Java with Generators

Tony Young, Student ID: 5383914, University of Auckland

***Abstract*—This report details the implementation of *generators* in Java – a specialized case of continuations that allow the execution of code to be paused within a method body at specific *yield* points. It introduces *linearization* as the primary technique for achieving this, with additional supplementary algorithms to transform code to adhere to the semantics of Java.**

I. Introduction

Generators are a language feature found in various modern languages. They enable sequential code to be paused at specific *yield* points, making it useful for tasks such as iterating through a binary tree or creating lazy, infinite lists (à la Haskell) without having to explicitly keep track of where execution is.

This implementation of generators extends Java syntax in two ways:

- An extension to the Java method syntax. Before the method name, we allow an optional $\star$ token that signifies the method is a generator that will yield values of the type specified for the method.
- The introduction of a `yield` statement. It is allowed only in starred generator methods and takes an expression that will be returned a subsequent execution of the generator.

For simplicity's sake, generators in this implementation are unidirectional and only yield values out (such as in C#), unlike the generators in Python or ECMAScript Harmony can have values "sent" to them.

II. Transformation

The code is transformed in a multi-pass fashion – the bulk of transformation is done during the linearization process (section II-C), but some pre- and post-processing passes are employed to make linearization simpler.

A. Loop Desugaring

The first phase of generator transformation is loop desugaring. In this phase, we transform all loops into a *canonical form*, which is proposed to be the following:

```
for (;;) { /* loop body */ }
```

The canonical loop form represents an infinite loop – however, the loop body may contain `break` and `continue` statements to mimic sugared loop forms.

As such, automated conversion can be employed from other forms of loops (such as `while`, `for` and `do`). For a full list of conversions from canonical loop forms, refer to appendix A.

In the special case of a non-canonical `for` loop, we may need to move an outer label into the inner canonical `for` loop. This is facilitated by remembering the position of labels before entering their statement nodes as well as which statement node was entered then, during processing of the statement node, we can indicate that we have moved the label into a loop and that the label needs to be removed.

```
function ScopeMangle(node)
    id ← 0
    for all b ∈ node's blocks do
        for all d ∈ b's declarations do
            Prepend(id, d's variable name)
        end for
        for all e ∈ b's expressions do
            for all n ∈ e's referenced names do
                Prepend(n's block's id, n)
            end for
        end for
        id ← id + 1
    end for
end function
```

Figure 1. Scope mangling algorithm.

B. Scope Mangling

Java employs block-level scoping for variables – for example, variables declared inside an `if` block will not be visible upon exit of the `if` block. Due to linearization (section II-C), we cannot preserve blocks and, consequently, we cannot use variable names in the form they are suggested by the code, for example:

```
{
    int i = 0;
}
{
    float i = 0.0f;
}
```

`i` is typed as an `int` in the first scope, but `float` in the second. If the statements are placed into a single block from both the blocks, we will find that we have a variable name clash, as well as the variable `i` having its type changed.

As such, we mangle variable names in deeper scopes to ensure they don't conflict with variable names in higher scopes. Figure 1 details a basic algorithm for name mangling. We also perform partial semantic analysis of variable declaration at this point, ensuring that local variables are not redeclared. We maintain a stack of block scopes that represents the current block of code we're in, as well as all the ancestral blocks. The block scopes contain symbol tables that keep track of variable declarations in this scope – if we find that a variable has been redeclared either in this scope or any ancestral scope, this is a semantic error.

function MARKLINEARIZATION(node)
  if node $\in$ yields $\cup$ breaks $\cup$ continues $\cup$ returns **then**
    MARK(node)
    return True
  end if
  for all n $\in$ node's children **do**
    if MARKLINEARIZATION(n) **then**
      MARK(n)
    end if
  end for
  if nodes were marked **then**
    return True
  else
    return False
  end if
end function

Figure 2. Linearization marking algorithm.

