DynamicJava

An earlier proposal for special-casing `java.dyn.Dynamic` provides a minimal set of functionality to support interoperability with dynamically typed values, assuming those values are mostly produced and manipulated in other languages.

By assigning canonical invokedynamic calls to other uses (in Java) of dynamic expressions, additional possibilities open up for integrating Java with other languages. The extra steps include defining, for additional constructs which involve dynamic subexpressions. Each such construct is transformed into one or more invokedynamic calls.

Here are the invokedynamic calls for each additional syntax that allows dynamic subexpressions. For each expression, the variables "x" and "y" are dynamic. Variables "a" and "b" are of arbitrary types "A" and "B", which may or may not be the type `Dynamic`.

The unary arithmetic and bitwise operators ! ~ are transformed according to this pattern:

```
!x                  Dynamic.<Dynamic>#"operator:!"(x)
```

The binary arithmetic and bitwise operators + - * / & | ^ % << >> >>> are transformed according to this pattern:

```
x+a                 Dynamic.<Dynamic>#"operator:+"(x, a)
a+x                 Dynamic.<Dynamic>#"operator:+"(a, x)
```

The binary relational operators > < == <= >= != are transformed according to this pattern:

```
x!=a                Dynamic.<boolean>#"operator:!="(x, a)
a!=x                Dynamic.<boolean>#"operator:!="(a, x)
```

Normal method calls are transformed as specified above; the name is not augmented in any way, and is therefore distinct from any other invokedynamic selectors generated by these transformations:

```
x.foo(a,b,...)      Dynamic.<Dynamic>foo(a,b,...)
```

The field and array element selection syntaxes, when a Dynamic-typed value appears to the left of the dot or array bracket, is transformed according to this pattern:

```
x.foo               Dynamic.<Dynamic>#"field:foo"(x)
x[a]                Dynamic.<Dynamic>#"element:"(x,a)
```

As a result of the symmetry between the two syntaxes for invokedynamic expressions, the previous transformations are more compactly expressed as follows:

```
x.foo               x.#"field:foo"()
x[a]                x.#"element:"(a)
```

A simple assignment of a selection expression, when a dynamic expression appears to the left of the dot or array bracket, is transformed as follows:

```
x.foo=a             {t=Dynamic.<Dynamic>#"set:field:foo"(x,a); if(x!=t)x=t;} a
x[a]=b              {t=Dynamic.<Dynamic>#"set:element:"(x,a,b); if(x!=t)x=t;} b
```

The transformed code contains three separate operations: First, an invokedynamic expression processes the base reference and the incoming value (and the index value, if any). Second, the result of the invokedynamic is assumed to return the original base reference, or perhaps an updated version of it; it is in any case assigned to the original variable containing the reference, if it has changed. Finally, the assigned value is produced as the value of the whole expression.

If a dynamic expression appears as an element index, the transformation is similar, except that the array variable is not reassigned:

```
a[x]=b              {Dynamic.<void>#"set:element:"(a,x,b);} b
```

Similarly, a method call may be the subject of an assignment, if its receiver is dynamic. This produces a dynamic call by a transformation similar to assignments to selection expressions:

```
x.foo(a,b,...)=c    {x=Dynamic.<Dynamic>#"set:foo"(x,a,b,...,c);} c
```

The compound assignment operators **`+= -= *= /= &= |= ^= %= <<= >>= >>>=`** are transformed according to this pattern:

```
x+=a                x = Dynamic.<Dynamic>#"operator:+="(x, a)
a+=x                a = Dynamic.<A>#"operator:+="(a, x)
```

If the left-hand operand of the compound assignment is itself a dynamic selection expression, the simple assignment in the transformation of the compound assignment is further transformed. Therefore, the final results are as follows:

```
x.foo+=a            {t=x.#"field:foo"().#"operator:+="(a); x=x.#"set:field:foo"(t)} t
x[a]+=b             {t=x.#"element:"(a).#"operator:+="(b); x=x.#"set:element:"(a,t)} t
a[x]+=b             {t=Dynamic.#"element:"(a,x).#"operator:+="(b); Dynamic.#"set:element:"(a,x,t)} t
```

The unary assignment operators **`++ --`** are transformed according to this pattern:

```
++x                 x = Dynamic.<Dynamic>#"operator:++"(x)
x++                 {x = Dynamic.<Dynamic>#"operator:++"(x,null);} x /*previous value*/
```

Uses of dynamic values in constructs requiring booleans is allowed. They are converted to booleans as according to this pattern:

```
if (x)              if (Dynamic.<boolean>#"as:"(x))
x ? a : b           Dynamic.<boolean>#"as:"(x) ? a : b
```

This pattern holds for if, while, do/while, for, assert, and the first operands of the boolean expressions **`?: && ||`**.

A ternary expression is dynamic if and only if its second or third operand is dynamic.

A sequential logical expression **`&& ||`** is dynamic if and only if one of its two subexpressions is dynamic. In that case (and in any case), its meaning is defined by the following transformations:

```
a || b              a ? a : b
a && b              a ? b : a
```

This treatment of operators scales easily to other, non-Java operators, if an expression parser were able to parse them. There have been proposals (such as Borneo) to allow disciplined extension the set of operators in Java-like languages. The conventions given here would adapt easily, and allow a flexible (if statically untyped) way of supplying semantics to those operators.

If the expression of the for-each statement is `Dynamic`, the statement transformed as follows:

```
for (A a : x)       for (A a : Dynamic.<Iterable<A>>#"for:"(A.class, x))
```

Note that the type parameter for `A` is made explicit in the dynamic call.

In all these transformations, temporaries are generated as needed to prevent side effects from being doubled.

The operator names are rendered in invokedynamic call sites by prepending the string "operator:" and then adding the operator spelling, with the following (arbitrarily fixed) translations of dangerous identifier characters. Each translation of a dangerous character is a two-character pair which begins with a backslash character, as follows:

```
dangerous:          /   <   >   [   ]
safe replacement:   \|  \^  \_  \{  \}
```

For example, the **`<<=`** operator is safely (if rudely) spelled as **`#"\^\^="`**.

There is no special transformation for other statement types. The switch, throw, synchronized, and try statements do not accept dynamic expressions.

(A dynamic expression cannot be cast or converted to a parameterized type instance. This is a problem; the erased information must be reified as an extra argument to the MOP. Likewise, erasing argument types before calling invokedynamic loses important information.)