JMatch: Java plus Pattern Matching

Jed Liu Andrew C. Myers
Computer Science Department
Cornell University

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 technical report, we present the design and implementation of JMatch, a new object-oriented language that extends Java [GJSB00] 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]

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

This research was supported in part by DARPA Contract and 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. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

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 technical report 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.

This technical report is an expanded version of an earlier paper, including more examples and a more detailed description of the syntax and semantics of JMatch. The rest of the technical report 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 static checking of JMatch, including type checking, multiplicity checking, and static mode selection. Section 5 describes the implementation of the prototype compiler. Section 6 discusses related work. Section 7 summarizes and concludes with a discussion of useful extensions to JMatch.

Appendices A–C give a semantics for JMatch as a translation to Java. This semantics includes support for *interruptible iterators*,which is not described in this technical report, though it is described elsewhere [LM05]. The specific constructs include interrupts and interrupt handlers.

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 extends 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(Integer x,
              List(Integer y, List rest)),
         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 3.5).

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 (see Section 5).

2.6 Pattern methods

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 `theEmptyTree`. 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`.

```
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):
       ...
}
```

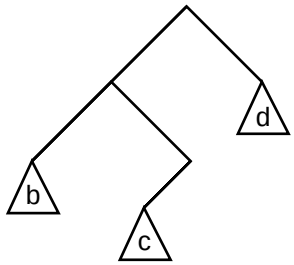
Figure 2: Reversible list append

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.)

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.




```
interface Tree {
    Tree node(Tree left, Tree right, Object o)
        returns(left, right, 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 RedBlackNode implements Tree {
    RedBlackNode lf, rg;
    int color;  // RED or BLACK
    Object value;
    Tree node(Tree left, Tree right)
      ( false )  // forward mode
      returns (left, right) (
        left = lf &&
        right = rg
      )
    Tree empty() returns()
      ( false )  // both modes
    ...
}
class RedBlackEmpty implements Tree {
    static Tree theEmptyTree = RedBlackEmpty();
    Tree node(Tree left, Tree right)
      returns (left, right)
        ( false )  // both modes
    Tree empty()
      returns() ( result = theEmptyTree )
    ...
}
```

(implementation)

Figure 1: Deep abstract pattern matching

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 ρ_1 a_2 ρ_2 ... ρ_{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. If `<` or `<=`, the iteration ascends, otherwise it descends. 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`).

```
class Node extends RBTree {
 int value;
 int color;  // RED or BLACK
 RBTree left, right;
..
 public RBTree insert(int x) {
   let Node(_,int v,RBTree l,RBTree r) = ins(x);
   return new Node(BLACK, v, l, r)
 }
 Node ins(int x) {  // internal insert
   if (x == value) return this;
   if (x < value) return balance(color, value,
                                 left.ins(x), right);
   else return balance(color, value,
                                 left, right.ins(x));
 }
 protected 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);
 }
```

Figure 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` [GJSB00] 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 Parsing

Abstract pattern matching can make parsing convenient, as shown in the code of Figure 4, which parses Lisp-style S-expressions [McC60].

```
static String sexp(String rest, Object AST)
  returns (rest, AST) (
    String s = stripWS(result) && (
      s = atom(rest, String name) && name = AST
    else
      s = "(" + list(")"+rest, List l) && l = AST
    )
  )
static String list(String rest, List AST)
  returns(rest, AST) (
    String s = stripWS(result) && (
      s.charAt(0) = ')' && s = rest && AST = null
    else
      s = sexp(list(rest, List l), Object o) &&
      AST = List(o, l)
    )
  )
static String atom(String rest, String name)
  returns (rest, name) (
    String s = stripWS(result) &&
      s = char c + atom(rest, String suffix) &&
      atomChar(c) &&
      name = c + suffix
  )
```

Figure 4: Parsing s-expressions

The code consists of three methods that correspond directly to the grammar for s-expressions. In its backward mode, the method `sexp` parses an s-expression that is a prefix of the result string it is matched against. The parsed s-expression is returned in `AST`, and the unconsumed suffix of the string is left in `rest`. The other two methods operate similarly but parse lists of s-expressions and atoms, respectively. For example, solving `"(a (b c)) d" = sexp(String r, Object AST)` results in `r` being bound to `d` and `AST` being bound to the list `["a", ["b", "c"]]` (the latter is not legal Java syntax).

This code relies on two elided methods, `stripWS` and `atomChar`, which respectively strip leading white space and report whether a character can be part of an atom. Note the extensive use of pattern matching on string concatenation.

3.5 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 5 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

```
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) )
}
```

Figure 5: Peano natural numbers ADT

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.

4 Syntax and static semantics

4.1 JMatch syntactic extensions

The following grammar productions describe how the Java 1.4 grammar [GJSB00] is extended in a backwards-compatible way to become the JMatch grammar. The notation is EBNF: large brackets are used to indicate optional terminals or nonterminals, large parentheses are used as a grouping structure, and Kleene star is used to indicate zero or more repetitions.

The grammar described here includes syntax for supporting interruptible iterators [LM05].

4.1.1 Method and constructor declarations

Java method declaration headers are extended to include interrupt and trap-exception declarations:

MethodDeclaratorRest →
    FormalParameters [`[]`] (*Throws* | *Traps*)* (*MethodBody* | `;`)
InterfaceMethodDeclaratorRest →
    FormalParameters [`[]`] (*Throws* | *Traps*)* `;`
Throws → `throws` *QualifiedIdentifierList*
Traps → `traps` *QualifiedIdentifier* [`:` *QualifiedIdentifierList*]

Method declaration bodies are extended to include mode declarations and implementations:

MethodBody →
    [*DefaultImpl*] (*Impl*)* [*AbstractModes* `;`]
    (but not empty)
DefaultImpl → *ImplBody*
Impl → (*Mode*)* *ImplBody*
ImplBody → *Block* | `(` *Formula* `)`
AbstractModes → (*Mode*)*
Mode →
    (`iterates` | `returns`) `(` [*Identifier*, ... , *Identifier*] `)`

Similar extensions exist for Java constructor declarations:

ConstructorDeclaratorRest →
    FormalParameters (*Throws* | *Traps*)* *ConstructorBody*
ConstructorBody → [*DefaultImpl*] (*Impl*)*
    (but not empty)

4.1.2 Statement extensions

A few new statements are added:

Statement → ...
    | `foreach (` *Formula* `)` *Statement*
    | `let` *Formula* `;`
    | `yield ;`
    | `cond {` (`(` *Formula* `)` *Statement*)*
      [`else` *Statement*] `}`
    | `raise` *Expression* `;`
    | `resume` (`break` | `continue` | `yield`)
      (*TrapClause*)*
TrapClause → `trap (` *FormalParameter* `)` *Block*

The `let` and `foreach` statements were described in Section 2. The `cond` statement (from LISP [Ste90]), is similar to `if` except that it supports multiple conditions. The `raise` and `resume` statements are used for raising and handling interrupts.

