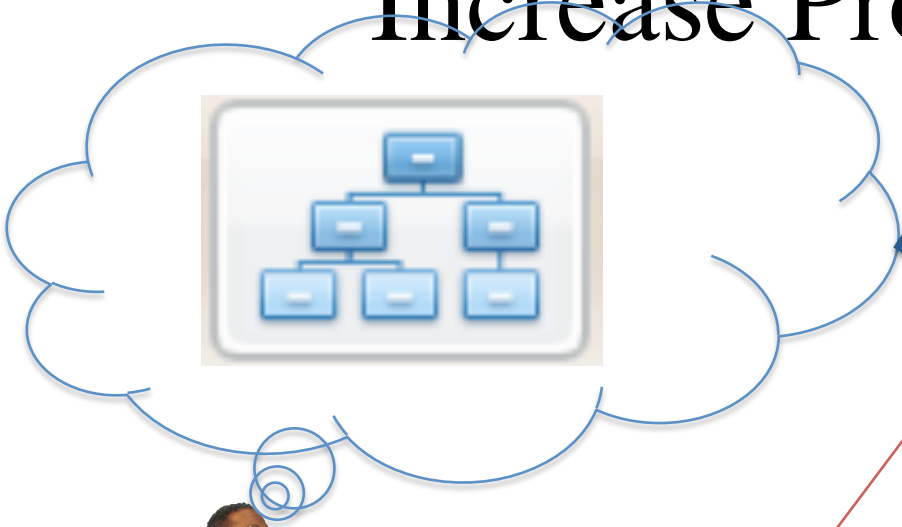
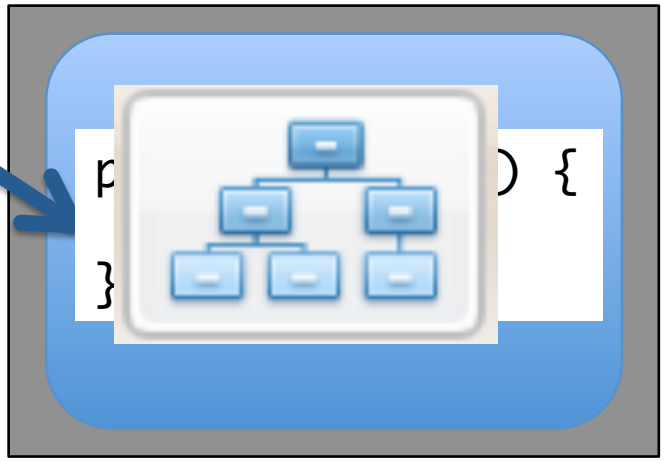
Agile and Efficient Domain-Specific Languages using Multi-stage Programming in Java Mint

Edwin Westbrook, Mathias Ricken,
and Walid Taha

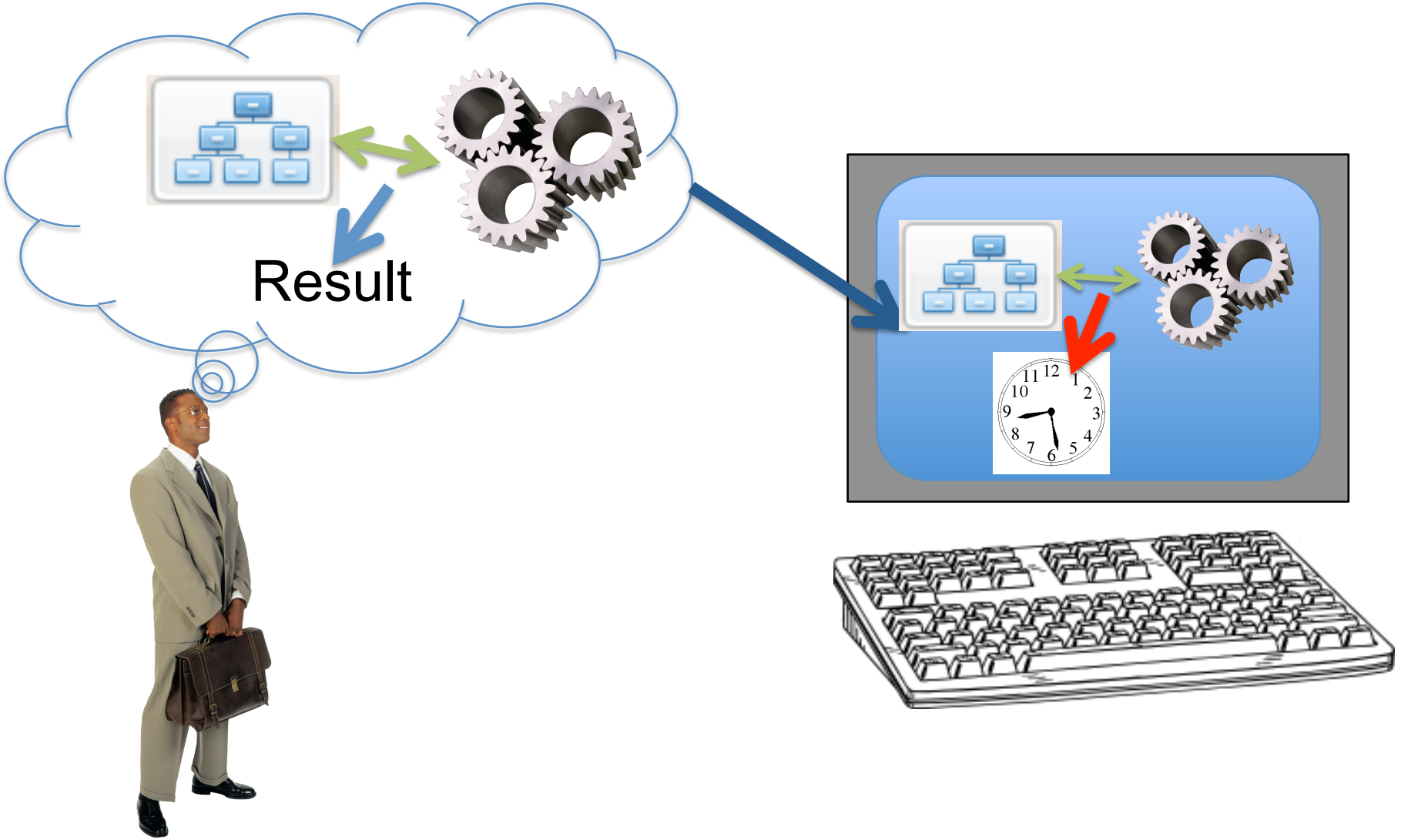
Domain-Specific Languages Increase Productivity



Time + Effort



DSLs Must Be Agile and Efficient



How to Implement DSLs?