In practice, the block ID is not a simple monotonically increasing number. For the implementation provided, the blocks are first numbered off by their offset to the start of the parent. The variables are then mangled using successive prepending of block numbers from the block the variable is declared in to the root of the method – for example, in the example given earlier, the three variables will be mangled to `s0$i`, `s1$i`. If a variable `j` was declared inside the second block inside a block, then it will be mangled to `s1$s0$j`.

C. Linearization

The linearization (pertaining to loop linearization) pass processes the control flow graph and transforms it into a state machine. This is required because we can then create states that represents the pausing of the state machine, at given yield points. Additionally, linearized control structures enable us to perform arbitrary jumps into the code – such as jumps into code after the yield point to simulate resumes.

Before linearization, we employ a node marking scheme to avoid excessive linearization. The algorithm is detailed in figure 2 details which nodes are selected to be linearized. In particular, any node containing a `yield`, `break` or `continue` must be linearized, as well as all of their ancestors.

Once we have the nodes for linearization, we apply the straightforward linearization transforms to control flow as described in appendix B. States are in the form of `case` statements, which denote a state number that can be jumped to by setting the appropriate state number and `break`ing from the case.

For yield statements, we create a state for resuming execution when we resume the state machine. In the previous state, we create a deferred jump to the resume state, set the yielded value as the current value emitted by the state machine and then return `true` from the state machine to signify that the current value has changed.

We keep track of two things during linearization: labels and loops. A loop can be considered an implicit label that is given a name using the enumeration order. A label refers to a statement that can be jumped to, but only from a child statement of the labeled statement. With loop and label state, the following conditions apply to the linearization transformer:

- When the transformer encounters a label statement, we push the label onto the stack of labels with its name, as well as onto the map of labels indexed by string. The label will also have the start state associated with it, i.e. the point we will jump to if we `continue` to it. After we process the label's child statement, we know the state at which the statement ends so we set that as the point that we jump to if we attempt to `break` to it. During transformation of child `break` and `continue` statements, we verify that we only ever attempt to `break` or `continue` to labels found in parent node. If a label is directly adjacent to a loop, the label is marked as being a loop label (and consequently allows `continues`).
- When the transformer encounters a canonical loop, we repeat the same procedure we did with the labels with the exception that we do not add them to the map of labels, nor do we name the loop labels. The transformer will then transform any label-less `break` and `continue` statements it finds into labeled ones, using special labels prefixed with `.loop`, followed by the loop number as dictated by enumeration order. We only allow `continue` statements to jump to an ancestral loop block.

After all the code has been linearized, we create a trap state that always deferred-jumps to itself and returns `false`. This ensures we don't execute random code in the state machine.

1) Labeled Jump Resolution: The initial linearization pass should have already verified that we aren't performing jumps to non-ancestral blocks. As such, we just look up labels in the linearization context and generate jumps to them for `break` statements. We do the same for `continue` statements, except we use the continue points instead of the break points.

At this point, we expect no more `breaks` or `continues` that haven't been dereferenced.

D. Wrapping

Linearization will have created a full state machine, which must then be wrapped in a `switch` statement wrapped in an infinite loop to facilitate jumping between states. This implementation supplies a runtime class, `genja.rt.Generator`, that wraps the generator to conform to Java's iterator interface. This class adopts slightly different semantics from standard iterators, due to the fact that generators are slightly different to Java iterators:

- A `hasNext()` call on a `genja.rt.Generator` will resume the underlying state machine and set the last state machine success result if and only if the generator's stale flag is set to true. If the stale flag is not set, we simply return the last state machine success result.
- A `next()` call on a `genja.rt.Generator` will make a call to `hasNext()` if and only if the generator's stale flag is set to true. Otherwise, we return the current value yielded from the state machine and set the stale flag to true.

III. Known Issues

Due to the time constraints of this assignment, the following non-essential features have been considered but not implemented. I believe sufficient functionality has been supplied in the implementation provided for a useful implementation of generators.

