove these assertions means that the method does not respect its contract. As a result, a warning for the violation is issued to the programmer.

5.3 Verifying disjoint patterns

JMatch 2.0 verifies multiplicity of formulas and patterns, ensuring that they generate at most one solution in non-iterative modes. Disjoint pattern disjunctions allow disjunctions to be expressed without generating multiple solutions, but this property must be verified. We also overload the `|` symbol as a logical operator; the formula f_1 `|` f_2 is a disjunction that may be used only if at most one of f_1 or f_2 is satisfiable. Let U be the set of unsolved unknowns in p_1 `|` p_2, and let p_1' be the result of substituting each unsolved unknown in p_1 with a fresh variable, and similarly for p_2'. Patterns p_1 and p_2 are disjoint if $(\mathcal{V}_{\mathcal{F}}[\![x = p_1']\!]\ U\ \mathsf{true}) \wedge (\mathcal{V}_{\mathcal{F}}[\![x = p_2']\!]\ U\ \mathsf{true})$, where x is a fresh variable, is unsatisfiable. Similiarly, when `|` is used as a logical operator, formulas f_1 and f_2 are disjoint if $(\mathcal{V}_{\mathcal{F}}[\![f_1']\!]\ U\ \mathsf{true}) \wedge (\mathcal{V}_{\mathcal{F}}[\![f_2']\!]\ U\ \mathsf{true})$ is unsatisfiable.

Consider the examples in Section 3.3. The pattern `1 | 2` is disjoint because $x = 1 \wedge x = 2$ is unsatisfiable. The disjunction `y-1 | y+1` is disjoint when `y` is known. When `y` is unknown, the verification procedure renames `y` in each arm to a fresh variable, yielding $x = \mathtt{y}_1 - 1 \wedge x = \mathtt{y}_2 + 1$, which is satisfiable, so the compiler generates a warning.

5.4 Soundness

As in most functional programming languages, we consider failures of exhaustiveness not as errors but rather as a reason to warn the programmer. Our goal is to help programmers be *effective*. Therefore, some unsoundness or incompleteness may be tolerable or even desirable if it rarely limits or annoys the programmer. Our verification procedures establish two main sources of unsoundness, possibly leading to erroneous warnings or lack of warnings. An obvious source is that JMatch is an imperative language, yet the reasoning procedures described here do not take side effects into account. We do not consider this a serious problem because JMatch encourages a programming style in which side effects are used sparingly and are encapsulated inside data abstractions. A second source of unsoundness arises from recursively defined methods, which are discussed in Section 6.2. In some cases, the compiler may report that it cannot prove exhaustiveness or lack of redundancy. This does not seem to be a problem in practice.

6. Implementation

We have built a prototype implementation of JMatch 2.0 by extending the JMatch 1.1.6 compiler [18] to add the new pattern matching

```
interface Nat {
  boolean zero() returns();
  boolean succ(Nat n) returns(n);
  ...
}
class PSucc implements Nat {
  Nat pred;
  boolean zero() returns() ( false )
  static PSucc create$zero() ( false )
  boolean succ(Nat n) returns(n)
    ( pred = n )
  static PSucc create$succ(Nat n)
    ( result = PSucc() && result.pred = n )
  ...
}
```

Figure 9. Translation of named constructors.

features in Section 3 and the static annotations in Section 4, and to use the Z3 theorem prover [3] to verify exhaustiveness, totality, and multiplicity.

6.1 Translating new features

Each named constructor `foo(...)` defined in class `C` is translated into two JMatch methods having the same visibility as that of `foo`. The first method is `boolean foo(...)` and contains all the modes where `result` is known. The other method is `static C create$foo(...)` and contains the remaining modes, whose body requires creating a fresh object. For named constructors defined in an interface, the latter translation is omitted. An invocation of a named constructor is also transformed to use one of the translated methods accordingly, with the exception of invocations appearing in invariants and `matches` and `ensures` clauses. These invocations are retained as a type object of the class and will be used directly during verifications. An example translation of `Nat` and `PSucc` is shown in Figure 9.

