Interruptible Iterators

Jed Liu Aaron Kimball Andrew C. Myers

Department of Computer Science
Cornell University
{liujed,ak333,andru}@cs.cornell.edu

Abstract

This paper introduces interruptible iterators, a language feature that makes expressive iteration abstractions much easier to implement. Iteration abstractions are valuable for software design, as shown by their frequent use in well-designed data structure libraries such as the Java Collections Framework. While Java iterators support iteration abstraction well from the standpoint of client code, they are awkward to implement correctly and efficiently, especially if the iterator needs to support imperative update of the underlying collection, such as removing the current element. Some languages, such as CLU and C# 2.0, support iteration through a limited coroutine mechanism, but these mechanisms do not support imperative updates. Interruptible iterators are more powerful coroutines in which the loop body is able to interrupt the iterator with requests to perform updates. Interrupts are similar to exceptions, but propagate differently and have resumption semantics. Interruptible iterators have been implemented as part of the JMatch programming language, an extended version of Java. A JMatch reimplementation of the Java Collections Framework shows that implementations can be made substantially shorter and simpler; performance results show that this language mechanism can also be implemented efficiently.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features— Coroutines; D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.1.6 [*Programming Techniques*]: Logic Programming; D.3.2 [*Programming Languages*]: Language Classifications— Multiparadigm languages

General Terms Design, Languages

Keywords Coroutine, Exception, Java, JMatch, Logic Programming

1. Introduction

Iteration abstractions are operations intended to support convenient iteration over the elements of some sequence. Iteration abstractions are an essential aspect of well-designed interfaces, which is demonstrated by their prevalence in modern collection class libraries. Mainstream languages, such as Java [GJSB00] and C# [Mic01], have been evolving to include features explicitly for iteration abstraction [Sun04, HWG03]. It is therefore important to understand how to support iteration abstractions well in imperative, object-oriented languages.

Different languages take different approaches to iteration abstraction. The Java approach of using iterator objects (cursor objects) is common; however, iteration abstractions can also be implemented using higher-order functions, as in ML [MTH90], or through a built-in language feature, as in CLU [LSAS77] or C# 2.0 [HWG03]. Most mechanisms provide iteration abstractions that are convenient for client code to use. For example, Java iterator objects have methods `next` and `hasNext` for use in `for` loops.

However, it is often not appreciated that iterators are remarkably difficult to implement in most object-oriented languages. An iterator object must record the state of the iteration in a way that permits the iteration to be restarted. This requirement leads to awkward code reminiscent of continuation-passing style [HFW86]. Consequently, programmers commonly use iterators as clients, but usually avoid defining their own iteration abstractions, leading to interfaces that are less clean and simple than they could be.

A number of languages (e.g., [L$^+$81, GHK81, MMPN93, MOSS96, DTH04, HWG03]) support more convenient implementation of iterators with a structured coroutine mechanism in which iterator code can `yield` iterated values back to the loop. However, this mechanism has limitations: in particular, it prevents the client from modifying the underlying collection during iteration. For example, Java iterators usually support a `remove` operation that removes the last element yielded by the iterator. Languages supporting coroutine iterators do not allow these updates.

Implementing a Java iterator object that correctly handles `remove` is even more challenging than implementing a simple non-mutating iterator. The iterator must be able to handle a `remove` request at any point during iteration, update the underlying collection state, and leave the iterator object ready to handle the next request. At present there is no good way to implement such iterators.

In this paper, we describe *interruptible iterators*, a way to conveniently implement efficient iterators that can handle additional requests such as element removal. Code using an interruptible iterator can raise *interrupts* that request the iterator to perform additional operations such as `remove`. Like exceptions, interrupts are a nonlocal control mechanism; unlike exceptions, a handled interrupt results in resumption, and interrupts propagate logically downwards in the call stack rather than upward.

This mechanism has been implemented in the JMatch language; an implementation is available for public download [LM02]. JMatch introduces interruptible iterators in the context of Java, but the goal of this paper is not to propose an extension to Java per se, but rather to introduce a new iteration mechanism compatible with imperative languages in general.

The rest of the paper is structured as follows. Section 2 gives background on previous iterator mechanisms and their limitations. Section 3 introduces interruptible iterators and describes how they work in the JMatch programming language. JMatch also supports

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.
Copyright © 2006 ACM 1-59593-027-2/06/0001. . . $5.00.

implementation of iterators through a simple logic-programming mechanism; the integration of interrupts with this mechanism is covered in Section 4. Section 5 discusses some interesting static checking issues. Section 6 discusses details of the implementation. Results on the expressiveness and performance of JMatch are given in Section 7. Related work is covered in Section 8, and the paper concludes in Section 9. The static semantics of interruptible iterators is given in the appendix.

2. Iteration abstractions

Iteration abstractions are important for modular software design. When used as part of the interface to an abstract data type, an iteration abstraction promotes declarative programming and avoids exposure of the ADT implementation. For this reason, contemporary data structure libraries such as STL [MDS01], the Java Collections Framework [GJSB00], and the Standard ML Basis Library [GR04] make extensive use of iteration abstractions.

Like any abstraction, an iteration abstraction defines the interaction between the *client code* that uses the abstraction and the *implementation* of the abstraction. The key property of an iteration abstraction is that client code is able to obtain, on demand, another element from a finite or infinite sequence of values, without causing the computation of the entire sequence ahead of time. Typically, an iteration abstraction is invoked from a loop in the program. On each loop iteration the iterator is invoked to obtain a value for the next iteration.

This paper refers to any abstraction usable in this way as an iterator, regardless of whether it happens to be implemented as an object (as in Java), a higher-order function (as in SML), or some specialized construct (as in CLU). The form iterators take depends on the language and programming style in use. The challenge for the language designer is to make iterators convenient for both the client and the implementer. This is particularly difficult if imperative updates may be needed during the use of the iterator.

2.1 Iterator objects

As part of its standard collection classes, the Java language [GJSB00] provides an interface `Iterator`, modeled on the "Iterator" design pattern in which iteration abstraction is provided by iterator objects (also known as *cursor objects*). The value of iterator objects is that they require no special language support.

The Java `Iterator` interface is defined as follows:

```
interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

The `next` method advances the iterator to the next element in sequence, and returns it. The `hasNext` method determines whether there *is* a next element, but has no visible side-effect. And `remove` removes the current element, if any. An iterator in Java is an object whose class implements this interface; iteration is performed by invoking these methods.

For example, the following Java code iterates over the Collection `c`, printing out all its elements, and while doing so, removes all elements that are `null`:

```
for (Iterator it = c.iterator(); it.hasNext(); ) {
    Object o = it.next();
    if (o == null) it.remove();
    System.out.println(o);
}
```

This example shows the expressive power and convenience of Java iterators for the user of the abstraction.

However, Java iterators are difficult to implement. A Java iterator must be able to accept `hasNext`, `next`, and `remove` methods at every point during iteration, and therefore it must record enough state to allow the iteration computation to be restarted wherever it last left off. This often leads to an awkward style of programming in which the iterator is implemented as a complicated state machine. For example, consider implementing an in-order iterator for a binary search tree without parent node pointers. The iterator has at least five different states:

1. before any tree elements have been produced,
2. iterating over the elements of the left child,
3. having just produced the element stored at the current node,
4. iterating over the elements of the right child, and
5. at the end of the iteration.

In fact, the iterator has even more states, because the current element might have been removed by the `remove` method. The state machine necessary to correctly handle all possible requests in all these states is complex, particularly if the iterator is to produce each element in asymptotically optimal $O(1)$ time. An iterator for even a data structure as simple as a binary tree is difficult to write correctly; it is little wonder that programmers shy away from defining their own iteration abstractions.

2.2 Imperative update

In an imperative language like Java, it is important to be able to update data structures—even while they are being iterated over, as in the above example. However, operations that change the underlying data structure directly are unsafe during iteration, because a change to the data structure might violate invariants that the iterator depends on. For example, deleting a binary search tree node might leave the iterator pointing to a node that is no longer part of the tree. Therefore, any updates must go through the iterator so it can adjust its internal state to compensate; this is why many iterators in the Java Collections Framework support the `remove` method.

Supporting the `remove` method is even more challenging than implementing a simple non-mutating iterator. The iterator must be able to handle the `remove` request at any point during iteration, update the underlying collection state, and leave the iterator object ready to handle the next element request.

2.3 Nested iterators and compositionality

Abstract data types are frequently implemented using other abstract data types, and iterators of the higher-level abstraction are frequently implemented using the iterators of the lower-level abstraction. For example, a hash table can be implemented using arrays and linked lists. An iterator over the key–value pairs in the hash table might be implemented by an iteration over the elements of the array, with an inner loop iterating over the elements of the linked list. At run time, this *nested iterator* coding style results, in general, in a stack of iterators in which higher-level iterators invoke lower-level iterators, possibly filtering or transforming the values from the lower level. The same stack of iterators can be seen in iterators for inductively defined types, such as binary trees, as described in Section 2.1.

Nested iterators create significant challenges for iterator mechanisms. In fact, the Java `Iterator` interface is not rich enough to fully support nested iterators, although this seems not to have been observed before. The problem lies in an interaction between the `hasNext`, `next`, and `remove` methods.

When iterators are nested, implementation of `hasNext` for a higher-level abstraction requires, in general, use of `next` on the lower-level abstraction, because some elements of the lower-level abstraction may be filtered out by the higher level, and must be skipped over. For example, in a hash table it may be necessary to skip over empty buckets to determine whether there is a non-empty bucket still to come. But this creates a conflict with `remove`, which is supposed to remove the current element. If the lower-level iterator has been advanced by calling `next`, its current element can no longer be removed by calling its `remove` operation. Thus, Java iterators are not compositional.

2.4 Coroutine iterators

An alternative to iterator objects is coroutine-style iterators, which were introduced by CLU [L$^+$81] and are found in other languages such as ICON [GHK81], Sather [MOSS96], Python [vR03], Ruby [DTH04], and C# 2.0 [Mic01]. In this approach, iterator abstractions are implemented by special iterator methods that *yield* values back to the receiving context (typically, a loop body). These methods are a limited form of coroutine because whenever a new value is requested from the iterator, execution of the iterator method continues immediately after the last `yield` statement.

The JMatch language [LM03], an extension to Java, supports coroutine iterators. For example, in JMatch an in-order binary tree iterator can be written succinctly and clearly:

```
class Node {
  Node left, right;
  int val;
  int elements() iterates(result) {
    foreach (left != null && int elt = left.elements())
        yield elt;
    yield val;
    foreach (right != null && int elt = right.elements())
        yield elt;
  }
}
```

The method `elements` is an iterator, as indicated by the clause `iterates(result)`. Using a JMatch `foreach` statement, the method iterates over and yields the elements of the left child, then yields the value in the tree node, then iterates over and yields the elements of the right child. The code is shorter and clearer than its Java counterpart because the various iterator states described in Section 2.1 are now encoded in the next iterator program point to resume.

In the general case of nested iterators, a *stack* of coroutines results, as shown in Figure 1. The top of the stack is the top-level iterator that is being used directly by the client code, the next coroutine on the stack is for an iterator that the top-level iterator is using internally, and so on.

Coroutine iterators are clear, concise, and compositional. They can also be implemented efficiently, especially when they are invoked from a `foreach` construct, because in that case they can be allocated on the program stack even when nested [LAS78]; however, when iterators are first-class (generators, streams, cursor objects) their activation records must be heap-allocated. The weakness of coroutine iterators is that they do not support imperative update such as element removal during iteration. Imperative updates must be deferred until there are no iterators observing the updated data structure.

2.5 Iterators as higher-order functions

Another way to provide iteration abstraction, followed in the functional languages community, is to use higher-order functions. For

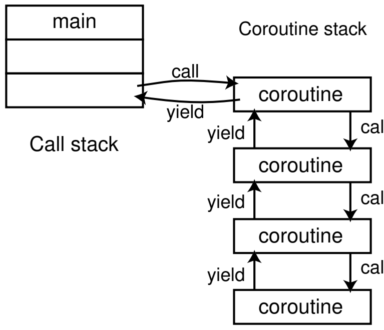

Figure 1. A stack of coroutines.

example, a "`fold`" abstraction for iterating over values of type α might be declared in SML as follows:

```
fun fold (body: α * β → β, initial: β) : β
```

The first argument to this function is the body of the loop that iteratively receives elements of type α, implemented as a function with type $\alpha * \beta \to \beta$. The type variable β is the type of information that is passed from one loop iteration to the next. The second argument is the initial information that begins the iteration; the result of the whole iteration is the result (of type β) last computed by the loop body. Unlike Java iterators, this kind of iterator cannot be used as an abstract stream from which elements can be obtained; the entire use of the iterated values must be expressed in the loop body that is passed as an argument.

In a functional style, deletion of collection elements is usually accomplished through an operation like SML's `filter`. Filtering results in needless copying when applied to imperative data structures like hash tables, and like `fold` it has less expressive power than Java iterators.

Iteration (and coroutines) can also be implemented using more powerful constructs such as threads and continuations. Threads introduce the problem of reasoning about concurrency, and their implementations usually do not support the efficient, fine-granularity context switching needed for iteration. Using and reasoning about continuations also seems to be difficult for many programmers. First-class continuations place many demands on the compiler and run-time system; and make optimization (of memory management, for example) more difficult.

3. Interruptible iterators

Coroutine-style iterators are convenient for the user and also easy to implement. However, they do not support imperative update operations like `remove`. This paper introduces *interruptible iterators*, a mechanism that extends coroutine iterators to handle update operations through an exception-like mechanism called *interrupts* (not to be confused with the existing Java method `Thread.interrupt`).

3.1 Raising interrupts

The core idea is straightforward: when client code wants to tell the iterator to perform some operation such as removing an element, it *interrupts* the iterator. The iterator handles the interrupt, performs the operations, and returns control to the place where the interrupt was raised, typically within a loop body.

To raise an interrupt in an iterator, JMatch adds a `raise` statement. As with Java exceptions, the `raise` statement takes an object as an argument that indicates the kind of interrupt, and hence the operation that is intended.

```
class List implements Collection {
  Object head; List tail;
  ...
  Object elements()
    traps SetValue iterates(result) {
      List cur = this;
      while (cur != null) {
        try {
          yield cur.head;
        } trap (SetValue s) {
          cur.head = s.value;
        }
        cur = cur.tail;
      }
    } ...
}
```

Figure 2. An iterator with imperative update

For instance, the Java code example in Section 2.1 can be written in JMatch as follows:

```
foreach (Object o = c.elements()) {
  if (o == null) raise new Remove();
  System.out.println(o);
}
```

Here, the `elements` method is an iterator that successively binds the variable `o` to the elements of the collection `c`. The `raise` statement generates a `Remove` interrupt which is received by the iterator, causing the iterator to update the collection `c` accordingly. Once the iterator has finished removing the null element, control is returned to the loop body immediately after the `raise`.

3.2 Interrupt propagation

Iterator interrupts and Java exceptions are similar in that they are signals with non-local handlers, but there are two key differences. The first is that exceptions have termination semantics while interrupts have *resumption semantics* [Goo75]. The `raise` statement in the example above does not terminate the `foreach` loop. Once the `Remove` interrupt is handled, control returns to the point after the `raise` statement, and the rest of the `foreach` body is executed.

The second difference is the direction of propagation. Recall that in general an iteration is implemented by a stack of coroutines, shown in Figure 1. Whereas Java exceptions propagate *up* the call stack, interrupts propagate *down* the coroutine stack. Just as a Java exception can propagate up multiple stack frames before a handler is found, an interrupt may descend several frames in the coroutine stack before a handler is found.

In the example above, the `Remove` interrupt is handled by the `c.elements` coroutine, rather than by coroutines lower on the stack that are used to implement `c.elements`. These are the only reasonable semantics because the client code is making the `Remove` request to the `c.elements` iterator; if a lower-level iterator tried to handle the remove request, it might well break data structure invariants involving the higher-level abstraction.

For example, in a hash table implemented as an array of linked lists, an iterator for the hash table might use the iterator of the underlying linked-list data structure. Suppose some client code using the hash table iterator raises an interrupt to insert an element. Inserting into the current linked list would be wrong because the new element might hash to a different location. To preserve the hash table invariant, the hash table iterator must be able to intercept the interrupt, and this requires downward interrupt propagation.

Note that when iterators are implemented as higher-order functions ("fold" operations), ordinary exceptions as in SML [MTH90] have exactly the reverse behavior: they propagate upward from the bottom-most to the topmost iterator on the stack. This makes them unsuitable for implementing imperative update.

3.3 Declaring handlers

JMatch iterators explicitly declare which interrupts they can handle, so the compiler can determine that every interrupt will be handled by a single handler. This follows the design of Java, which requires that all thrown exceptions be handled, except those corresponding to run-time errors. Handling of an interrupt is declared with a `traps` clause in the method header. For example, the following `Collection` interface has an `elements` method that removes an element when it traps a `Remove` interrupt:

```
interface Collection {
  ...
  Object elements()
    traps Remove iterates(result);
}
```

The `traps Remove` clause specifies that the iterator `elements` can handle any interrupt that is an instance of a subtype of `Remove`. In this example, `Remove` is a user-defined class; in JMatch, any object type can be used for interrupts.

3.4 Handling interrupts

To the iterator code, interrupts appear to be raised just after `yield` statements, because these are the points in the code where control is handed back to the client code. To handle interrupts from a `yield`, the `yield` is placed in the `try` block of a `try`...`trap` statement. Figure 2 shows an iterator for a linked list that can replace list elements during iteration when it traps a `SetValue` interrupt.

In this example, when the handler is finished executing, the iterator sets its control state so that iteration resumes at the variable assignment following the `try`. It then returns program control to the caller. If the client code raises several interrupts in succession before resuming iteration, the same set of interrupt handlers is reused for each interrupt. An alternate set of handlers can also be specified, as discussed in Section 3.6.

It is also possible to delegate interrupt handling to an iterator lower on the coroutine stack. Here is a less efficient recursive version of the `elements` iterator, which creates a new iterator for every node in the linked list:

```
Object elements() traps SetValue
                  iterates(result) {
  try {
    yield head;
  } trap (SetValue v) {
    head = v.value;
  }
  if (tail != null)
    foreach (Object elt = tail.elements())
      yield elt;
}
```

Here, a `SetValue` interrupt raised during the second `yield` statement is handled by the `foreach` statement, which passes the interrupt down the coroutine stack until it reaches the instance in which the statement `yield head` has just been executed. A `foreach` statement handles the same interrupts as its condition expression.

Dynamic propagation of interrupts proceeds downward in the coroutine stack, as described. Within each method body, an interrupt propagates lexically outward from the last `yield` statement executed, invoking the innermost handler for the given interrupt. If a method is declared to handle an interrupt, every `yield` must be lexically surrounded by either an appropriate `try...trap` or another iterator that handles the interrupt. As discussed in Section 5, this is statically checked by the compiler.

3.5 Resuming iteration

In the above examples, the interrupt handlers transfer control back to the client code by executing to the end of the handler. Interrupt handlers can use the `resume` statement to complete interrupt handling by explicitly suspending the iterator and transferring control back to the client code. The `resume` statement has a few variants that differ in how they set the control state of the suspended iterator.

The `resume break` statement transfers control to the end of the associated `try` statement. As in the linked list examples above, an interrupt handler ends with an implicit `resume break` if control would otherwise reach the end of the handler.

A `resume continue` statement returns control to the beginning of the associated `try` statement. This is shown in the following integer stream iterator example in which a `Reset` interrupt resets the iterator to the beginning of the stream:

```
int elements()
  traps Reset iterates(result) {
    try {
      int i = 0;
      while (true) {
        yield i;
        i++;
      }
    } trap (Reset) {
      resume continue;
    }
  }
```

When the iterator traps a `Reset` interrupt, it uses the `resume continue` to set the control state back to the beginning of the `try` block and return program control back to the client code.

Finally, a `resume yield` statement resumes executing the client but returns iterator control to where it was before the interrupt handler was invoked.

3.6 Exceptions

Handling an interrupt may require transferring control either back up or further down the coroutine stack. A common instance of this is element removal in data structures. For example, removing an element from a balanced binary search tree may cause non-local modifications to the tree.

Control can be transferred down the coroutine stack by raising a new interrupt. To transfer control back up the stack, a handler throws a *trap exception*, which is simply an exception that is thrown in an interrupt handler. Like ordinary Java exceptions, trap exceptions have termination semantics. Trap exceptions are distinguished from regular exceptions because it is illegal for an iterator to yield a value to client code during interrupt handling. Trap exceptions can only be handled by *trap exception handlers*.

JMatch follows Java's design choice of preferring static errors over dynamic ones. Thus, the compiler checks that control flow from a trap exception handler must lead to a `resume` statement, or else terminate the entire iteration with an exception. Control flow from a regular exception must lead to a `yield` statement, or else terminate the entire iteration. If a trap exception propagates back to the context of the original `raise` statement (that is, it reaches a context that does not handle interrupts), the trap exception becomes an ordinary Java exception that terminates the iteration.

```
int elements() traps Remove : Nullify
               iterates(result) {
  try {
    foreach (left != null && int x = left.elements())
      yield x;
  } catch (trap Nullify) {
    left = null;
  }
  boolean deleted = false;
  try {
    yield val;
  } trap (Remove) {
    if (deleted) throw IllegalStateException();
    deleted = true;
    if (deleteVal()) resume continue;
  }
  try {
    foreach (right != null && int x = right.elements())
      yield x;
  } catch (trap Nullify) {
    right = null;
  }
}
...
// Returns true when an element is moved up from
// the right side of the tree.  Throws Nullify when
// deleting a leaf node.
boolean deleteVal() throws Nullify { ... }
```

Figure 3. Binary search tree iterator with element removal

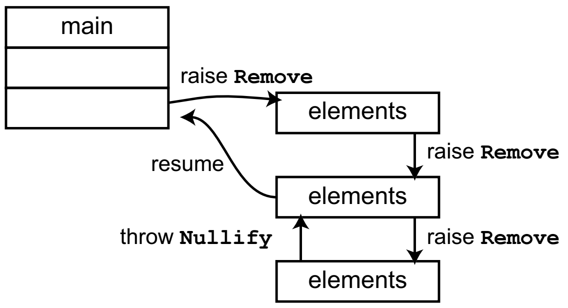

Figure 4. The binary search tree example in Figure 3, handling a `Remove` interrupt.

Figure 3 shows how one might remove elements in an unbalanced binary search tree where the `elements` iterator is implemented recursively. To remove a tree node, the JMatch code refers to an elided `deleteVal` method. This method removes the value at the node in the usual way, but with two minor differences. First, if the node being removed is a leaf, the method throws a `Nullify` exception to indicate that the leaf node should be replaced with a `null` value. The call to `deleteVal` does not occur in a `try` statement that handles a `Nullify` trap exception, so the trap exception propagates to the iterator's client for handling. Therefore, the method header declares `traps Remove : Nullify` to indicate that the handler for `Remove` can throw a `Nullify` trap exception.

Second, `deleteVal` returns a boolean indicating whether a node in the right subtree was used to replace the deleted node, in which case the value at the current node has not been yielded yet. Therefore, `resume continue` is used by the `Remove` handler to ensure that all elements are yielded by the iterator exactly once.

Figure 4 illustrates the interprocedural control flow that occurs when client code raises `Remove` to remove a leaf node. The interrupt propagates down the coroutine stack until it finds the tree node containing the last element yielded. Splicing out the leaf node requires access to the previous node, so the `deleteVal` method throws a `Nullify` trap exception to walk up one coroutine stack frame. There the exception is caught and the leaf node spliced out; iteration continues from that point in the tree.

Client code can raise several successive interrupts before resuming iteration, and coroutine methods must be prepared to handle these interrupts. A `resume` statement may have `trap` clauses to specify how to handle the next interrupt. In this example, neither the `resume continue` statement nor the implicit `resume break` statements are annotated with trap clauses. This means the original set of interrupt handlers handle multiple successive interrupts.

4. Declarative iterators

In addition to coroutine iterators, JMatch has simple logic programming features that enable a more declarative implementation of iteration abstractions. Combining interrupts and logic programming features enables a single piece of JMatch code to concisely implement multiple Java methods and iterators, while supporting imperative update. For details on the logic programming and pattern matching capabilities of JMatch, see [LM03].

4.1 Predicate methods

In JMatch, an iterator can be implemented as a method whose return type is declared as `boolean`. The body of the method is a logical formula that expresses the conditions under which the iterator should yield a value—that is, when the formula would evaluate to `true`. For example, the simple list iterator of Section 3.4 can be written even more compactly as a logical formula, ignoring for now the issue of imperative update:

```
boolean contains(Object o) iterates(o) (
  head == o ||
  tail != null && tail.contains(o)
)
```

When this formula is treated as an iterator, the disjunction operator `||` defines different ways to satisfy the condition that `o` is contained in the list. Each way yields values to the client. The right-hand side is a recursive invocation of the `contains` predicate, yielding all the elements in the rest of the list. A disjunction is always explored left-to-right.

4.2 Formulas

In certain contexts, such as in the condition of a `foreach` statement, a boolean formula may be used if JMatch is able to automatically satisfy the formula by choosing values for variables declared in it. This capability is used to implement iterators like `contains`; the JMatch compiler generates code to find all satisfying assignments to the variables (in this case, the variable `o`), and these satisfying assignments are yielded to the client.

For example, the following code executes the loop body with `x` bound to `1`, and also executes it with `x` bound to 4 if `p` is non-null. The variable `p` is not solved for because it is bound outside the loop; its value is already known.

```
foreach(int x == 1 || x == 4 && p != null) { ... }
```

4.3 Modes

A formula expresses a relation among its variables; given that the values of some variables are known and some are unknown, it may be possible to find satisfying assignments for the unknowns. In general, a single formula can be evaluated in several *modes*, in which different subsets of variables are treated as knowns or unknowns. The ability to be evaluated in several modes allows a single formula to implement several iterators at once.

In a given mode, a formula's variables are either *knowns* or *unknowns*. In the forward mode, all formal arguments are knowns and the formula is evaluated as a boolean expression, as in Java. In backward modes, the formula behaves like an iterator—some variables are unknowns and satisfying assignments for them are sought. The JMatch compiler generates code to solve for unknowns.

4.4 Modal abstraction

Predicate methods can implement multiple modes with a single boolean formula. For example, the `contains` method given in Section 4.1 has a clause `iterates(o)`. This indicates that the method can be used to solve for the argument `o`, treating it as an unknown. However, there is also a default, *forward* mode that corresponds to the ordinary Java interpretation of the method signature: a mode in which `o` is known and the method returns `true` or `false`. The body of `contains` is a boolean formula that correctly implements both the backward, iterative mode and the forward, non-iterative mode.

This ability to write code that abstracts over multiple modes can make code considerably more compact. It also can simplify interfaces. For example, in the Java Collections Framework, the Java `Collection` interface declares separate methods for finding all elements and for checking whether a given object is an element:

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

The JMatch `contains` method in Section 4.1 provides both of these functionalities in one piece of code. In addition, it ensures that they agree with one another.

In any correct Java implementation, there is an equational relationship between the two operations: any object `o` produced by the iterator object satisfies `contains(o)`, and any object satisfying `contains(o)` is eventually generated by the iterator. When writing the specification for Java `Collection`, the specifier must describe this relationship so implementers can do their job correctly. In JMatch, the equational relationship is specified simply by the fact that these are modes of the same method, and implementing them with a single formula enforces this.

In general, a method may declare additional modes that the method implements, beyond the default, forward mode. For example, a mode `iterates(`$\vec{x_i}$`)` means that the method iterates over a *set* of satisfying assignments to $\vec{x_i}$. In iterative modes, results are returned from the method by the `yield` statement rather than by `return`.

The modes of a method may be implemented by separate formulas or using the statement language described in Section 3. The statement language is useful when no single boolean formula is solvable for all modes or when it would lead to inefficient code. For example, the following code separately implements the two modes of `List.contains`. The `foreach` statement used in this example loops through the satisfying assignments of a formula.

```
boolean contains(Object o) {
  return head == o || tail.contains(o);
} iterates(o) {
  yield head;
  if (tail != null)
    foreach (tail.contains(Object tmp)) {
      yield tmp;
    }
}
```

From this declaration, the compiler generates separate code for each mode. Whenever the `contains` predicate is used, the compiler generates a call to the code for the appropriate mode.

4.5 Handling interrupts

To handle interrupts in declarative iterators, the `try...trap` and `try...catch` constructs may be used within boolean formulas. Here, the body of the `try` is a boolean formula, whereas the `trap` handler is a statement. For example, consider implementing the `contains` method of a hash table. In the absence of `Remove` interrupts, both the forward and backward mode can be implemented compactly in a single boolean formula:

```
boolean contains(Object key, Object value)
    iterates(key) iterates(key, value)
(
  int hash = Math.abs(key.hashCode() % table.length) &&
  Bucket b = table[hash] &&
  b != null &&
  b.contains(hash, key, value)
)
```

In the forward mode, the hash code is computed and used to look up a `Bucket`; this bucket then checks using the forward mode of `Bucket.contains` to determine if it holds a mapping for `key`. In the two backward modes, the array index operation is used to enumerate all elements of `table`; the iterative mode of `Bucket.contains` is used to enumerate all elements in the bucket and the values of `hash`. This single method supersedes at least three methods of the Java `Map` interface.

To support element removal, there must be a handler for the `Remove` interrupt. In fact, the interrupt handler can be placed inside the `Bucket` class, which is simply a linked list, as shown in Figure 5. `Bucket` throws a `Replace` exception when the head of the list is removed; the hash table catches the exception and updates its array accordingly:

```
boolean contains(Object key, Object value)
    traps Remove
    iterates(key) iterates(key, value)
(
 (assume(int hash = abs(key.hashCode() % table.length)) &&
  Bucket b = table[hash] &&
  b != null &&
  b.contains(hash, key, value))
    catch (trap Replace r) {
      table[hash] = r.entry;
      resume continue
        trap Remove {
          throw new IllegalStateException();
        }
    }
)
```

Note that the code for handling remove requests is cleanly factored out from the iteration code. When a table index is updated, the `resume continue` restarts the solving of the formula to which the `catch` clause is attached. Also, the `resume continue` has an interrupt handler that traps attempts to remove the same entry twice.

This code contains one user-specified optimization: the `assume` applied to the first conjunct tells the JMatch compiler that it can assume the conjunct is true if it doesn't need to solve for variables in it. This conjunct is guaranteed to be true if the hash table satisfies its own data structure invariant; the `assume` prevents the compiler from generating unnecessary code to check the invariant.

