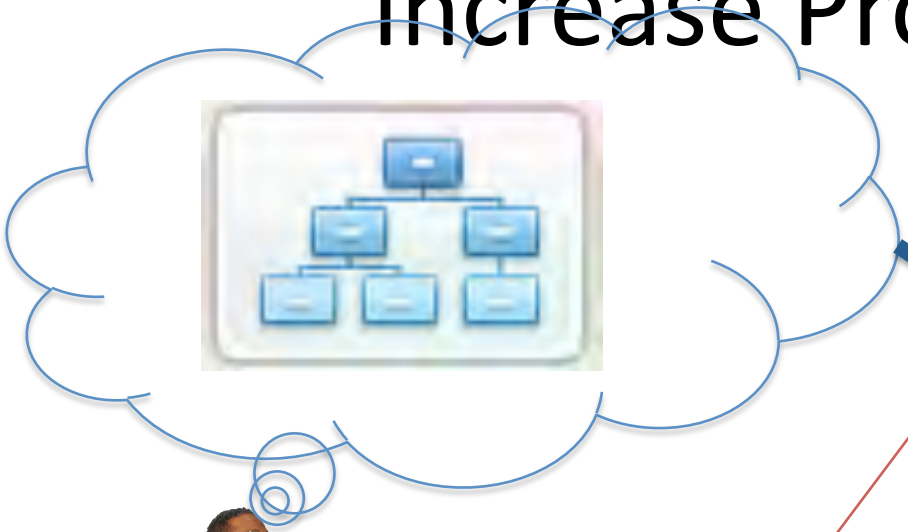
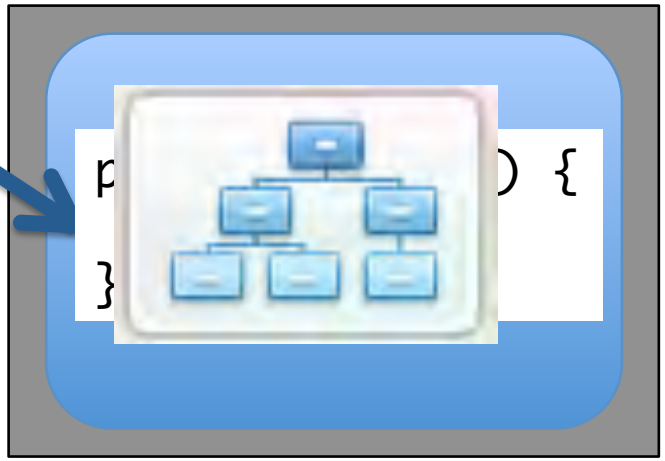
Agile and Efficient Domain-Specific Languages using Multi-stage Programming in Java Mint

Edwin Westbrook, Mathias Ricken,
and Walid Taha

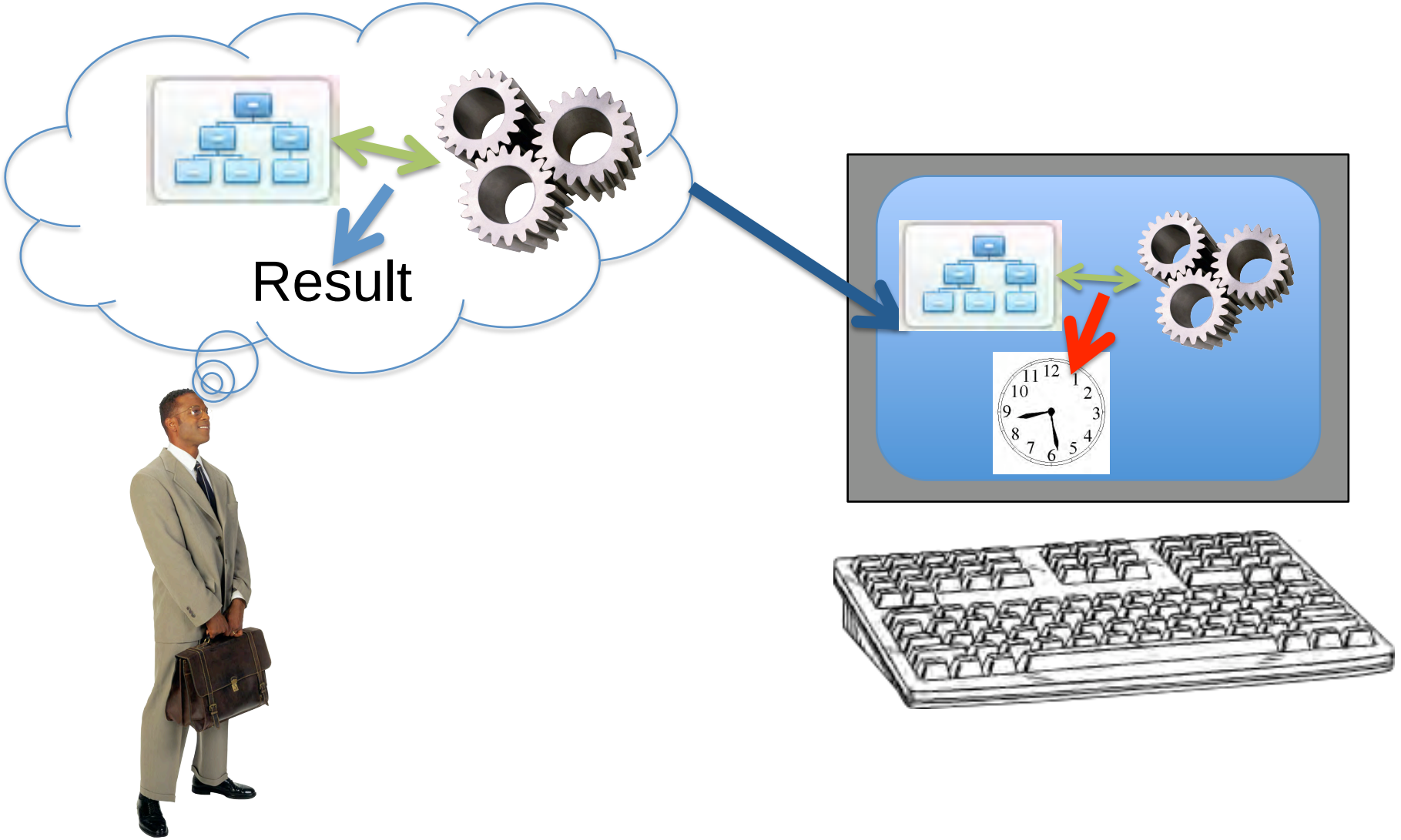
Domain-Specific Languages Increase Productivity