In the JMatch implementation, when a variable w of type T_w is matched against a value x of type T_x, only an `instanceof` check is introduced if T_w is not a supertype of T_x. To use the equality constructor, JMatch 2.0 further checks whether an equality constructor accepting one argument of type T_x exists in the implementation of T_w and invokes it on x if the `instanceof` check fails.

6.2 Handling recursion

The verification functions defined in Section 5 unwind all method invocations appearing in a formula being translated into assertions expressed in terms of the `matches` and `ensures` clauses of the methods. In general, these translations may not be well-founded when the `matches` and `ensures` clauses of methods are mutually dependent, or in invariants of mutually recursive types. Nevertheless, the verification may be successful without fully unrolling all facts about method calls and types. We use Z3's external theory plugin to implement lazy assertions by introducing interpreted theory predicates and functions. Our external theory for Z3 expands facts about type invariants and about matching preconditions and postconditions only when instances of the theory predicates are assigned a truth value. For example, if an instance of the predicate on procedure invocation is assigned false, the negation of the `matches` clause of the associated procedure is asserted on the procedure inputs. If the instance is assigned true, the `ensures` clause is asserted. An interpreted theory function is used to enforce the uniqueness of procedure outputs when the procedure is a (partial) function.

Because Z3 treats each asserted axiom as global, every instantiated axiom is asserted as an implication whose premise is the assigned predicate. Z3 also keeps track of every asserted theory predicate in its logical context, which allows proving exhaustiveness using class invariants without unrolling them entirely. To prevent unbounded unrolling, iterative deepening [12] is used to unroll as deeply as possible within a time budget. Since the theory will not further expand facts beyond the maximum depth, Z3 concludes that no satisfying assignment exists. If this happens when checking exhaustiveness, the compiler warns that it did not find a counterexample to exhaustiveness, but that there might be one.

7. Evaluation

Our evaluation of JMatch 2.0 aims to answer three kinds of questions:

- Is the extended language expressive? In particular, does it permit concise implementations? What annotation burden is incurred by programmers using the new verification procedures?
- Is the verification performed by our implementation effective on different kinds of code?
- What is the compile-time overhead of verification?

7.1 Code examples

We have evaluated our prototype JMatch 2.0 implementation on a variety of different coding problems. For each of these code examples, we have shown that the compiler correctly performs the three verification tasks described above, and we have measured the time taken by verification and compared it to total compilation time. To evaluate expressiveness, we have also implemented each example as concisely as we could using Java.

Natural numbers The implementations of natural numbers shown earlier in the paper are also used for our evaluation.

Lists A JMatch 2.0 interface `List` for immutable lists is shown in Figure 10. We implement this interface in four very different ways: the empty list (`EmptyList`), regular cons lists (`ConsList`), snoc lists (`SnocList`) in which elements are appended to the end, and lists with an array representation (`ArrList`), in which the underlying array is imperatively updated by `cons` but is shared as much as possible across multiple lists that record indices into it. To give the flavor of these implementations, the figure shows how the multimodal named constructor `snoc` is defined for `ConsList`. As the remaining code in the figure shows, these four list implementations interoperate smoothly, and list operations, even including `reverse`, can be used as patterns.

CPS We implement CPS conversion of a simple abstract syntax tree (AST) for lambda calculus; though Figure 5 shows only the key code, the implementation also includes AST classes.

Type inference We implement unification-based type inference over the same ASTs, augmented with type declarations. The code for type inference is placed within the AST node classes.

Trees A JMatch 2.0 interface `Tree` for binary trees is shown in Figure 11. We implement the AVL tree based on this interface. The `rebalance` method, also shown in the figure, returns the balanced version of the input subtree having `v` as the value at the root and `l` and `r` as its children. The invariant of `Tree` and the `ensures` clause of `branch` are crucial for the JMatch 2.0 compiler to verify that the formula in `rebalance` covers all the possible input subtrees. Checking the disjoint disjunctions also ensures that there is only one way to match each tree.