```
class Bucket {
  int hash;
  Object key, value;
  Bucket next;
  ...
  boolean contains(int h, Object k, Object v)
      traps Remove: Replace
      iterates(k,v)
  (
    (h = hash && k = key && v = value)
        trap (Remove) { throw new Replace(next); }
    || next != null && next.contains(h,k,v)
        catch (trap Replace r) {
          next = r.entry;
          resume continue
            trap Remove {
              throw new IllegalStateException();
            }
        }
  )
}
```

Figure 5. Hash table bucket implemented as a linked list.

5. Static checking

Introducing interrupts creates new obligations for static checking. Raised interrupts and thrown trap exceptions need to be checked to ensure they are eventually handled. Additionally, interrupt and trap-exception handlers need to be checked to ensure that all execution paths lead to either a `resume` statement or an exception being thrown, so that control is transferred back to the client in a way that the client expects. This section briefly describes how these checks are performed; the formal details are given in the appendix.

Ensuring that raised interrupts are eventually handled involves checking two properties. First, every `raise` statement should occur lexically in a `foreach` statement whose formula condition handles the interrupt. A formula handles an interrupt if, for each solution it generates, there is a unique handler for the interrupt among the formula terms used to generate the solution. This can be checked statically by ensuring that all disjuncts in a disjunction have handlers and that exactly one term in any other formula or pattern has a handler.

Second, iterative method modes must have handlers for the interrupts declared by the method. For a formula implementation, this means checking that the formula can handle the interrupts. In a statement-based implementation, every `yield` must occur lexically in `foreach` or `try...trap` statements that can handle the interrupts. Furthermore, each `resume` statement must either have a handler for each interrupt or occur in a lexical context that can handle the interrupts.

Checking that thrown trap exceptions are eventually handled is similar to ordinary Java exception checking. The only difference is the trap exception declarations in method signatures need to be verified. This requires that each trap exception is annotated with the interrupt that caused it.

Finally, a simple control-flow analysis is sufficient to verify that all execution paths in interrupt and trap-exception handlers lead to either a `resume` statement or an exception being thrown.

All static analyses performed are strictly intra-procedural and can be done in a modular fashion. To allow separate compilation, JMatch `.class` files record extra information for each method: the modes supported, the traps handled, and the trap-exceptions thrown.

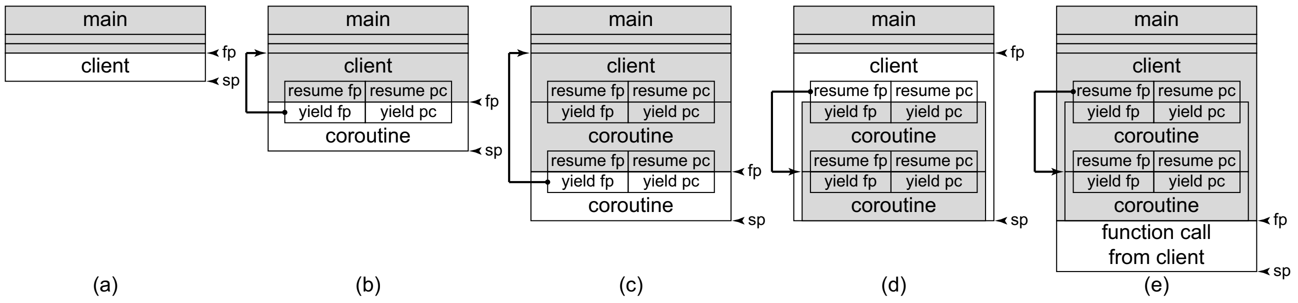

Figure 6. Possible stack layouts of a coroutine. The coroutine's activation records are kept entirely on the call stack.

6. Implementation

The JMatch compiler has been implemented using the Polyglot extensible compiler framework, which is designed to support Java language extensions [NCM03]. The compiler checks JMatch code and generates Java source code as its output, handing this source to a Java compiler. It supports separate compilation by embedding extra type information into the Java output. This implementation of JMatch is publicly available for download [LM02].

Because Java lacks support for low-level control over memory and stack layout, the performance of code generated by the released compiler suffers. To obtain an accurate assessment of the performance of interruptible iterators, we designed a translation from JMatch to C++. The differences between the Java- and C++-based implementations are discussed in Section 6.2.

6.1 Translation to Java

The JMatch implementation translates JMatch into Java in two major steps, briefly described here. A more detailed account is given in [LM03].

In the first pass, modal abstraction is removed by translating logical formulas into statements, using the `yield` statement to explicitly send results to the client code. Each declared method mode that is implemented as a logical formula is translated to distinct code consisting of JMatch statements.

In the second pass, iterative method modes are translated into classes that implement the Java `Iterator` interface; this transformation is similar to continuation-passing style (CPS) translation. An iterator object stores a state index that records the program point for resuming execution when it gets control back from client code. On entry to the iterator, a `switch` statement is used to jump to the appropriate code.

When an iterator receives an interrupt, the existing state index is mapped to the index corresponding to the appropriate interrupt handler. When an interrupt handler ends with a `resume`, it computes the appropriate state index at which to continue iterator execution.

6.2 Translation to C++

Translating JMatch iterators to Java results in the run-time allocation of many short-lived heap-allocated objects representing iterator activation records (see Figure 1). Results from profiling the generated code suggest that performance is hurt substantially by the additional garbage collection overhead that these objects add.

The key insight is that allocation of iterator activation records follows a LIFO stack discipline, and it is therefore possible to allocate these objects on the stack, reducing memory management overhead. We exploited this insight in designing a translation from JMatch to C++ [Str87], with a few macros containing inlined x86 assembly code. A prototype back end for this translation has been developed.

In the translation to C++, any iterator invoked with the `foreach` statement keeps its activation record entirely on the program call stack, as shown in Figure 6. The first time an iterator is entered, it is called as an ordinary function (b). Nested calls are handled similarly (c).

When the iterator yields back to the client code (d), it adjusts the program counter (`pc`) and frame pointer (`fp`) to resume execution in the client code. Since the iterator restores the frame pointer, but not the stack pointer (`sp`), the bottom of the stack grows to accommodate the coroutine frames as though they were part of the client's stack frame. To allow the client to resume execution in the iterator, the program counter and frame pointer for the iterator are stored in the client's activation record, allowing the client to resume execution in the iterator. Function calls from client code allocate at the bottom of the stack (e).

Further speedup is gained by inlining non-recursive iterators that have been declared `final`.

6.3 Tail-yield optimization

JMatch makes it convenient to compose iterators; however, if iterators are not implemented carefully, this convenience can come at a cost. For example, if implemented naively, the `Buckets` iterator of Figure 5 would take $O(n^2)$ time to iterate over the list.

Because of the recursion, the coroutine stack grows by one frame with each iteration. As shown in Figure 1, the request travels from the client code down to the bottom of the stack, and in a naive implementation, the result values are passed all the way back up the stack. Thus, each element request would incur a cost linear in the size of the coroutine stack.

JMatch solves this by performing a *tail-yield* optimization, with the result indicated in Figure 6(c). If no computation is performed on result values on their way back up to client code, the result values are passed directly to the client code and the next element request goes directly to the bottom of the stack. Most of the iterator code examples given so far benefit from this tail-yield optimization. For example, with the optimization the recursive linked-list iterator takes linear time, not quadratic time, to traverse a list.

When a nested iterator call is tail-yield optimized in the C++ translation, the client's program counter and frame pointer are passed to the nested iterator, as shown in Figure 6(c). This allows it to yield directly back to the client, and the client to directly resume execution in the nested iterator.

Traditional tail-call optimizations are performed only when a method directly returns the result of a call. To make the tail-yield optimization effective, it is important to accommodate simple transformations on the yielded result. For example, a set can be implemented by storing its elements as keys in a hash table. An iterator

	ArrayList			LinkedList			HashMap			TreeMap		
	JMatch	Java	PolyJ	JMatch	Java	PolyJ	JMatch	Java	PolyJ	JMatch	Java	PolyJ
Insert	47	56 (16%)	56 (16%)	55	64 (14%)	111 (50%)	55	99 (44%)	77 (29%)	100	229 (56%)	116 (14%)
Delete	21	28 (25%)	29 (28%)	50	53 (6%)	104 (52%)	41	73 (44%)	67 (39%)	177	200 (12%)	174 (-2%)
Lookup	14	48 (71%)	30 (53%)	58	78 (26%)	104 (44%)	40	83 (52%)	62 (35%)	56	88 (36%)	75 (25%)
Update	13	17 (24%)	16 (19%)	29	31 (6%)	66 (56%)	50	92 (46%)	70 (29%)	95	149 (36%)	106 (10%)
Iterate	65	124 (48%)	134 (51%)	59	87 (32%)	105 (44%)	40	133 (70%)	122 (67%)	245	251 (2%)	255 (4%)
Views	—	— (—)	— (—)	—	— (—)	— (—)	47	104 (55%)	81 (42%)	149	300 (50%)	268 (44%)
Total	112	204 (45%)	207 (46%)	155	249 (38%)	242 (36%)	158	434 (64%)	356 (56%)	472	805 (41%)	647 (27%)

Table 1. Lines of code in various ADT implementations. Totals are smaller than actual column sums because of code sharing. Percentages show how much smaller the corresponding JMatch code is.

