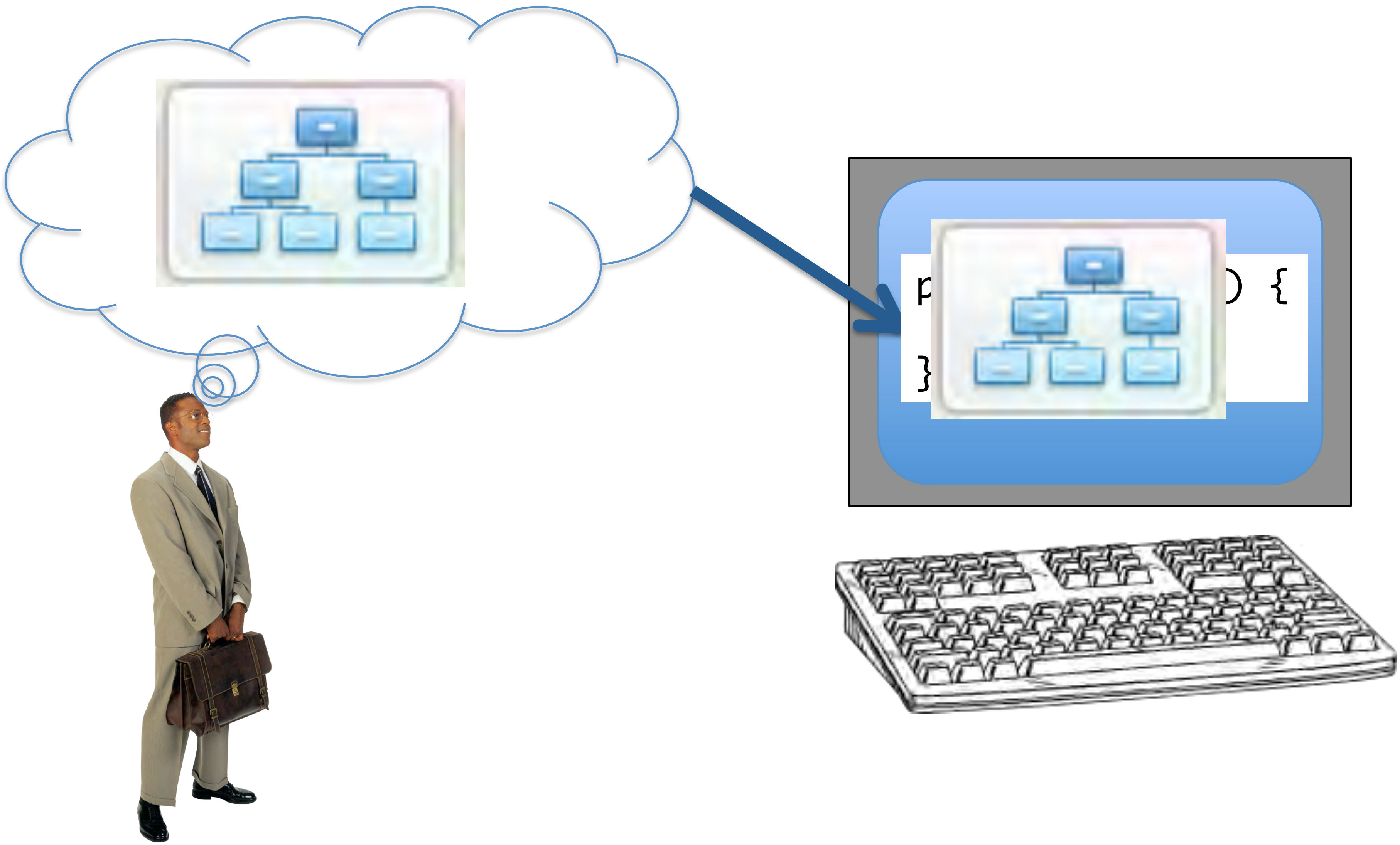
Agile and Efficient Domain-Specific Languages using Multi-stage Programming in Java Mint