Collections We convert the prior JMatch reimplementation of the key collection classes from the Java collections framework [17] into JMatch 2.0. This code base includes implementations of various data structures: hash tables, red-black trees, and resizable arrays.

```
interface List {
  invariant(this = nil() | cons(_, _));
  constructor nil() matches(notall(result));
  constructor cons(Object hd, List tl)
    matches(notall(result)) returns(hd, tl);
  constructor snoc(List hd, Object tl)
    matches ensures(cons(_, _)) returns(hd, tl);
  constructor equals(List l);
  constructor reverse(List l) matches(true) returns(l);
  boolean contains(Object elem) iterates(elem);
  int size();
}
constructor snoc(List h, Object t) // in ConsList
  matches ensures(cons(_, _)) returns(h, t) (
    h = EmptyList.nil() && cons(t, h)
  | h = cons(Object hh, List ht) && cons(hh, snoc(ht, t))
)
static int length(List l) {
  switch (l) {
    case nil(): return 0;
    case snoc(List t, _): return length(t) + 1;
    case cons(_, List t): return length(t) + 1;
      // detected as redundant
  }
}
List l = EmptyList.nil();  // l = []
l = SnocList.cons(0, l);   // l = [[], 0]
l = ConsList.snoc(l, 1);   // l = [0, [1, []]]
l =  ArrList.snoc(l, 2);   // l = [0, 1, 2]
l = ConsList.cons(3, l);   // l = [3, [0, 1, 2]]
let l = reverse(List r1);  //r1 = [2, [1, [0, [3, []]]]]
l =  ArrList.cons(4, l);   // l = [4, 3, 0, 1, 2]
let l = reverse(List r2);  //r2 = [2, 1, 0, 3, 4]
```

Figure 10. List interface and sample usage.

Implementation	JMatch	Java	w/o verif	w/ verif
Nat	41 (21)	29	0.100	0.104
PZero	85	189	0.258	0.331
PSucc	98	226	0.280	0.435
ZNat	161	319	0.377	0.459
List	114 (72)	91	0.129	0.123
EmptyList	164	455	0.416	0.510
ConsList	309	1007	0.807	2.47
SnocList	311	1006	1.05	3.36
ArrList	473	1208	0.864	1.90
Expr	96 (57)	80	0.710	0.846
Variable	192	434	0.689	0.852
Lambda	239	500	1.20	1.52
TypedLambda	86	92	1.38	1.57
Apply	232	506	1.15	2.31
CPS	325	1279	7.88	8.37
Type	154	187	0.218	0.307
BaseType	73	163	0.350	0.443
ArrowType	82	189	0.357	0.444
UnknownType	154	245	0.372	0.490
Environment	211	310	0.695	0.862
Tree	114 (44)	69	0.165	0.170
Leaf	124	351	0.420	0.510
Branch	202	553	0.529	0.682
AVLTree	439	720	2.84	9.01
ArrayList	773	1098*	1.67	1.81
LinkedList	886	1232*	2.00	2.20
HashMap	1082	1874*	3.41	3.66
TreeMap	3606	3955*	5.90	6.43

Table 1. The number of tokens for implementations in JMatch 2.0 versus Java. Interface token counts are reported both with and without (in parentheses) `matches` and `ensures` clauses. Verification overhead is given in seconds as the average of 24 runs, with a standard deviation of at most 15%. Some comparisons (*) are versus a PolyJ [21] implementation that is more concise than the Java one. For example, the PolyJ `TreeMap` is 20% shorter than the Java equivalent [17].

7.2 Expressiveness

We can assess the expressiveness of JMatch 2.0 by comparing the number of language tokens needed to implement each of the examples. The resulting token counts shown in Table 1 indicate that JMatch 2.0 code is considerably more concise than in Java: 42.9% shorter on average. This conciseness is largely due to the JMatch support for modal abstraction and for equality constructors.

7.3 Effectiveness

There are three new verification tasks. First, `switch` and related constructs (`let`, `cond`, etc.) should be exhaustive. Second, method implementations must be correct with respect to both their declared `matches` clause and their `ensures` clause. Third, disjoint disjunctions must indeed be disjoint, to verify multiplicity.