Time + Effort



DSLs Must Be Agile and Efficient



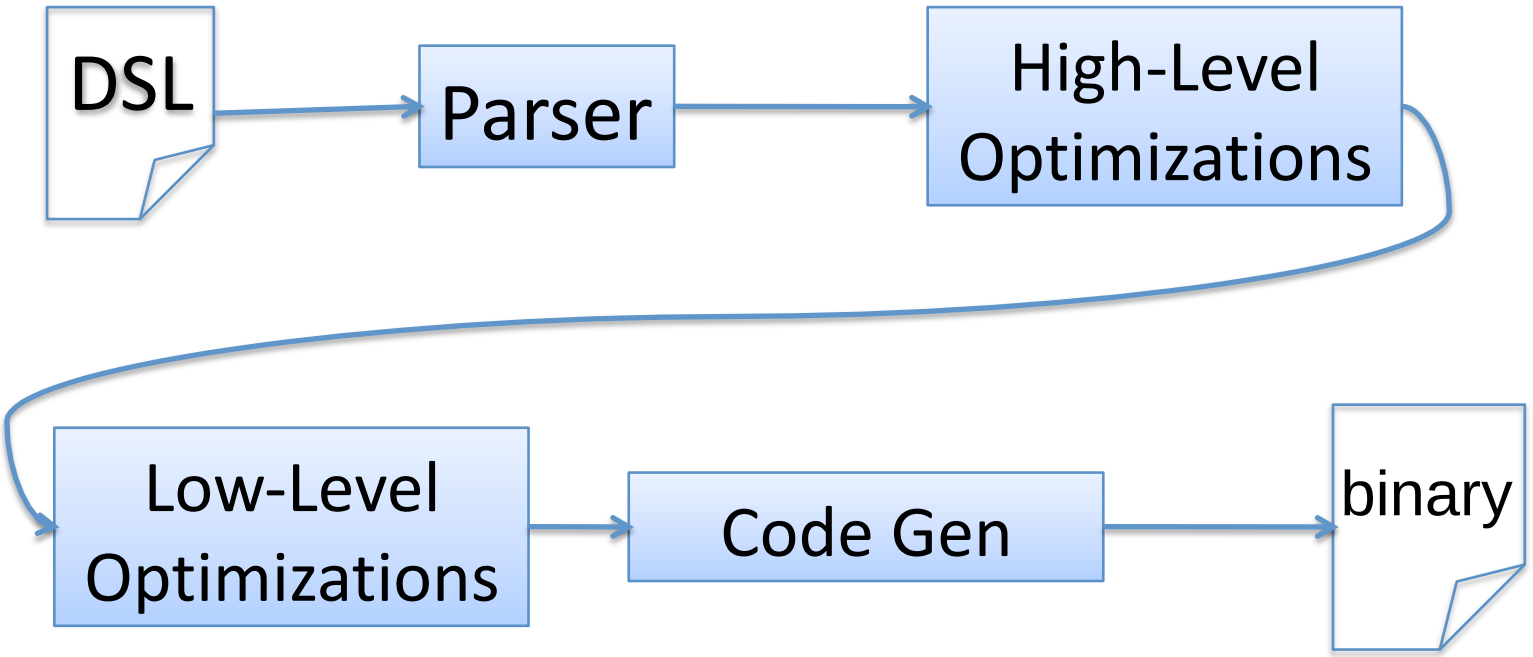
How to Implement DSLs?

- Interpreters: easier to write and modify
- Compilers: more efficiency
- Why can’t we have both?

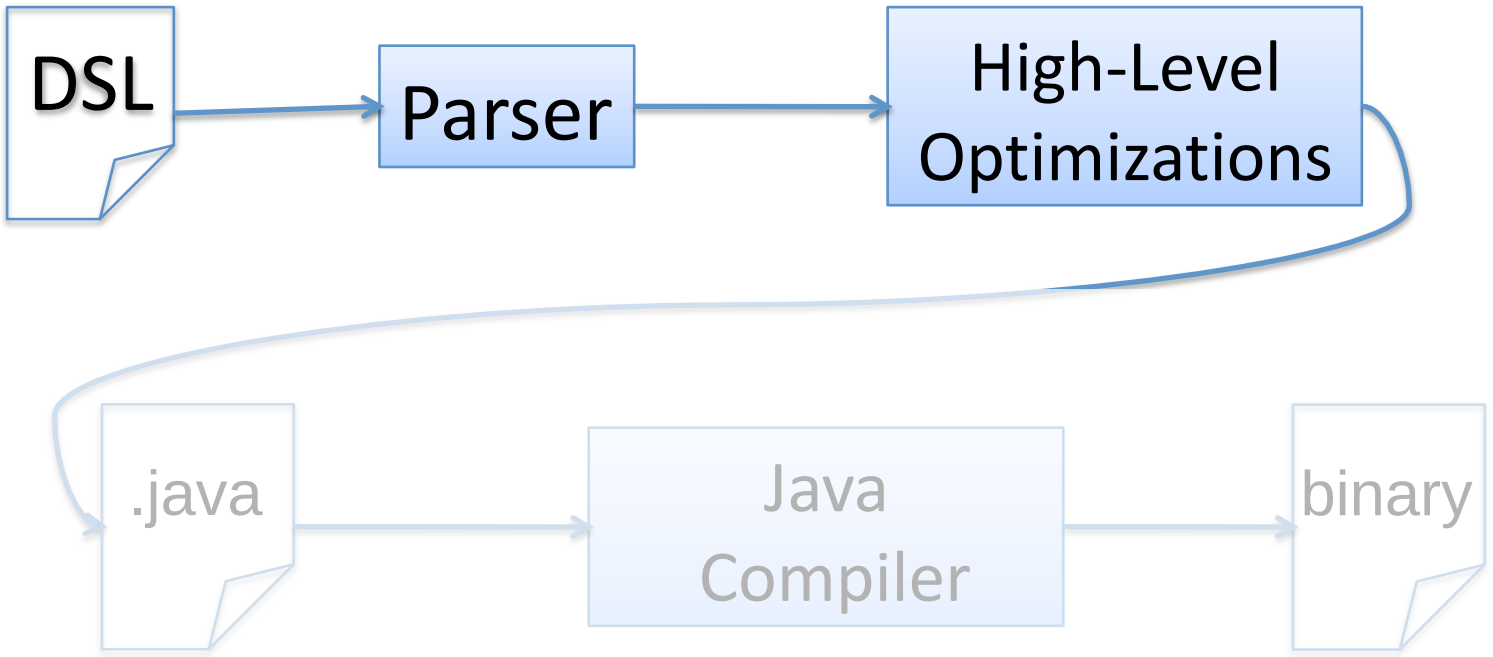
Multi-Stage Programming

- Can turn your interpreter into a compiler
- By adding *staging annotations*
- Result: code-producing interpreter
  - Looks similar to original interpreter
  - Produces compiled code
- Generated code *guaranteed* to be well-typed

Compiler


DSL
Parser
High-Level Optimizations
Low-Level Optimizations
Code Gen
binary

Staged Interpreter in Mint



Outline

- What is MSP?
- Writing a staged interpreter
- Mint toolchain support
- Compiler optimizations in MSP:
  - Dynamic type inference / unboxing
  - Automatic loop parallelization

MSP Reduces the Cost of Abstractions

- MSP languages
  - provide constructs for runtime code generation
  - are statically typed: do not delay error checking until runtime

MSP in Mint

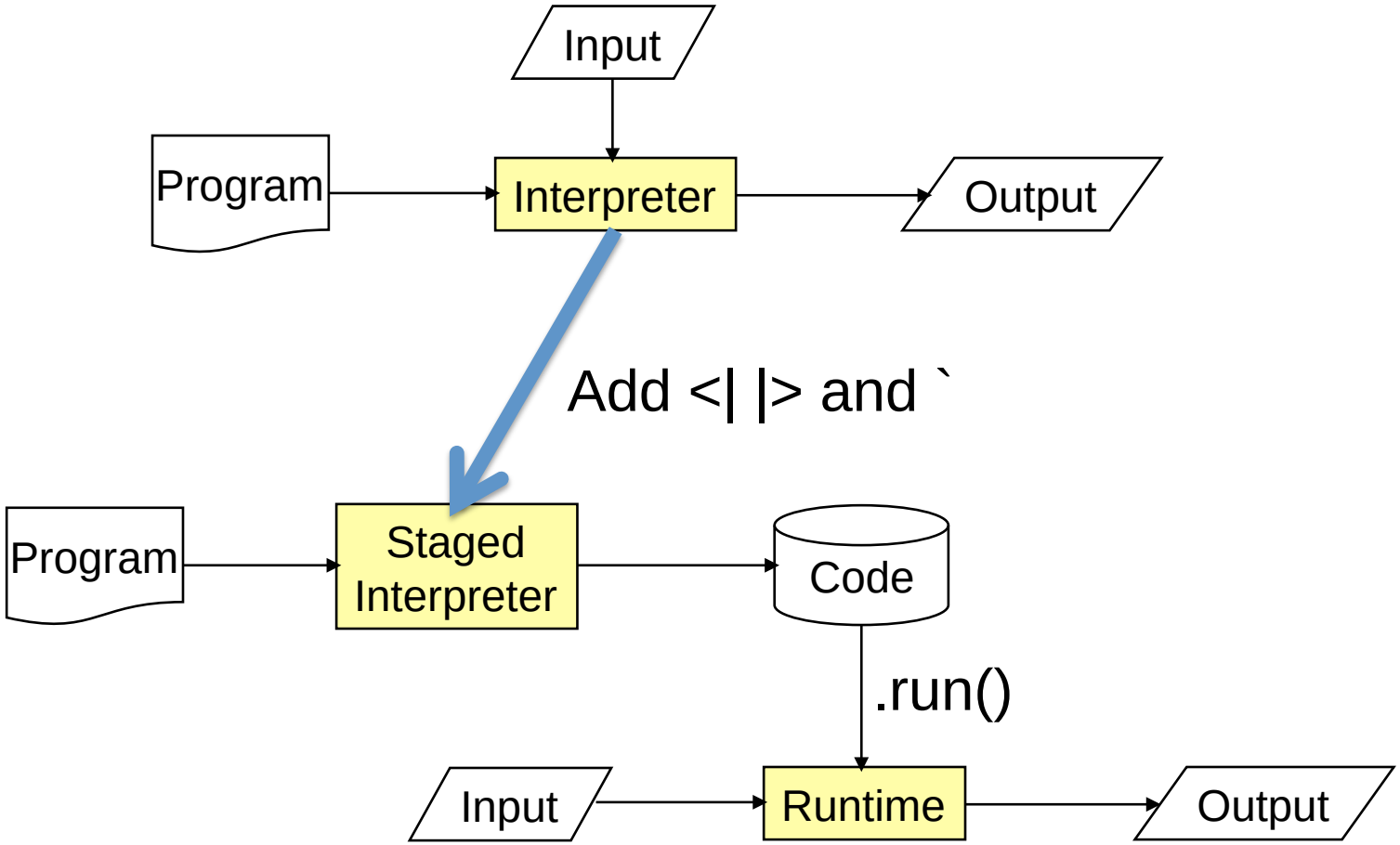
- Code has type Code<A>
- Code built with *brackets* <| e |>
- Code spliced with *escapes* `e
- Code compiled and run with run() method

```
Code<Integer> x = <| 1 + 2 |>;
Code<Integer> y = <| `x * 3 |>;
Integer z = y.run(); // z = 9
```