Interpreters

+ Easy to write/modify

- Slow

Compilers

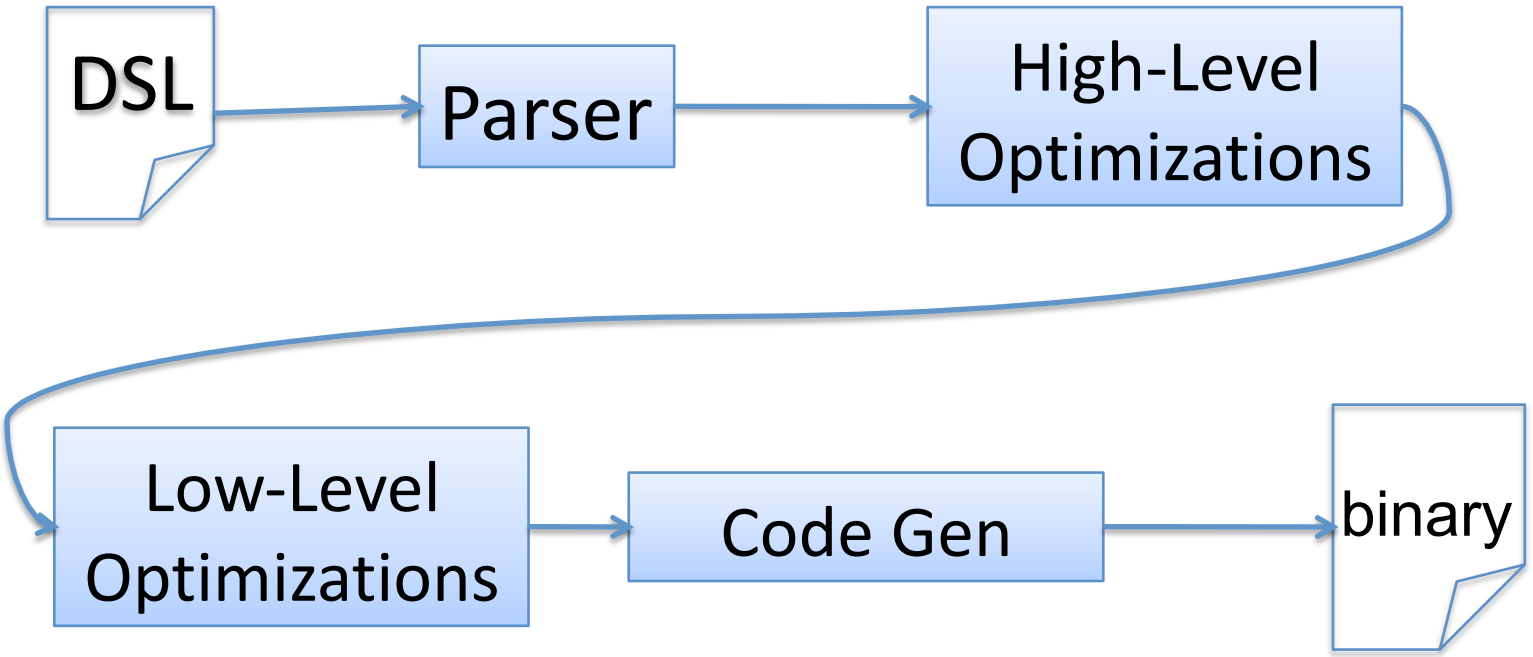
- Complex

+ Efficient code

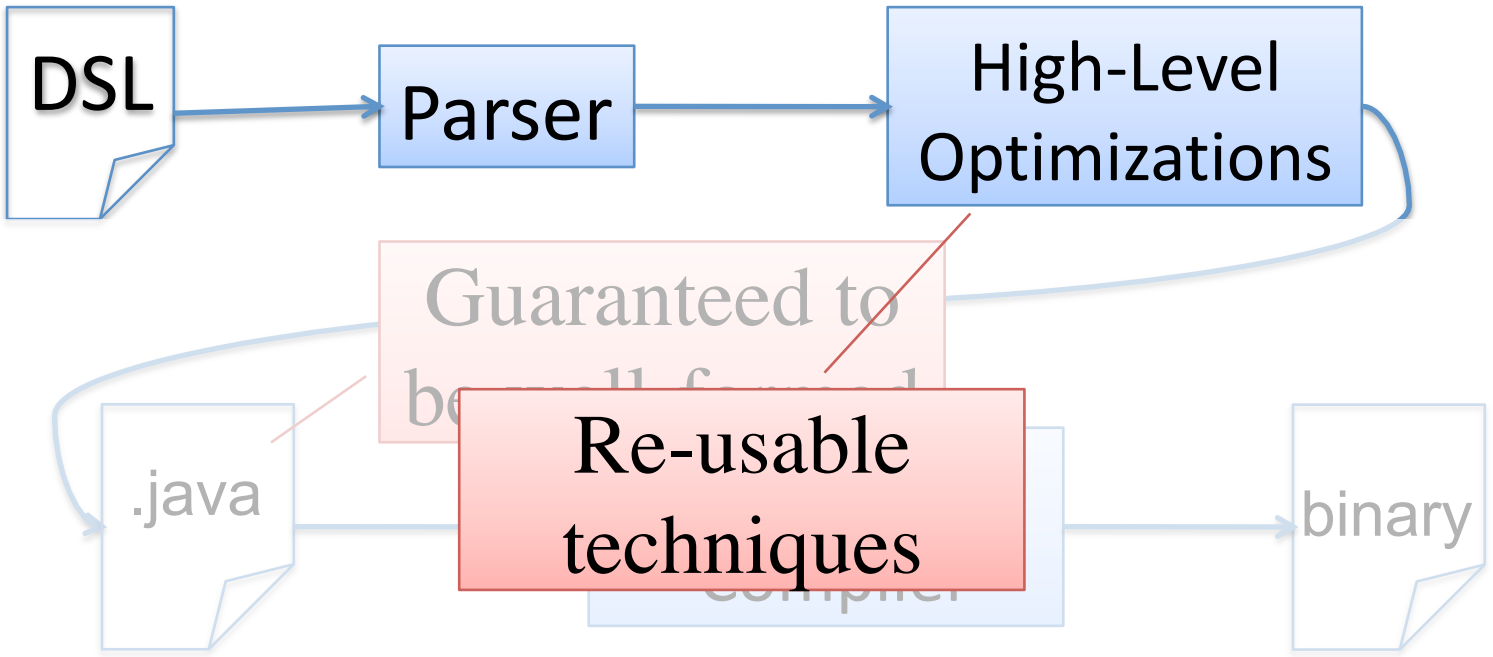
Why can't we have both?

Multi-Stage Programming

Compiler


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

Staged Interpreter in Mint



Outline

- Background: a simple interpreter
- What is MSP?
- Writing a staged interpreter
- Dynamic type inference / unboxing in MSP
- Staged interpreter for an image processing DSL
- Automatic loop parallelization in MSP

A Simple Dynamically-Typed PL

```
(Defun fact (x)
  (If (Equals (Var x) (Val (IntValue 0)))
     (Val (IntValue 1))
     (Mul (Var x)
          (App fact (Sub (Var x)
                         (Val (IntValue 1)))))))
```

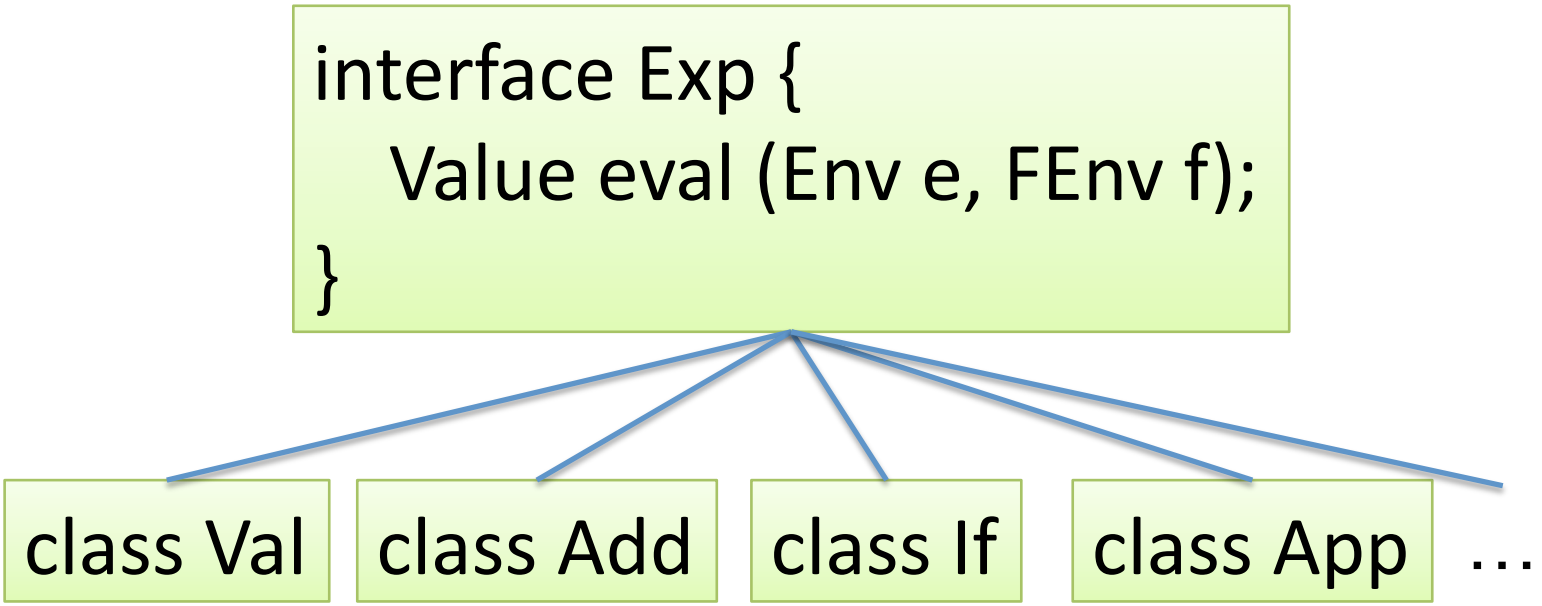
The Value Type

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

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

```
class BooleanValue {
    boolean b;

    // … similar …

}
```

