Consider an operation such as a pre- or post-increment or -decrement:

++i;

In this case, “i” is obviously an l-value, such as a variable or a property. In Java, such a simple expression / statement is actually terribly complex from a compiler point-of-view. For example, consider the following *comment* from IncExpression.java in the Java compiler used by the TDE:

// compilation of "++var":

// ?load var // i for byte/char/short/int, l for long,

// // f for float, d for double

// ?const\_1 // i for byte/char/short/int, l for long,

// // f for float, d for double

// ?add // i for byte/char/short/int, l for long,

// // f for float, d for double

// // ?sub if pre-dec

// i2? // b for byte, c for char, s for short

// // (n/a for int/long/float/double)

// dup? // if not discarded; dup2 for long/double,

// // dup for byte/char/short/int/float

// ?store var // i for byte/char/short/int, l for long,

// // f for float, d for double

//

// optimization of "++var" for int:

// iinc var,1 // -1 if pre-dec

// iload var // if not discarded

//

// compilation of "var++":

// ?load var // i for byte/char/short/int, l for long,

// // f for float, d for double

// dup? // if not discarded; dup2 for long/double,

// // dup for byte/char/short/int/float

// ?const\_1 // i for byte/char/short/int, l for long,

// // f for float, d for double

// ?add // i for byte/char/short/int, l for long,

// // f for float, d for double

// // ?sub if pre-dec

// i2? // b for byte, c for char, s for short

// // (n/a for int/long/float/double)

// ?store var // i for byte/char/short/int, l for long,

// // f for float, d for double

//

// optimization of "++var" for int:

// iload var // if not discarded

// iinc var,1 // -1 if pre-dec

//

// Note: The only difference between "++<expr>" and "<expr>++" is

// the location of the dup when the result is not discarded. This

// holds true for fields and arrays as well as with variables. For

// postfix increment and decrement in which the result is not disc-

// arded, the dup instruction is moved from after the conversion to

// before the constant increment of 1.

//

// compilation of "++field":

// getstatic field

// ?const\_1 // i for byte/char/short/int, l for long,

// // f for float, d for double

// ?add // i for byte/char/short/int, l for long,

// // f for float, d for double

// // ?sub if pre-dec

// i2? // b for byte, c for char, s for short

// // (n/a for int/long/float/double)

// dup? // if not discarded; dup2 for long/double,

// // dup for byte/char/short/int/float

// putstatic field

//

// compilation of "++ref.field":

// <ref>

// dup

// getfield field

// ?const\_1 // i for byte/char/short/int, l for long,

// // f for float, d for double

// ?add // i for byte/char/short/int, l for long,

// // f for float, d for double

// // ?sub if pre-dec

// i2? // b for byte, c for char, s for short

// // (n/a for int/long/float/double)

// dup?\_x1 // if not discarded; dup2\_x1 for long/double,

// // dup\_x1 for byte/char/short/int/float

// putfield field

//

// compilation of "++array[index]":

// <array>

// <index>

// dup2

// ?aload // b for byte, c for char, s for short, i for int,

// // l for long, f for float, d for double

// ?const\_1 // i for byte/char/short/int, l for long,

// // f for float, d for double

// ?add // i for byte/char/short/int, l for long,

// // f for float, d for double

// // ?sub if pre-dec

// i2? // b for byte, c for char, s for short

// // (n/a for int/long/float/double)

// dup?\_x2 // if not discarded; dup2\_x2 for long/double,

// // dup\_x2 for byte/char/short/int/float

// ?astore // b for byte, c for char, s for short, i for int,

// // l for long, f for float, d for double

That’s a lot of variation for what is essentially the same thing.

What if a type incorporated the concept of being incrementable? In Java, variables of the int type have

this concept – hence the iinc byte code. What if this could be expressed in an interface, such as:

interface Incrementable<T> {

T preIncrement();

T postIncrement();

void blindIncrement();

T preDecrement();

T postDecrement();

void blindDecrement();

}

Now it becomes trivially simple to compile an increment operator against any type that implements Incrementable, but this begs the question, how do L-Values take on the aspects of the types that they contain? Consider the L-Values from the compiler comment above: Local Variables, Object Fields, and Array Elements. In the case of the local variable, it is apparent that the compiler would know the compile-time type of the variable, and that it would implement Incrementable, so compilation of the previous example “++i;” could be as simple as:

invoke Incrementable.blindIncrement i

In the case of “i” being a property, it could be as simple as an additional dereference:

getprop this "i" -> tmp0

invoke Incrementable.blindIncrement tmp0

Similarly for array elements, as in "++i[5];":

arrayload i 5 -> tmp0

invoke Incrementable.blindIncrement tmp0

In the two de-referencing examples, the “tmp0” temporary variable is known by the compiler to have a type of Incrementable. There is something else at work, though, because in the first example, “i” was the variable and thus the L-Value, while in the other examples, “tmp0” appears to be a *reference to* the L-Value to increment, but it actually must *be* the L-Value to increment. More correctly, the first example specifies a *variable* L-Value that is referenced by the register “i”, while the second case specifies a *property* L-Value that is referenced by the register “tmp0”, and the third case specifies an *array element* L-Value that is referenced by the register “tmp0”.

--

Partially bound

--

Arrays and Constructors

Array creation is a function of three things:

* Element Type;
* Array Size (Element Count); and
* Initial R-Value for each Element

In other words:

array = f<Type T, int count>(T value)

Imagine a set of partially bound constructors, such as:

int\_array\_constructor = f<int, ?>(0)

Such that a line of code:

array = new int[10];

Is actually translated as:

array = int\_array\_constructor.new<count: 10>();