All of the examples shown in the table, and all prior examples shown in the paper, are successfully verified for exhaustiveness, (non-)redundancy, and multiplicity. The compiler caught several subtle exhaustiveness bugs during development of this code, such as incorrect order of arguments to methods and invocation to an unexpected implementation of overloaded or overridden methods. In case of `TreeMap`, the absence of red-black tree invariants results in a nonexhaustive warning in the `balance` method.

7.4 Efficiency

Table 1 shows that verification time is reasonable for all of the code examples, even with our unoptimized prototype implementation. The reported numbers include compilation time of dependencies but exclude the overhead of initializing the compiler (689 ms) and the Z3 solver (680 ms). On average, the verification overhead on the evaluated code is 37.5% compared to the regular compilation time.

The speed of verification is not surprising, because verification is performed one method at a time. Verification is simple and tractable because the abstraction mechanisms we introduced to JMatch allow both programmers and the SMT solver to reason locally about code.

Because JMatch 2.0 does not significantly change the dynamic semantics of JMatch, the translation to Java is essentially unchanged. The performance of the compiled programs is therefore similar to one in the previous evaluation [17].

8. Related work

Integrating pattern matching with objects and data abstraction has been the subject of quite a few research efforts.

Case classes in the Scala programming language [22], as in Pizza [23], provide pattern matching by allowing case-class constructors in case arms. Scala uses *sealed classes* to limit the number of case classes that can inherit them. This makes exhaustiveness easy to verify, but sacrifices extensibility because only one implementation is allowed per declaration of a sealed class. Our `invariant` declaration achieves the same level of exhaustiveness checking but allows programmers to extend classes freely. Closely related approaches include extensible algebraic data types [32] and polymorphic variants [8], which support some extensibility and

```
interface Tree {
  invariant(this = leaf() | branch(_,_,_));
  constructor leaf() matches(height() = 0) ensures(height() = 0);
  constructor branch(Tree l, int v, Tree r) matches(height() > 0) ensures(height() > 0 &&
      (height() = l.height() + 1 && height() > r.height() || height() > l.height() && height() = r.height() + 1))
    returns(l, v, r);
  int height() ensures(result >= 0);
}

static Tree rebalance(Tree l, int v, Tree r) matches(true) (                                    // in AVLTree
  result = Branch(Branch(Tree a, int x, Tree b), int y, Branch(Tree c, int z, Tree d))
    && ( l.height() - r.height() > 1 && d = r && z = v                                          // rotation from left
           && ( l = branch(Tree ll, y, c) && ll = branch(a, x, b) && ll.height() >= c.height()  // case 1: single rot.
              | l = branch(a, x, Tree lr) && lr = branch(b, y, c) && a.height() < lr.height())  // case 2: double rot.
       | r.height() - l.height() > 1 && a = l && x = v                                          // rotation from right
           && ( r = branch(Tree rl, z, d) && rl = branch(b, y, c) && rl.height() > d.height()   // case 3: double rot.
              | r = branch(b, y, Tree rr) && rr = branch(c, z, d) && b.height() <= rr.height())) // case 4: single rot.
  | abs(l.height() - r.height()) <= 1 && result = Branch(l, v, r)
)
```

Figure 11. Tree interface and the AVL tree `rebalance` method, which uses the interface to check for totality.

deep pattern matching, but tie pattern matching to the data representation more than is ideal.

Wadler's views [30] were an early, influential generalization of pattern matching. Views require an explicitly defined bijection between the abstract view and the representation. Unlike in our language, views do not reconcile pattern matching with subtyping and do not allow matching without knowing the identity of the implementation.

Extractors are introduced in [5] as an alternative to case classes that is compatible with data abstraction. Each extractor contains `apply` and `unapply` methods, called implicitly during construction and pattern matching. There is no check that these methods are inverses, however. Modal abstractions in JMatch are less verbose and reduce the chance of such errors. No exhaustiveness checking was proposed for extractors. Dotta et al. [4] verify extractors by relying on sealed classes, and support user-defined constructor patterns. Their work does check for pattern disjointness; abstraction prevents us from making this guarantee.

Active patterns in F# [29] are similar to extractors, but support exhaustiveness checking by allowing the declaration that a set of patterns is complete. Because they offer only a backward mode, they do not support algebraic reasoning in the same way as modal abstractions. They also do not support object-oriented extensibility.