Expressions and eval



Example Expression Classes

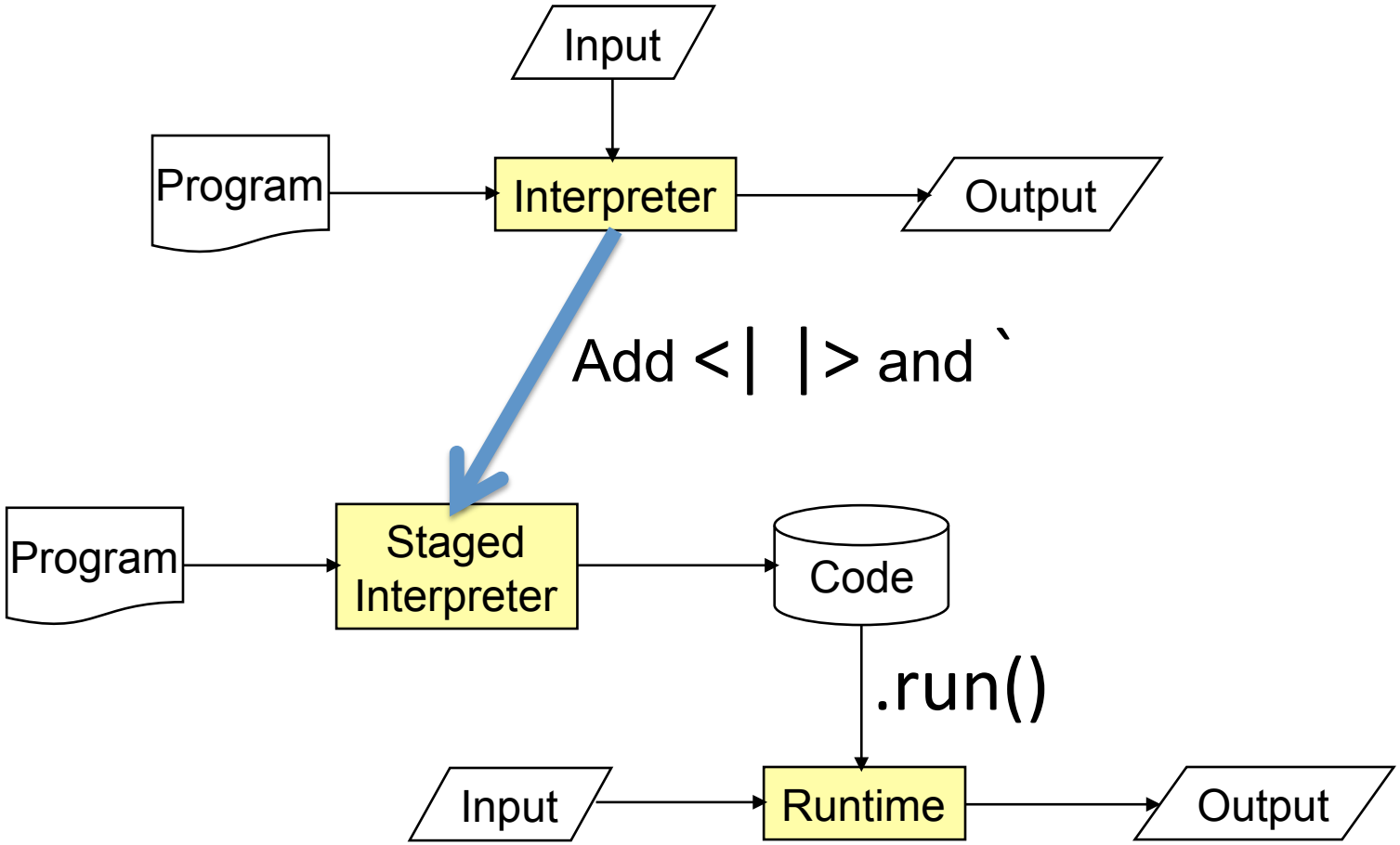
```
class Var implements Exp {
    String x;
    Value eval (Env e, FEnv f) {
        return e.get (x);
    }
}

class Add implements Exp {
    Exp left, right;
    Value eval (Env e, FEnv f) {
        return new IntValue ( left.eval(e,f).intValue() +
                              right.eval(e,f).intValue() );
    }
}
```

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

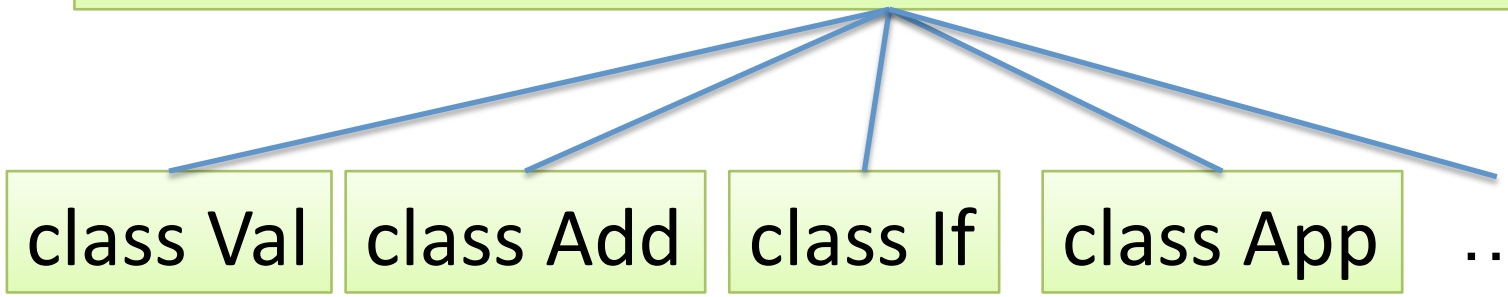
```
Code<Integer> x = <| 1 + 2 |>;
Code<Integer> y = <| (1 + 2) * 3 |>;
Integer z = y.run();   // z = 9
```

Staging the Interpreter



Staging the Interpreter

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



Staging the Expression Classes

```
class Var implements Exp {
    String x;
    separable Code<Value> eval (Env e, FEnv f) {
        return e.get (x);
    }
}
class Add implements Exp {
    Exp left, right;
    separable Code<Value> eval (Env e, FEnv f) {
        return <| new IntValue
                  (`(left.eval (e,f)).intValue () +
                   `(right.eval (e,f)).intValue ()) ) |>;
    }
}
```

Side Note: Weak Separability

- Escapes must be *weakly separable*
  - Prevents *scope extrusion* [Westbrook et al. '10]
- Limits the allowed assignments:
  - Variables with code must be block-local
- Can only call **separable** methods

Preventing Scope Extrusion

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

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

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

Side Note: Cross-Stage Persistance

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

v “persists” in generated code

Side Note: Cross-Stage Persistance

- CSP only allowed for *code-free* classes
  - Primitive types OR implementers of CodeFree
- Mint prohibits use of Code in CodeFree classes
- Prevents scope extrusion; see paper for details

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

DSL Implementation Toolchain

- Abstract `Compiler` and `Runner` classes
  - Template method design pattern
  - ~25 lines each, easily updated for new DSLs
  - Compiles code to a .jar file using Mint's save() method
  - Command-line:

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

Results

- Generated code (factorial):

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

- 3x speedup relative to unstaged (x = 10)
  - (2.67 GHz Intel i7, RHEL 5, 4 GB RAM)