Edwin Westbrook, Mathias Ricken,
and Walid Taha

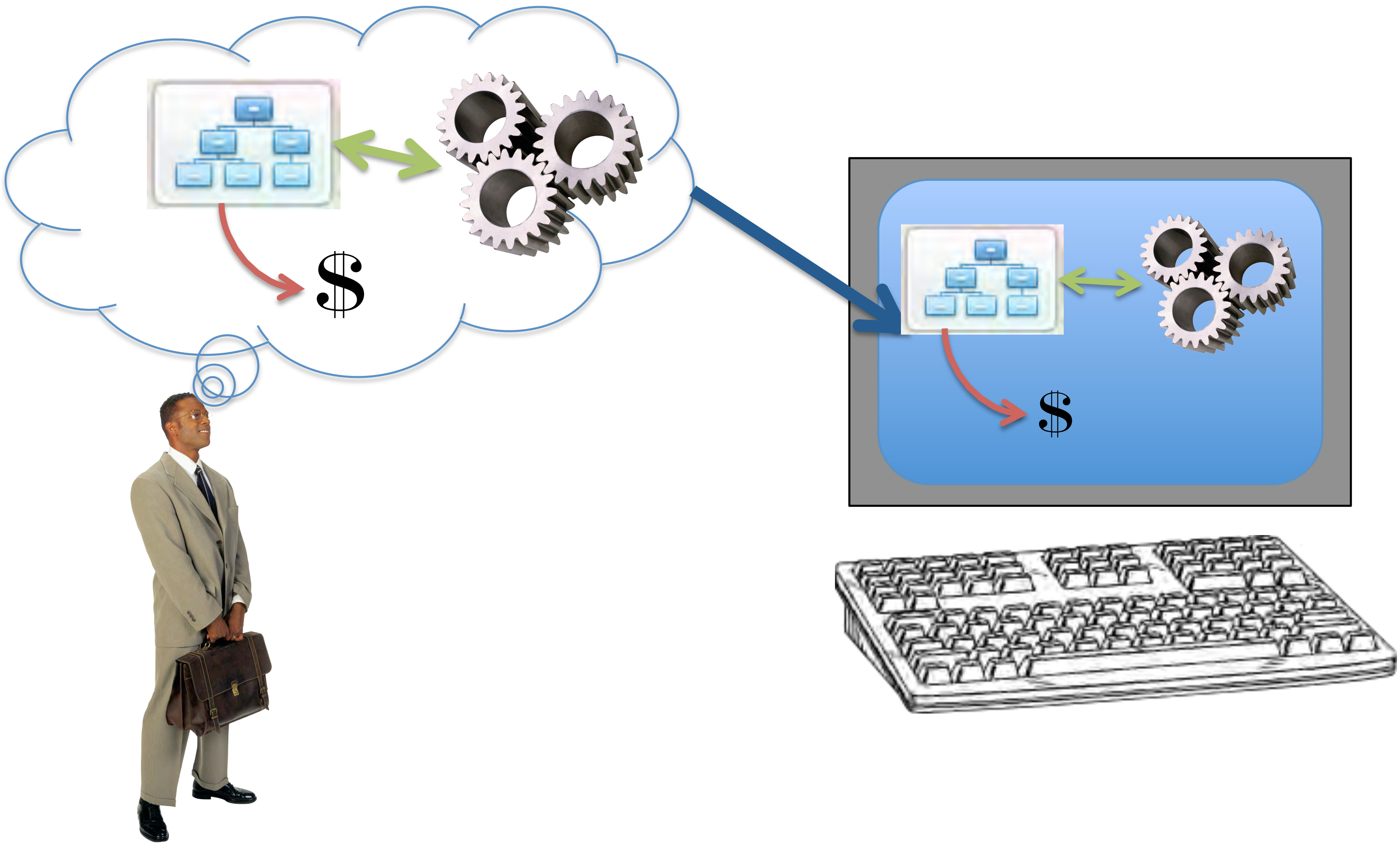
Why Domain-Specific Languages?