The RINV language of Wang et al. [31] also uses invertible computation to implement pattern matching that is compatible with data abstraction. Rather than extracting computations from a logical characterization of the computation, RINV instead uses a restricted language for abstraction functions that guarantees invertibility. These functions are bidirectional rather than fully multimodal and do not support iterative modes. RINV analyzes exhaustiveness via specifications of complete sets of constructors, but does not verify these specifications. RINV supports neither subtyping nor extensibility.

Suter et al. [27] also use abstraction functions to reduce algebraic data types to abstract values such as multisets, and use known theories of these abstract values to reason about data types. Methods may be annotated with a postcondition in terms of abstraction functions. Leon [28] extends this reasoning to recursive programs. Used in conjunction with sealed classes, these decision procedures assist in a more precise analysis of pattern exhaustiveness by taking type refinement into account. These decision procedures do not support modal abstraction.

An orthogonal approach to integrating pattern matching into object-oriented languages is predicate dispatch [6, 20], which extends multimethods with the ability to choose an implementation based on general predicates over the arguments. Predicate dispatch appears to be largely orthogonal and complementary to the pattern matching mechanisms described here. The predicates in prior work on predicate dispatch are, however, less expressive than those we have explored here. OOMatch [24] uses pattern matching in predicate dispatch. Its deconstructors are similar to the backward mode of JMatch constructors. OOMatch's pattern matching differs in that it can appear only in method headers as part of predicate dispatch, and no separation of specification and implementation is provided. HydroJ [15] uses predicate dispatch to express extensible communication patterns in distributed systems; however, pattern matching is done over concrete data structures, tagged trees.

Matchete [10] extends Java with pattern matching operators similar to extractors, but matches on regular expressions and other specialized expressions. It does not analyze exhaustiveness.

The Thorn language integrates patterns to make code more concise and robust [1]. Its rich set of patterns includes boolean combinations of patterns, general list patterns, regular expressions, and first-class patterns. First-class patterns in Thorn provide pattern abstraction that supports evolution of the data structure used in pattern matching, but Thorn does not support multiple implementations. As a dynamic language, Thorn does not check exhaustiveness.

Harmony and the Boomerang language [2, 7] support bidirectional computations over trees and strings through domain-specific lens combinators. The types in these languages support reasoning about the totality of transformations in these domains, but data abstraction is not a feature of these languages.

One focus of research on pattern matching has been on how to generate efficient code that shares computation across different patterns (e.g., [13]). Such optimizations are orthogonal to this work.

9. Conclusions

A clean integration of pattern matching into the object-oriented setting could simplify many programming tasks. Prior work has not managed to provide expressive pattern matching with strong data abstraction and subtyping, along with statically checked exhaustiveness. This is the first work that manages to combine these important features. We improved the integration of pattern matching with object-oriented programming, yet showed that even with this more powerful pattern matching, it is possible to reason statically about exhaustiveness, redundancy, totality, and multiplicity.

The most important insight was that programmers need to be able to specify the precondition for successful pattern matching in an abstract way. We showed that it is possible to do this while keeping the annotation burden low, by automatically extracting matching preconditions. The specification techniques introduced may be helpful for other models of multidirectional computation.

Acknowledgments

Soam Vasani and Denis Bueno got the JMatch exhaustiveness project off to a good start. Leo de Moura helped with using Z3. Ross Tate, Owen Arden, Aslan Askarov, Jed Liu, Harry Terkelsen, Mark Reitblatt, Robert Soulé, Mike George, and Danfeng Zhang offered many helpful suggestions for the presentation.

This research was supported in part by Office of Naval Research Grants N00014-09-1-0652 and N00014-13-1-0089, and by a grant administered by the Air Force Research Laboratory.

References

[1] Bard Bloom and Martin J. Hirzel. Robust scripting via patterns. In *Proc. 8th symposium on Dynamic languages (DLS)*, pp. 29–40, 2012.

[2] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *Proc. 35th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 407–419, January 2008.

[3] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[4] Mirco Dotta, Philippe Suter, and Viktor Kuncak. On static analysis for expressive pattern matching. Technical report, École Polytechnique Fédérale de Lausanne, February 2008.

