While searching for something else, I happened to stumble across an old proposal for JVM tail calls from 1998 (updated 2002)! Since Cameron apparently mentioned them at the JVM summit, I couldn't resist sending this around for your amusement.

--Guy

Proposal for JVM tail call support
Guy Steele, May 15, 1998
updated January 3, 2002
updated January 8, 2002
Summary: phase in support for tail calls in a manner that is compatible with existing JVM implementations (except that such implementations might run out of storage when executing code that relies heavily on tail calls).
This is done by adding a "TailCall" attribute to methods in class files. A "goto" statement is added to the Java programming language to allow tail calls to be expressed in source code.

(1) At the JVM level:
Introduce a new attribute of the method_info structure: the "TailCall" attribute. The info of this attribute is a list of pc locations within the Code attribute for the same method. Each pc location must be the location of an invokeinterface, invokespecial, invokestatic, or invokevirtual instruction; moreover, that instruction must be immediately followed by a return instruction. If these constraints are not met, the class file is considered malformed and may be rejected by a JVM.
An invokeinterface, invokespecial, invokestatic, or invokevirtual instruction that is identified by the TailCall attribute is intended to be executed as a tail call if possible. This is done by first following all the usual rules for locating the method to be invoked, and then (in effect) popping the arguments (and perhaps an objectref) from the stack. At this point, a test is made to see whether a tail call is permitted (see below). If the test succeeds, then the current stack frame is removed from the stack as if by an appropriate return instruction; this also restores the PC to a point within the "caller" of the current method. Then, whether the test succeeded or failed, the rest of the invoke operation is carried out by creating a new stack frame, installing the arguments (and perhaps an objectref) as local variables, etc.
Here is the test mentioned in the preceding paragraph:
(1) The test is permitted to fail or to succeed, at the discretion of the implementation, if the called method is native. It is the responsibility of the implementation to ensure that security policies are not compromised if a tail call is used.
(2) Otherwise, the test is permitted to fail or to succeed, at the discretion of the implementation, if the class loader of either the calling method or the called method is the system class loader (null). It is the responsibility of the implementation to ensure that security policies are not compromised if a tail call is used.
(3) Otherwise, the test definitely fails if the class loader of the calling method and the class loader of the called method are different and neither is the system class loader (null).
(4) Otherwise, the test definitely fails if the protection domain of the calling method and the protection domain of the called method are different.
(5) Otherwise, the test definitely succeeds.
Thus, an application programmer can rely on a tail call consuming no net stack space if the JVM supports tail calls and the called method has the same class loader and protection domain as the calling method. An important special case is that tail calls always "work" if the JVM supports tail calls and the called method is defined in the same class as the calling method.
It is permitted, though discouraged, for a JVM to ignore the TailCall attribute. In this case, code will be executed in a semantically equivalent manner but may fail to complete execution if it runs out of storage for stack frames. (This is permitted primarily to allow some measure of compatibility with existing JVM implementations. The intent, however, is to phase in implementations that all properly
support tail calls so that, after a certain point in time, programmers will be able to rely on it as a programming mechanism.

(This is exactly analogous to the observation that it is permitted, but discouraged, for a JVM not to have a garbage collector; it could just allocate new objects out of a finite pool of storage and then give up when the pool is exhausted. But that's not considered a quality implementation.)

(2) At the JLS level:
Introduce a new statement:

```java
goto <MethodInvocation> ;
```

This may appear in exactly the places that a return statement may appear.

(Perhaps, out of an abundance of caution, we should not allow it to be used within a constructor.)

If the type of the `<MethodInvocation>` expression is void, then this statement may appear within the body of a method whose return type is void [or within the body of a constructor?]. This statement is then semantically equivalent to

```java
{ <MethodInvocation>; return; }
```

If the type of the `<MethodInvocation>` expression is not void, then this statement may appear within the body of a method whose return type is the same as the type of the `<MethodInvocation>` expression. This statement is then semantically equivalent to

```java
return <MethodInvocation> ;
```

In each case, however, the "goto" form additionally specifies a requirement, or at least the strong desire, that the implementation should recycle the stack frame of the currently running method as it performs the invocation of the called method. The intent is that it should be possible to execute an indefinitely large number of "goto" calls without running out of space by reason of stack overflow.