Benefits of DSLs:

- Productivity: write programs easier and faster
- Modularity: DSL implementation vs. programs
- Correctness: easier to see and fix bugs

DSLs Must Be Agile



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
  – No need to worry about compiler back-ends

Outline

- What is MSP?
- Writing a staged interpreter
- “Compiler optimizations” in MSP:
  - Dynamic type inference to reduce boxing/unboxing
  - Automatic loop parallelization
- Future Work: Monadic Staging

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

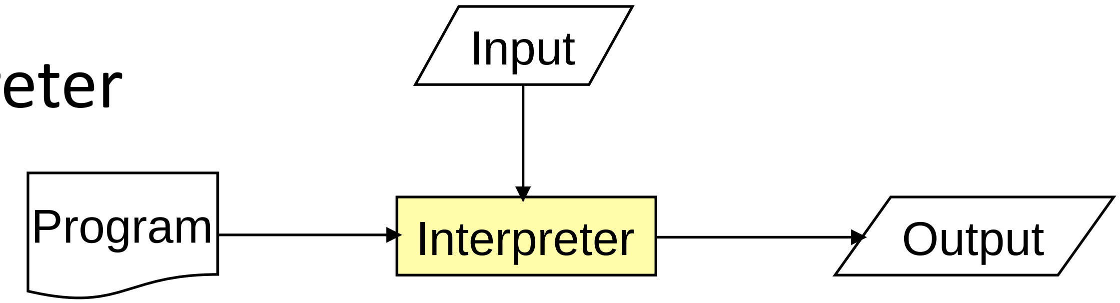
```
Code<Integer> x = <| 1 + 2 |>;
Code<Integer> y = <| `x * 3 |>;
Integer z = y.run(); // z = 9
```

MSP Applications

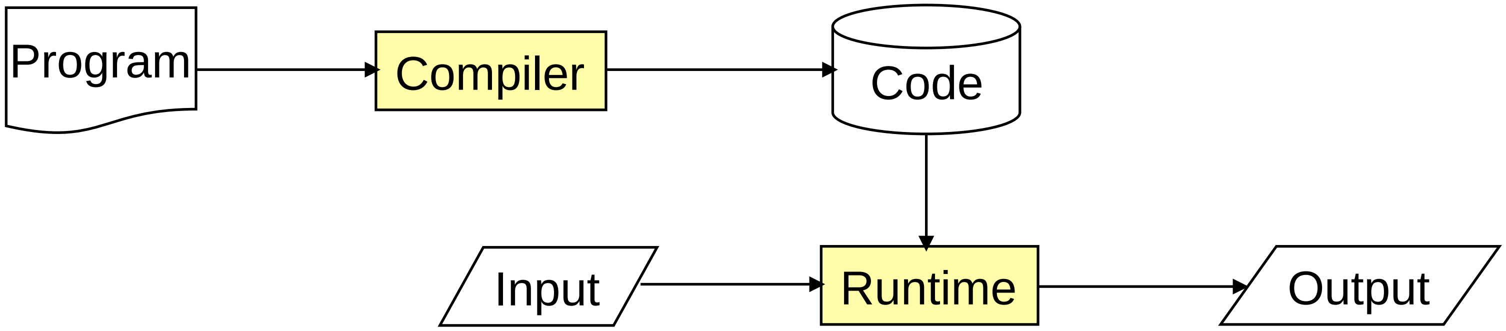
- Sparse matrix multiplication
  - Specialize for elements of value 0 or 1
- Array Views
  - Habanero Java's way of mapping multiple dimensions into 1-dimensional array
  - Removal of index math
- Killer example: Staged interpreters