Overhead of Unnecessary Boxing

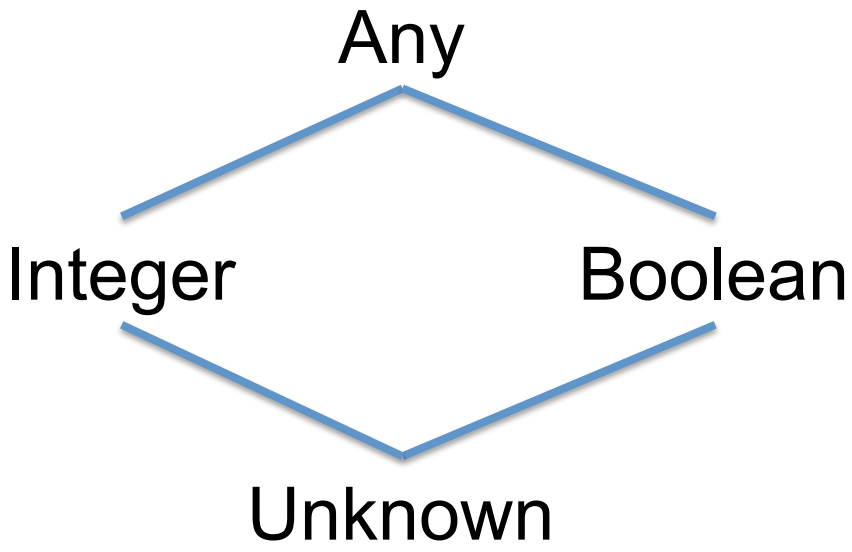
- Problem: eval() always returns Code<Value>:

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

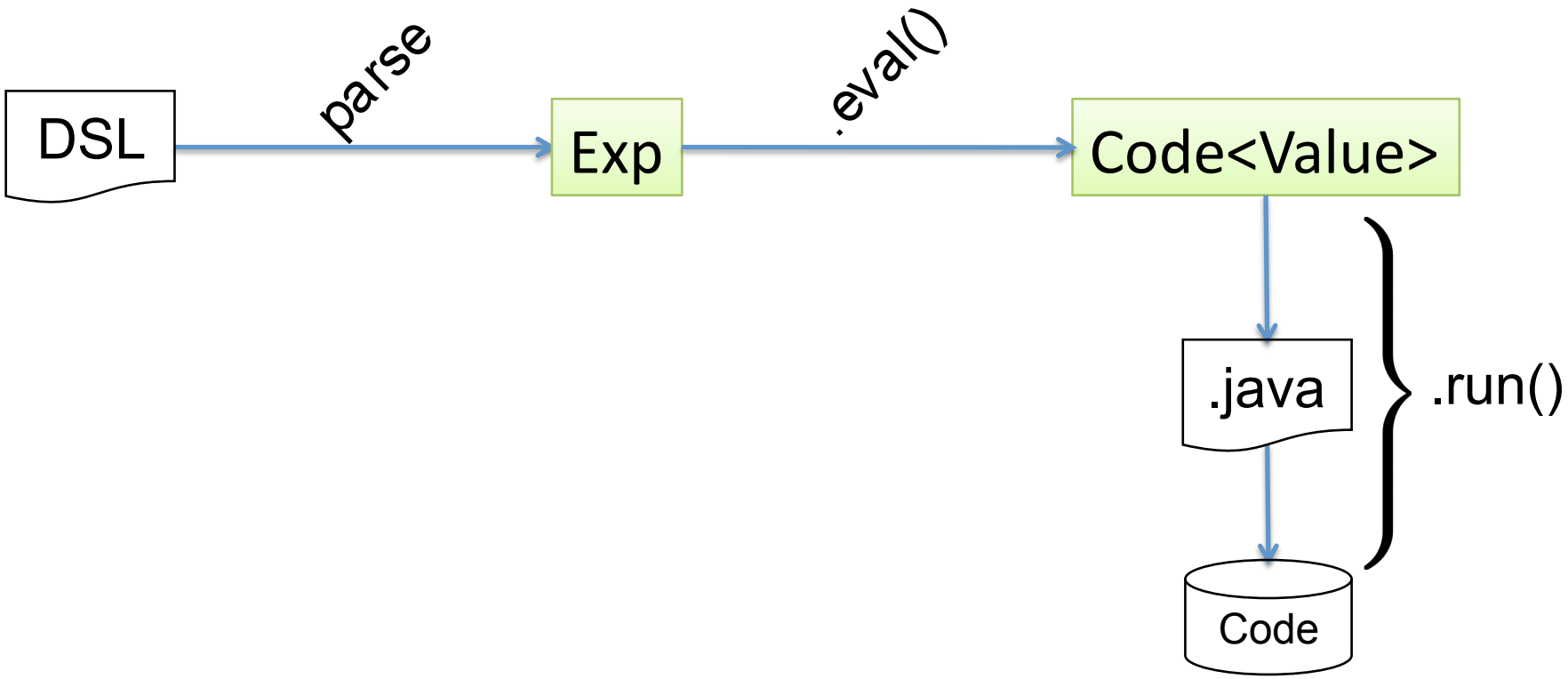
IntValue always built at runtime!

Solution: Dynamic Type Inference

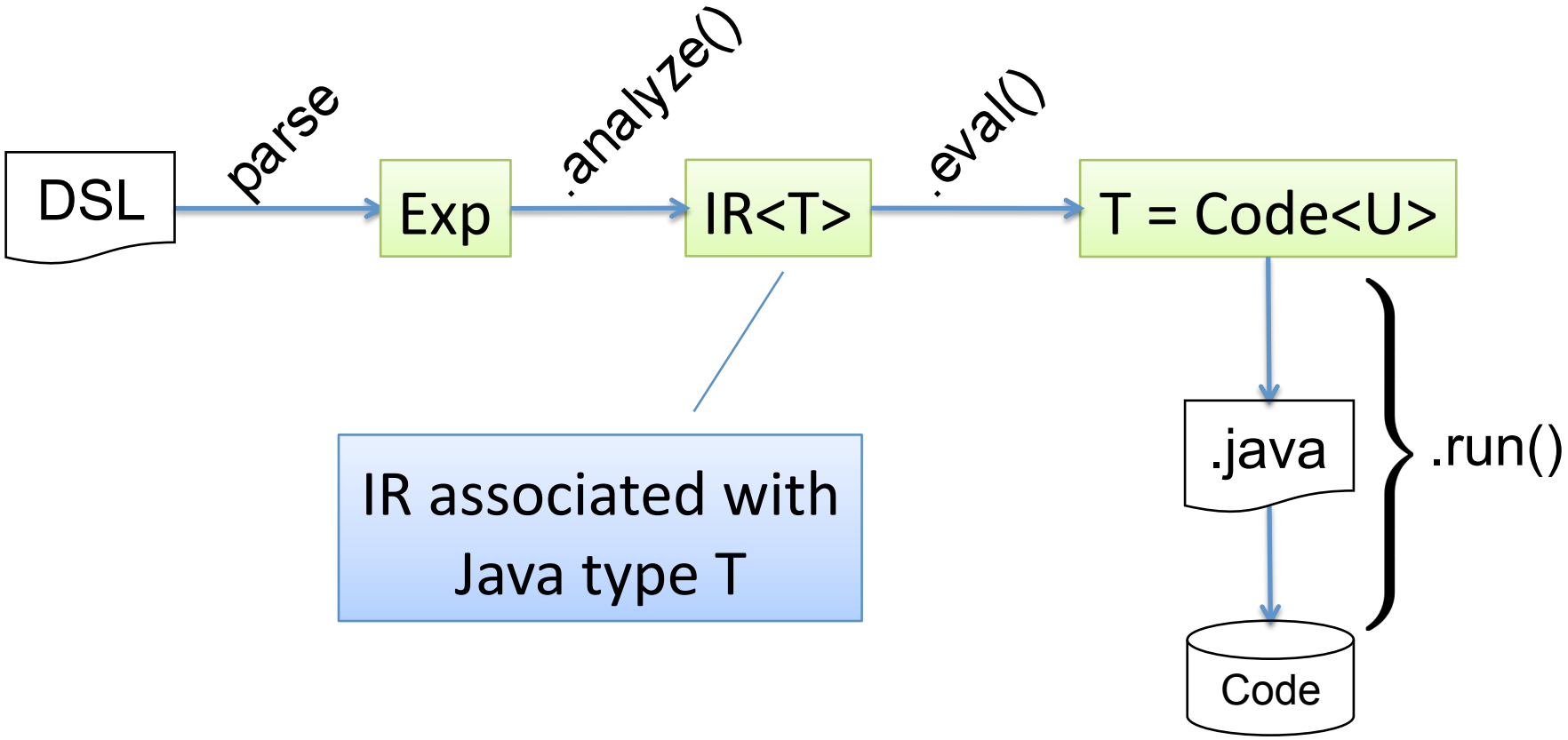
- Statically determine the type of an expression
- Elide coercions to/from boxed Value type
- Can be expressed as a dataflow problem:



Dynamic Type Inference in MSP

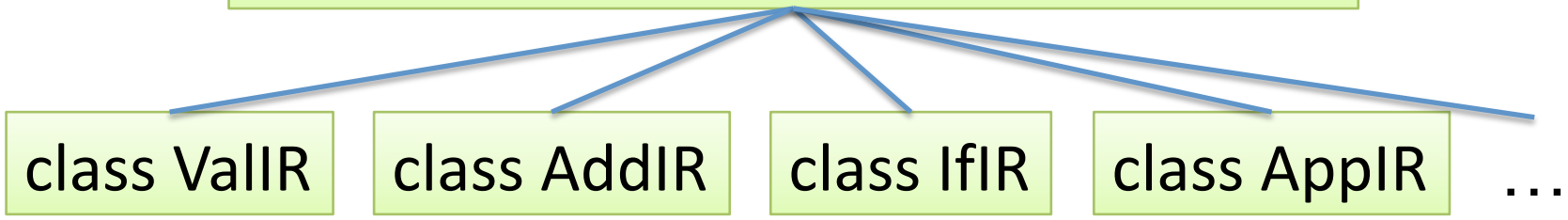


Dynamic Type Inference in MSP



Typed Intermediate Representation

```
abstract class IR<T> {
    DSLType<T> tp;
    separable T eval (Env e, FEnv f);
}
```



Example: AddIR

```
class AddIR extends IR<Code<Integer>> {
   IR<Code<Integer>> left, right;
   separable Code<Integer> eval (Env e, FEnv f) {
      return <| `(left.eval (e,f)) + `(right.eval (e,f)) |
>;
   }
}
```

No unnecessary boxing here!

Building IR Nodes: Analysis

```
abstract class Exp {
    IR<?> analyze (TEnv e, TFEnv f);
}
```

IR that produces some Java type

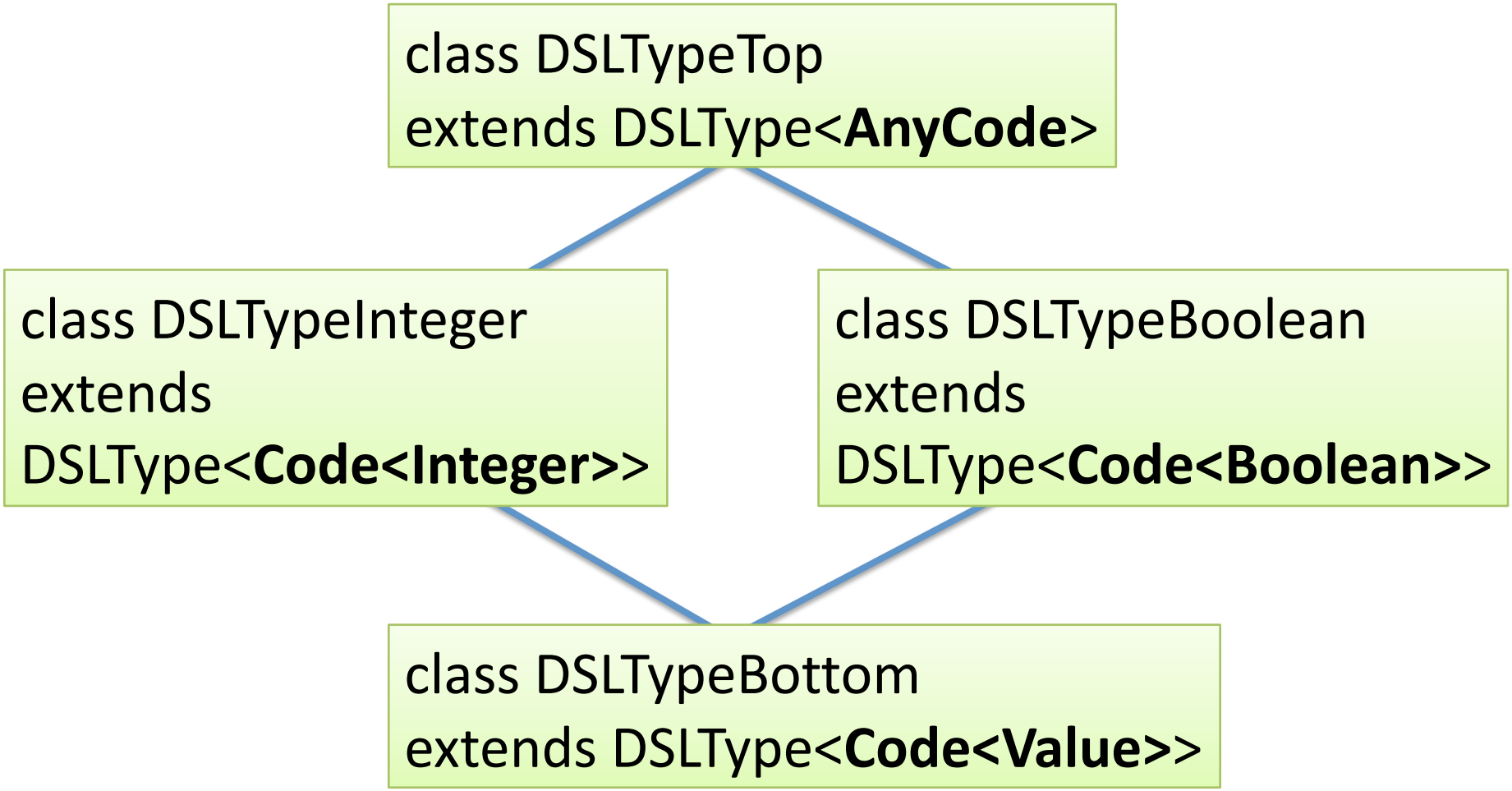
Types of variables and functions

Example: Add

```
class Add extends Exp {
    Exp left, right;
    separable IR<Code<Integer>>
    eval (TEnv e, TFEnv f) {
        return new AddIR
            (coerceToInt (left.analyze (e,f)),
             coerceToInt (right.analyze (e,f)));
    }
}
```

Coerce IR<?> to IR<Code<Integer>>

DSL Types → class DSLType<T>



Visitor Pattern = Typecase on IR<T>

```
interface Visitor<R> {
    R visitTop (IR<AnyCode> ir);
    R visitInteger (IR<Code<Integer>> ir);
    R visitBoolean (IR<Code<Boolean>> ir);
    R visitBottom (IR<Code<Value>> ir)
}
class DSLTypeInteger {
    R accept (IR<Code<Integer>> ir, Visitor<R> v) {
        v.visitInteger (ir);
}}
```

Return an R for any IR type

Pass ir to the correct method

Example: Coercing to Integers

```
<T> IR<Code<Integer>> coerceToInt (IR<T> ir) {
  return ir.tp.accept (ir, new Visitor<IR<Code<Integer>>>() {
    IR<Code<Integer>> visitTop (IR<An
      return new BoxAnyInt (ir);  }
    IR<Code<Integer>> visitInteger (IR<Code<Integer>> ir) {
      return ir; }
    IR<Code<Integer>>                    Code<Boolean>> ir) {
      return new ErrorIR<Integer,Boolean> (ir) }
    IR<Code<Integer>> visit                  e>> ir) {
      return new UnboxIntIR (ir); }
}); }
```