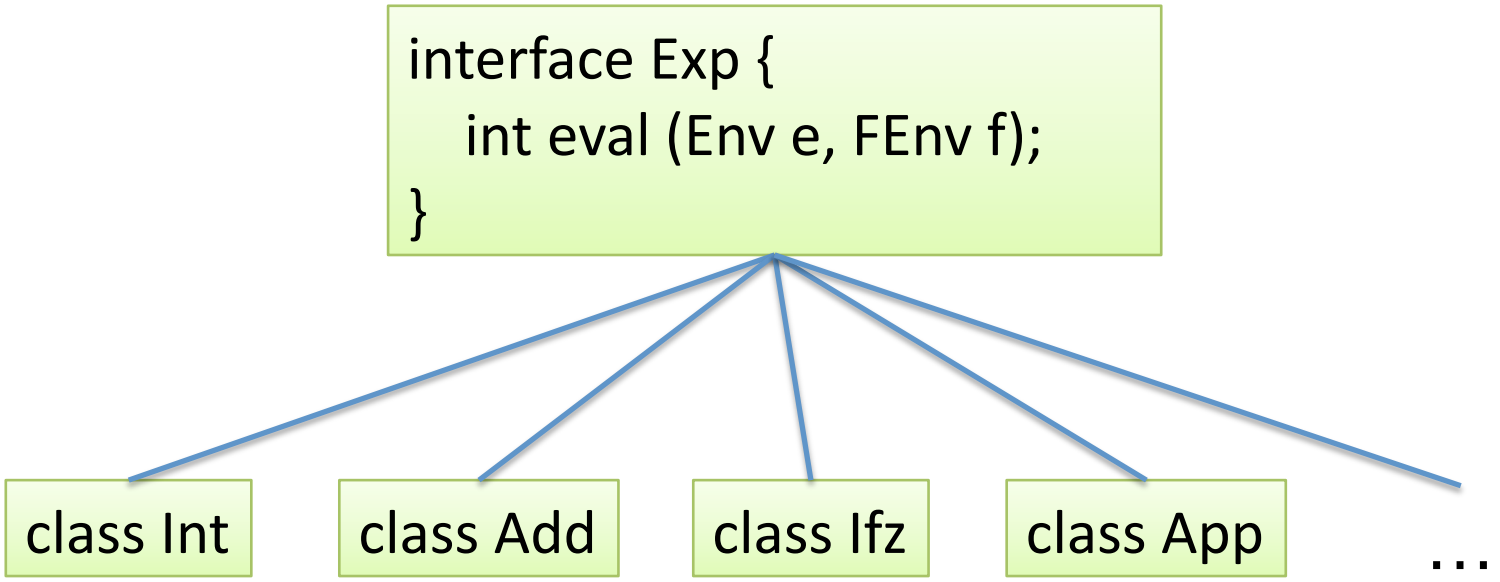
MSP Applications

- Sparse matrix multiplication
  - Specialize for elements of value 0 or 1
- Array Views
  - Habanero Java's way of mapping multiple dimensions into 1-dimensional array
  - Removal of index math
- Killer example: Staged interpreters

Staging an Interpreter



Unstaged Interpreter in Java

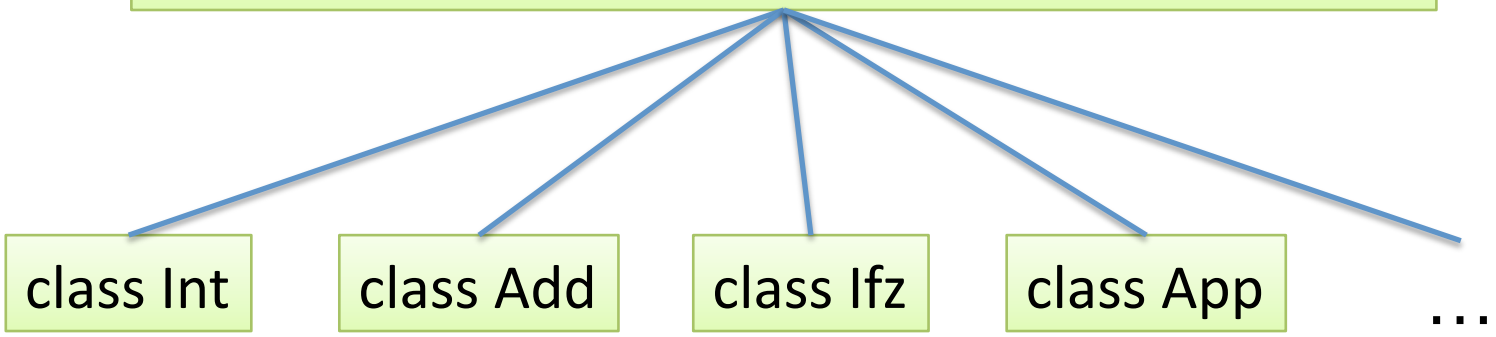


Unstaged Interpreter in Java

```
public class Int implements Exp {
    private int _v;
    public int eval(Env e, FEnv f) {
        return _v;
    }
}

public class Add implements Exp {
    private Exp _left, _right;
    public int eval(Env e, FEnv f) {
        return _left.eval(e,f) + _right.eval(e,f);
    }
}
```

Staging the Interpreter

```
interface Exp {
    separable Code<Integer> eval (Env e, FEnv f);
}
```



Staging the Interpreter

```
public class Int implements Exp {
    private Code<Integer> _v;
    public separable Code<Integer> eval(Env e, FEnv f) {
        return _v;
    }
}

public class Add implements Exp {
    private Exp _left, _right;
    public separable Code<Integer> eval(Env e, FEnv f) {
        return <| `(_left.eval(e,f)) + `(_right.eval(e,f)) |>;
    }
}
```

Side Note: Weak Separability

- Escapes must be *weakly separable*
  - Prevents *scope extrusion* [Westbrook et al. ‘10]
- Limits the allowed assignments:
  - Block-local variables; OR
  - Variables with *code-free* type
- Can only call **separable** methods