The syntax of some other Java statements is modified:

IfStatement → `if (` *Formula* `)` *Statement* [`else` *Statement*]
SwitchStatement →
    `switch (` *Expression*, ... , *Expression* `)` *SwitchBlock*
SwitchLabel →
    `case` *Expression*, ... , *Expression* `where` *Formula* `:`
  | `default :`
Catches → (*CatchClause* | *TrapClause*)+
CatchClause → ...
  | `catch (` `trap` *FormalParameter* `)` *Block*

The `if` statement is extended to allow any formula as the conditional. The real grammar is more complicated because of the usual dangling-else problem.

In a `switch` statement, the corresponding `case` labels can be arbitrary patterns, with an optional side condition specified by a `where` clause.

The `try` statement is extended to include declarators for interrupt and trap-exception handlers.

4.1.3 Expression extensions

Formulas are boolean expressions, extended with the `else` operator.

Formula → *ConditionalOrExpression*
ConditionalOrExpression → ...
    | *ConditionalOrExpression* `else`
      ConditionalAndExpression

Conjunction expressions are extended to contain *try expressions* for specifying exception and interrupt handlers.

ConditionalAndExpression → *TryExpression*
| *ConditionalAndExpression TryExpression*

TryExpression → *InclusiveOrExpression*
| *InclusiveOrExpression* (*CatchClause* | *TrapClause*)$^+$

Expressions are extended to allow the declarations of variables to solve for, as well as some other new expression forms:

PostfixExpression →
...
| *Type VariableDeclaratorId*
| `iterate` *Identifier* `(` *Formula* `)`
| `single` `(` *Expression* `)`
| `_`

The `single` construct can be used on formula or pattern to prevent more than one set of solutions; it turns a formula or pattern with multiple solutions into one with a single solution.

As in ML [MTH90], the underscore (`_`) is a wildcard pattern that can be matched against any value.

To avoid parentheses around most equality tests, JMatch gives the operator `=` higher precedence than `&&` or `||`. This change requires juggling a few productions. Note that the pattern operator `as` shows up here.

StatementExpression → *Assignment*
| *RelationalExpression*
RelationalExpression →
RelationalExpression RelOp InstanceofExpression
RelOp → `==` | `=` | `!=` | `<` | `>` | `<=` | `>=`
EqualityExpression → *RelationalExpression*
InstanceofExpression → *ShiftExpression*
| *InstanceofExpression* `instanceof` *ReferenceType*
| *InstanceofExpression* `as` *ShiftExpression*

4.2 Type checking

Like Java programs, JMatch programs are type-checked statically. However, the introduction of modes creates new obligations for static checking. Type checking JMatch expressions, including formulas and patterns, is little different from regular Java type checking, since the types are the same in all modes, and the forward mode corresponds to ordinary Java evaluation.

The interface and abstract class conformance rules in Java are extended in a natural way to handle method modes: to implement an interface or to extend an abstract class, 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 superclass.

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.

4.2.1 Mode selection

For built-in and user-defined predicate and pattern methods, the compiler must select the appropriate mode for each use in a formula or pattern. A pattern is solvable if a value can be found for the pattern, along with corresponding assignments to the variables it declares, even in the absence of a value to match against. If a value is needed to match against, the pattern is matchable.

Mode selection puts patterns into a canonical form in which they can solved in left-to-right order. In canonical form, conjuncts occur in the order in which they are to be solved. In addition, equality tests are ordered so that the pattern on the left side is solvable and the right side is matchable, using the solved value of the left side.

When a pattern or formula is solved, it provides values for all unknown variables used by the formula. The only exception to this occurs in the case of disjunctions, where values are provided only for those variables used in all arms of the disjunction.

For constructor and method invocations, there may be more than one mode that permits solution of the pattern or formula. The first step is to determine the set of *usable* modes. A mode is usable if all the arguments provided to known formal parameters are solvable patterns.

Given the set of usable modes, the best mode is selected, based on an ordering of modes. Each mode has a *multiplicity*, which is either **1** (single) or $*$ (multiple), depending on whether it has at most one solution or possibly many solutions (the `returns` and `iterates` modes, respectively.)

A partial ordering of predicate modes is defined as follows. First an ordering on multiplicities is defined: $\mathbf{1} \leq *$. If modes $M_1(U_1)$ and $M_2(U_2)$ have respective multiplicities M_1 and M_2 and respective unknown sets U_1 and U_2, then $M_1(U_1) \leq M_2(U_2)$ iff $M_1 \leq M_2$ and $U_1 \subseteq U_2$. From the subset of the usable modes that are minimal in this order, the first mode declared in the receiver type is selected. Usually there is only one minimal usable mode, so declaration order does not matter.

4.2.2 Multiplicity checking

In JMatch it is a static error to use a formula or pattern with multiple solutions when a single solution is expected, because solutions might be silently discarded. Multiplicity checking ensures that a formula or pattern with multiplicity $*$ cannot be used in a context (such as an `if` or `let`) where a single solution is expected. The `single` operator may be used to explicitly convert the multiplicity.

Usually, the multiplicity of a formula or pattern is simply the join of the multiplicities of its subformulas or subpat-

terns. Any formula, pattern, constructor, or method used in the forward direction is singular. Otherwise:

- Disjunctions have multiplicity $*$.
- Wildcard patterns have multiplicity **1**.
- The `single` of a formula or pattern has multiplicity **1**.
- Constructor, method and built-in operator invocations have a multiplicity defined by the mode used.

5 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.

5.1 Translating JMatch to Java$_{\texttt{yield}}$

In the prototype 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 `yield`, `try ...trap`, and `resume` statements [LM05] that can only be used to implement 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. Appendix A provides this translation, which is straightforwardly defined using a few mutually inductively defined syntax-directed functions.

5.2 Translating Java$_{\texttt{yield}}$ to Java

Java$_{\texttt{yield}}$ code that does not contain the `yield` statement is translated as is. In iterator implementations, the `yield` statement is eliminated by converting the iterator code into a state machine. The form of code for an iterator class `ITER` that returns variables x_i is shown in Figure 6, which refers to the framework class `jmatch.runtime.Iterator` given in Appendix B.

The code of the iterator, broken up into `yield`-free fragments, appears within the `switch` statement in the method `peek`. The variable `$state$` explicitly captures the control state of the iterator so it can be restarted by branching to the appropriate case arm. Each `yield` statement causes the corresponding `case` arm to terminate by updating the variable `$state$` and returning to the caller. The translated code can also jump among the various `case` arms by changing the value of `$state$` and executing a `continue`.

