JMatch: Iterable Abstract Pattern Matching for Java

Jed Liu and Andrew C. Myers

Computer Science Department
Cornell University, Ithaca, New York

Abstract. The JMatch language extends Java with *iterable abstract pattern matching*, pattern matching that is compatible with the data abstraction features of Java and makes iteration abstractions convenient. JMatch has ML-style deep pattern matching, but patterns can be abstract; they are not tied to algebraic data constructors. A single JMatch method may be used in several modes; modes may share a single implementation as a boolean formula. Modal abstraction simplifies specification and implementation of abstract data types. This paper describes the JMatch language and its implementation.

1 Introduction

Object-oriented languages have become a dominant programming paradigm, yet they still lack features considered useful in other languages. Functional languages offer expressive pattern matching. Logic programming languages provide powerful mechanisms for iteration and backtracking. However, these useful features interact poorly with the data abstraction mechanisms central to object-oriented languages. Thus, expressing some computations is awkward in object-oriented languages.

In this paper we present the design and implementation of JMatch, a new object-oriented language that extends Java [GJS96] with support for *iterable abstract pattern matching*—a mechanism for pattern matching that is compatible with the data abstraction features of Java and that makes iteration abstractions more convenient. This mechanism subsumes several important language features:

- convenient use and implementation of iteration abstractions (as in CLU [L+81], ICON [GHK81], and Sather [MOSS96].)
- convenient run-time type discrimination without casts (for example, Modula-3's `typecase` [Nel91])
- deep pattern matching allows concise, readable deconstruction of complex data structures (as in ML [MTH90], Haskell [Jon99] and Cyclone [JMG+02].)
- multiple return values
- views [Wad87]
- patterns usable as first-class values [PGPN96,FB97]

Email: `jed@cs.washington.edu`, `andru@cs.cornell.edu`. Jed Liu is now at the University of Washington, Seattle, WA. This research was supported by DARPA Contract F30602-99-1-0533, monitored by USAF Rome Laboratory, by ONR Grant N00014-01-1-0968, and by an Alfred P. Sloan Research Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright annotation thereon.

JMatch exploits two key ideas: *modal abstraction* and *invertible computation*. Modal abstraction simplifies the *specification* (and use) of abstractions; invertible computation simplifies the *implementation* of abstractions.

JMatch constructors and methods may be modal abstractions: operations that support multiple *modes* [SHC96]. Modes correspond to different directions of computation, where the ordinary direction of computation is the "forward" mode, but backward modes may exist that compute some or all of a method's arguments using an expected result. Pattern matching uses a backward mode. A mode may specify that there can be multiple values for the method outputs; these can be easily iterated over in a predictable order. Modal abstraction simplifies the specification and use of abstract data type (ADT) interfaces, because where an ADT would ordinarily have several distinct but related operations, in JMatch it is often natural to have a single operation with multiple modes.

The other key idea behind JMatch is invertible computation. Computations may be described by boolean formulas that express the relationship among method inputs and outputs. Thus, a single formula may implement multiple modes; the JMatch compiler automatically decides for each mode how to generate the outputs of that mode from the inputs. Each mode corresponds to a different direction of evaluation. Having a single implementation helps ensure that the modes implement the abstraction in a consistent manner, satisfying expected equational relationships.

These ideas appear in various logic programming languages, but it is a challenge to integrate these ideas into an object-oriented language in a natural way that enforces data abstraction, preserves backwards compatibility, and permits an efficient implementation. JMatch is not a general-purpose logic-programming language; it does not provide the full power of unification over logic variables. This choice facilitates an efficient implementation. However, JMatch does provide more expressive pattern matching than logic-programming, along with modal abstractions that are first-class values (objects).

Although JMatch extends Java, little in this paper is specific to Java. The ideas in JMatch could easily be applied to other garbage-collected object-oriented languages such as C# [Mic01] or Modula-3 [Nel91].

A prototype compiler for JMatch is available for download. It is built using the Polyglot extensible Java compiler framework [NCM02], which supports source-to-source translation into Java.

The rest of this paper is structured as follows. Section 2 provides an overview of the JMatch programming language. Section 3 gives examples of common programming idioms that JMatch supports clearly and concisely. Section 4 describes the implementation of the prototype compiler. Section 5 discusses related work. Section 6 summarizes and concludes with a discussion of useful extensions to JMatch.

2 Overview of JMatch

JMatch provides convenient specification and implementation of computations that may be evaluated in more than one direction, by extending expressions to *formulas* and *patterns*. Named abstractions can be defined for formulas and patterns; these abstractions are called *predicate methods*, *pattern methods*, and *pattern constructors*. JMatch ex-

tends the meaning of some existing Java statements and expressions, and adds some new forms. It is backwards compatible with Java.

2.1 Formulas

Syntactically, a JMatch formula is similar to a Java expression of boolean type, but where a Java expression would permit a subexpression of type T, a formula may include a variable declaration with type T. For example, the expression `2 + int x == 5` is a formula that is satisfied when `x` is bound to `3`.