- `try-catch-finally`. As the generator enables pausing code at arbitrary points, we can enclose execution of the generator in an overarching `try` statement, and jump into `catch` clauses when an exception is encountered. Indeed, linearization is a variant of a continuation-passing style transform, and exceptions are variants of non-local jumps – consequently, we can use linearization-equivalent continuations (states) to represent non-local jumps and, in turn, to represent exceptions.
- Duplicate state elimination. The linearization pass generates states at each point where execution could jump to or be resumed at. These are often duplicated and can be eliminated after linearization.
- Unreachable jump elimination. Occasionally, the linearization pass generates unconditional jumps directly after unconditional jumps. These, of course, have no effect and can be removed.
- Selective loop desugaring. Currently, all loops are desugared. This could be eliminated with an additional annotation pass before desugaring, and making the desugaring pass aware of annotations.
- Selective loop linearization. Currently, all loops (except for infinite ones) are linearized completely. This is because the node annotator chooses to mark `break` and `continue` as statements that require a linearized block – as these statements can jump into linearized code from a non-linearized block.

IV. Conclusion

Through a multi-pass compilation process, we can effectively generate state machines in Java via linearization of control flow. These can then be in turn transformed into generators and made compatible with Java's iterators to facilitate various interesting constructs, such as infinite lazy lists and abandonable loop processing from yield points.

Appendix

A. Canonical Loop Forms

Here are the canonical loop forms for various types of loops.

1) `while`:

```
label: while (pred) {
    body;
}
```

becomes:

```
label: for (;;) {
    if (!pred) break; body;
}
```

2) `do`:

```
label: do {
    body;
} while (pred)
```

becomes:

```
label: for (;;) {
    body; if (!pred) break;
}
```

3) `for` (non-canonical loop):

```
label: for (init; pred; update) {
    body;
}
```

becomes:

```
{
    init;
    label: for (;;) {
        if (!pred) break; body;
        update;
    }
}
```

B. Linearizations

Here are linearized forms for various control structures, as well as `yield`. The case numbers only intend to denote flow.

1) `if`:

```
preamble;
if (pred) {
    consequent;
} else {
    alternate;
}
postamble;
```

becomes:

```
case 0:
    preamble;
    if (pred) {
        state = 1;
        break;
    } else {
        state = 2;
        break;
    }
case 1:
    consequent;
    state = 3;
    break;
case 2:
    alternate;
case 3:
    postamble;
```

2) `for` (canonical loop):

```
preamble;
for (;;) {
    body;
}
postamble;
```

becomes:

```
case 0:
    preamble;
case 1:
    body;
case 2:
    state = 1;
    break;
case 3:
    postamble;
```

3) *switch (with default)*:

```
preamble;
switch (x) {
    case A:
        bodyA;
        break;

    case B:
        bodyB;

    case C:
    case D:
        bodyD;
        break;

    default:
        bodyDefault;
}
postamble;
```

becomes:

```
case 0:
    preamble;
    switch (x) {
        case A:
            state = 1;
            break;
        case B:
            state = 2;
            break;
        case C:
            state = 3;
        case D:
            state = 4;
            break;
        default:
            state = 5;
            break;
    }
    break;
case 1:
    bodyA;
    state = 6;
    break;
case 2:
    bodyB;
case 3:
case 4:
    bodyD;
    state = 6;
    break;
case 5:
    bodyDefault;
case 6:
    postamble;
```

4) *switch (without default)*:

```
preamble;
switch (x) {
    case A:
        bodyA;
        break;

    case B:
        bodyB;

    case C:
    case D:
        bodyD;
        break;
}
postamble;
```

becomes:

```
case 0:
    preamble;
    switch (x) {
        case A:
            state = 1;
            break;
        case B:
            state = 2;
            break;
        case C:
            state = 3;
            break;
        case D:
            state = 4;
            break;
        default:
            state = 5;
            break;
    }
    break;
case 1:
    bodyA;
    state = 5;
    break;
case 2:
    bodyB;
case 3:
case 4:
    bodyD;
    state = 5;
    break;
case 5:
    postamble;
```

5) `yield`:

```
preamble;
yield x;
postamble;
```

becomes:

```
case 0:
    preamble;
case 1:
    state = 2;
    current = x;
    return true;
case 2:
    postamble;
```