Handlers for exceptions, interrupts, and trap exceptions [LM05] are similarly translated and appear in the `switch` statement in the method `peek`. Handler dispatch is

```
class ITER extends jmatch.runtime.Iterator {
  T_i $i;        // unknowns
  T_j z_j;       // local variables
  boolean peek() throws Throwable {
    while (true) {
      try {
        switch($state$) {
            case 0: ...
            case 1: ...
            ...
            case N: ...
        }
      } catch (Throwable $t$) {
        $thrown$ = $t$;
        if (!findHandler()) throw $t$;
        continue;
      } } }
  boolean findHandler() {
    while (true) {
      switch($state$) {
        case 0: ...
        case 1: ...
        ...
        case N: ...
      } } } }
```

Figure 6: Output of the translation from Java$_{\texttt{yield}}$

managed by the method `findHandler`: depending on the current control state of the iterator and the type of the exception or interrupt to be handled, the `$state$` variable is updated to dispatch control to the appropriate handler.

The translation, given in Appendix C, is essentially a conversion to continuation-passing style.

5.3 Implementation status

Most of the language described in this paper is implemented in the current JMatch compiler prototype found at `http://www.cs.cornell.edu/Projects/jmatch`. Certain features of JMatch have not yet been implemented, though their implementation has been designed. The reverse modes of the string concatenation operator are implemented in a user-defined library but the operator + syntactic sugar is not.

While the performance of the translated code is asymptotically acceptable, several easy optimizations would improve code quality.

6 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 modules and separate compilation. As in JMatch, Mercury predicates can have several modes. 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 has a more complex mode system that allows a distinction between exactly-one and at-most-one solution. As in JMatch, these declarations are checked statically. Mercury does not support objects.

Several other languages have expressive iteration constructs. The language CLU [L+81] first introduced iterators whose use and implementation are both convenient; the `yield` statement of JMatch was inspired by CLU. ICON [GHK81] also has "generators" that can produce more than one value. Implementation of ICON generators is similar to that in CLU, although there are some convenient built-in generators. ICON supports imperative programming and limited backtracking across statements. Sather provides iterator abstractions [MOSS96] with some extensions to the CLU model. None of these languages have pattern matching.

Juno-2 [NH97] is a constraint-based rendering language with both imperative and logic-programming features; it can solve formulas that include numerical equations; predicates are expanded at application time, so predicate arguments are not required to be ground values. It does not support data abstraction.

Pizza also extends Java with support for datatypes and ML-style pattern matching [OW97], by allowing a class to be implemented as an algebraic datatype. Because the datatype is not exposed outside the class, this design does not permit abstract pattern matching; it does allow a collection of related implementations to be conveniently packaged together with some code sharing and pattern matching. Forax and Roussel have also proposed a Java extension for simple pattern matching based on reflection [FR99].

Ernst et al. have developed predicate dispatching [EKC98], another way to add pattern matching to an object-oriented language. In their language, boolean formulas control the dispatch mechanism, which allows some encoding of 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 predicate abstractions that can implement a single new view of an object, but unlike JMatch, it does not unify predicates and methods. Predicates have some limitations: they may not be recursive or iterative and do not support modal abstraction or invertible computation.

7 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 useful suggestions. Nate Nystrom helped figure out how to implement JMatch using 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.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000. ISBN 0-201-31008-2.

[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.

[LM05] Jed Liu and Andrew C. Myers. Interruptible iterators. Submitted for publication, 2005.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Comm. of the ACM*, 3(4), April 1960.

[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.

[NH97] Greg Nelson and Allen Heydon. Juno-2 language definition. SRC Technical Note 1997-007, Digital Systems Research Center, June 1997.

[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.

[Ste90] Guy Steele. *Common LISP: the Language*. Digital Press, second edition, 1990. ISBN 1-55558-041-6.

[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.

A Translating JMatch to Java$_{\texttt{yield}}$

The translation from JMatch to Java$_{\texttt{yield}}$ consists of several syntax-directed functions that are mutually inductively defined. In the following, let f range over formulas and p range over patterns; s ranges over Java$_{\texttt{yield}}$ statements, w–z range over variables, and U ranges over sets of variables. The output of each of the translations is a sequence of Java$_{\texttt{yield}}$ statements.

- $\mathcal{S}[\![s]\!]$ is a Java$_{\texttt{yield}}$ statement that is equivalent to the JMatch statement s. It is the identity for statements that do not contain special JMatch features.
- $\mathcal{F}[\![f]\!]Us$ is the translation of a formula. It is a sequence of Java$_{\texttt{yield}}$ statements that solve the formula f and execute the statement s for each solution found. The argument U is the set of unknowns to be solved for.
- $\mathcal{M}_s[\![p]\!]Uxs$ and $\mathcal{M}_p[\![p]\!]Uxs$ are the pattern-matching forms of the pattern translation. $\mathcal{M}_s$ and $\mathcal{M}_p$ give the semantic-equality and pointer-equality semantics, respectively. Each produces a sequence of statements that solve the formula $p = x$, where x is a known variable containing the value to match p against. For each solution the output code executes s with the unknowns in U given satisfying assignments.
- $\mathcal{P}[\![p]\!]Uws$ is a pattern translation that solves for the value of the pattern and its unknowns without a value to match against. The output code executes s for every solution, where the unknowns in U are assigned to produce that value for p and the variable w is assigned the value of the pattern p.
- $\mathcal{D}[\![d]\!]$ gives the translation of JMatch iterators. It takes a JMatch method mode declaration and produces a corresponding Java iterator class.
- $\mathcal{E}[\![e]\!]$ and $\mathcal{C}[\![e]\!]$ together give the translation of an `iterate` expression. $\mathcal{E}$ translates the `iterate` expression into an equivalent Java iterator; $\mathcal{C}$ generates the container class for the result values.

Let μf and μp be the set of variables declared in the formula f or pattern p, respectively. Let φf and φp be the set of fields accessed in the formula f or pattern p, respectively. Let variables y denote fresh variables, and l denote fresh statement labels.