However, the programmer cannot expect the stack frame to be recycled in this manner if the call might fail the test (mentioned above in the JVM section).

(3) Small example: Searching a polymorphic list

Suppose there is a list with two or more kinds of nodes, and we want to ask whether there is a node in the list that matches a given string, and if so, return an associated value. The "obvious" way to do it is as follows:

```java
interface Chain { Value lookup(String name); }
class SimplePair implements Chain {
    String name;
    Value val;
    Chain next;
    ...
    Value lookup(String query) {
        if (name.equals(query)) return value;
        if (next == null) return null;
        return next.lookup(query);                 ;not me---pass the buck
    }
}
class ComputedValue implements Chain {
    String oldprefix;
    String newprefix;
    Chain next;
    ...
    Value lookup(String query) {
        if (query.startsWith(prefix))
            return new Value(newprefix + query.substring(prefix.length()));
        if (next == null) return null;
        return next.lookup(query);                ;not me---pass the buck
    }
}
```

Unfortunately, in current JVM implementations, this chews up a stack frame for every node in the chain. As a result, implementors often try to use an explicit while loop, which results in tearing apart the code for the lookup method in complicated ways.

With tail calls, just replace

```java
return next.lookup(query);
```

with
and both the algorithmic intent and the intended stack behavior are quite clear.

(4) Big example: A tiny Lisp evaluator

Here is an interpreter for a tiny lambda-calculus-based language that consists of numeric (BigInteger) literals, variables, an if-then-else expression, lambda expressions of one parameter, a rec (label) construct for naming a recursive function, and function calls with one argument. It provides add and multiply operators in curried form, so that one must say `((+ 3) 4)`, for example. There is also an explicit zero test primitive that returns true or false. (Internally, the number 0 is used as the false value for if-then-else.) Programs are of type Expression; the eval method produces a Value.

```java
interface Expression {
    Value eval(Environment env);
}

class LiteralNode implements Expression {
    final Value item;
    LiteralNode(Value item) { this.item = item; }
    public Value eval(Environment env) { return item; }
}

class VariableNode implements Expression {
    final String name;
    VariableNode(String name) { this.name = name; }
    public Value eval(Environment env) { return env.lookup(name); }
}

class IfNode implements Expression {
    final Expression test, thenpart, elsepart;
    IfNode(Expression test, Expression thenpart, Expression elsepart) {
        this.test = test; this.thenpart = thenpart; this.elsepart = elsepart;
    }
    public Value eval(Environment env) {
        if (!(test.eval(env).isZero()))
            goto thenpart.eval(env);
        else
            goto elsepart.eval(env);
    }
}

class LambdaNode implements Expression {
    final String name;
    final Expression body;
    LambdaNode(String name, Expression body) { this.name = name; this.body = body; }
    public Value eval(Environment env) { return new Closure(name, body, env); }
}

class RecNode implements Expression {
    final String name;
    final Expression body;
    RecNode(String name, Expression body) { this.name = name; this.body = body; }
    public Value eval(Environment env) {
        Environment newenv = env.push(name, null);
        Value item = body.eval(newenv);
        newenv.clobber(item);
        return item;
    }
}

class CallNode implements Expression {
    final Expression fn, arg;
    CallNode(Expression fn, Expression arg) { this.fn = fn; this.arg = arg; }
    public Value eval(Environment env) { goto fn.eval(env).invoke(arg.eval(env)); }
}

interface Value {
    Value invoke(Value arg);
    boolean isZero();
}

class IntVal implements Value {
    final java.math.BigInteger v;
    IntVal(java.math.BigInteger v) { this.v = v; }
    IntVal(int v) { this.v = java.math.BigInteger.valueOf(v); }
    static java.math.BigInteger zero = java.math.BigInteger.valueOf(0);
}
```
static java.math.BigInteger one = java.math.BigInteger.valueOf(1);
public boolean isZero() { return v.equals(zero); }
Value zeroTest() { return new IntVal(isZero() ? one : zero); }
Value add(Value that) { return new IntVal(this.v.add(((IntVal)that).v)) ; }
Value multiply(Value that) { return new IntVal(this.v.multiply(((IntVal)that).v)) ; }
public Value invoke(Value arg) { throw new Error("Can't invoke an integer"); }
public String toString() { return v.toString(); }
}
abstract class NonZeroValue implements Value { public boolean isZero() { return false; } }
class Closure extends NonZeroValue {
    final String name;
    final Expression body;
    final Environment env;
    Closure(String name, Expression body, Environment env) {
        this.name = name; this.body = body; this.env = env;
    }
    public Value invoke(Value arg) { goto body.eval(env.push(name, arg)); }
}
class ZeroTestPrimitive extends NonZeroValue {
    public Value invoke(Value arg) { return ((IntVal)arg).zeroTest(); }
}
class AddPrimitive extends NonZeroValue {
    public Value invoke(Value arg) { return new CurriedAdd(arg); }
}
class CurriedAdd extends NonZeroValue {
    final Value arg1;
    CurriedAdd(Value arg1) { this.arg1 = arg1; }
    public Value invoke(Value arg2) {
        return ((IntVal)arg1).add(arg2);
    }
}
class MultPrimitive extends NonZeroValue {
    public Value invoke(Value arg) { return new CurriedMult(arg); }
}
class CurriedMult extends NonZeroValue {
    final Value arg1;
    CurriedMult(Value arg1) { this.arg1 = arg1; }
    public Value invoke(Value arg2) {
        return ((IntVal)arg1).multiply(arg2);
    }
}
class Environment {
    final String name;
    Value item;
    final Environment next;
    static Environment empty = new Environment(null, null, null);
    private Environment(String name, Value item, Environment next) {
        this.name = name; this.item = item; this.next = next;
    }
    Environment push(String name, Value item) {
        return new Environment(name, item, this);
    }
    void clobber(Value item) { this.item = item; }
    Value lookup(String query) {
        if (this == empty) throw new Error("Name " + query + " not found");
        if (name.equals(query)) return item;
        goto next.lookup(query);
    }
}
/*
Here is test code that executes the expression
((rec fact (lambda (n) (if (= n) 1 (* n (fact (+ n -1)))))) j)
to compute the factorial function for various values of j.
*/
public class Foo {
    static Expression demo(int j) {
        return new CallNode(
            new RecNode("fact",
                new LambdaNode("n",
                    new IfNode(
                        new CallNode(...),
                        new IntVal(1),
                        new CallNode(...))));
    }
}
new LiteralNode(new ZeroTestPrimitive()),
    new VariableNode("n")
),
new LiteralNode(new IntVal(1)),
new CallNode(
    new CallNode(
        new LiteralNode(new MultPrimitive()),
        new VariableNode("n")
    ),
    new CallNode(
        new VariableNode("fact"),
        new CallNode(
            new CallNode(
                new LiteralNode(new AddPrimitive()),
                new VariableNode("n")
            ),
            new LiteralNode(new IntVal(-1))
        )
    )
),
new LiteralNode(new IntVal(-1));
}