```
boolean f(int x, int y)
  iterates(x,y) (
    g(y,x)
  )

boolean g(int a, int b)
  iterates(a,b) (
    a==1 && b==2
  )
```

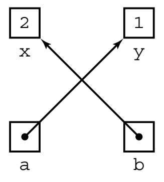

Figure 7. In a tail-yield optimized call, the sub-iterator receives a set of pointers to allow it to write result values directly to the client.

for the set might be implemented by an iteration over all key–value pairs in the hash table and dropping the value component.

JMatch allows result values to be reordered and omitted, making the tail-yield optimization more widely applicable. In the C++ implementation, the nested iterator is simply passed a set of pointers that map directly to the result locations from which the client will read (Figure 7).

6.4 First-class iterators

Like Java, JMatch supports first-class iterator objects that can be used as general streams. The `iterate` expression produces an object of a class implementing the Java `Iterator` interface, as shown in the following example:

```
Collection c = ... ;
Iterator it = iterate MyIter(Object o = c.elements());
while (it.hasNext()) {
  Object o = ((MyIter)it.next()).o;
  ...
}
```

As discussed in Section 2.3, Java iterators are not compositional because client code can remove elements after calling `hasNext`. JMatch addresses this issue by specializing the iterators created by `iterate`.

In general, a call to `hasNext` may need to invoke `next` on nested iterators, with arbitrary side effects. To make `hasNext` appear pure, side effects must be rolled back, which is straightforward as long as any side effects are updates to local variables of nested iterators. To support rollback, JMatch iterators extend the standard Java iterator interface with an automatically generated `checkpoint` method that records the state of local variables. A checkpoint is taken on every call to `hasNext`. If a `remove` operation is performed after a call to `hasNext` and before the corresponding call to `next`, the checkpoint is used to restore the previous values of local variables.

In the translation from JMatch to C++, first-class iterators are implemented as an `Iterator` object that allocates a block of heap memory to use as an internal stack. Iterator code runs on this separate stack, decoupling the iterator and client stacks without entailing individual heap allocation of iterator activation records.

7. Results

We evaluated interruptible iterators from the standpoint of both expressive power and performance.

7.1 Expressiveness

To explore the usability and expressive power of JMatch, we reimplemented the core collection classes from the Java Collections Framework. This reimplementation closely mirrors the functionality of the Sun Java Collections, including support for views. The only notable difference is that the JMatch Collections don't implement the bidirectional iteration functionality of the `List` classes. Additionally, the interfaces were redesigned to exploit modal abstraction, while keeping the functionality as described.

It is difficult to compare expressive power, but one measure is the amount of code that must be written. Table 1 shows that when measured in lines of code, the JMatch implementations of the collection classes are substantially shorter than the Sun implementations; in some cases, less than half as long. Line lengths in the two implementations are similar; blank lines and comments were not counted. The JMatch samples in this paper are representative of the coding style used.

To ensure that these results did not merely reflect verbosity on the part of the implementers at Sun, we also compared against a slightly more concise implementation of the Collections Framework written in PolyJ, an extension of Java that supports parametric polymorphism [MBL97, MLM98]. The JMatch implementations have 35% less code than the PolyJ implementations.

To avoid unfairly penalizing the Java and PolyJ code for features not included in the JMatch implementation, the code implementing these features was not included in the Java and PolyJ line counts.

The benefit from using JMatch results from a combination of two features: interruptible coroutine iterators and code sharing through modal abstraction. Operations such as lookup and iteration can usually be implemented with a single formula that implements multiple modes of a method. This is particularly true of `Map` implementations, which offer several modes of iteration. Interruptible iterators provide an expressive way to handle `remove` requests in both statement-based and formula-based implementations.

7.2 Performance

We measured performance to understand whether JMatch can be implemented efficiently. The results suggest it can.

Several JMatch iterators were translated into C++ using the translation described in Section 6.2. Their performance was compared with equivalent iterators from the C++ Standard Template Library (STL) [MDS01], and also with the corresponding Java iterators. Each JMatch iterator performed similarly to the corresponding STL iterator.

	50,000 elements				250,000 elements			
	ArrayList	LinkedList	HashMap	TreeMap	ArrayList	LinkedList	HashMap	TreeMap
JMatch (foreach)	137.0	55.9	4.2	3.5	135.0	56.1	3.7	3.1
JMatch (first class)	23.2	23.8	3.6	2.5	22.9	23.7	3.2	2.4
C++ STL	339.0	57.4	3.7	4.4	215.0	57.7	3.1	3.9
C++ (stack)	—	—	—	4.4	—	—	—	3.9
Java Collections	10.6	10.1	3.9	3.9	6.3	10.3	4.2	3.5

Table 2. Iterator performance over collections of 50,000 and 250,000 elements. Numbers reported are in millions of elements iterated per second. Each number is the average of eight measurements with a standard deviation of at most 5%.

Benchmarks

The Java Collections Framework consists of four core classes: `ArrayList`, `LinkedList`, `HashMap`, and `TreeMap`. The iterators for these classes were reimplemented in JMatch and used to compare iteration performance.

The code for the `TreeMap` iterator is similar to that listed in Section 2.4. This was compared to the STL `map` iterator, which iterates over a red-black tree. The STL iterator code exploits parent pointers stored in every node. For completeness, the `TreeMap` iterator was also compared against a second C++ tree implementation, which uses an internal stack of tree nodes to track iteration state.

The `ArrayList` and `LinkedList` iterators were compared to the STL `vector` and `list` iterators, respectively.

The `HashMap` iterator uses both the `ArrayList` and `LinkedList` iterators. This was compared to the `hash_map` iterator, found in the Sun extensions to the STL and included in Linux libc++ 3.4.3.

Each benchmark program timed the execution of several runs of each data iterator over collections of 50,000 and 250,000 elements.

The benchmarks were compiled using `gcc` 3.4.3 and executed on a Fedora Core 3 system. The STL iterators were compiled with the `-O2` flag to ensure a good baseline for comparing performance.

The code generated by the JMatch translation to C++ manipulates the program stack in a non-standard way. As a result, some of the optimizations made by `gcc` break the benchmark code; and it was not possible to use the full suite of gcc optimizations on the output from JMatch.

The Java iterator benchmarks were run using the Java 1.5 Hotspot JIT compiler.

Performance Results

The results in Table 2 demonstrate that JMatch iterators can be translated to code that performs comparably to conventional iteration techniques. JMatch iterators perform almost as well as STL iterators in all cases (faster in the case of the HashMap test), with the exception of `ArrayList`.