JMatch has a `let` statement that tries to satisfy a formula, binding new variables as necessary. For example, the statement `let 2 + int x == 5;` causes `x` to be bound to `3` in subsequent code (unless it is later reassigned). If there is no satisfying assignment, an exception is raised. To prevent an exception, an `if` statement may be used instead. The conditional may be any formula with at most one solution. If there is a satisfying assignment, it is in scope in the "then" clause; if there is no satisfying assignment, the "else" clause is executed but the declared variables are not in scope. For example, the following code assigns `y` to an array index such that `a[y]` is nonzero (the `single` restricts it to the first such array index), or to `-1` if there is no such index:

```
int y;
if (single(a[int i] != 0)) y = i;
else y = -1;
```

A formula may contain free variables in addition to the variables it declares. The formula expresses a relation among its various variables; in general it can be evaluated in several modes. For a given mode of evaluation these variables are either *knowns* or *unknowns*. In the *forward mode*, all variables, including bound variables, are knowns, and the formula is evaluated as a boolean expression. In backward modes, some variables are unknowns and satisfying assignments are sought for them. If JMatch can construct an algorithm to find satisfying assignments given a particular set of knowns, the formula is *solvable* in that mode. A formula with no satisfying assignments is considered solvable as long as JMatch can construct an algorithm to determine this.

For example, the formula `a[i] == 0` is solvable if the variable `i` is an unknown, but not if the variable `a` is an unknown. The modes of the array index operator `[]` do not include any that solve for the array, because those modes would be largely useless (and inefficient).

Some formulas have multiple satisfying assignments; the JMatch `foreach` statement can be used to iterate through these assignments. For example, the following code adds the indices of all the non-zero elements of an array:

```
foreach(a[int i] != 0) n += i;
```

In formulas, the single equals sign (`=`) is overloaded to mean equality rather than assignment, while preserving backwards compatibility with Java. The symbol `=` corresponds to semantic equality in Java (that is, the `equals` method of class `Object`). Formulas may use either pointer equality (`==`) or semantic equality (`=`); the difference between the two is observable only when an equation is evaluated in forward mode,

where the Java `equals` method is used to evaluate `=`. Otherwise an equation is satisfied by making one side of the equation pointer-equal to the other—and therefore also semantically equal. Because semantic equality is usually the right choice for JMatch programs, concise syntax is important. The other Java meanings for the symbol `=` are initialization and assignment, which can be thought of as ways to satisfy an equation.

2.2 Patterns

A pattern is a Java expression of non-boolean type except that it may contain variable declarations, just like a formula. In its forward mode, in which all its variables are knowns, a pattern is evaluated directly as the corresponding Java expression. In its backward modes, the value of the pattern is a known, and this value is used to reconstruct some or all of the variables used in the pattern. In the example above, the subexpression `2 + int x` is a pattern with type `int`, and given that its value is known to be `5`, JMatch can determine `x = 3`. Inversion of addition is possible because the addition operator supports the necessary computational mode; not all binary operators support this mode. Another pattern is the expression `a[int i]`. Given a value `v` to match against, this pattern iterates over the array `a` finding all indices `i` such that `v = a[i]`. There may be many assignments that make a pattern equal to the matched value. When JMatch knows how to find such assignments, the pattern is *matchable* in that mode. A pattern p is matchable if the equation $p = v$ is solvable for any value v.

The Java `switch` statement is extended to support general pattern matching. Each of the `case` arms of a `switch` statement may provide a pattern; the first arm whose pattern matches the tested value is executed.

The simplest pattern is a variable name. If the type checker cannot statically determine that the value being matched against a variable has the same type, a dynamic type test is inserted and the pattern is matched only if the test succeeds. Thus, a `typecase` statement [Nel91] can be concisely expressed as a `switch` statement:

```
Vehicle v; ...
switch (v) {
    case Car c: ...
    case Truck t: ...
    case Airplane a: ...
}
```

For the purpose of pattern matching there is no difference between a variable declaration and a variable by itself; however, the first use of the variable must be a declaration.

2.3 Pattern constructors

One way to define new patterns is *pattern constructors*, which support conventional pattern matching, with some increase in expressiveness. For example, a simple linked list (a “cons cell”, really) naturally accommodates a pattern constructor:

```
public class List implements Collection {
    Object head;
    List tail;
    public List(Object h, List t) returns(h, t) (
        head = h && tail = t
    )
    ...
}
```

This constructor differs in two ways from the corresponding Java constructor whose body would read `{head = h; tail = t; }`. First, the mode clause `returns(h,t)` indicates that in addition to the implicit forward mode in which the constructor makes a new object, the constructor also supports a mode in which the result object is a known and the arguments `h` and `t` are unknowns. It is this backward mode that is used for pattern matching. Second, the body of the constructor is a simple formula (surrounded by parentheses rather than by braces) that implements both modes at once. Satisfying assignments to `head` and `tail` will build the object; satisfying assignments to `h` and `t` will deconstruct it.