Side Note: Weak Separability

```
int i;
Code <Integer> d; = <| y + 3 |>

separable Code<Integer> foo (Code<Integer> c) {
    Code<Integer> c2 = <| `c + 1 |>;          // OK
    i += 2;                                   // OK
    d = <| `c + 3 |>;                         // BAD
    return c;
}

<| new A() {
        int bar (int y) {
            `(foo (<| y |>))
        }} |>;
```

Using the Staged Interpreter

```
interface IntFun {
    int apply (int x);
}

Code<IntFun> tenPlusXCode = <| new IntFun() {
    public int apply(final int x) {
        return  x + 10;

} |>;
```

DSL Implementation Toolchain

- Reflection-based S-expression parser

```
(Ifz (Var x) (Int 1) (Mul (Var x) (App f (Sub (Var x) (Int 1)))))
```

- Abstract compiler and runner
  - Template method design pattern
  - Compiles code to a .jar file using Mint's save() method
  - Command-line:

```
mint Compiler prog.dsl output.jar
mint Runner output.jar <input>
```

Compiler for Example DSL

```
public class Compiler extends util.ACompiler<Exp,IntFun> {
    public Exp parse(String source) {
        return parser.LintParser.parse(Exp.class, source);
    }

    public Code<IntFun> getFunction(final Exp funBody) {
        return <| new IntFun() {
            public separable int apply(final int param) {
                return `(bindParameter(funBody, <|param|>));
            }
        } |>;
    }
}
```

Results (1st Interpreter)

- Program: factorial

  (Ifz (Var x) (Int 1) (Mul (Var x) (App f (Sub (Var x) (Int 1)))))

- Generated code:

```
new IntFun() {
    public Value apply(final Value x) {
        return (x == 0) ? 1 : (x * apply (x - 1));
}}
```

- 27x speedup relative to unstaged (x = 10)
  - 2.4 GHz Intel Core 2 Duo, OS X 10.5.8, 2 GB RAM

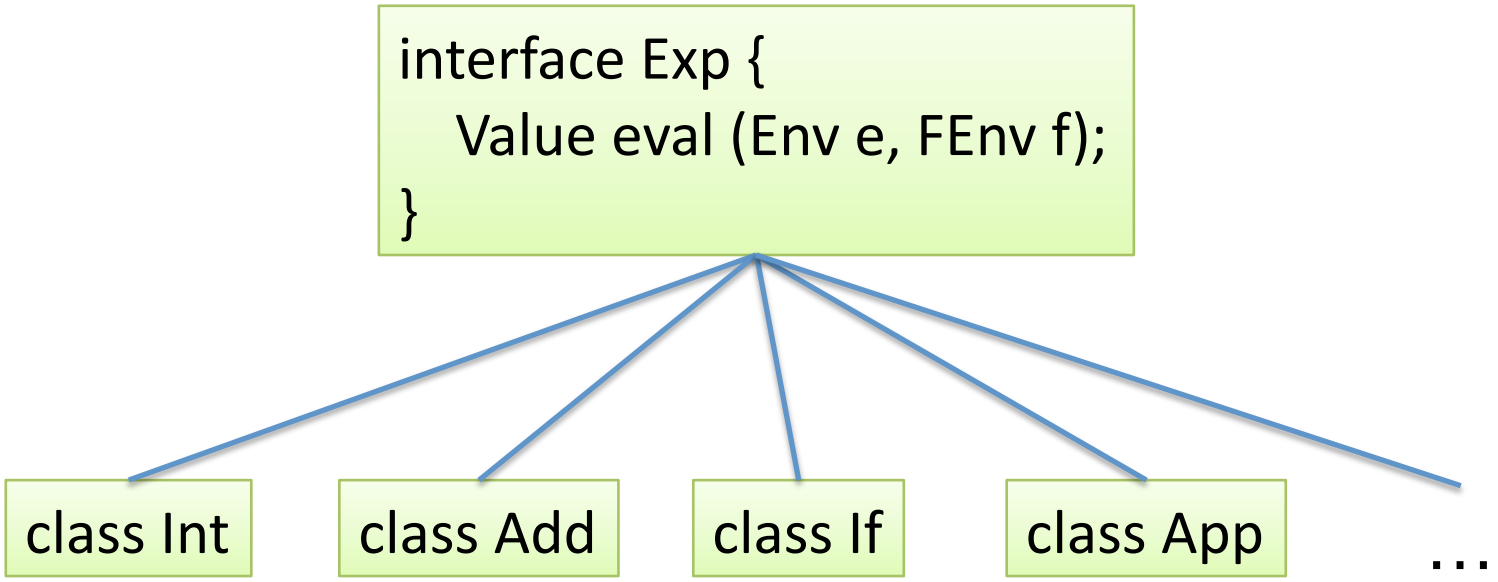
Multiple Data Types

```
abstract class Value implements CodeFree {
    int intValue();
    boolean booleanValue();
}
```

```
class IntValue {
    int _i;
    int intValue() { return _i; }
    boolean booleanValue() {
        throw Error;
    }
}
```

```
class BooleanValue {

    // … similar …

}
```

Multiple Data Types

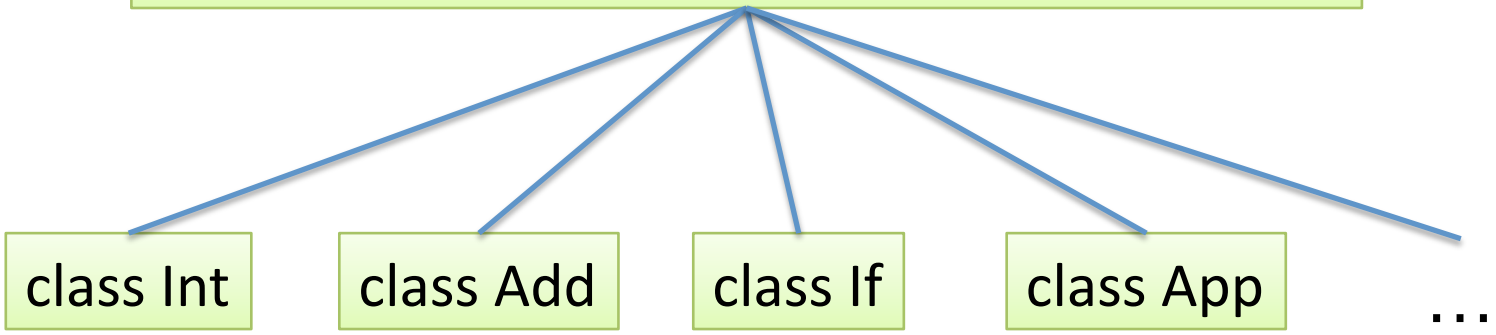


Unstaged Interpreter in Java

```
public class Val implements Exp {
    private Value _v;
    public Value (Value v) { _v = v; }
    public Value eval(Env e, FEnv f) {
        return _v;
    }
}

public class Add implements Exp {
    private Exp _left, _right;
    public Value eval(Env e, FEnv f) {
        return new IntValue (_left.eval(e,f).intValue() +
                             _right.eval(e,f).intValue ());
    }
}
```

Staging the Interpreter

```
interface Exp {
    separable Code<Value> eval (Env e, FEnv f);
}
```



Staging the Interpreter

CSP of Value type; must be CodeFree

```
public class Val implements Exp {
    private Code<Value> _v;
    public Val (final Value v) { _v = <| v |>; }
    public separable Code<Value> eval(Env e, FEnv f) {
        return _v;
    }
}

public class Add implements Exp {
    private Exp _left, _right;
    public separable Code<Value> eval(Env e, FEnv f) {
        return <| new IntValue (`(_left.eval(e,f)).intValue() +
                                `(_right.eval(e,f)).intValue()) |>;
    }
}
```

Results (Multiple Datatypes)

- Generated code (factorial):

```
public Value apply(final Value x) {
   return (new BooleanValue(x.valueEq(new IntValue(0))).booleanValue()) ?
             new IntValue(1) :
             new IntValue(x.intValue() *
                          apply(new IntValue(x.intValue() -
                                             new IntValue(1).intValue())).intValue());
}
```