A.1 Statement translations

$$\mathcal{S}[\![\texttt{foreach}\ (f)\ s]\!] \ = \ \mathcal{F}[\![f]\!](\mu f)(\mathcal{S}[\![s]\!])$$

$$\begin{array}{l}\mathcal{S}[\![\texttt{if}\ (f)\ s_1\ \texttt{else}\ s_2]\!] \ = \\ \quad \texttt{boolean}\ y = \texttt{true}; \\ \quad \mathcal{F}[\![f]\!](\mu f)(y = \texttt{false};\ \mathcal{S}[\![s_1]\!]); \\ \quad \texttt{if}\ (y)\ \mathcal{S}[\![s_2]\!]\end{array}$$

$$\begin{array}{l}\mathcal{S}[\![\texttt{cond}\ \overrightarrow{(f_i)\ s_i}\ \texttt{else}\ s]\!] \ = \\ \quad \mathcal{S}[\![\texttt{if}\ (f_1)\ s_1\ \texttt{else if}\ (f_2)\ s_2\ \texttt{else if}\ \ldots\ \texttt{else}\ s]\!]\end{array}$$

$$\begin{array}{l}\mathcal{S}[\![\texttt{switch}\ (e)\ \{\overrightarrow{\texttt{case}\ p_i\ \texttt{where}\ f_i :\ s_i}\ \texttt{default :}\ s\}]\!] \ = \\ \quad \texttt{Object}\ y \ = \ e; \\ \quad \mathcal{S}[\![\ \ \texttt{if}\ (y = p_1\ \&\&\ f_1)\ s_1 \\ \qquad \texttt{else if}\ (y = p_2\ \&\&\ f_2)\ s_2 \\ \qquad \ldots \\ \qquad \texttt{else}\ s\ ]\!]\end{array}$$

A.2 Formula translations

Formulas are assumed to be in canonical form.

$$\mathcal{F}[\![f_1 \texttt{ \&\& } f_2]\!]Us = \mathcal{F}[\![f_1]\!](U \cap \mu f_1)(\mathcal{F}[\![f_2]\!](U \setminus \mu f_1)s)$$

$$\mathcal{F}[\![f_1 || f_2]\!]Us = \mathcal{F}[\![f_1]\!]Us\,;\ \mathcal{F}[\![f_2]\!]Us$$

$$\begin{array}{l}\mathcal{F}[\![f_1 \texttt{ else } f_2]\!]Us = \\ \quad \texttt{boolean } y = \texttt{true}; \\ \quad \mathcal{F}[\![f_1]\!]U(\{y = \texttt{false};\ s;\}) \\ \quad \texttt{if } (y)\ \mathcal{F}[\![f_2]\!]Us\end{array}$$

$$\begin{array}{l}\mathcal{F}[\![p_1 = p_2]\!]Us = \\ \quad \texttt{Object } y; \\ \quad \mathcal{P}[\![p_1]\!](U \cap \mu p_1)y(\mathcal{M}_s[\![p_2]\!](U \setminus \mu p_1)ys)\end{array}$$

$$\begin{array}{l}\mathcal{F}[\![p_1 == p_2]\!]Us = \\ \quad \texttt{Object } y; \\ \quad \mathcal{P}[\![p_1]\!](U \cap \mu p_1)y(\mathcal{M}_p[\![p_2]\!](U \setminus \mu p_1)ys)\end{array}$$

$$\begin{array}{l}\mathcal{F}[\![p_1 \texttt{ != } p_2]\!]Us = \\ \quad \texttt{Object } y_1, y_2; \\ \quad \mathcal{P}[\![p_1]\!](U \cap \mu p_1)y_1(\mathcal{P}[\![p_2]\!](U \setminus \mu p_1)y_2(\{\texttt{if } (y_1 \texttt{ != } y_2)\ s\}))\end{array}$$

$$\begin{array}{l}\mathcal{F}[\![p_1 < p_2]\!]Us = \\ \quad \tau_1\ y_1;\ \tau_2\ y_2; \\ \quad \mathcal{P}[\![p_1]\!](U \cap \mu p_1)y_1(\mathcal{P}[\![p_2]\!](U \setminus \mu p_1)y_2(\{\texttt{if } (y_1 < y_2)\ s\}))\end{array}$$

where τ_1 and τ_2 are the types of p_1 and p_2, respectively, and `<` can be replaced with `<=`, `>=` and `>`.

$$\mathcal{F}[\![\texttt{single } (f)]\!]Us = l : \{\mathcal{F}[\![f]\!]U(\{s;\ \texttt{break } l;\})\}$$

$$\begin{array}{l}\mathcal{F}[\![f\ \overrightarrow{\texttt{trap } (\tau_i\ t_i)\ \{\ s_i;\ \texttt{resume } (f_i);\ \}}\ \overrightarrow{\texttt{catch}_j}]\!]Us = \\ \quad \texttt{int } y = 0;\ \overrightarrow{\tau_i\ t_i = \texttt{null}}; \\ \quad \texttt{try } \{ \\ \qquad \mathcal{F}[\![y = 0 \texttt{ \&\& } f \texttt{ || } \overrightarrow{y = i \texttt{ \&\& } f_i}]\!]Us \\ \quad \overrightarrow{\}\ \texttt{trap } (\tau_i\ y_i)\ \{y = i;\ t_i = y_i;\ \texttt{resume continue};\}}\ \overrightarrow{\texttt{catch}_j}\end{array}$$

A.3 Pattern translations

An expression with no unknowns can be evaluated directly:

$$\mathcal{M}_s[\![p]\!]\emptyset xs = \{T_x\ y\texttt{=}p;\ \texttt{if } (x.\texttt{equals}(y))\ s\}$$

$$\mathcal{M}_p[\![p]\!]\emptyset xs = \{T_x\ y\texttt{=}p;\ \texttt{if } (x == y)\ s\}$$

$$\mathcal{P}[\![p]\!]\emptyset ws = \{w\texttt{=}p;\ s\}$$

Wildcards require no work and bind no variables:

$$\mathcal{M}[\![\_]\!]\emptyset xs = s$$

If a variable w of type T_w is matched against a value in x of type T_x, a runtime check is introduced if T_w is not a supertype of T_x:

$$\mathcal{M}[\![w]\!]\{w\}xs = \texttt{if } (x \texttt{ instanceof } T_w)\ \{T_w\ w\texttt{=}(T_w)x; s\}$$

Otherwise, the value can be assigned directly.