For example, this pattern constructor can be applied in ways that will be familiar to ML programmers:

```
List l;
...
switch (l) {
    case List(Integer x, List(Integer y, List rest)): ...
    default: ...
}
```

The `switch` statement extracts the first two elements of the list into variables `x` and `y` and executes the subsequent statements. The variable `rest` is bound to the rest of the list. If the list contains zero or one elements, the `default` case executes with no additional variables in scope. Even for this simple example, the equivalent Java code is awkward and less clear. In the code shown, the constructor invocations do not use the `new` keyword; the use of `new` is optional.

The `List` pattern constructor also matches against subclasses of `List`; in that case it inverts the construction of only the `List` part of the object.

It is also possible to match several values simultaneously:

```
List l1, l2; ...
switch (l1, l2) {
    case List(Object x, List(Integer y,List r)), List(y, _): ...
    default: ...
}
```

The first case executes if the list `l1` has at least two elements, and the head of list `l2` exists and is an `Integer` equal to the second element of `l1`. The remainder of `l2` is matched using the wildcard pattern "`_`".

In this example of a pattern constructor, the constructor arguments and the fields correspond directly, but this need not be the case. More complex formulas can be used to implement views as proposed by Wadler [Wad87] (see Section 3.4).

The example above implements the constructor using a formula, but backwards compatibility is maintained; a constructor can be written using the usual Java syntax.

2.4 Methods and modal abstraction

The language features described so far subsume ML pattern matching, with the added power of invertible boolean formulas. JMatch goes further; pattern matching coexists with abstract data types and subtyping, and it supports iteration.

Methods with `boolean` return type are *predicate methods* that define a named abstraction for a boolean formula. The forward mode of a predicate method expects that all arguments are known and executes the method normally. In backward modes, satisfying assignments to some or all of the method arguments are sought. Assuming that the various method modes are implemented consistently, the corresponding forward invocation using these satisfying assignments would have the result `true`.

Predicate methods with multiple modes can make ADT specifications more concise. For example, in the Java Collections framework the `Collection` interface declares separate methods for finding all elements and for checking if a given object is an element:

```
boolean contains(Object o);
Iterator iterator();
```

In any correct Java implementation, there is an equational relationship between the two operations: any object `x` produced by the iterator object satisfies `contains(x)`, and any object satisfying `contains(x)` is eventually generated by the iterator. When writing the specification for `Collection`, the specifier must describe this relationship so implementers can do their job correctly.

By contrast, a JMatch interface can describe both operations with one declaration:

```
boolean contains(Object o) iterates(o);
```

This declaration specifies two modes: an implicit forward mode in which membership is being tested for a particular object `o`, and a backward mode declared by `iterates(o)`, which iterates over all contained objects. The equational relationship is captured simply by the fact that these are modes of the same method.

An interface method signature may declare zero or more additional modes that the method implements, beyond the default, forward mode. A mode `returns`$(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are argument variable names, declares a mode that generates a satisfying assignment for the named variables. A mode `iterates`$(x_1, \ldots, x_n)$ means that the method iterates over a *set* of satisfying assignments to the named variables.

Invocations of predicate methods may appear in formulas. The following code iterates over the Collection `c`, finding all elements that are lists whose first element is a green truck; the loop body executes once for each element, with the variable `t` bound to the `Truck` object.

```
foreach (c.contains(List(Truck t, _)) && t.color() = GREEN)
    System.out.println(t.model());
```

2.5 Implementing methods

A linked list is a simple way to implement the `Collection` interface. Consider the linked list example again, where the `contains` method is no longer elided:

```
public class List implements Collection {
    Object head; List tail;
    public List(Object h, List t) returns(h, t) ...
    public boolean contains(Object o) iterates(o) (
        o = head || tail.contains(o)
    )
}
```

As with constructors, multiple modes of a method may be implemented by a formula instead of a Java statement block. Here, the formula implements both modes of `contains`. In the forward mode there are no unknowns; in the backward mode the only unknown is `o`, as the clause `iterates(o)` indicates.

In the backward mode, the disjunction signals the presence of iteration. The two subformulas separated by `||` define two different ways to satisfy the formula; both will be explored to find satisfying assignments for `o`.