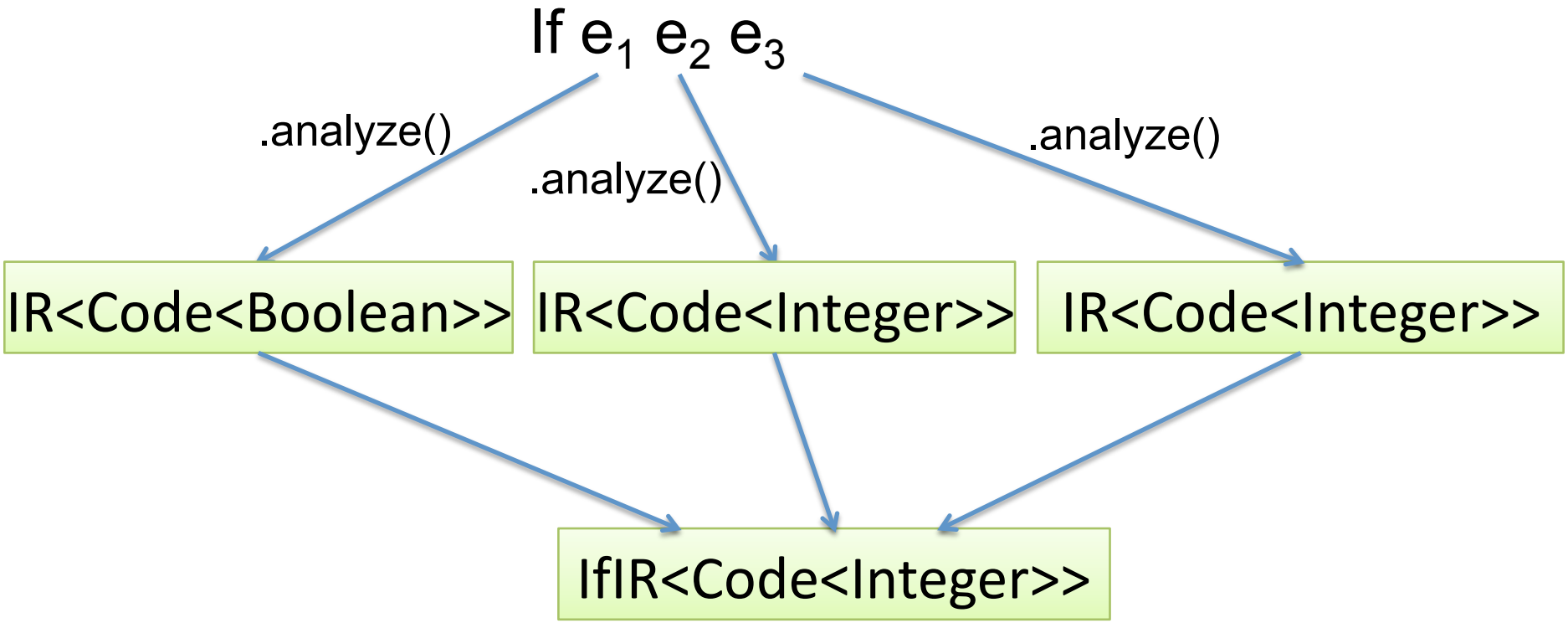
Cast AnyCode to Integer

Just return Integer

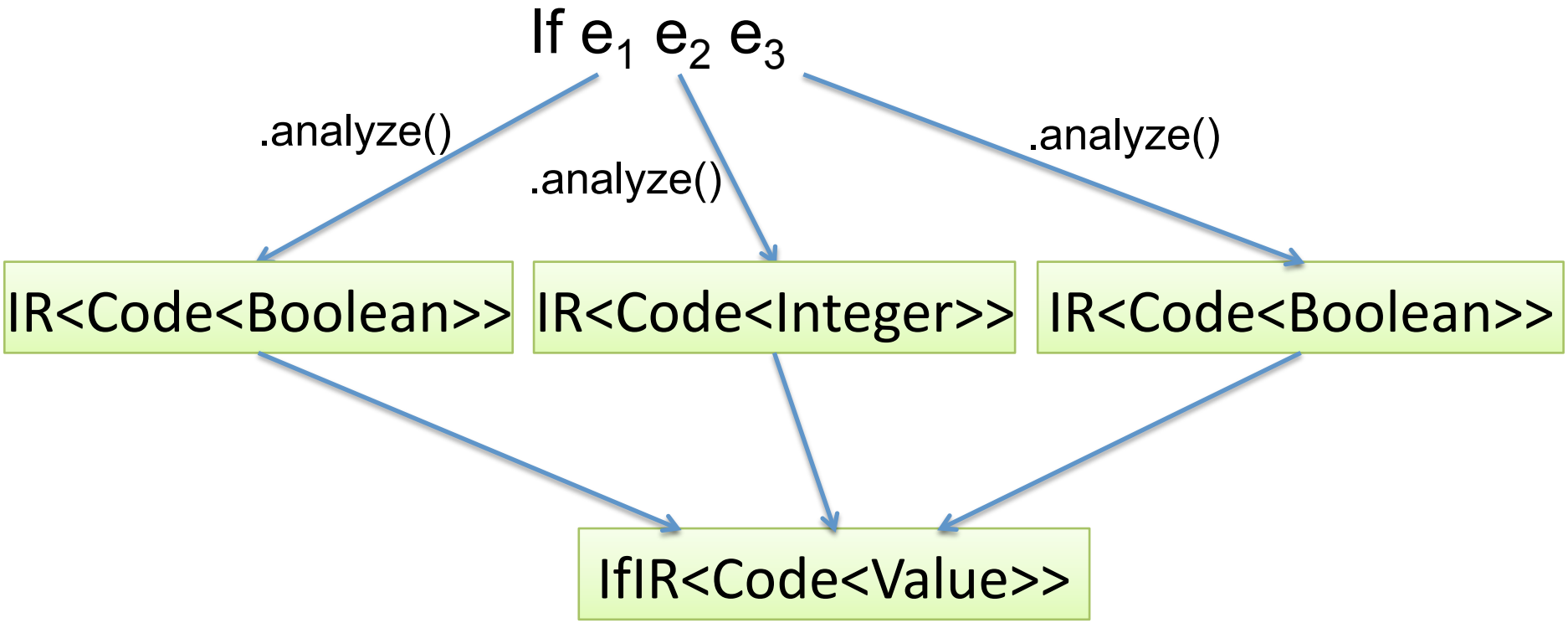
Throw errors for Boolean

Conditionally unbox Values

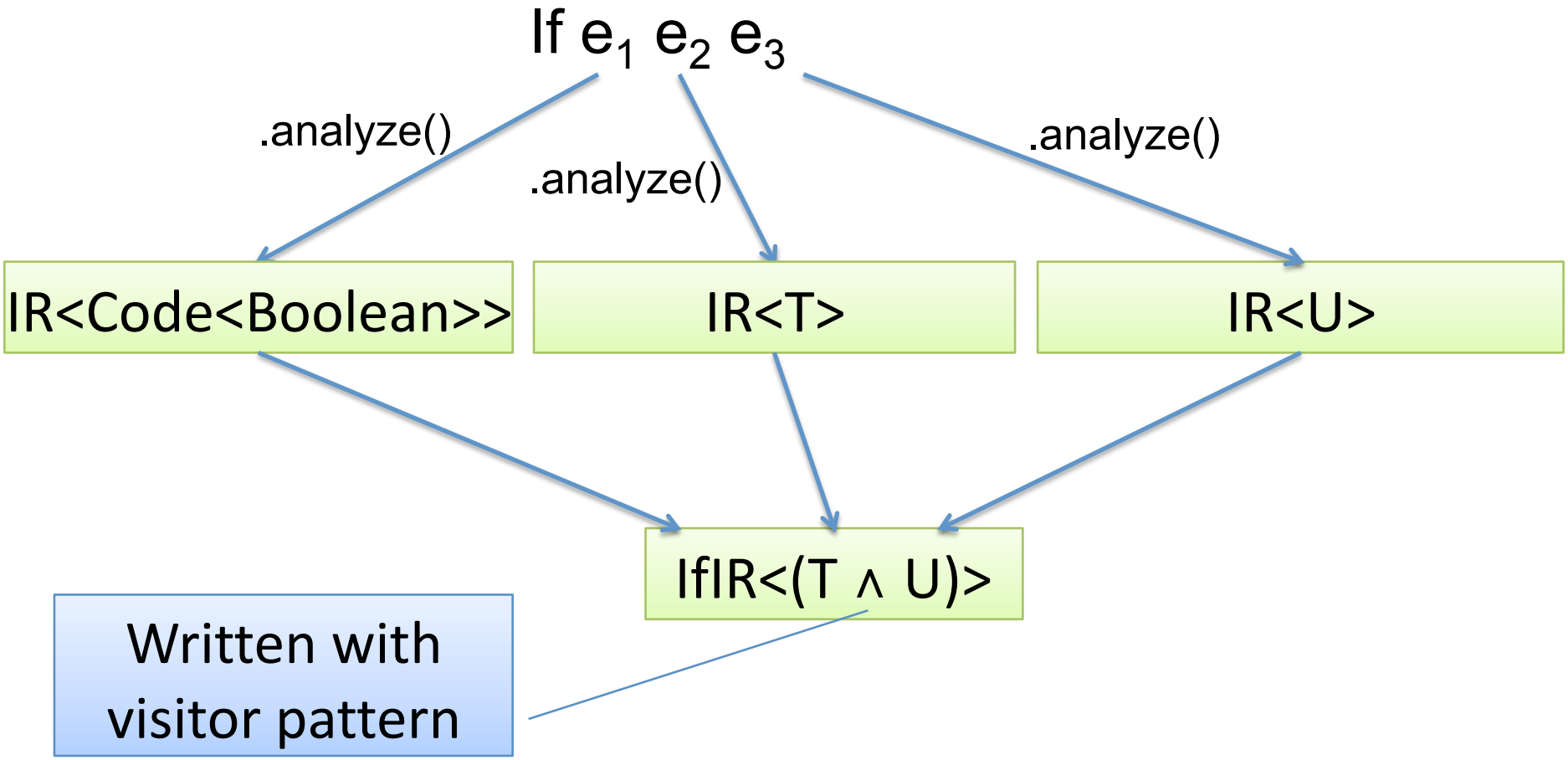
Staging Join Points



Staging Join Points



Staging Join Points



Per-Datatype Functions

(Defun fact (x) …)

```
new Fun() {
   applyInt (int x) {  …  }
   applyBool (boolean b) {  …  }
   applyValue (Value x) {
      /* applyInt or applyBool */
   }
}
```

Iterated Dataflow for Back Edges

(Defun T fact (I x)

I (If B (Equals I (Var x) I (Val (IntValue 0)))

I (Val (IntValue 1))

I (Mul I (Var x)

T (App fact I (Sub I (Var x)

I (Val (IntValue 1))))))

Iterated Dataflow for Back Edges