The discrepancy in performance between the JMatch `TreeMap` iterator and that of the STL appears related to the inability to inline the (recursive) JMatch iterator. By contrast, the JMatch `HashMap` iterator performs about 15% faster than the STL iterator; this is due to implementation choices made in the STL iterator's `advance` function. Finally, the JMatch `ArrayList` iterator is significantly slower than that of the STL (though still much faster than Java) because the STL iterator is just an element pointer directly manipulated by the client code.

First-class iterator objects in JMatch are somewhat slower because iterator state is checkpointed, as discussed in Section 6.4, and because iterator code can no longer be inlined. However, the performance is probably still acceptable.

To determine the effectiveness of the tail-yield optimization, the `TreeMap` implementation was translated to C++ with and without the tail-yield optimization. The speedup from the tail-yield optimization was a factor of four for both 50k and 250k elements.

8. Related work

CLU [L$^+$81], ICON [GHK81], Python [vR03], Ruby [DTH04], and C# 2.0 [HWG03] each support coroutine iterators whose use and implementation are convenient; the `yield` statement of JMatch was inspired by CLU. None of these iterator mechanisms, however, support imperative updates.

Sather [MOSS96] extends the CLU iterator mechanisms to provide limited support for imperative update, through "hot arguments" that can be used to transmit information to an iterator in progress. This mechanism does not support invocation of update requests (e.g., `remove`) during a loop body. We are unaware of any evaluation of the effectiveness of the Sather mechanism for complex data structure manipulations during iteration.

Simula [Kir89], Modula-2 [Wir85] and BETA [MMPN93] support coroutines that can be used to implement iteration abstractions.

Alphard [SWL77] supports iteration through *generators*, which are essentially iterator (cursor) objects, and similarly difficult to implement.

Cedar [Lam83] supports both termination- and resumption-style exceptions, though not in the context of coroutines, and without static checking.

Various languages have used logic programming to enable the declarative implementation of a computation that produces multiple possible results; the best-known is Prolog [WPP77]. Mercury is a good modern example; as in JMatch, predicates can have several modes, a feature originating in some versions of Prolog (e.g., [Gre87]). Although logic programming languages give the ability to concisely express computations with multiple results, in these languages iteration occurs only at top level; there is no way to use the iteration within a larger program.

Other languages such as SML/NJ [SML], Scheme [KCe98], and Ruby [DTH04] support first-class continuations, which can be used to implement interruptible iterators and coroutines more generally [HFW86].

9. Conclusions

Iteration abstraction is important and yet poorly supported by the commonly used imperative object-oriented languages. Coroutine iterators are concise, readable and can be implemented efficiently. However, they do not support the full functionality desired by programmers, as evidenced by the Java `Iterator` interface. The problem of how to properly support iteration abstraction in imperative languages has remained unsolved.

This work shows that imperative update operations such as the Java `remove` method can be accommodated in a coroutine iterator framework, through *interrupts*, a new non-local control mechanism. Unlike Java iterators, the resulting iterators are compositional. The same interrupt mechanism can be applied to iterators implemented as declarative logical formulas, making code more concise through modal abstraction.

Using JMatch to implement the most complex classes in the Java Collections Framework has shown that interruptible iterators lead to substantial improvements in code size and, arguably, clarity. Measurements show that interruptible iterators have performance comparable to other iterator techniques, because the implementation can exploit the LIFO allocation discipline of coroutine stacks.

The static and dynamic semantics of the current version of JMatch have been formalized; the static checking of interrupts is described in the appendix, and the evaluation of JMatch programs has been described formally as a translation to Java [LM02]. These results suggest that interruptible iterators offer a reasonable way to solve the problem of implementing iterator abstractions.

A. Interrupt and trap checking

The following typing rules describe how interrupt checking is done in JMatch. The goal is to ensure that every raised interrupt or thrown trap exception is eventually handled, and that control is returned properly to the client through `resume`.

A.1 Notation

There are two judgments, for expressions and statements:

- Expressions: $\vdash_e \cdot : I, E$
- Statements: $\vdash_s \cdot : I, E, U$

I is the set of interrupts handled. This can be $\bot$ to indicate that there is no yield point in the given expression or statement (and hence no need to handle interrupts):

$$\bot \cup S = \bot \cap S = S \setminus \bot = S$$

$E = \{(e_j, i_j)\}$ is a conservative overapproximation of the set of exceptions thrown, each tagged with the interrupt that caused it. The interrupt tag i_j can be none if the exception is an ordinary one, or unk if the cause of the exception is as yet undetermined. U is the set of interrupts that have been `raise`d in an enclosed lexical context, but not yet handled.

$E|_S = \{(e, i) : (e, i) \in E \text{ and } i \in S\}$ filters a tagged exception set E against a set S of interrupts.

$E_{[\tau \mapsto \tau']} = \bigcup_{(e,\tau)\in E}\{(e, \tau')\} \cup \bigcup_{(e,i)\in E : i \neq \tau}\{(e, i)\}$ updates interrupt tags in E, remapping the tag τ to τ'.

Set subtraction is extended to include subtypes:

$$E \setminus E' = \{(\tau_1, \tau_2) \in E \mid \nexists (\tau_1', \tau_2') \in E'.\ \tau_1 \leq \tau_1' \text{ and } \tau_2 \leq \tau_2'\}$$

$$U \setminus U' = \{\tau \in U \mid \nexists \tau' \in U'.\ \tau \leq \tau'\}$$

A.2 Method declarations

To ensure that every raised interrupt or thrown trap exception is eventually handled, we verify that each iterative method mode adheres to the method signature's interrupt and trap exception declarations. In a method that declares $\overrightarrow{\texttt{traps}\ \tau_i :\ \tau_i'}$, each iterative mode must satisfy:

- $\vdash_e e : \{\overrightarrow{\tau_i}\}, \{\overrightarrow{\tau_i', \tau_i}\}$, where the mode is implemented with the formula e; or else
- $\vdash_s s : \{\overrightarrow{\tau_i}\}, \{\overrightarrow{\tau_i', \tau_i}\}, \emptyset$, where the mode is implemented with the statement s.

A.3 Formulas

$$\vdash_e \texttt{false} : \bot, \emptyset$$

Conjunction:

$$\frac{\vdash_e e_i : \bot, \emptyset \qquad \vdash_e e_j : I, E}{\vdash_e e_1 \,\&\&\, e_2 : \bot, \emptyset} \qquad \{i, j\} = \{1, 2\}$$

$$\frac{\vdash_e e_1 : I_1, E_1 \qquad \vdash_e e_2 : I_2, E_2}{\vdash_e e_1 \,\&\&\, e_2 : I_1 \cup I_2, E_1 \cup E_2} \qquad I_1 \neq \bot; I_2 \neq \bot; I_1, I_2 \text{ disjoint}$$

Binary comparisons, assignments, and binary patterns:

$$\frac{\vdash_e e_1 : I_1, E_1 \qquad \vdash_e e_2 : I_2, E_2}{\vdash_e e_1\ \texttt{op}\ e_2 : I_1 \cup I_2, E_1 \cup E_2} \qquad I_1, I_2 \text{ disjoint}$$

where: $\texttt{op} \in \{+, -, *, /, \%, \&, |, \hat{}, =, ==, !=, <, <=, >=, >, \texttt{as}\}$

Disjunction:

$$\frac{\vdash_e e_1 : I_1, E_1 \qquad \vdash_e e_2 : I_2, E_2}{\vdash_e e_1\ \texttt{op}\ e_2 : I_1 \cap I_2, (E_1 \cup E_2)|_{(I_1 \cap I_2) \cup \{\mathsf{none}, \mathsf{unk}\}}} \quad \texttt{op} \in \{||, \texttt{else}\}$$

Constructor invocation:

$$\frac{\vdash_e e_i : I_i, E_i}{\vdash_e \texttt{new}\ \tau(\overrightarrow{e_i}) : \bigcup_i I_i, (\bigcup_{\tau' \in E} (\tau', \mathsf{unk})) \cup (\bigcup_i E_i)} \qquad \begin{array}{l}\tau(\overrightarrow{\tau_{e_i}})\ \texttt{throws}\ E \\ I_i \text{ disjoint}\end{array}$$

Method invocation:

$$\frac{\vdash_e e_i : I_i, E_i}{\vdash_e m(\overrightarrow{e_i}) : \{\overrightarrow{\tau_j}\} \cup (\bigcup_i I_i), \mathcal{E}}$$

where:

$\mathcal{E} = (\bigcup_{\tau \in E}(\tau, \mathsf{unk})) \cup (\bigcup_i E_i) \cup (\bigcup_{j, \tau \in E_j}\{(\tau, \tau_j)\})$
iterative mode chosen for m
$m\ \texttt{throws}\ E\ \texttt{traps}\ \overrightarrow{\tau_j : E_j}$
I_i and $\{\overrightarrow{\tau_j}\}$ disjoint

All other expressions e: $\qquad \vdash_e e : \emptyset, \emptyset$

A.4 Statements

$$\vdash_s \texttt{yield} : \emptyset, \emptyset, \emptyset \qquad\qquad \frac{\vdash_e e : \bot, E}{\vdash_s \texttt{raise}\ e : \bot, E, \{\tau_e\}}$$

$$\frac{\vdash_e e : \bot, E}{\vdash_s \texttt{throw}\ e : \bot, E \cup \{(\tau_e, \mathsf{unk})\}, \emptyset}$$

$$\frac{\vdash_e e : I_e, E_e \qquad \vdash_s s : I_s, E_s, U_s}{\begin{array}{l}\vdash_s \texttt{foreach}\ (e)\ s : \\ \quad I_e \cup I_s, E_s \cup (\bigcup_{(\tau, \tau') \in E_e, \tau' \in U_s \cup \{\mathsf{unk}, \mathsf{none}\}} \{(\tau, \mathsf{unk})\}), U_s \setminus I_e\end{array}}$$

Statement sequencing:

$$\frac{\vdash_s s_1 : I_1, E_1, U_1 \qquad \vdash_s s_2 : I_2, E_2, U_2}{\vdash_s s_1; s_2 : I_1 \cap I_2, (E_1 \cup E_2)|_{(I_1 \cap I_2) \cup \{\mathsf{none}, \mathsf{unk}\}}, U_1 \cup U_2}$$

Try:

$$\frac{\vdash_s s : \bot, E, U \qquad \vdash_s s_i : I_i, E_i, U_i}{\vdash_s \texttt{try}\ \{s\}\ \overrightarrow{\texttt{catch}\ (\tau_i)\ \{s_i\}} : \bigcap_i I_i, \mathcal{E}, U \cup (\bigcup_i U_i)}$$

where:

$$\mathcal{E} = E \setminus \{\overrightarrow{(\tau_i, \mathsf{unk})}, \overrightarrow{(\tau_i, \mathsf{none})}\} \cup (\bigcup_i E_i)|_{(\bigcap_i I_i) \cup \{\mathsf{none}, \mathsf{unk}\}}$$

$$\frac{\begin{array}{c}\vdash_s s : I, E, U \\ \vdash_s s_i : I_i, E_i, U_i \qquad \vdash_s s_j : I_j, E_j, U_j \qquad \vdash_s s_k : I_k, E_k, U_k\end{array}}{\begin{array}{c}\vdash_s \texttt{try}\ \{s\}\ \overrightarrow{\texttt{trap}\ (\tau_i)\ \{s_i\}}\ \overrightarrow{\texttt{catch}\ (\texttt{trap}\ \tau_j)\ \{s_j\}} \\ \overrightarrow{\texttt{catch}\ (\tau_k)\ \{s_k\}} : \mathcal{I}, \mathcal{E}, \mathcal{U}\end{array}}$$

where:

$$\begin{array}{rcl}
\mathcal{I} & = & (I \cup \{\overrightarrow{\tau_i}\}) \cap (\bigcap_i I_i) \cap (\bigcap_j I_j) \cap (\bigcap_k I_k) \\
\mathcal{E} & = & ((E \setminus \{\overrightarrow{(\tau_j, \texttt{Object})}, \overrightarrow{(\tau_k, \mathsf{unk})}, \overrightarrow{(\tau_k, \mathsf{none})}\}) \\
 & \cup & (\bigcup_i E_{i[\mathsf{unk} \mapsto \tau_i]}) \\
 & \cup & (\bigcup_{(\tau, \mathsf{unk}) \in E_j, (\tau_j, \tau') \in E}\{(\tau, \tau')\}) \\
 & \cup & (\bigcup_{(\tau, \tau') \in E_j, \tau' \neq \mathsf{unk}}\{(\tau, \tau')\}) \cup (\bigcup_k E_k))|_{\mathcal{I}} \\
\mathcal{U} & = & U \cup (\bigcup_i U_i) \cup (\bigcup_j U_j) \cup (\bigcup_k U_k)
\end{array}$$

Resume: The rules for `resume break` and `resume continue` are identical to the following rules for `resume yield`.

$$\vdash_s \texttt{resume}\,\texttt{yield} : \bot, \emptyset, \emptyset$$

$$\frac{\vdash_s s_i : I_i, E_i, U_i}{\begin{array}{l}\vdash_s \texttt{resume}\,\texttt{yield}\,\overrightarrow{\texttt{trap}\,(\tau_i)\,\{s_i\}} :\\ \quad \{\overrightarrow{\tau_i}\} \cap (\bigcap_i I_i), (\bigcup_i (E_{i\,[\mathsf{unk} \mapsto \tau_i]}))|_{\{\overrightarrow{\tau_i}\} \cap (\bigcap_i I_i)}, \bigcup_i U_i\end{array}}$$

All other statements s: $\quad \vdash_s s : \bot, \emptyset, \emptyset$

Acknowledgments

Steve Chong, Michael Clarkson, Nate Nystrom and Lantian Zheng gave many useful comments on the presentation.

This research was supported in part by ONR Grant N00014-01-1-0968, by NSF Grants 0208642, 0133302, and 0430161, 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 funding agencies. Aaron Kimball was supported by an Engineering Learning Initiatives grant funded by Intel.

References

[DTH04] Chad Fowler Dave Thomas and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. The Pragmatic Programmers, 2nd edition, 2004. ISBN 0-974-51405-5.

[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.

[Goo75] John B. Goodenough. Exception handling design issues. *SIGPLAN Notices*, 10(7):41–45, 1975.

[GR04] Emden R. Gansner and John H. Reppy. *The Standard ML Basis Library*. Cambridge University Press, October 2004.

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

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

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley, 1st edition, October 2003. ISBN 0321154916.

[KCe98] Richard Kelsey, William Clinger, and Jonathan Rees (editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, October 1998.

[Kir89] B. Kirkerud. *Object-Oriented Programming with SIMULA*. Addison-Wesley, 1989.

[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.

[Lam83] Butler Lampson. A description of the Cedar language: A Cedar language reference manual. Technical Report CSL-83-15, Xerox PARC, December 1983.

[LAS78] B. Liskov, R. Atkinson, and R. Scheifler. Aspects of implementing CLU. In *Proceedings of the Annual Conference*. ACM, 1978.

[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`.

[LM03] Jed Liu and Andrew C. Myers. JMatch: Abstract iterable pattern matching for Java. In *Proc. 5th Int'l Symp. on Practical Aspects of Declarative Languages (PADL)*, pages 110–127, New Orleans, LA, January 2003.

[LSAS77] B. Liskov, A. Snyder, R. Atkinson, and J. C. Schaffert. Abstraction mechanisms in CLU. *Comm. of the ACM*, 20(8):564–576, August 1977. Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems*.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 132–145, Paris, France, January 1997.

[MDS01] David R. Musser, Gillmer J. Derge, and Atul Saini. *The STL Tutorial and Reference Guide*. Addison-Wesley, 2nd edition, 2001. ISBN 0-201-37923-6.

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

[MLM98] Andrew C. Myers, Barbara Liskov, and Nicholas Mathewson. PolyJ: Parameterized types for Java. Software release, at `http://www.cs.cornell.edu/polyj`, July 1998.

[MMPN93] O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[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.

[NCM03] Nathaniel Nystrom, Michael Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, number 2622 in Lecture Notes in Computer Science, pages 138–152, Warsaw, Poland, April 2003. Springer-Verlag.

[SML] The SML/NJ Fellowship. *Standard ML of New Jersey*. http://www.smlnj.org/.

[Str87] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[Sun04] Sun Microsystems. *JDK 5 Java Language Documentation*, 2004. http://java.sun.com/j2se/1.5.0/docs/guide/language.

[SWL77] M. Shaw, W. Wulf, and R. London. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Comm. of the ACM*, 20(8), August 1977.

[vR03] Guido van Rossum. *The Python Language Reference Manual*. Network Theory, Ltd., September 2003.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, Berlin, 3rd edition, 1985.

[WPP77] David H. D. Warren, Luis M. Pereira, and Fernando Pereira. Prolog—the language and its implementation compared with Lisp. In *Proc. 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 109–115, 1977.