public static void main(String args[])
    
    System.out.println(demo(0).eval(Environment.empty).toString());
    System.out.println(demo(5).eval(Environment.empty).toString());
    System.out.println(demo(8).eval(Environment.empty).toString());
    System.out.println(demo(1000).eval(Environment.empty).toString());
    System.out.println(demo(1000000).eval(Environment.empty).toString());

    Under the existing JVM, I tested the code exactly as shown except that I replaced the keyword "goto" with the keyword "return". Here is the output:

    livia 69 =>java Foo
    1
    24
    120
    40320
    402387260077093773547032439320398571937486421074163254799910429938512398629209520442084869694048004799886101
    9719058631666872994808559901323829696445909974245040870737599188236277218873521977959059599527612087947596429
    7043601418527890464469291056393887437886487373191981045825783647849977012476632888935957534532151382546386075
    5574091142641477434947553428646576661166779379666880299212073791438531795882498081266878837475597317461360853795
    34524221586593201928090879229370834139284440281231558610367968013537042161687476907587134831202545893892076791
    91324484262361314512087802000026168315102734182797707478463586817016436502415369139821268410210397671261448966359
    9287051149649541990934222156683257208082133318161185361583654984046708975602909053761675847724218896976
    42644954156076535408198901385442487984959953391917233555566021394503997362807501378376153017271264920864903435262
    5200015888553147331611701201039617592151090778001939317811419454525722386554146106289218796022383971476088506276
    862961714667469756291123408243920816015378088993996451826324367166762179168097899190375403127462289980051954
    44414282021873617459926249558174662830295557029902432415318161721204565320367869611762015878352075151628242554
    026517048330242261439742869330616098979684825902125458327168262458606562687969586562682278070757813985187888652028
    164348344825932660433760176996996283186078883861650279465955131156552031609399818061218558600301435694527224063
    44631779460594862571307908402443243846565724501404828189525247093519062092292313649327497556519358792055965422
    87497740114343696627154228458623773875382304838656889764619273838149001407673104466402598994902222176590439901
    886018566562485065179902356193970178600408118879299813102171122984590164192106888437812185566124960798722085
    1929681937286442614839657382291123520524186649353413497013742853192649875337218940694281431481520581508141233482
    80150533996492901534830776445696099073152423237822882698460278986432113908350621709500257938865542771967422248
    757586767532443220207573630569498825087968921627538486399699958268209561245099478107120414564166263079293901
    208890869420285106401821543994571568059418727489980945274472135824010636774045957471878516085223013535808814009699
    63725242305605859037006242712434169909004153690105933983857799394109700277534720000000000000000000000000000000000000000
    0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    Exception in thread "main" java.lang.StackOverflowError: c stack overflow