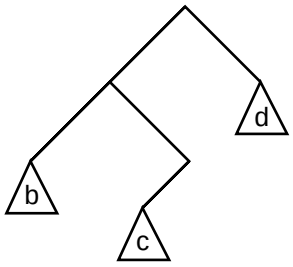
The modes of a method may be implemented by separate formulas or by ordinary Java statements, which is useful when no single boolean formula is solvable for all modes, or it leads to inefficient code. For example, the following code separately implements the two modes of `contains`:

```
public boolean contains(Object o) {
    if (o.equals(head)) return true;
    return tail.contains(o);
} iterates(o) {
    o = head;
    yield;
    foreach (tail.contains(Object tmp)) {
        o = tmp;
        yield;
    }
}
```

For backward modes, results are returned from the method by the `yield` statement rather than by `return`. The `yield` statement transfers control back to the iterating context, passing the current values of the unknowns. While this code is longer and no faster than the formula above, it is simpler than the code of the corresponding Java iterator object. The reason is that iterator objects must capture the state of iteration so they can restart the iteration computation whenever a new value is requested. In this example, the state of the iteration is implicitly captured by the position of the `yield` statement and the local variables; restarting the iteration is automatic. In essence, iteration requires the expressive power of coroutines [Con63,L$^+$81,GHK81]. Implementing iterator objects requires coding in continuation-passing style (CPS) to obtain this power [HFW86],

which is awkward and error-prone [MOSS96]. The JMatch implementation performs a CPS conversion behind the scenes.

2.6 Pattern methods




```
interface Tree {
  Tree node(Tree l, Tree r, Object o)
    returns(l, r, o);
  Tree empty();
}

Tree a = ...;
...
switch (a) {
  case Tree.node(
    Tree.node(Tree b,
              Tree.node(Tree c,
                Tree.empty())),
    Tree d):
      ...
}
```

(specification and use)

```
class RBNode implements Tree {
    RBNode lf, rg;
    int color;  // RED or BLACK
    Object value;
    Tree node(Tree l, Tree r)
      ( false )  // forward mode
      returns (l, r) (
        l = lf &&
        r = rg
      )
    Tree empty() returns()
      ( false )  // both modes
    ...
}
class RBEmpty implements Tree {
    static Tree _empty = RBEmpty();
    Tree node(Tree l, Tree r)
      returns (l, r)
        ( false )  // both modes
    Tree empty()
      returns() ( result = _empty )
    ...
}
```

(implementation)

Fig. 1. Deep abstract pattern matching

JMatch methods whose return type is not boolean are *pattern methods* whose result may be matched against other values if the appropriate mode is implemented. Pattern methods provide the ability to deconstruct values even more abstractly than pattern constructors do, because a pattern method declared in an interface can be implemented in different ways in the classes that implement the interface.

For example, many data structure libraries contain several implementations of trees (e.g., binary search trees, red-black trees, AVL trees). When writing a generic tree-walking algorithm it may be useful to pattern-match on tree nodes to extract left and right children, perhaps deep in the tree. This would not be possible in most languages with pattern matching (such as ML or Haskell) because patterns are built from constructors, and thus cannot apply to different types. An abstract data type is implemented in

these languages by hiding the actual type of the ADT values; however, this prevents any pattern matching from being performed on the ADT values. Thus, pattern matching is typically incompatible with data abstraction.

By contrast, in JMatch it is possible to declare pattern methods in an interface such as the `Tree` interface shown on the left side of Figure 1. As shown in the figure, these pattern methods can then be used to match the structure of the tree, without knowledge of the actual `Tree` implementation being matched.

An implementation of the pattern methods `node` and `empty` for a red-black tree is shown on the right side of Figure 1. Here there are two classes implementing red-black trees. For efficiency there is only one instance of the empty class, called `_empty`. The `node` and `empty` pattern methods are only intended to be invoked in the backwards mode for pattern-matching purposes. Thus, the ordinary forward mode is implemented by the unsatisfiable formula `false`.

As this example suggests, the rule for resolving method invocations is slightly different for JMatch. A non-static pattern method m of class T can be invoked using the syntax $T.m$, in which case the receiver of the method is the object being matched. JMatch has a pattern operator `as`; the pattern (P_1 `as` P_2) matches a value if both P_1 and P_2 match it. A pattern T`.`m`()` is syntactic sugar for the pattern `(`T y `as` y`.`m`())` where y is fresh.

Within a pattern method there is a special variable `result` that represents the result of the method call. Mode declarations may mention `result` to indicate that the result of the method call is an unknown. In the default, forward mode the only unknown is the variable `result`. During the method calls shown in Figure 1, the variable `result` will be bound to the same object as the method receiver `this`. This need not be true if the pattern method is invoked on some object other than the result—which allows the receiver object to be used as a first-class pattern. (The expression `this` is always a known in non-static methods.)

```
class List {
  Object head; List tail;
  static List append(List prefix, Object last) returns(prefix, last) (
      prefix = null &&                              // single element
      result = List(last, null)
    else                                            // multiple elements
      prefix = List(Object head, List ptail) &&
      result = List(head, append(ptail, last))
  )
}
List l; ...
switch(l) {
  case List.append(List.append(_, Object o1), Object o2): ...
}
```

Fig. 2. Reversible list append

Figure 2 shows an example of a static pattern method; `append` appends an element to the list in the forward direction but inverts this operation in the backward direction, splitting a list into its last element and a prefix list. In this version of `List`, empty lists are represented by `null`. The `append` method is static so that it can be invoked on empty lists. The `switch` statement shows that pattern matching can extract the last two elements of a list.