- 2.7x speedup relative to unstaged (x = 10)

Overhead of Unnecessary Boxing

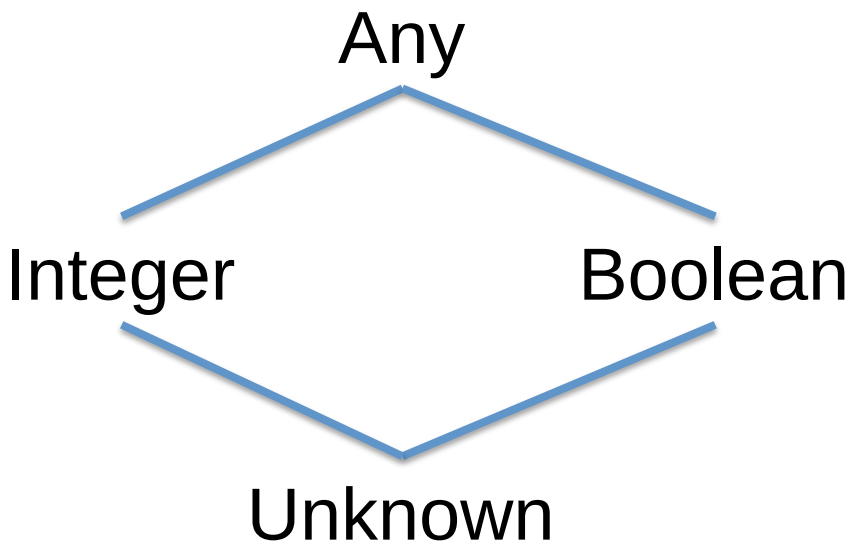
- eval() always returns Code<Value>:

```
public static class Add {
    public separable Code<Value> eval(Env e, FEnv f) {
        return <| new IntValue(`(_left.eval(e,f)).intValue() +
                               `(_right.eval(e,f)).intValue()) |>;
    }
}
```

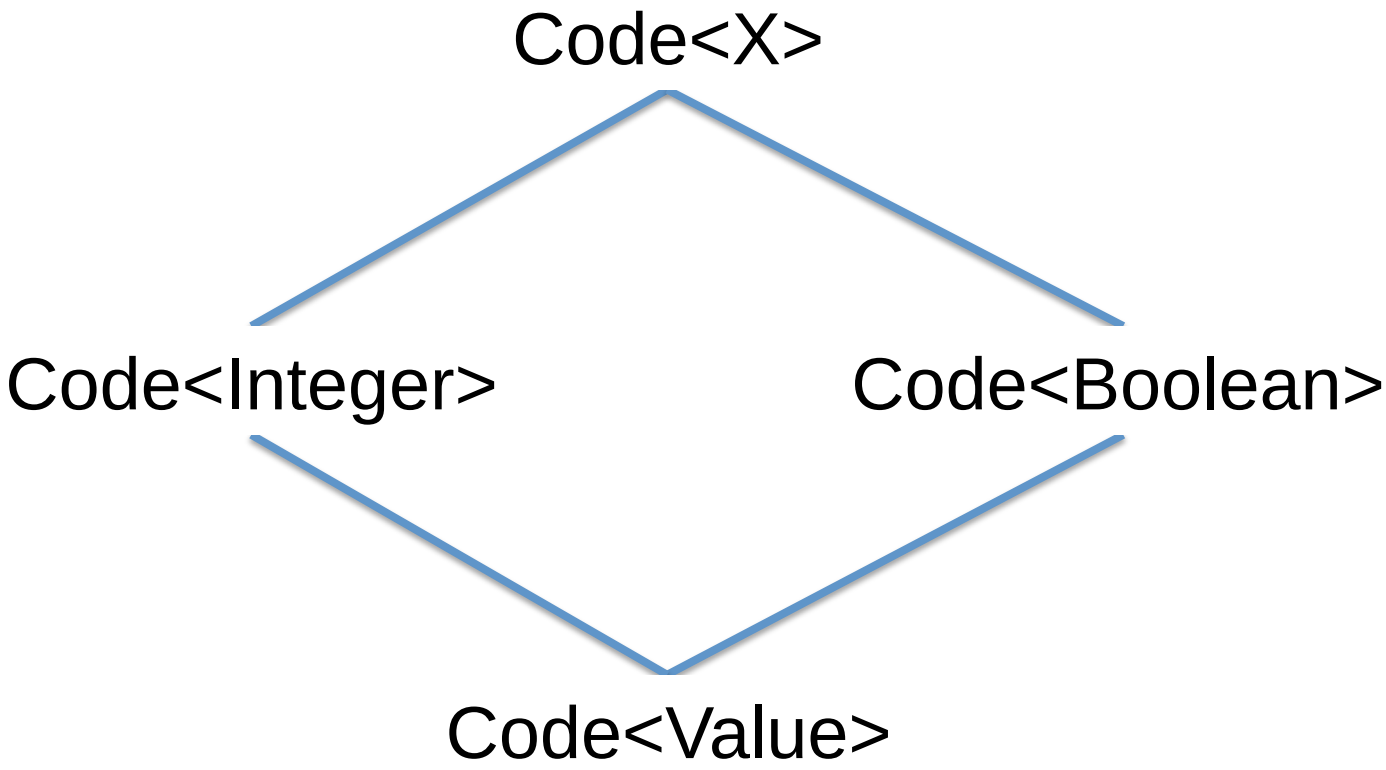
IntValue always built at runtime!

Solution: Unboxing

- Statically determine the type of an expression
- Elide coercions to/from boxed Value type
- Can be expressed as a dataflow problem:



Dataflow in MSP: Nodes → Types



“Staging the Value Type”

```
abstract class SValue {
    <X> Code<X> anyCodeValue();
    Code<Integer> intCodeValue();
    Code<Boolean> booleanCodeValue();
    Code<Value> codeValue();   }
```

```
class SIntValue {
    Code<Integer> _i;
    codeValue() {
        return <| new IntValue(`_i) |>;
}
```

```
class SBooleanValue {
    Code<Boolean> _b;
}
```

```
    intCodeValue() {
        return <| `c.intValue() |>;
    }
}
```

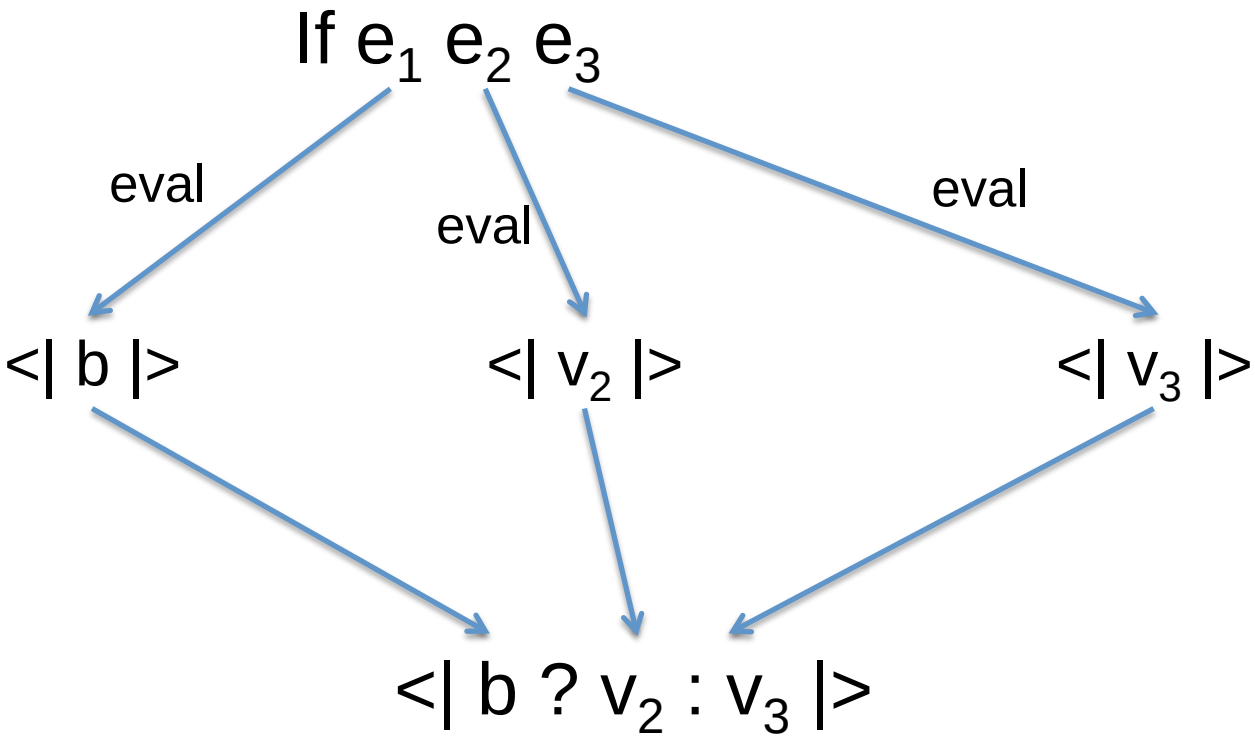
```
nyValue {
t <X> Code<X>
CodeValue();}
```