[5] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In Erik Ernst, editor, *Proc. 21st European Conf. on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pp. 273–298. Springer Berlin Heidelberg, 2007.

[6] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *Proc. 12th European Conf. on Object-Oriented Programming*, pp. 186–211, 1998.

[7] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[8] Jacques Garrigue. Typing deep pattern-matching in presence of polymorphic variants. In *JSSST Workshop on Programming and Programming Languages*, 2004.

[9] Ralph E. Griswold, David R. Hanson, and John T. Korb. Generators in Icon. *ACM Transaction on Programming Languages and Systems*, 3(2):144–161, April 1981.

[10] Martin Hirzel, Nathaniel Nystrom, Bard Bloom, and Jan Vitek. Matchete: Paths through the pattern matching jungle. In *Proc. 10th Int'l Conf. on Practical Aspects of Declarative Languages*, pp. 150–166, 2008.

[11] Chinawat Isradisaikul and Andrew C. Myers. Reconciling exhaustive pattern matching with objects. Technical report, Computing and Information Science, Cornell University, March 2013. `http://hdl.handle.net/1813/33123`.

[12] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, September 1985.

[13] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proc. 6th ACM SIGPLAN Int'l Conf. on Functional Programming*, pp. 26–37, 2001.

[14] Gary T. Leavens and Peter Müller. Information hiding and visibility in interface specifications. In *Proc. 33rd Int'l Conf. on Software Engineering (ICSE)*, pp. 385–395, 2007.

[15] Keunwoo Lee, Anthony LaMarca, and Craig Chambers. HydroJ: object-oriented pattern matching for evolvable distributed systems. In *Proc. 18th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pp. 205–223, 2003.

[16] Barbara Liskov, Russell Atkinson, Toby Bloom, J. Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Springer-Verlag, 1984. Also published as Lecture Notes in Computer Science 114, G. Goos and J. Hartmanis, Eds., Springer-Verlag, 1981.

[17] Jed Liu, Aaron Kimball, and Andrew C. Myers. Interruptible iterators. In *Proc. 33rd ACM Symposium on Principles of Programming Languages (POPL)*, pp. 283–294, January 2006.

[18] Jed Liu and Andrew C. Myers. JMatch: Java plus pattern matching. Technical Report TR2002-1878, Computer Science Department, Cornell University, October 2002. Software release at `http://www.cs.cornell.edu/projects/jmatch`.

[19] Jed Liu and Andrew C. Myers. JMatch: Iterable abstract pattern matching for Java. In *Proc. 5th International Symposium on Practical Aspects of Declarative Languages*, pp. 110–127, January 2003.

[20] Todd Millstein. Practical predicate dispatch. In *Proc. 19th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pp. 345–364, 2004.

[21] Andrew C. Myers, Barbara Liskov, and Nicholas Mathewson. PolyJ: Parameterized types for Java. Software release, at `http://www.cs.cornell.edu/polyj`, July 1998.

[22] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[23] Martin Odersky and Philip Wadler. Pizza into Java: translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 146–159, 1997.

[24] Adam Richard and Ondřej Lhoták. OOMatch: pattern matching as dispatch in Java. In *Companion 22nd ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pp. 771–772, 2007.

[25] Clyde Ruby and Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. In *Proc. 15th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pp. 208–228, 2000.

[26] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1–3):17–64, October–December 1996.

[27] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In *Proc. 37th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 199–210, 2010.

[28] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *Proc. 18th Int'l Conf. on Static Analysis*, pp. 298–315, 2011.

[29] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *Proc. 12th ACM SIGPLAN Int'l Conf. on Functional Programming*, pp. 29–40, 2007.

[30] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proc. 14th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 307–313, 1987.

[31] Meng Wang, Jeremy Gibbons, Kazutaka Matsuda, and Zhenjiang Hu. Refactoring pattern matching. *Science of Computer Programming*, in press. Available online 24 August 2012.

[32] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proc. 6th ACM SIGPLAN Int'l Conf. on Functional Programming*, pp. 241–252, September 2001.