This example uses a disjunctive logical connective, `else`, which behaves like `||` except that the right-hand disjunct generates solutions only if the left-hand disjunct has not. An `else` disjunction does not by itself generate multiple solutions in backward modes; both `else` and `||` are short-circuit operators in the forward mode where the proposed solution to the formula is already known.

This example also demonstrates reordering of conjuncts in different modes. The algorithm for ordering conjuncts is simple: JMatch solves one conjunct at a time, and always picks the leftmost solvable conjunct to work on. This rule makes the order of evaluation easy to predict, which is important if conjuncts have side effects. While JMatch tends to encourage a functional programming style, it does not attempt to guarantee that formulas are free of side-effects, because side-effects are often useful.

In this example, in the backward mode the first conjunct is not initially solvable, so the conjuncts are evaluated in reverse order—in the multiple-element case, `result` is first broken into its parts, then the prefix of its tail is extracted (recursively using the `append` method), and finally the new prefix is constructed.

Pattern methods and pattern constructors obey similar rules; the main difference is that when `result` is an unknown in a pattern constructor, the variable `result` is automatically bound to a new object of the appropriate type, and its fields are exposed as variables to be solved. The list-reversal example shows that pattern methods can construct and deconstruct objects too.

2.7 Built-in patterns

Many of the built-in Java operators are extended in JMatch to support additional modes. As mentioned earlier, the array index operator `[]` supports new modes that are easy to specify if we consider the operator on the type `T[]` (array of `T`) as a method named `operator[]` after the C++ idiom:

```
static T operator[](T[] array, int index)
  iterates(index, result)
```

That is, an array has the ability to automatically iterate over its indices and provide the associated elements. Note that other than the convenient syntax of array indexing and the type parameterization that arrays provide, there is no special magic here; it is easy to write code using the `yield` statement to implement this signature, as well as for the other built-in extensions.

The arithmetic operations `+` and `-` are also able to solve for either of their arguments given the result. In Java, the operator `+` also concatenates strings. In JMatch the concatenation can be inverted to match prefixes or suffixes; all possible matching prefix/suffix pairs can also be iterated over.

Within formulas, relational expressions are extended to support a chain of relational comparisons. Certain integer inequalities are treated as built-in iterators: formulas of the form ($a_1\ \rho_1\ a_2\ \rho_2\ \ldots\ \rho_{n-1}\ a_n$), where a_1 and a_n are solvable, and all of the ρ_i are either `<` or `<=` (or else all `>` or `>=`). These formulas are solved by iteration over the appropriate range of integers between a_1 and a_n. For example, the following two statements are equivalent except that the first evaluates `a.length` only once:

```
foreach (0 <= int i < a.length) { ... }
for (int i = 0; i < a.length; i++) { ... }
```

2.8 Iterator objects

Java programmers are accustomed to performing iterations using objects that implement the `Iterator` interface. An `Iterator` is an object that acts like an input stream, delivering the next object in the iteration whenever its `next()` method is called. The `hasNext()` method can be used to test whether there is a next object.

Iterator objects are usually unnecessary in JMatch, but they are easy to create. Any formula F can be converted into a corresponding iterator object using the special expression syntax `iterate` C`(`F`)`. Given a formula with unknowns $x_1, \ldots, x_n$, the expression produces an iterator object that can be used to iterate over the possible solutions to the formula. Each time the `next()` method of the iterator is called, a container object of class C is returned that has public fields named $x_1, \ldots, x_n$ bound to the corresponding solution values.

Iterator objects in Java sometimes implement a `remove` method that removes the current element from the collection. Iterators with the ability to remove elements can be implemented by returning the (abstract) context in which the element occurs. This approach complicates the implementation of the iterator and changes its signature. Better support for such iterators remains future work.

2.9 Exceptions

The implementation of forward modes by boolean formulas raises the question of what value is returned when the formula is unsatisfiable. The `NoSuchElementException` exception is raised in that case.

Methods implemented as formulas do not have the ability to catch exceptions raised during their evaluation; a raised exception propagates out from the formula to the context using it. If there is a need to catch exceptions, the method must be implemented as a statement block instead.

In accordance with the expectations of Java programmers, exceptions raised in the body of a `foreach` iteration cannot be intercepted by the code of the predicate being tested.

3 Examples

A few more detailed examples will suggest the added expressive power of JMatch.