New eval for Unboxing

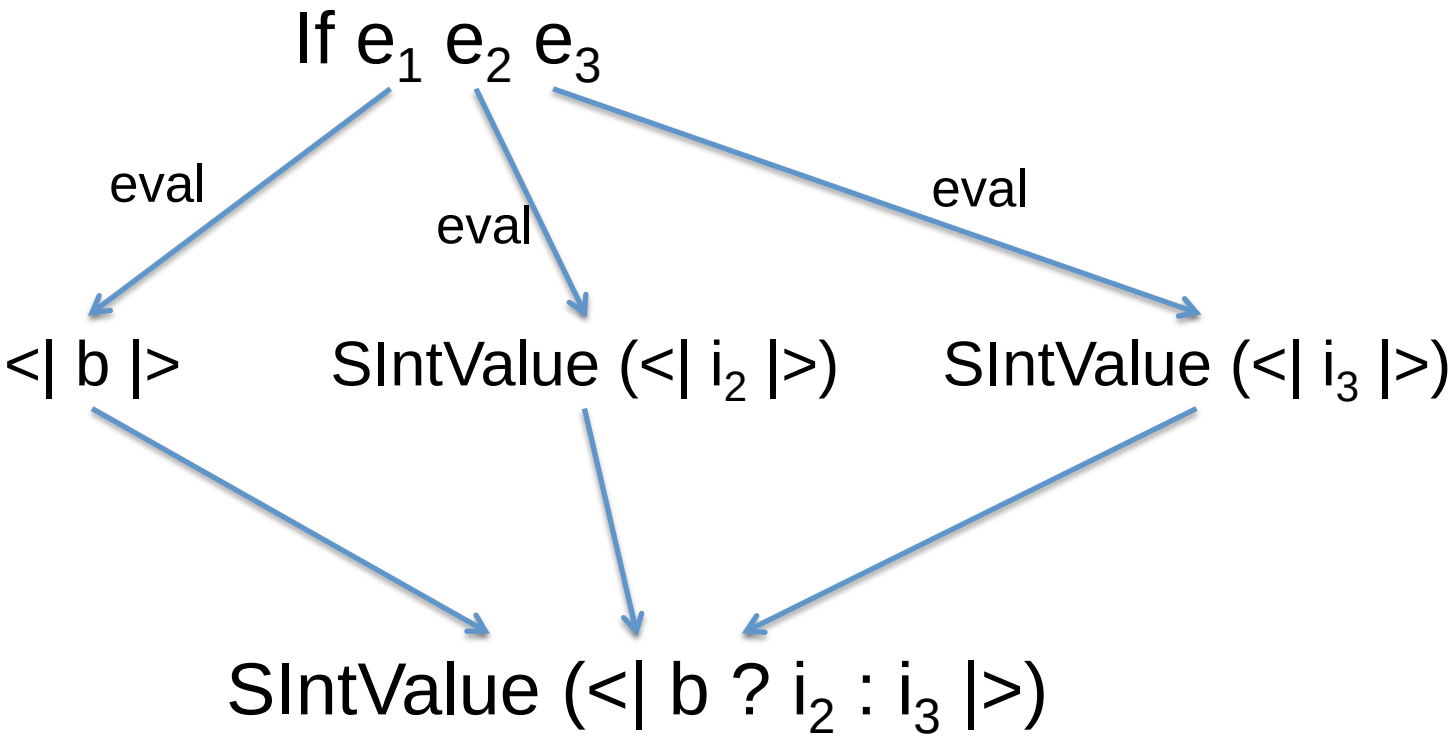
```
public interface Exp {
    public separable SValue eval(Env e, FEnv f);
}

public static class Add {
    //…
    public separable SValue eval(Env e, FEnv f) {
        return new SIntValue(<| `(_left.eval(e,f).intCodeValue())
                                + `(_right.eval(e,f).intCodeValue()) |>);
    }
}
```

Boxing/unboxing only inserted where needed

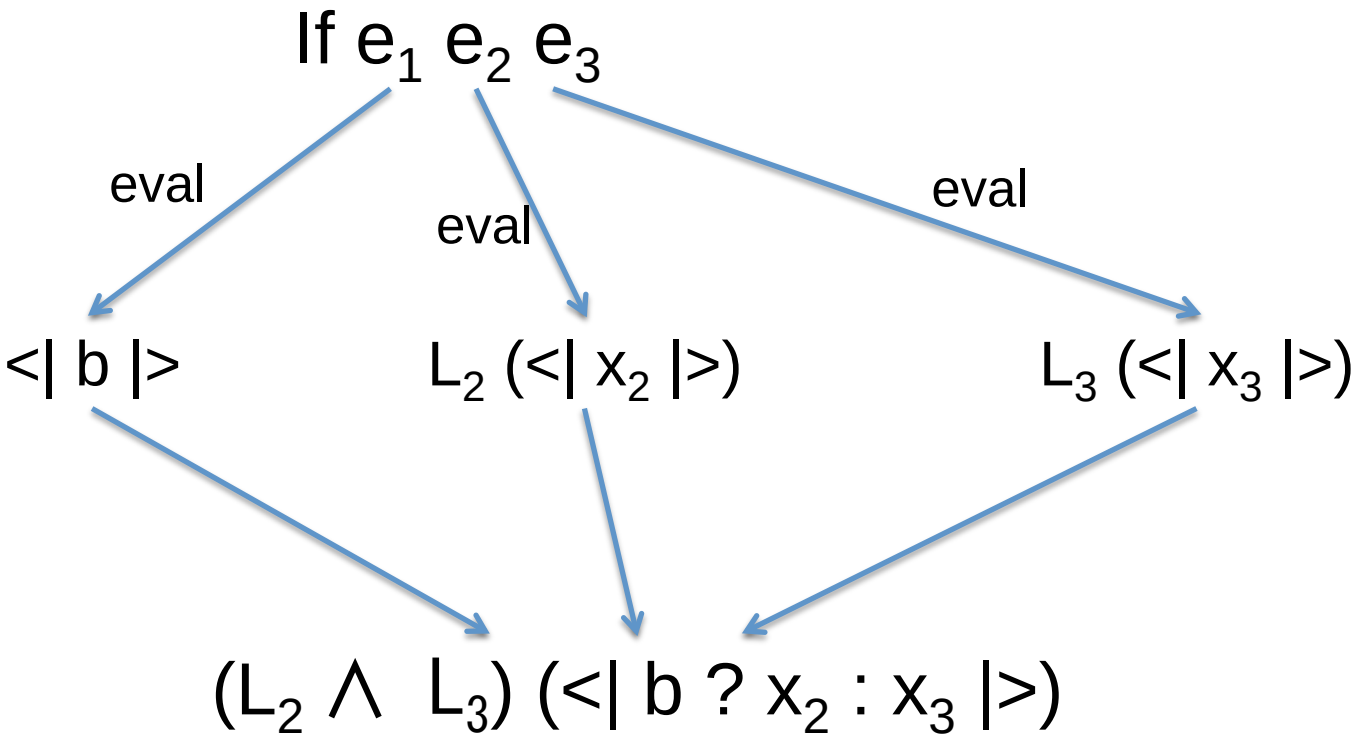
Staging Join Points



Staging Join Points



Staging Join Points



Results (Staged Value Type)