If a type T is matched against a value in x of type T_x, a runtime check is introduced if T is not a supertype of T_x:

$$\mathcal{M}_p[\![T]\!]\emptyset xs = \texttt{if } (x \texttt{ instanceof } T)\ s$$

Otherwise, the statement s can be executed directly.

$$\begin{array}{l}\mathcal{M}_p[\![p_1 \texttt{ as } p_2]\!]Uxs = \\ \quad \mathcal{M}_p[\![p_1]\!](U \cap \mu p_1)x(\mathcal{M}_p[\![p_2]\!](U \setminus \mu p_1)xs)\end{array}$$

$$\begin{array}{l}\mathcal{M}_p[\![\tau.m(\overrightarrow{p_i})]\!]Uxs = \\ \quad \mathcal{M}_p[\![\tau\ y \texttt{ as } y.m(\overrightarrow{p_i})]\!](U \cup \{\tau\ y\})xs\end{array}$$

$$\mathcal{M}_p[\![\texttt{single}(p)]\!]Uxs = l : \{\mathcal{M}_p[\![p]\!]Ux(\{s;\ \texttt{break } l;\})\}$$

$$\mathcal{P}[\![\texttt{single}(p)]\!]Uws = l : \{\mathcal{P}[\![p]\!]Uw(\{s;\ \texttt{break } l;\})\}$$

Built-in operators such as `+` and `[ ]` support pattern matching. For brevity, the semantic-equality pattern-matching semantics have been omitted.

$$\begin{array}{l}\mathcal{M}_p[\![e\texttt{[}p\texttt{]}]\!]Uxs = \\ \quad \texttt{Object[ ] } y_a = e; \\ \quad \texttt{for } (\texttt{int } y_i = 0;\ y_i < y_a.\texttt{length};\ y_i\texttt{++})\{ \\ \qquad \texttt{if } (x\texttt{==}y_a\texttt{[}y_i\texttt{]})\ \mathcal{M}_p[\![p]\!]Uy_is \\ \quad \}\end{array}$$

$$\begin{array}{l}\mathcal{P}[\![e[p]]\!]Uws = \\ \quad \texttt{Object[ ] } y_a = e; \\ \quad \texttt{for } (\texttt{int } y_i = 0;\ y_i < y_a.\texttt{length};\ y_i\texttt{++})\{ \\ \qquad w = y_a\texttt{[}y_i\texttt{]}; \\ \qquad \mathcal{M}_p[\![p]\!]Uy_is \\ \quad \}\end{array}$$

$$\begin{array}{l}\mathcal{M}_p[\![\texttt{-}p]\!]Uxs = \\ \quad \tau\ y \texttt{=-}x; \\ \quad \mathcal{M}_p[\![p]\!]Uys\end{array}$$

where τ is the type of p and "`-`" can be replaced with "`+`" and "`~`".

$$\begin{array}{l}\mathcal{P}[\![\texttt{-}p]\!]Uws = \\ \quad \mathcal{P}[\![p]\!]Uw(\{w \texttt{=-}w;\ s\})\end{array}$$

where "`-`" can be replaced with "`+`" and "`~`".

For brevity only the semantics for additive patterns are shown. The semantics for other arithmetic patterns follow analogously.

$$\begin{array}{l}\mathcal{M}_p[\![p_1 + p_2]\!]Uxs = \\ \quad \tau_1\ y_1;\ \tau_2\ y_2; \\ \quad \mathcal{P}[\![p_1]\!](U \cap \mu p_1)y_1(\{y_2 = x - y_1; \mathcal{M}_p[\![p_2]\!](U \setminus \mu p_1)y_2s\})\end{array}$$

where τ_1 and τ_2 are the types of p_1 and p_2, respectively.

$$\begin{array}{l}\mathcal{P}[\![p_1 + p_2]\!]Uws = \\ \quad \tau_1\ y_1;\ \tau_2\ y_2; \\ \quad \mathcal{P}[\![p_1]\!](U \cap \mu p_1)y_1(\mathcal{P}[\![p_2]\!](U - \mu p_1)y_2(\{w = y_1 + y_2;\ s\}))\end{array}$$

where τ_1 and τ_2 are the types of p_1 and p_2, respectively.

A.4 Expression translations

For simplicity, the checkpointing portion of the translation is elided.

$$\begin{array}{l}\mathcal{C}[\![\texttt{iterate } c(f)]\!] = \\ \quad \texttt{class } c\ \{ \\ \qquad \overrightarrow{\tau_i\ x_i}; \\ \quad \}\end{array}$$

where $(\overrightarrow{\tau_i\ x_i}) = \mu f$.

$\mathcal{E}[\![\texttt{iterate}\ c(f)]\!] =$
```
new jmatch.runtime.Iterator() {
  τ_i x'_i;
  protected Object pack() {
    c result = new c();
    result.x_i = this.x'_i;
    return result;
  }
  protected boolean peek() {
    τ_i x_i;
    F[[f]](μf){ x'_i = x_i; yield; }
    return false;
  }
  protected boolean findHandler() {
    // body filled in during translation from Java_yield
  }
}
```
where $(\overrightarrow{\tau_i\ x_i}) = \mu f$ and $x'_1, \ldots, x'_n$ are fresh variables.

This translation refers to the framework class `jmatch.runtime.Iterator` that contains code common to all JMatch iterators. The code for this class is given in Appendix B.

A.5 Methods

Translation of method bodies defined as formulas is based on the $\mathcal{F}$ translation. Each time a solution is found to the unknowns of a backward mode, it is saved and control is yielded to the calling context.

$\mathcal{D}[\![\texttt{boolean}\ p(\overrightarrow{\tau_i\ x_i})\ \texttt{iterates}(U)\ (f)]\!] =$
```
class p$U extends jmatch.runtime.Iterator {
  τ_{u_i} $i;      // unknown result values
  τ_{k_i} x_{k_i};   // these store the known values

  p$U(τ_{k_i} x_{k_i}) { this.x_{k_i} = x_{k_i}; }

  public boolean peek() {
    τ_{u_i} x_{u_i};
    F[[f]]U({this.$i = x_{u_i}; yield;})
    return false;
  }

  protected boolean findHandler() {
    // body filled in during translation from Java_yield
  }
}
```
where $\{\vec{x}_{\mathtt{k}_i}\} = U$ are known and $\{\vec{x}_{\mathtt{u}_i}\} = \{\vec{x}_i\} \setminus U$ are unknown.

$\mathcal{D}[\![\texttt{boolean}\ p(\overrightarrow{\tau_i\ x_i})\ \texttt{returns}()\ (f)]\!] =$
```
boolean p(τ_i x_i) {
  F[[f]](μf)(return true;)
  return false;
}
```
Correspondingly, a call to a predicate method in the forward mode is translated by solving for the arguments and performing the call for each set of arguments obtained:

$\mathcal{F}[\![p(p_1, \ldots, p_n)]\!]Us =$
$\mathcal{P}[\![p_{a_1}]\!]U_1 y_{a_1}($
$\mathcal{P}[\![p_{a_2}]\!]U_2 y_{a_2}($
$\ldots$
$\mathcal{P}[\![p_{a_n}]\!]U_n y_{a_n}($
$\texttt{if}(p(y_1, \ldots, y_n))\ s)$

where: $\{1, \ldots, n\} = \{a_1, \ldots, a_n\}$
$V_1 = U,\ V_i = V_{i-1} \setminus \mu p_{a_{i-1}},\ U_i = V_i \cap \mu p_{a_i}\ (1 < i \le m)$

A call to a predicate method in a backward mode is translated as a loop that uses the iterator implemented by this method body:

$\mathcal{F}[\![p_0.f(p_1, \ldots, p_n)]\!]Us =$
$\mathcal{P}[\![p_{a_1}]\!]U_1 y_{a_1}($
$\mathcal{P}[\![p_{a_2}]\!]U_2 y_{a_2}($
$\ldots$
$\mathcal{P}[\![p_{a_m}]\!]U_m y_{a_m}($
`jmatch.runtime.Iterator` I `=`
`new` $p\$M(y'_1, \ldots, y'_m)$;
`Throwable` e `= null`;
$\texttt{while}^{I,e}_{\langle\tau_1,\ldots,\tau_r\rangle}(I.\texttt{advance}())$ {
$\mathcal{M}[\![p_{b_1}]\!]U'_1(I.\$1)($
$\mathcal{M}[\![p_{b_2}]\!]U'_2(I.\$2)($
$\ldots$
$\mathcal{M}[\![p_{b_k}]\!]U'_k(I.\$k)(s)\ldots))$
}
`if` $(e$ `instanceof` $\tau'_1)$ `throw` (τ'_1)`e`;
`if` $(e$ `instanceof` $\tau'_2)$ `throw` (τ'_2)`e`;
$\ldots$
`if` $(e$ `instanceof` $\tau'_t)$ `throw` (τ'_t)`e`;
`if` $(e$ `!= null)` `throw e`;$)\ldots))$

where: $\{0, \ldots, n\} = \{a_1, \ldots, a_m, b_1, \ldots, b_k\}$
$\{\tau_1, \ldots, \tau_r\}$ are the interrupts handled by the iterator
$\{\tau'_1, \ldots, \tau'_t\}$ are the trap exceptions thrown by the iterator
$[y'_1, \ldots, y'_m] = [y_0, \ldots, y_n] \cap \{y_{a_1}, \ldots, y_{a_m}\}$
$V_1 = U,\ V_i = V_{i-1} \setminus \mu p_{a_{i-1}},\ U_i = V_i \cap \mu p_{a_i}\ (1 < i \le m)$
$V'_1 = V_m \setminus \mu p_{a_m},\ V'_i = V'_{i-1} \setminus \mu p_{b_{i-1}},\ U'_i = V'_i \cap \mu p_{b_i},\ (1 < i \le k)$

The $n + 1$ patterns to be evaluated are divided into those that are in a known position in the mode selected (indices $a_1, \ldots, a_m$) and those that are not ($b_1, \ldots, b_k$). The former are solved directly, as much in a left-to-right order as possible; the latter are matched using the results provided by the iterator $p\$M$ that implements the predicate method. The intersection of a list and a set is a new list whose elements are a set intersection but preserve the original order. The translation of pattern method invocations is similar.

The translation above includes code and annotations used for interrupt and trap exception dispatch. In the final translation output, the body of the while loop will save into e any trap exceptions thrown by I and break out of the loop. Thus, to facilitate the translation of `raise` statements in the final translation to Java, the `while` loop is annotated with the variable I that contains the iterator object, the variable e that is to store the trap exceptions, and the set of interrupts supported by the iterator.

B Iterator Framework Class

The translations for `iterate` expressions in Appendix A.4 and methods in Appendix A.5 refer to the following class that contains code common to all JMatch iterators.

```
package jmatch.runtime;
public abstract class Iterator {
   protected boolean advanced;    // Whether the iterator has advanced via a hasNext() call.
   protected boolean haveNext;    // The cached result of hasNext().
   protected boolean removeSupported;
   protected Throwable thrown;    // The exception that needs to be handled.
   protected Object raised;       // The interrupt that needs to be handled.
   public int $state$;
   public int $yieldpt$;
   public int $resumept$;

   public Iterator() {
     this.raised = this.thrown = null;
     advanced = false;
   }

   public boolean hasNext() {
     if (advanced) return haveNext;
     if (removeSupported) checkpoint();
     advanced = true;
     return haveNext = advance();
   }

   public Object next() {
     if (advanced ? !hasNext : !advance()) throw new java.util.NoSuchElementException();
     advanced = false;
     return pack();
   }

   public void remove() {
     if (!removeSupported) throw new UnsupportedOperationException();
     if (advanced) { advanced = false; restoreState(); }
     Throwable t = trap(new jmatch.runtime.Remove());
     if (t instanceof RuntimeException) throw (RuntimeException)t;
     if (t != null) throw new jmatch.runtime.TrapException(t);
   }

   public boolean advance() {
     try {
       return peek();
     } catch (RuntimeException e) {
       throw e;
     } catch (Throwable t) {
       throw new jmatch.runtime.Error(t);
     }
   }

   /** Handles the given trap and returns any resulting exception. */
   public Throwable trap(Object trap) {
     if (trap == null) return new NullPointerException();
     raised = trap; thrown = null;
     try {
       if (!findHandler()) {
         raised = null;
         return new jmatch.runtime.Error("unhandled trap: " + trap);
       }
       peek();
     } catch (Throwable t) {
       return t;
     }
   }
```

```
  /** Packs the iterator's output data into a single object. */
  protected Object pack() { return null; }

  /**
   * Advances the iterator state.  Returns true iff there is a next element.  Throws any unhandled
   * exceptions.
   */
  protected abstract boolean peek() throws Throwable;

  /**
   * Sets up the internal state of the iterator to handle the interrupt in "raised" or
   * the exception in "thrown".  Returns true iff an appropriate handler was found.
   */
  protected abstract boolean findHandler();

  protected void checkpoint() { ...  }
  protected void restoreState() { ...  }
}
```

C Translating Java$_{\texttt{yield}}$ to Java

The form of the output of this translation is shown in Figure 6. Table 1 gives the inputs and outputs of the translation function. The final Java translation of an iterator body s is obtained by evaluating

$$(n, s', h, L) = \mathcal{T}[\![s]\!](\{\})1\{\mathsf{b} = 0, \mathsf{c} = 0, \mathsf{try} = \mathsf{unit}, \mathsf{rb} = 0, \mathsf{rc} = 0, \mathsf{handler} = (\lambda\tau.(\epsilon, \epsilon, 0))\}.$$

The resulting statements s' become part of the body of the `peek` method, whereas the statements h become part of the body of the `findHandler` method:

```
public boolean peek() {
  while (true) {
    try {
      switch ($state$) {
        case 0: s'
      }
    } catch (Throwable $t$) {
      $thrown$ = $t$;
      if (!findHandler()) throw $t$;
      continue;
    } } }
```

```
public boolean findHandler() {
  while (true) {
    switch ($state$) {
      h
      default:
        $state$ = $yieldpt$;
        return false;
    } } }
```

This translation is different from what is implemented in the JMatch compiler. The prototype implementation includes a few optimizations, including an important *tail-yield* optimization needed for good asymptotic performance [LM05]. A simplified version of the translation is provided here to illustrate the semantics of Java$_{\texttt{yield}}$.