```
(Defun I fact ( I x)
I (If B (Equals I (Var x) I (Val (IntValue 0)))
   I (Val (IntValue 1))
   I (Mul I (Var x)
            I (App fact I (Sub I (Var x)
                                 I (Val (IntValue 1))))))
```

Results (Dynamic Type Inference):

- Generated code (factorial):

```
Integer applyInt (final int x) {
   final /* type omitted */ tthis = this;
   return (x == 0) ? 1 : (x * tthis.applyInt (x - 1));
}

Boolean applyBool (final boolean x) {
   final /* type omitted */ tthis = this;
   return (let l = x; let r = 0; false) ? 1 : /* ERROR */
}
```

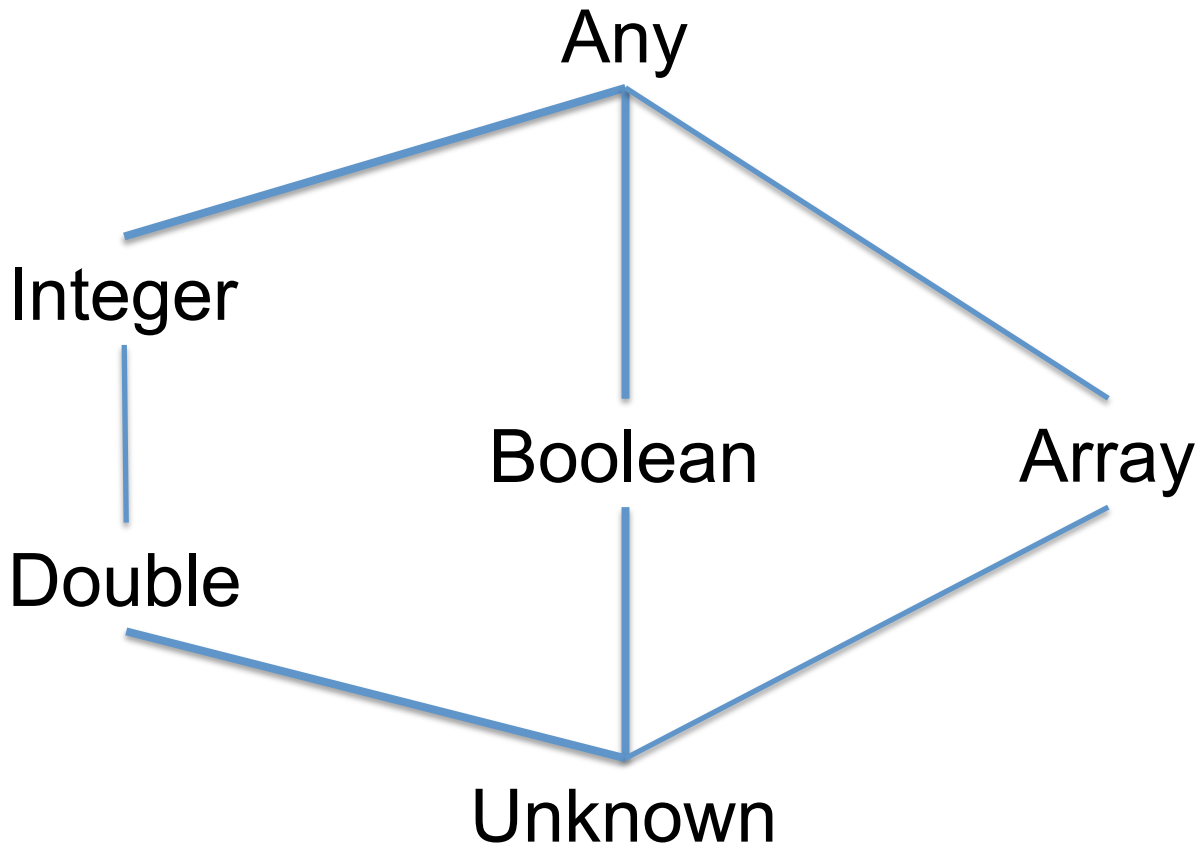
- 12x speedup relative to unstaged (x = 10)

  (2.67 GHz Intel i7, RHEL 5, 4 GB RAM)