- Generated code (factorial):

```
public Value apply(final Value x) {
    return new IntValue
        (((x instanceof IntValue) ? (x.intValue() == 0) : (/* error */))
          ? 1
          : (x.intValue()
             * apply (new IntValue (x.intValue() - 1)).intValue()));
}
```

- 4.1x speedup relative to unstaged (x = 10)

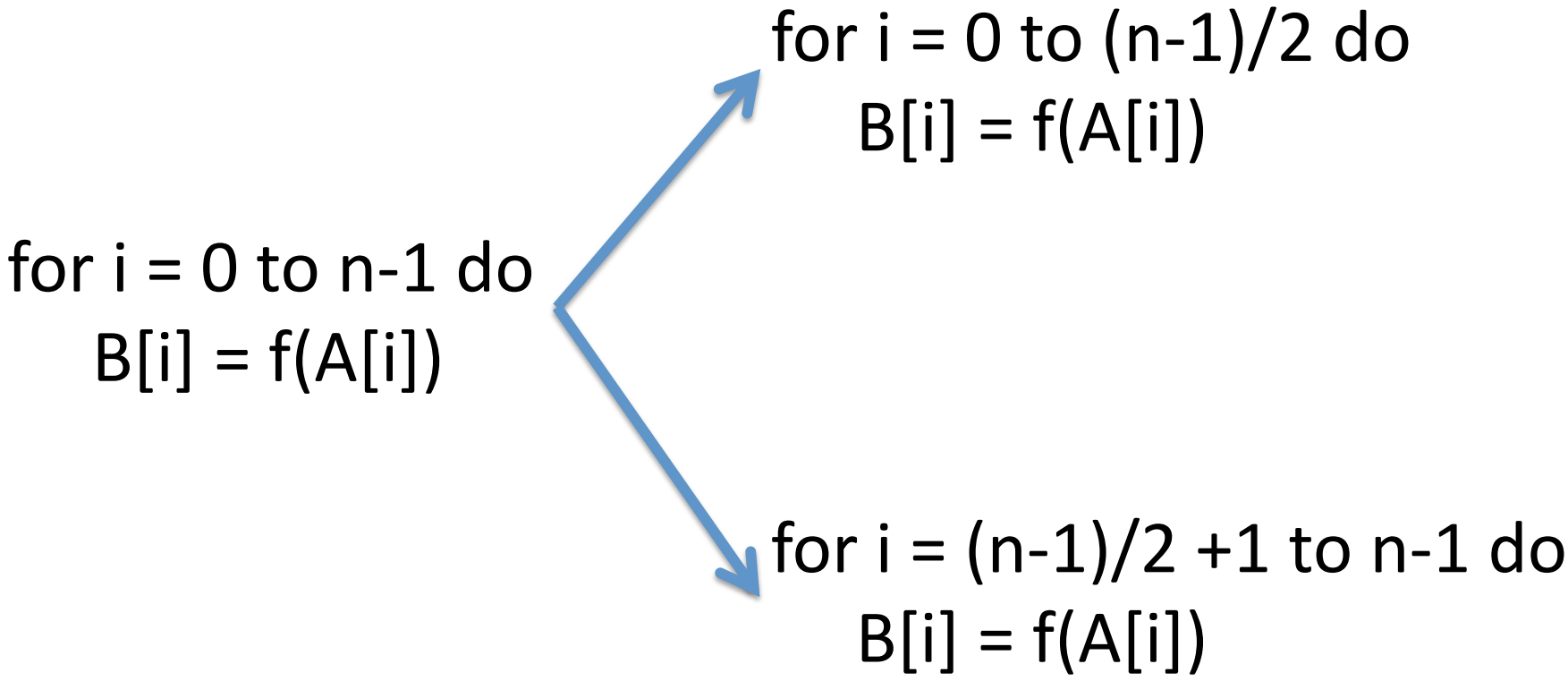
One More Optimization: Type-dependent Methods

- Generated code (factorial):

```
public Value applyInt(int x) {
    return new IntValue
        (x == 0) ? 1 : x * applyInt (x - 1).intValue();
}

public Value applyBoolean(boolean b) { (/* error */) }
```

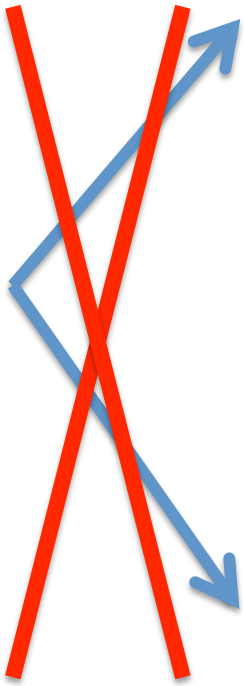
- 7.8x speedup relative to unstaged (x = 10)

Automatic Loop Parallelization



Not That Easy…

```
for i = 0 to n-1 do
    B[i] = f(B[i-1])
```



```
for i = 0 to (n-1)/2 do
    B[i] = f(B[i-1])
```

Could set B[(n-1)/2 + 1] before B[(n-1)/2]!

```
for i = (n-1)/2+1 to n-1 do
    B[i] = f(B[i-1])
```

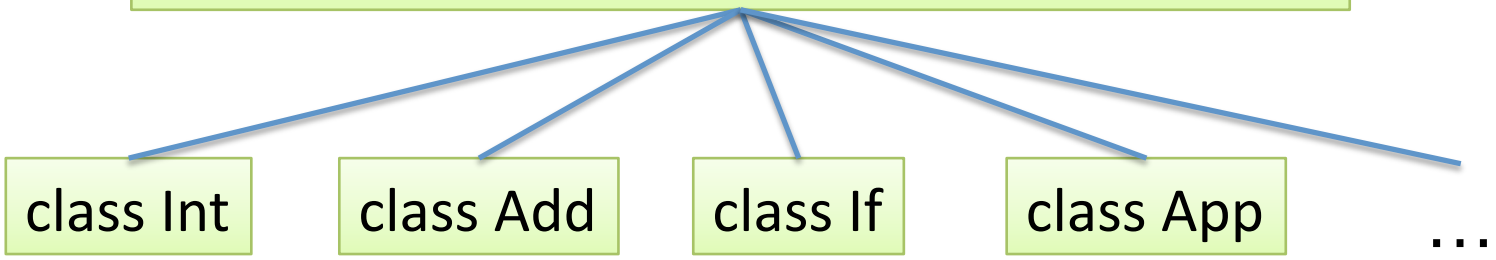
Embarrassingly Parallel Loops

```
for i = 0 to n-1 do
    … = R[i]
    W[i] = …
```

{ Reads R } ∩ { Writes W } = ∅

Staging the Interpreter

```
interface Exp {
    separable Code<Value> eval (Env e, FEnv f);
    separable RWSet rwSet ();
}
```



Implementation: rwSet Method

```
public class Int implements Exp {
    //…
    public RWSet rwSet () { return new RWSet (); }
}

public class Add implements Exp {
    //…
    public RWSet rwSet () {
        return _left.rwSet().union (_right.rwSet ());
    }
}
```

Implementation: rwSet Method

```
public class AGet implements Exp {
    //…
    public RWSet rwSet () {
        if (_arr instanceof Var) {
            return _index.rwSet().addR (((Var)_arr)._s);
        } else {
            return _index.rwSet().completeR ();
        }
    }
}
```

Staged Parallel For

```
public class For implements Exp {
    //…
    public SValue eval (Env e, FEnv f) {
        return new SIntValue
            (<| LoopRunner runner = /* … loop code … */;
                `(rwOverlaps (e, _body.rwSet())) ?
                    runner.run () : runner.run_parallel ()  |>);
}
```