    livia 70 =>

    Note that it correctly computed all the factorial values but the last,
and ran out of stack trying to compute the last one---and rightly so, ha ha, for this computation is not tail-recursive! But now consider this test code, which performs a factorial computation tail-recursively:

```java
public class Bar {
    public static void main(String[] args) {
        System.out.println(demo(0).eval(Environment.empty).toString());
        System.out.println(demo(5).eval(Environment.empty).toString());
        System.out.println(demo(8).eval(Environment.empty).toString());
        System.out.println(demo(1000).eval(Environment.empty).toString());
        System.out.println(demo(1000000).eval(Environment.empty).toString());
    }
    public static Expression demo(int j) {
        return new CallNode(
            new RecNode("fact",
                new LambdaNode("n"),
                new LambdaNode("a"),
                new IfNode(
                    new CallNode(
                        new CallNode(
                            new CallNode(
                                new LiteralNode(new EqualsPrimitive()),
                                new VariableNode("n"))
                        ,
                        new LiteralNode(new IntVal(0))
                    ),
                    new VariableNode("a"),
                    new CallNode(
                        new CallNode(
                            new CallNode(
                                new VariableNode("fact"),
                                new CallNode(
                                    new CallNode(
                                        new LiteralNode(new AddPrimitive()),
                                        new VariableNode("n"))
                                ,
                                new LiteralNode(new IntVal(-1))
                            ),
                            new CallNode(
                                new CallNode(
                                    new LiteralNode(new MultPrimitive()),
                                    new VariableNode("n"))
                            ),
                            new VariableNode("a")
                        )
                    )
                ));
            new LiteralNode(new IntVal(j)),
            new LiteralNode(new IntVal(1)));
        System.out.println(demo(0).eval(Environment.empty).toString());
        System.out.println(demo(5).eval(Environment.empty).toString());
        System.out.println(demo(8).eval(Environment.empty).toString());
        System.out.println(demo(1000).eval(Environment.empty).toString());
        System.out.println(demo(1000000).eval(Environment.empty).toString());
    }
    static Expression demo(int j) {
        return new CallNode(
            new RecNode("fact",
                new LambdaNode("n"),
                new LambdaNode("a"),
                new IfNode(
                    new CallNode(
                        new CallNode(
                            new CallNode(
                                new LiteralNode(new EqualsPrimitive()),
                                new VariableNode("n"))
                        ,
                        new LiteralNode(new IntVal(0))
                    ),
                    new VariableNode("a"),
                    new CallNode(
                        new CallNode(
                            new CallNode(
                                new VariableNode("fact"),
                                new CallNode(
                                    new CallNode(
                                        new LiteralNode(new AddPrimitive()),
                                        new VariableNode("n"))
                                ,
                                new LiteralNode(new IntVal(-1))
                            ),
                            new CallNode(
                                new CallNode(
                                    new LiteralNode(new MultPrimitive()),
                                    new VariableNode("n"))
                            ),
                            new VariableNode("a")
                        )
                    )
                ));
            new LiteralNode(new IntVal(j)),
            new LiteralNode(new IntVal(1)));
        return new CallNode(
            new RecNode("fact",
                new LambdaNode("n"),
                new LambdaNode("a"),
                new IfNode(
                    new CallNode(
                        new CallNode(
                            new CallNode(
                                new LiteralNode(new EqualsPrimitive()),
                                new VariableNode("n"))
                        ,
                        new LiteralNode(new IntVal(0))
                    ),
                    new VariableNode("a"),
                    new CallNode(
                        new CallNode(
                            new CallNode(
                                new VariableNode("fact"),
                                new CallNode(
                                    new CallNode(