Statement sequencing:

$\mathcal{T}[\![s_1;\ s_2]\!]j_s n r =$
  let
    $(n_1, s_1', h_1, L_1) = \mathcal{T}[\![s_1]\!](\{\texttt{\$state\$ = } n\texttt{; continue;}\})(n+1)r$
    $(n_2, s_2', h_2, L_2) = \mathcal{T}[\![s_2]\!]j_s n_1 r$
  in
    $(n_2,$ s_1'
      `case` n`:` s_2',
     $h_1; h_2, n :: L)$
  end

Inputs:	
$\text{Java}_{\texttt{yield}}$	The $\text{Java}_{\texttt{yield}}$ statement to translate.
Java	Java code to be inserted after the translation of the $\text{Java}_{\texttt{yield}}$ statement.
int	The next unused case label.
{b : int,	The case label targets for unlabeled `break` statements. For simplicity, we omit the treatment of labeled `break`s and `continue`s.
c : int,	The case label targets for unlabeled `continue` statements.
try : int + unit,	The case label for the innermost try block containing the $\text{Java}_{\texttt{yield}}$ statement being translated.
rb : int,	The case label target for `resume break` statements.
rc : int,	The case label target for `resume continue` statements.
handler : $\kappa \rightarrow$ String $\times$ String $\times$ int}	A function for finding the handler for a `raised` interrupt. It maps the interrupt type to: • the name of the variable holding the iterator that will handle the interrupt, • the name of the variable to which any resulting trap exceptions should be assigned, and • the case label to which control should be transferred if a trap exception is thrown.
Outputs:	
int	The next unused case label.
Java	The translated $\text{Java}_{\texttt{yield}}$ statement with the Java code applied. This is the main output of the translation.
Java	A series of Java statements to be included in the body for the `findHandler()` method.
int list	The set of case labels used to translate the top-level block in the given $\text{Java}_{\texttt{yield}}$ statement.

Table 1: The signature of the $\text{Java}_{\texttt{yield}}$ translation function.

If:

$\mathcal{T}[\![\texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2]\!]j_s r =$

let

$(n_1, s_1', h_1, L_1) = \mathcal{T}[\![s_1]\!](\texttt{\{\$state\$ = }n+1\texttt{; continue;\}})(n+2)r$

$(n_2, s_2', h_2, L_2) = \mathcal{T}[\![s_2]\!](\texttt{\{\$state\$ = }n+1\texttt{; continue;\}})n_1 r$

in

$(n_2,$ `if (`e`) {$state$ =` n`; continue; }`
s_2'
`case` n`:` s_1'
`case` $n+1$`:` j_s`,`
$h_1; h_2, [n, n+1] \cdot L_1 \cdot L_2)$

end

Branching:

$\mathcal{T}[\![\texttt{break}]\!]j_s n r = (n, \texttt{\{\$state\$ = \#b }r\texttt{; continue;\}}, \epsilon, [\,])$

$\mathcal{T}[\![\texttt{continue}]\!]j_s n r = (n, \texttt{\{\$state\$ = \#c }r\texttt{; continue;\}}, \epsilon, [\,])$

Return:

$\mathcal{T}[\![\texttt{return}]\!]j_s n r =$

$(n+2,$ `case` n`: $yieldpt$ = $state$ =` $n+1$`;`
`return true;`
`case` $n+1$`: return false;`,
$\epsilon, [n, n+1])$

Yield:

$\mathcal{T}[\![\texttt{yield}]\!] j_s n r =$

$(n+2,$ `case` n`: $yieldpt$ = $state$ =` $n+1$`; return true;`
`case` $n+1$`:` j_s,
$\epsilon, [n, n+1])$

Raise:

$\mathcal{T}[\![\texttt{raise}\ e]\!] j_s n r =$

let
$(p, t, n') = (\#\text{handler}\ r)\tau_e$
in
$(n,$ `if ((`t `= `p`.trap(`e`)) != null) { $state$ =` n'`; continue; }`
j_s,
$\epsilon, [\,])$
end

Try:

$\mathcal{T}[\![$`try {`s`} catch (`$\tau_1\ x_1$`) {`s_1`} catch (`$\tau_2\ x_2$`) {`s_2`}...catch (`$\tau_k\ x_k$`) {`s_k`}`
`catch (trap `$\tau_{k+1}\ x_{k+1}$`) {`s_{k+1}`} catch (trap `$\tau_{k+2}\ x_{k+2}$`) {`s_{k+2}`}...catch (trap `$\tau_m\ x_m$`) {`s_m`}`
`trap (`$\tau_{m+1}\ x_{m+1}$`) {`s_{m+1}`} trap (`$\tau_{m+2}\ x_{m+2}$`) {`s_{m+2}`}...trap (`$\tau_p\ x_p$`) {`s_p`}`$]\!] j_s n r =$

let
fall_through =
case t of unit ⇒ {return false;}
| int i ⇒ {$state$ = i; continue;}
end
$(n', s', h, L) = \mathcal{T}[\![s]\!](\{\$state\$ = n+p+2;\ \texttt{continue;}\})(n+p+3)(r[\text{try} \mapsto n])$
$(n_1, s'_1, h_1, L_1) = \mathcal{T}[\![s_1]\!](\{\$state\$ = n+1;\ \texttt{return true;}\})n'(r[\text{rb} \mapsto n+p+2][\text{rc} \mapsto n])$
$(n_2, s'_2, h_2, L_2) = \mathcal{T}[\![s_2]\!](\{\$state\$ = n+1;\ \texttt{return true;}\})n_1(r[\text{rb} \mapsto n+p+2][\text{rc} \mapsto n])$
…
$(n_p, s'_p, h_p, L_p) = \mathcal{T}[\![s_p]\!](\{\$state\$ = n+1;\ \texttt{return true;}\})n_{p-1}(r[\text{rb} \mapsto n+p+2][\text{rc} \mapsto n])$
in
$(n_p,$ `case` n`:` s'
`case` $n+1$`: $state$ = $n+p+2$; continue;`
`case` $n+2$`:` $\tau_1\ x_1$ `= (`τ_1`) $thrown$;` s'_1
`case` $n+3$`:` $\tau_2\ x_2$ `= (`τ_2`) $thrown$;` s'_2
⋮
`case` $n+m+1$`:` $\tau_m\ x_m$ `= (`τ_m`) $thrown$;` s'_m
`case` $n+m+2$`:` $\tau_{m+1}\ x_{m+1}$ `= (`τ_{m+1}`) $signal$;` s'_{m+1}
`case` $n+m+3$`:` $\tau_{m+2}\ x_{m+2}$ `= (`τ_{m+2}`) $signal$;` s'_{m+2}
⋮
`case` $n+p+1$`:` $\tau_p\ x_p$ `= (`τ_p`) $signal$;` s'_p
`case` $n+p+2$`:` j_s,
$h; h_1; \ldots; h_p;$
`case` n`:`
`case` $n+1$`:`
`case` L`: if ($thrown$ instanceof` τ_1`) $state$ =` $n+2$`;`
`else if ($thrown$ instanceof` τ_2`) $state$ =` $n+3$`;`
`else ...if ($thrown$ instanceof` τ_m`) $state$ =` $n+m+1$`;`
`else if ($signal$ instanceof` τ_{m+1}`) $state$ =` $n+m+2$`;`
`else ...if ($signal$ instanceof` τ_p`) $state$ =` $n+p+1$`;`
`else` *fall_through*
`return true;`,
$[n+2, n+3, \ldots, n+k+1] \cdot L_1 \cdot \cdots \cdot L_p)$
end