3.1 Functional red-black trees

A good example of the power of pattern matching is the code for recursively balancing a red-black tree on insertion. Cormen et al. [CLR90] present pseudocode for red-black tree insertion that takes 31 lines of code yet gives only two of the four cases necessary. Okasaki [Oka98a] shows that for functional red-black trees, pattern matching can reduce the code size considerably. The same code can be written in JMatch about as concisely. Figure 3 shows the key code that balances the tree. The four cases of the red-black rotation are handled by four cases of the `switch` statement that share a single `return` statement, which is permitted because they solve for the same variables (`a`–`d`, `x`–`z`).

```
static Node balance(int color, int value, RBTree left, RBTree right) {
  if (color == BLACK) {
    switch (value, left, right) {
      case int z,
           Node(RED, int y, Node(RED,int x,RBTree a,RBTree b), RBTree c),
           RBTree d:
      case z, Node(RED,x,a,Node(RED,y,b,c)), d:
      case x, c, Node(RED,z,Node(RED,y,a,b),d):
      case x, a, Node(RED,y,b,Node(RED,z,c,d)):
          return Node(RED,y,Node(BLACK,x,a,b), Node(BLACK,z,c,d));
    }
  }
  return new Node(color, value, left, right);
}
```

Fig. 3. Balancing red-black trees

3.2 Binary search tree membership

Earlier we saw that for lists, both modes of the `contains` method could be implemented as a single, concise formula. The same is true for red-black trees:

```
public boolean contains(int x) iterates(x) (
  left != null && x < value && left.contains(x) ||
  x = value ||
  right != null && x > value && right.contains(x)
)
```

In its forward mode, this code implements the usual $O(\log n)$ binary search for the element. In its backward mode, it iterates over the elements of the red-black tree in ascending order, and the tests `x < value` and `x > value` superfluously check the data-structure invariant. Automatic removal of such checks is future work.

3.3 Hash table membership

The hash table is another collection implementation that benefits in JMatch. Here is the `contains` method, with three modes implemented by a single formula:

```
class HashMap {
  HashBucket[] buckets;
  int size;
  ...
  public boolean contains(Object key, Object value)
        returns(value) iterates(key, value) (
    int n = key.hashCode() % size &&
    HashBucket b = buckets[n] &&
    b.contains(key, value)
  )
}
```

In the forward mode, the code checks whether the (key,value) binding is present in the hash table. In the second mode, a key is provided and a value efficiently located if available. The final mode iterates over all (key,value) pairs in the table. The hash table has chained buckets (`HashBucket`) that implement `contains` similarly to the earlier `List` implementation. In the final, iterative mode, the built-in array iterator generates the individual buckets `b`; the check `n = hash(key)` becomes a final consistency check on the data structure, because it cannot be evaluated until `key` is known.

The signature of the method `HashBucket.contains` is the same as the signature of `HashMap.contains`, which is not surprising because they both implement maps. The various modes of `HashMap.contains` use the corresponding modes of `HashBucket.contains` and different modes of the built-in array index operator. This coding style is typical in JMatch.

A comparison to the standard Java collection class `HashMap` [GJS96] suggests that modal abstraction can substantially simplify class signatures. The `contains` method provides the functionality of methods `get`, `iterator`, `containsKey`, `containsValue`, and to a lesser extent the methods `keySet` and `values`.

3.4 Simulating views

Wadler has proposed views [Wad87] as a mechanism for reconciling data abstraction and pattern matching. For example, he shows that the abstract data type of Peano natural numbers can be implemented using integers, yet still provide the ability to pattern-match on its values. Figure 4 shows the equivalent JMatch code. Wadler also gives an example of a view of lists that corresponds to the modes of the method `append` shown in Section 2.6.

In both cases, the JMatch version of the code offers the advantage that the forward and backwards directions of the view are implemented by a single formula, ensuring consistency. In the views version of this code, separate *in* and *out* functions must be defined and it is up to the programmer to ensure that they are inverses.

```
class Peano {
    private int n;
    private Peano(int m) returns(m) ( m = n )
    public Peano succ(Peano pred) returns(pred) (
        pred = Peano(int m) && result = Peano(m+1)
    )
    public Peano zero() returns() ( result = Peano(0) )
}
```

Fig. 4. Peano natural numbers ADT

4 Semantics and Implementation

We now touch on some of the more interesting details of the semantics of JMatch and its implementation. The JMatch compiler is built using the Polyglot compiler framework for Java language extensions [NCM02]. Polyglot supports both the definition of languages that extend Java and their translation into Java. For more details see the technical report on JMatch and the implementation notes available with the current version of the compiler [LM02].

4.1 Static semantics

Type-checking JMatch expressions, including formulas and patterns, is little different from type-checking Java expressions, since the types are the same in all modes, and the forward mode corresponds to ordinary Java evaluation.

The Java interface and abstract class conformance rules are extended in a natural way to handle method modes: a JMatch class must implement all the methods in all their modes, as declared in the interface or abstract class being implemented or extended. A method can add new modes to those defined by the super class.

The introduction of modes does create a new obligation for static checking. In JMatch it is a static error to use a formula or pattern with multiple solutions in a context (such as a `let`) where a single solution is expected, because solutions might be silently discarded. Thus, the JMatch type system is extended so that every expression has a multiplicity in addition to its ordinary Java type. The `single` operator may be used to explicitly discard the extra solutions of an expression and reduce its static multiplicity.