Staging an Interpreter

- Unstaged interpreter

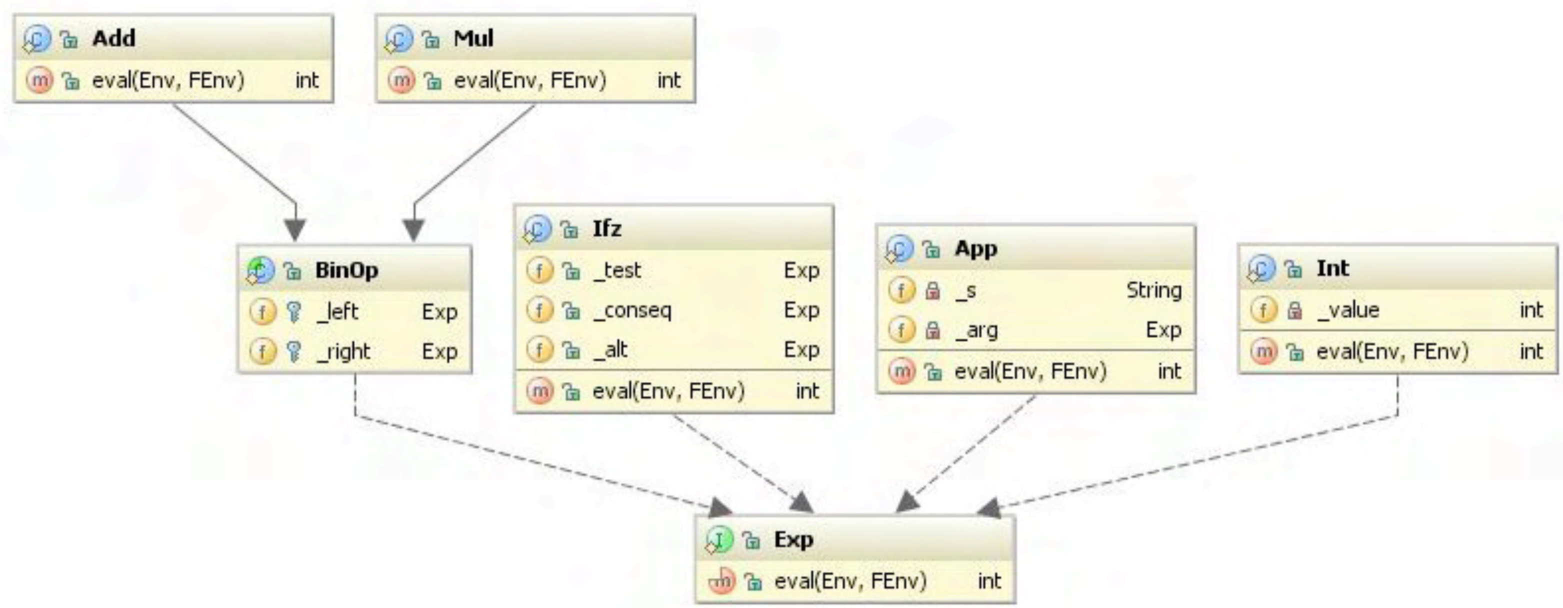


- Staged interpreter (compiler)



Unstaged Interpreter in Java

- Integer arithmetic, recursive unary functions
- Implementation
  - Interpreter and Composite design patterns



Unstaged Interpreter in Java

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

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

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

Using the Unstaged Interpreter

```
// evaluate an expression

Exp e = new Add(new Int(10), new Int(20));
int i = e.eval(env0, fenv0); // i = 10 + 20 = 30


// create a function that can be evaluated for any x

IntFun tenPlusX = new IntFun() {
    public int apply(int x) {
        Exp e = new Add(new Int(10), new Int(x));
        int i = e.eval(env0, fenv0);
        return i;
    }
};
int j = tenPlusX.apply(20); // j = 10 + 20 = 30
```

Staged Interpreter

```
public interface Exp {
    public separable Code<Integer> eval(Env e, FEnv f);
}
public class Int implements Exp {
    private Code<Integer> _v;
    public Int(final int value) { _v = <| value |>; }
    public separable Code<Integer> eval(Env e, FEnv f) {
        return _v;
    }
}
public class Add implements Exp {
    private Exp _left, _right;
    public Add(Exp l, Exp r) { _left = l; _right = r; }
    public separable Code<Integer> eval(Env e, FEnv f) {
        return <| `(_left.eval(e,f)) + `(_right.eval(e,f)) |>;
    }
}
```