Inserts runtime check for whether

{ Reads R } ∩ { Writes W } = ∅

Test Program: Parallel Array Set

```
(Let a (AllocArray (Var x) (Val (IntValue 1)))
    (Let dummy (For i (Var x)
                    (ASet (Var a) (Var i) (Var i)))
        (Var a)))
```

- 5x speedup relative to unstaged ($x = 10^6$)
  almost no speedup relative to serial staged
  - 2.67 GHz Intel i7, RHEL 5, 4 GB RAM

Results (Loop Parallelization):

- Generated code (array set):

```
public Value apply(final Value p) {
  return let final Value temp = new ArrayValue(
                        allocArray(p.intValue(), new IntValue(1)));
  let final int notUsed = let LoopRunner runner =
      new LoopRunner(){
        public int run(int lower, int upper) { /* next slide */ }
      };
  let int bound = p.intValue(); let boolean run_single = (false);
  run_single ? runner.run(0, bound) : runner.runParallel(bound);
  temp;
}
```

Results (Loop Parallelization):

- Generated code (array set):

```
public Value apply(final Value p) {
  ...
    public int run(int lower, int upper) {
        for (int i = lower; i < upper; ++i) {
           Value unused = new IntValue(
             setArray(temp.arrayValue(), i, new IntValue(i)));
        }
        return 0;
    }
  ...
}
```

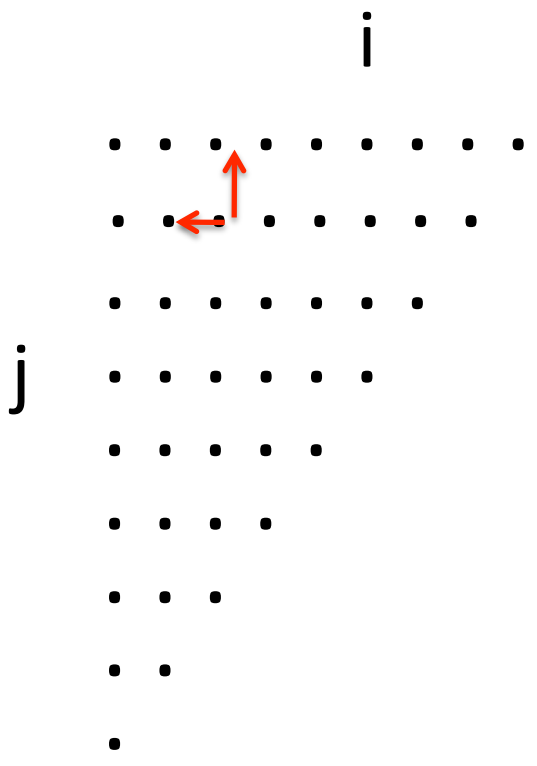
Conclusion

- MSP for DSL implementation
  - Agile: as simple as writing an interpreters
  - Efficient: performance of compiled code
- Dataflow-based compiler optimizations
  - Insert runtime checks
- Helpful toolchain for implementing DSLs
- Available for download:

  http://www.javamint.org

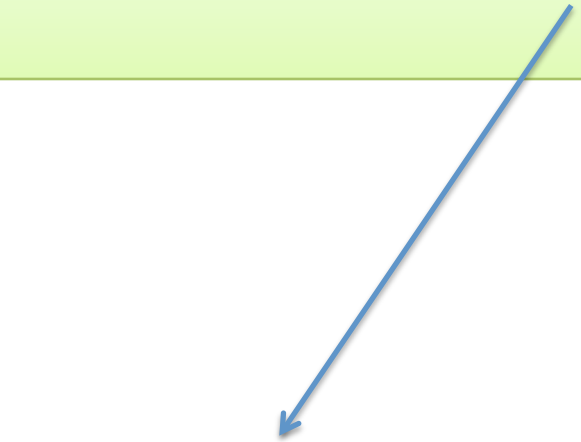
Future Work: Iteration Spaces

```
for i from 1 to N do
   for j from 1 to i do
      A[i,j] = f (A[i-1,j], A[i,j-1])
```



Iteration Spaces in MSP

```
interface Exp {
   separable Pair<SValue,Deps> eval (Env e, FEnv f, IterSpace i);
}
```



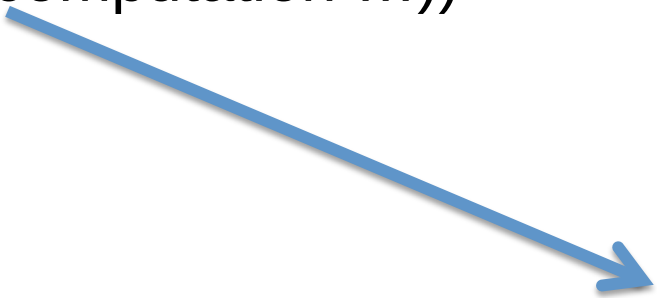
{ (i-1,j), (i,j-1) }

{ i → [1, N);
  j → [1, i) }

Thank You!

Let Removes Code Duplication

```
(Mul (… complex computation …)
     (… complex computation …))
```



```
(Let y (… complex computation …)
   (Mul y y))
```

FIXME: remove or shorten let discussion

Staging Let Expressions

```
public static class Let implements Exp {
    // …

    public Value eval(Env e, FEnv f) {
        Env newenv = ext (e, _var, _rhs.eval (e, f));
        return _body.eval (newenv, f);
    }
}
```

Naïve Staging of Let Expressions

```
public static class Let implements Exp {
    // …

    public separable SValue eval(Env e, FEnv f) {
        Env newenv = ext (e, _var, _rhs.eval (e, f));
        return _body.eval (newenv, f);
    }
}
```

Causes code duplication!

Proper Staging of Let

```
public static class L
    // …

    public separable SValue e      Env e, FEnv f) {
        SValue rhsVal = _rhs.e     (e, f);
        if (rhsVal instanceof SIntValue) {
            return new SCodeValue
              (<| let int temp = `(rhsVal.intCodeValue ());
                    `(_body.eval
                      (ext (e, _var, new SIntValue(<|temp|>)),
                       f).codeValue ())|>);
        }

        else if (rhsVal instanceof SBooleanValue) {
            //…
        }
    }
}
```

Loss of type information!

Demo:
Code Duplication

Adding Loops and Mutable Arrays

```
public static class ArrayValue extends Value {
    private Value[] _data;
    public ArrayValue(Value[] data) { _data = data; }
    public Value[] arrayValue() { return _data; }
}

public static class ASet implements Exp {
    public Exp _arr, _index, _val;
    public ASet(Exp arr, Exp index, Exp val) {
        _arr = arr;
        _index = index;
        _val = val;
    }
    public Value eval(Env e, FEnv f) {
        _arr.eval (e, f).arrayValue()[_index.eval (e, f).intValue()]
            = _val.eval (e, f);
        return new IntValue (0);
    }
}
```

FIXME: just give an example program with loops

Adding Loops and Mutable Arrays

```
public static class For implements Exp {
    String _var;
    public Exp _bound, _body;
    public For(String var, Exp bound, Exp body) {
        _var = var;
        _bound = bound;
        _body = body;
    }
    public Value eval(Env e, FEnv f) {
        int bound = _bound.eval (e, f).intValue ();
        for (int i = 0; i < bound; ++i) {
            _body.eval (ext (e, _var, new IntValue (i)), f);
        }
        return new IntValue (0);
    }
}
```

Future Work: Monadic Staging

- Monads: powerful abstraction for interpreters
  - Hide “features” in monad
  - More modular, easier to add/change “features”

```
abstract class M<X> {
   X runMonad (Env e, FEnv f);

   static <Y> M<Y> ret (Y in);
   <Y> M<Y> bind (Fun<X,M<Y>> f);
}
```

```
interface Exp {
    M<Value> eval ();
}
```

Future Work: Monadic Staging

- Idea: Staged Monads
  - Hide “staging-related features”
  - Modularity in staged interpreter
  - More information out of Let expressions…?

```
abstract class SMInt extends SM<Value> {
    Code<Int> runMonadInt (Env e, FEnv f);

    // …
}
```