For each invocation of a built-in or user-defined predicate or pattern method, the compiler must select a mode to use to solve the expression in which the invocation appears. There may be more than one usable mode; the compiler selects the best mode according to a simple ordering. Modes are considered better if (in order of priority) they are not iterative, if they avoid constructing new objects, if they solve for fewer arguments, and if they are declared earlier.

One change to type checking is in the treatment of pattern method invocations. When a non-static method is invoked with the syntax $T.m$, it is a pattern method invocation of method m of type T. It would be appealing to avoid naming T explicitly but this would require type inference.

4.2 Translation to Java

In the current implementation, JMatch is translated into Java by way of an intermediate language called Java$_{\texttt{yield}}$, which is the Java 1.4 language extended with a limited `yield` statement that can only be used to implement Java iterator objects. Executing `yield` causes the iterator to return control to the calling context. The iterator object constructor and the methods `next` and `hasNext` are automatically implemented in Java$_{\texttt{yield}}$. Each subsequent invocation of `next` on the iterator returns control to the point just after the execution of the previous `yield` statement.

The benefit of the intermediate language is that the translation from JMatch to Java$_{\texttt{yield}}$ is straightforwardly defined using a few mutually inductively defined syntax-directed functions. The translation from Java$_{\texttt{yield}}$ to Java 1.4 is also straighforward; it is essentially a conversion to continuation-passing style. While the performance of the translated code is acceptable, several easy optimizations would improve code quality. See the technical report [LM02] for more details on the translation.

5 Related Work

Prolog is the best-known declarative logic programming language. It and many of its descendents have powerful unification in which a predicate can be applied to an expression containing unsolved variables. JMatch lacks this capability because it is not targeted specifically at logic programming tasks; rather, it is intended to smoothly incorporate some expressive features of logic programming into a language supporting data abstraction and imperative programming.

ML [MTH90] and Haskell [HJW92,Jon99] are well-known functional programming languages that support pattern matching, though patterns are tightly bound to the concrete representation of the value being matched. Because pattern matching in these languages requires access to the concrete representation, it does not coexist well with the data abstraction mechanisms of these languages. However, an advantage of concrete pattern matching is the simplicity of analyzing *exhaustiveness*; that is, showing that some arm of a `switch` statement will match.

Pattern matching has been of continuing interest to the Haskell community. Wadler's views [Wad87] support pattern matching for abstract data types. Views correspond to JMatch constructors, but require the explicit definition of a bijection between the abstract view and the concrete representation. While bijections can be defined in JMatch, often they can be generated automatically from a boolean formula. Views do not provide iteration.

Burton and Cameron [BC93] have also extended the views approach with a focus on improving equational reasoning. Fähndrich and Boyland [FB97] introduced first-class pattern abstractions for Haskell, but do not address the data abstraction problem. Palao Gonstanza et al. [PGPN96] describe first-class patterns for Haskell that work with data abstraction, but are not statically checkable. Okasaki has proposed integrating views into Standard ML [Oka98b]. Tullsen [Tul00] shows how to use combinators to construct first-class patterns that can be used with data abstraction. Like views, these proposals do not provide iterative patterns, modal abstraction, or invertible computation.

A few languages have been proposed to integrate functional programming and logic programming [Han97,Llo99,CL00]. The focus in that work is on allowing partially instantiated values to be used as arguments, rather than on data abstraction.

In the language Alma-0, Apt et al. [ABPS98] have augmented Modula-2, an imperative language, with logic-programming features. Alma-0 is tailored for solving search problems and unlike JMatch, provides convenient backtracking through imperative code. However, Alma-0 does not support pattern matching or data abstraction.

Mercury [SHC96] is a modern declarative logic-programming language with modularity and separate compilation. As in JMatch, Mercury predicates can have several modes, a feature originating in some versions of Prolog (e.g., [Gre87]). Modal abstractions are not first-class in Mercury; a single mode of a predicate can be used as a first-class function value, but unlike in JMatch, there is no way to pass several such modes around as an object and use them to uniformly implement another modal abstraction. Mercury does not support objects.

CLU [L$^+$81], ICON [GHK81], and Sather [MOSS96] each support iterators whose use and implementation are both convenient; the `yield` statement of JMatch was inspired by CLU. None of these languages have pattern matching.

Pizza also extends Java by allowing a class to be implemented as an algebraic datatypes and by supporting ML-style pattern matching [OW97]. Because the datatype is not exposed outside the class, Pizza does not permit abstract pattern matching. Forax and Roussel have also proposed a Java extension for simple pattern matching based on reflection [FR99].

Ernst et al. [EKC98] have developed predicate dispatching, another way to add pattern matching to an object-oriented language. In their language, boolean formulas control the dispatch mechanism, which supports encoding some pattern-matching idioms although deep pattern matching is not supported. This approach is complementary to JMatch, in which object dispatch is orthogonal to pattern matching. Their language has limited predicate abstractions that can implement a single new view of an object, but unlike JMatch, it does not unify predicates and methods. The predicates may not be recursive or iterative and do not support modal abstraction or invertible computation.