Resume:

$\mathcal{T}[\![\texttt{resume yield}]\!]j_s nr = (n, \{\texttt{\$state\$ = \$yieldpt\$; return true;}\}, \epsilon, [\,])$

For $R \in \{\texttt{break}, \texttt{continue}, \texttt{yield}\}$:

$\mathcal{T}[\![\texttt{resume } R \texttt{ trap } (\tau_1\ x_1)\ \{s_1\} \ldots \texttt{trap } (\tau_k\ x_k)\ \{s_k\}]\!]j_s nr =$

let

$fall\_through =$

case #try r of unit ⇒ {return false;}

| int i ⇒ {$state$ = i; continue;}

end

$set\_resumept =$

case R of yield ⇒ {$resumept$ = $yieldpt$;}

| _ ⇒ ϵ

end

$set\_state =$

case R of break ⇒ {$state$ = #rb r;}

| continue ⇒ {$state$ = #rc r;}

| yield ⇒ {$state$ = $resumept$;}

end

$(n_1, s_1', h_1, L_1) = \mathcal{T}[\![s_1]\!]\epsilon(n+k+2)r$

$(n_2, s_2', h_2, L_2) = \mathcal{T}[\![s_2]\!]\epsilon n_1 r$

$\ldots$

$(n_k, s_k', h_k, L_k) = \mathcal{T}[\![s_k]\!]\epsilon n_{k-1} r$

in

$(n_k,$ case n: $set\_resumept$
$\$yieldpt\$ = \$state\$ = n+1;$
return true;
case $n+1$: set_state
continue;
case $n+2$: $\tau_1\ x_1$ = (τ_1) $signal$; s_1'
case $n+3$: $\tau_2\ x_2$ = (τ_2) $signal$; s_2'
$\vdots$
case $n+k+1$: $\tau_k\ x_k$ = (τ_k) $signal$; s_k',
$h_1; \ldots; h_k;$
case $n+1$: if ($signal$ instanceof τ_1) $state$ = $n+2$;
else if ($signal$ instanceof τ_2) $state$ = $n+3$;
else ...if ($signal$ instanceof τ_k) $state$ = $n+k+1$;
else $fall\_through$
return true;,
$[n, n+2, \ldots, n+k+1] \cdot L_1 \cdot \cdots \cdot L_k)$

end

Loops:

$\mathcal{T}[\![\texttt{do } s \texttt{ while } e]\!]j_s nr =$

let

$(n', s', h, L) = \mathcal{T}[\![s]\!](\{\texttt{\$state\$ = } n+1\texttt{; continue;}\})(n+3)(r[\mathsf{b} \mapsto n][\mathsf{c} \mapsto n+1])$

in

$(n',$ case n: s'
case $n+1$: if (e) { $state$ = n; continue; }
case $n+2$: j_s,
$h, [n, n+1, n+2] \cdot L)$

end

$\mathcal{T}[\![\texttt{for (}s_1\texttt{; }e\texttt{; }s_2\texttt{) }s_3]\!]j_s n r =$

let

$(n', s_3', h, L) = \mathcal{T}[\![s_3]\!](\{\texttt{\$state\$ = }n+2\texttt{; continue;}\})(n+4)(r[\mathsf{b} \mapsto n+3][\mathsf{c} \mapsto n+2])$

in

$(n_2,$ s_1
`case` n`: if (!`e`) { $state$ =` $n+3$`; continue; }`
`case` $n+1$`:` s_3'
`case` $n+2$`:` s_2`; $state$ =` n`; continue;`
`case` $n+3$`:` s_c',
$h, [n, n+1, n+2, n+3] \cdot L)$

end

$\mathcal{T}[\![\texttt{while (}e\texttt{) }s]\!]j_s n r =$

let

$(n', s', h, L) = \mathcal{T}[\![s]\!](\{\texttt{\$state\$ = }n\texttt{; continue;}\})(n+2)(r[\mathsf{b} \mapsto n+1][\mathsf{c} \mapsto n])$

in

$(n',$ `case` n`: if (!`e`) { $state$ =` $n+1$`; continue; }`
s'
`case` $n+1$`:` j_c,
$h, [n, n+1] \cdot L)$

end

$\mathcal{T}[\![\texttt{while}^{p,t}_{\langle\tau_1,\ldots,\tau_k\rangle}\texttt{ (}e\texttt{) }s]\!]j_s n r =$

let

$fall\_through =$
case t of unit $\Rightarrow$ `{return false;}`
| int $i \Rightarrow$ `{$state$ =` i`;}`
end

$h' = \lambda\tau.\mathsf{if}\ \tau \in \{\vec{\tau_i}\}\ \mathsf{then}\ (p, t, n+2)\ \mathsf{else}\ (\#\mathsf{handler}\ r)\tau$

$r' = r[\mathsf{b} \mapsto n+2][\mathsf{c} \mapsto n][\mathsf{try} \mapsto n][\mathsf{handler} \mapsto h']$

$(n', s', h, L) = \mathcal{T}[\![s]\!](\{\texttt{\$state\$ = }n\texttt{; continue;}\})(n+3)r'$

in

$(n',$ `case` n`: if (!`e`) { $state$ =` $n+2$`; continue; }`
s'
`case` $n+1$`: if ((`t `= `p`.trap($signal$)) == null) {`
`$state$ = $yieldpt$;`
`return true;`
`}`
`case` $n+2$`:` s_2,
h;
`case` n`:`
`case` L`: if ($signal$ instanceof` τ_1
`|| $signal$ instanceof` τ_2
`|| ...`
`|| $signal$ instanceof` τ_k`) $state$ =` $n+1$`;`
`else` *fall_through*
`continue;`,
$[n+1, n+2])$

end