                                        new LiteralNode(new AddPrimitive()),
                                        new VariableNode("n"))
                                ,
                                new LiteralNode(new IntVal(-1))
                            ),
                            new CallNode(
                                new CallNode(
                                    new LiteralNode(new MultPrimitive()),
                                    new VariableNode("n"))
                            ),
                            new VariableNode("a")
                        )
                    )
                ));
            new LiteralNode(new IntVal(j)),
            new LiteralNode(new IntVal(1)));
    }
    public static void main(String[] args) {
        System.out.println(demo(0).eval(Environment.empty).toString());
        System.out.println(demo(5).eval(Environment.empty).toString());
        System.out.println(demo(8).eval(Environment.empty).toString());
        System.out.println(demo(1000).eval(Environment.empty).toString());
        System.out.println(demo(1000000).eval(Environment.empty).toString());
    }
    static Expression demo(int j) {
        return new CallNode(
            new RecNode("fact",
                new LambdaNode("n"),
                new LambdaNode("a"),
                new IfNode(
                    new CallNode(
                        new CallNode(
                            new CallNode(
                                new LiteralNode(new EqualsPrimitive()),
                                new VariableNode("n"))
                        ,
                        new LiteralNode(new IntVal(0))
                    ),
                    new VariableNode("a"),
                    new CallNode(
                        new CallNode(
                            new CallNode(
                                new VariableNode("fact"),
                                new CallNode(
                                    new CallNode(
                                        new LiteralNode(new AddPrimitive()),
                                        new VariableNode("n"))
                                ,
                                new LiteralNode(new IntVal(-1))
                            ),
                            new CallNode(
                                new CallNode(
                                    new LiteralNode(new MultPrimitive()),
                                    new VariableNode("n"))
                            ),
                            new VariableNode("a")
                        )
                    )
                ));
            new LiteralNode(new IntVal(j)),
            new LiteralNode(new IntVal(1)));
        return new CallNode(
            new RecNode("fact",
                new LambdaNode("n"),
                new LambdaNode("a"),
                new IfNode(
                    new CallNode(
                        new CallNode(
                            new CallNode(
                                new LiteralNode(new EqualsPrimitive()),
                                new VariableNode("n"))
                        ,
                        new LiteralNode(new IntVal(0))
                    ),
                    new VariableNode("a"),
                    new CallNode(
                        new CallNode(
                            new CallNode(
                                new VariableNode("fact"),
                                new CallNode(
                                    new CallNode(
                                        new LiteralNode(new AddPrimitive()),
                                        new VariableNode("n"))
                                ,
                                new LiteralNode(new IntVal(-1))
                            ),
                            new CallNode(
                                new CallNode(
                                    new LiteralNode(new MultPrimitive()),
                                    new VariableNode("n"))
                            ),
                            new VariableNode("a")
                        )
                    )
                ));
            new LiteralNode(new IntVal(j)),
            new LiteralNode(new IntVal(1)));
    }
}
```

This, actually, uses up stack less quickly and so runs out of memory trying to compute really huge BigIntegers. So I changed the starting value 1 to 0, and changed * to +, so that it computes triangular numbers instead of factorials, and got this output:

```
livia 87 => java Bar
0
15
36
500500
Exception in thread "main" java.lang.StackOverflowError: c stack overflow
livia 88 =>
```
and I claim that that last stack overflow need not occur if only tail calls were supported in Java.

In the code for the evaluator, note how the statements

\[
\text{goto thenpart.eval(env);} \\
\text{and} \\
\text{goto elsepart.eval(env);} \\
\]

in the code for the eval method of IfNode make clear the intent to process the chosen part of an "if" expression tail-recursively. Similarly for the invocation of a function by a CallNode and the evaluation of a body by a Closure.