6 Conclusions

JMatch extends Java with the ability to describe modal abstractions: abstractions that can be invoked in multiple different modes, or directions of computation. Modal abstractions can result in simpler code specifications and more readable code through the use of pattern matching. These modal abstractions can be implemented using invertible boolean formulas that directly describe the relation that the abstraction computes. In its forward mode, this relation is a function; in its backward modes it may be one-to-many or many-to-many. JMatch provides mechanisms for conveniently exploring this multiplicity.

JMatch is backwards compatible with Java, but provides expressive new features that make certain kinds of programs simpler and clearer. While for some such programs, using a domain-specific language would be the right choice, having more features in a

general-purpose programming language is handy because a single language can be used when building large systems that cross several domains.

A prototype of the JMatch compiler has been released for public experimentation, and improvements to this implementation are continuing.

There are several important directions in which the JMatch language could be usefully extended. An exhaustiveness analysis for switch statements and `else` disjunctions would make it easier to reason about program correctness. Automatic elimination of tests that are redundant in a particular mode might improve performance. And support for iterators with removal would be useful.

Acknowledgments

The authors would like to thank Brandon Bray and Grant Wang for many useful discussions on the design of JMatch and some early implementation work as well. Greg Morrisett and Jim O'Toole also made several useful suggestions. Nate Nystrom supported the implementation of JMatch on Polyglot. Readers of this paper whose advice improved the presentation include Kavita Bala, Michael Clarkson, Dan Grossman, Nate Nystrom, and Andrei Sabelfeld.

References

[ABPS98] Krzysztof R. Apt, Jacob Brunekreef, Vincent Partington, and Andrea Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM Transactions on Programming Languages and Systems*, 20(5):1014–1066, September 1998.

[BC93] F. W. Burton and R. D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, 1993.

[CL00] K. Claessen and P. Ljungl. Typed logical variables in Haskell. In *Haskell Workshop 2000*, 2000.

[CLR90] Thomas A. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[Con63] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[EKC98] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *12th European Conference on Object-Oriented Programming*, pages 186–211, Brussels, Belgium, July 1998.

[FB97] Manuel Fähndrich and John Boyland. Statically checkable pattern abstractions. In *Proc. 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 75–84, June 1997.

[FR99] Remi Forax and Gilles Roussel. Recursive types and pattern matching in java. In *Proc. International Symposium on Generative and Component-Based Software Engineering (GCSE '99)*, Erfurt, Germany, September 1999. LNCS 1799.

[GHK81] Ralph E. Griswold, David R. Hanson, and John T. Korb. Generators in ICON. *ACM Transaction on Programming Languages and Systems*, 3(2), April 1981.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996. ISBN 0-201-63451-1.

[Gre87] Steven Gregory. *Parallel Programming in PARLOG*. Addison-Wesley, 1987.

[Han97] Michael Hanus. A unified computation model for functional and logic programming. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 80–93, Paris, France, January 1997.

[HFW86] C. T. Haynes, D. P. Friedman, and M. Wand. Obtaining coroutines from continuations. *Journal of Computer Languages*, 11(3–4):143–153, 1986.

[HJW92] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[JMG+02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002. See also http://www.cs.cornell.edu/projects/cyclone.

[Jon99] Haskell 98: A non-strict, purely functional language, February 1999. Available at http://www.haskell.org/onlinereport/.

[L+81] B. Liskov et al. CLU reference manual. In Goos and Hartmanis, editors, *Lecture Notes in Computer Science*, volume 114. Springer-Verlag, Berlin, 1981.

[Llo99] John W. Lloyd. Programming in an integrated functional and logic programming language. *Journal of Functional and Logic Programming*, 3, March 1999.

[LM02] 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`.

[Mic01] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, 2001. ISBN 0-7356-1448-2.

[MOSS96] Stephan Murer, Stephen Omohundro, David Stoutamire, and Clemens Szyperski. Iteration abstraction in Sather. *ACM Transactions on Programming Languages and Systems*, 18(1):1–15, January 1996.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[NCM02] Nathaniel Nystrom, Michael Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. Technical Report 2002-1883, Computer Science Dept., Cornell University, 2002.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[Oka98a] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. ISBN 0-521-63124-6.

[Oka98b] Chris Okasaki. Views for Standard ML. In *Workshop on ML*, pages 14–23, September 1998.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 146–159, Paris, France, January 1997.

[PGPN96] Pedro Palao Gostanza, Ricardo Pena, and Manuel Núñez. A new look at pattern matching in abstract data types. In *Proc. 1st ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, PA, USA, June 1996.

[SHC96] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, October–December 1996.

[Tul00] Mark Tullsen. First-class patterns. In *Proc. Practical Aspects of Declarative Languages, 2nd International Workshop (PADL)*, pages 1–15, 2000.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings, 14th Symposium on Principles of Programming Languages*, pages 307–312. Association for Computing Machinery, 1987.