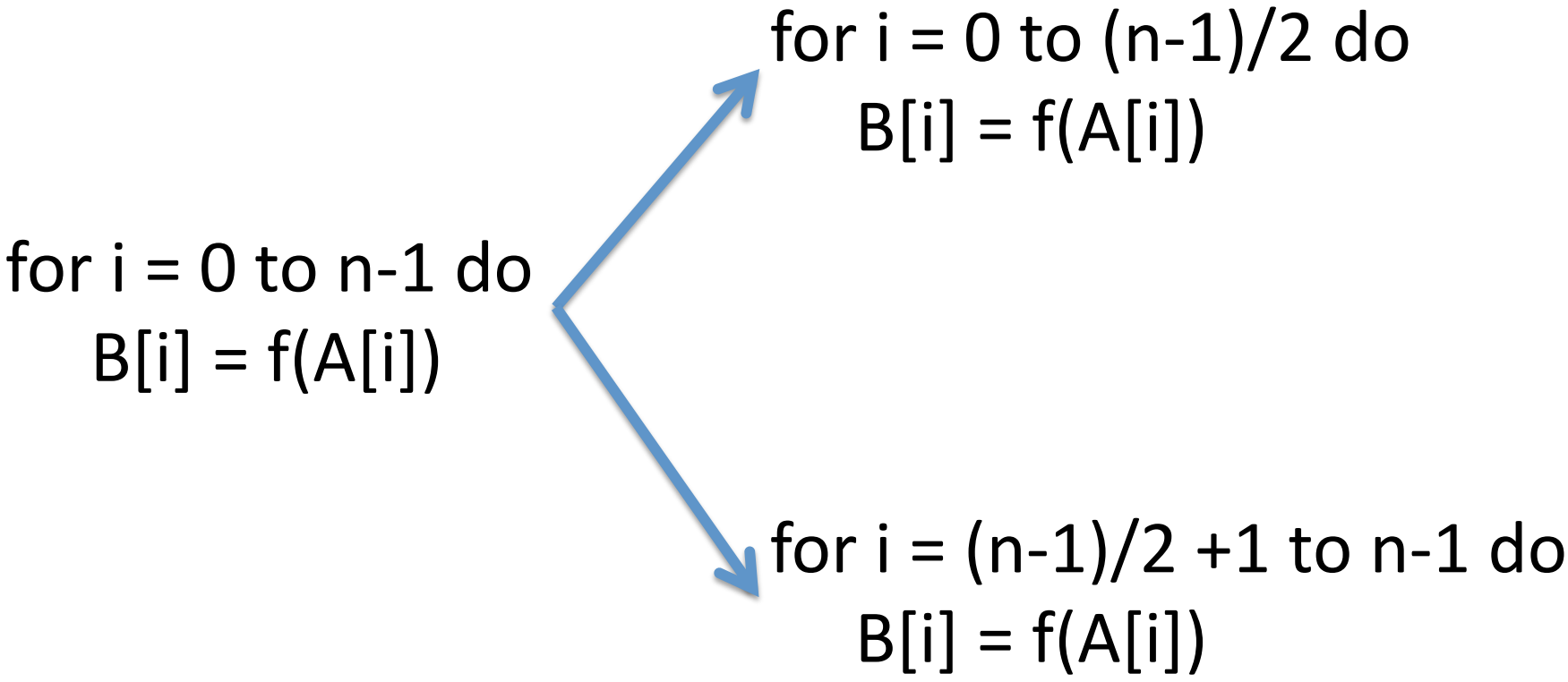
An Image Processing Language

- Double-precision floating point arithmetic
- Arrays
  - Allocate, get, set
- For loop
- Loading/displaying raster images

Adding New Types

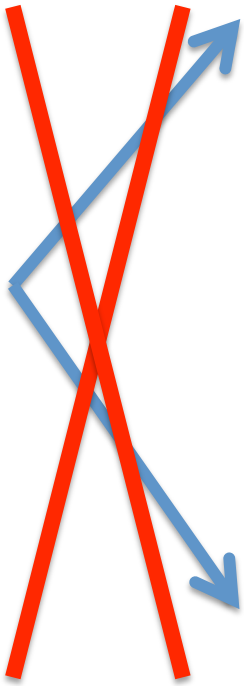


Automatic Loop Parallelization



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 } = ∅

Adding R/W Sets to the Analysis

```
class IR<T> {
    DSLType<T> tp;
    RWSet rwSet;
    separable T eval (Env e, FEnv f);
}
```

Staged Parallel For

```
public class ForIR extends IR<Code<Integer>> {
   //…
   public Code<Integer> eval (Env e, FEnv f) {
      return <| LoopRunner r = /* loop code */;
                `(rwOverlaps (e, _body.rwSet())) ?
                 runner.runSerial(…) :
                 runner.runParallel(…)  |>);
}
```

Inserts runtime check for whether
{ Reads R } ∩ { Writes W } = ∅

Test Program: Parallel Array Set

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

- 3.3x speedup relative to unstaged ($x = 10^5$)
- 2.4x speedup relative to staged serial
  - 2.67 GHz Intel i7, RHEL 5, 4 GB RAM

Test Results

Mandelbrot (200x200, max 256 iterations)

- 58x speedup relative to unstaged
- 2x speedup relative to staged serial

Gaussian Blur (2000x2000)

- 33x speedup relative to unstaged
- 1.5x speedup relative to staged serial

(2.67 GHz Intel i7, RHEL 5, 4 GB RAM)

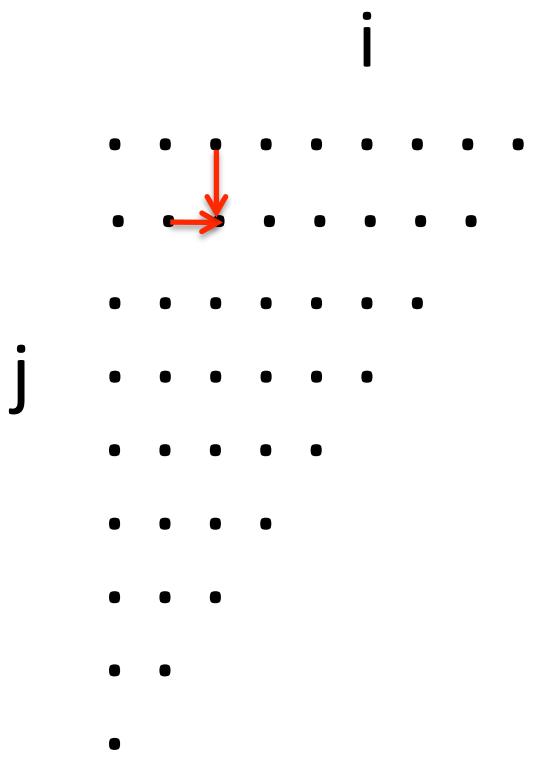
Conclusion

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

```
abstract class Exp {
    IR<?> analyze (TEnv e, TFEnv f, IterSpace i);
}
```

```
abstract class IR<T> {
    DSLType<T> tp;
    Deps deps;
}
```

i → [1, N)
j → [1, i)

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

Thank You!

http://www.javamint.org

Results (Loop Parallelization):

- Generated code (array set):

```
public Value apply(final int size) {
  return let final Value arr = new ArrayValue(
                    allocArray(size, new IntValue(1)));
  let final int notUsed = let LoopRunner r =
      new LoopRunner() { /* next slide */ };
  let int bound = size; let boolean runSerial = false;
  runSerial ? r.runSerial(bound) : r.runParallel(bound);
  arr;
}
```

Results (Loop Parallelization):

- Generated code (array set):

```
... let LoopRunner r = new LoopRunner() {
   public int run(int lower, int upper) {
      for (int i = lower; i < upper; ++i) {
        Value unused = new IntValue(
          setArray(arr.arrayValue(), i,
                        new DoubleValue(getRandom())));
      }
      return 0;
   } ...
}
```

Test Program: Mandelbrot

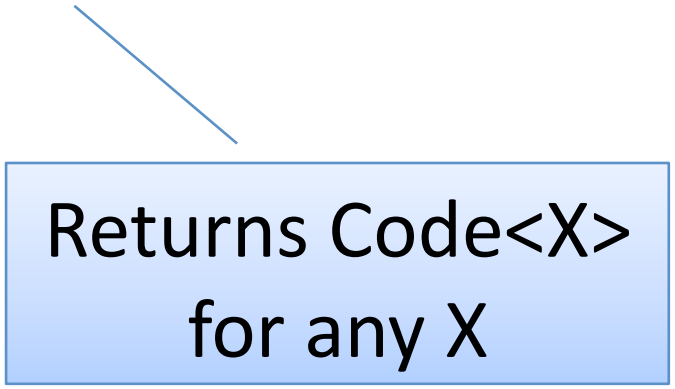
x and y loops can run in parallel

```
(Let …
  (For y (Var n)
    (For x (Var n) …
      (For k (Val (IntValue 256)) …
        (ASet (Var acc) … (Add (Val (IntValue 1))
                               (AGet (Var acc) …)))
        …
      (ASet (Var out) …)
```

k loop cannot run in parallel

The AnyCode Class

```
interface AnyCode {
  separable <X> Code<X> cast (Class<X> c);
}
```



MSP Applications

- Sparse matrix multiplication
- Array Views
  - Removal of index math

- Killer example: Staged interpreters