Using the Staged Interpreter

```
// evaluate an expression

Exp e = new Add(new Int(10), new Int(20));
Code<Integer> c = e.eval(env0, fenv0); // c = <| 10+20 |>
int i = c.run(); // i = 30


// create a function that can be evaluated for any x

Code<IntFun> tenPlusXCode = <| new IntFun() {
    public int apply(final int x) {
        return `(new Add(new Int(10), new Int(<| x |>)).
                    eval(env0, fenv0)); }
} |>;

IntFun tenPlusX = tenPlusXCode.run();
int j = tenPlusX.apply(20); // j = 10 + 20 = 30
```

Using the Staged Interpreter

```
// evaluate an

Exp e = new A
Code<Integer                                             |>
int i = c.run();
```

```
<| new IntFun() {
    public int apply(final int x) {
        return 10 + x;
    }
} |>
```

```
// create a function th   can be evaluated for any x

Code<IntFun> tenPlusXCode = <| new IntFun() {
    public int apply(final int x) {
        return `(new Add(new Int(10), new Int(<| x |>)).
                 eval(env0, fenv0)); }
} |>;

IntFun tenPlusX = tenPlusXCode.run();
int j = tenPlusX.apply(20); // j = 10 + 20 = 30
```

Framework for DSL Implementation

- Reflection-based S-expression parser
  - Works for all our DSLs

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

- Abstract interpreter, compiler and runner
  - Work for all our DSLs
  - Template method design pattern to specify the variant behavior (e.g. which parser to use)

Compiler for Example DSL

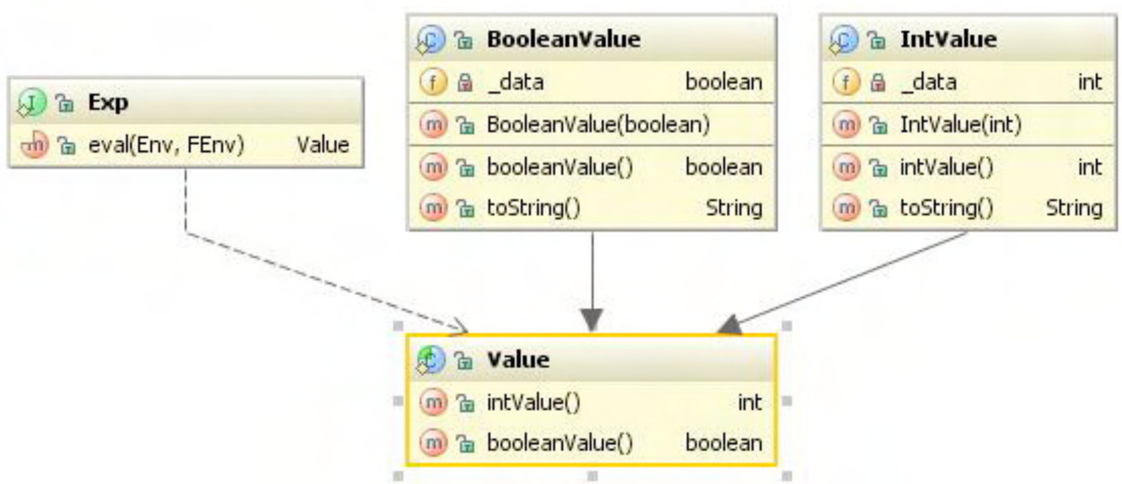
```
public class Compiler extends util.ACompiler<Exp,IntFun> {
    public Exp parse(String source) {
        return parser.LintParser.parse(Exp.class, source);
    }

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

Demo and Benchmarks: Integer Interpreter

Multiple Data Types

- Integer arithmetic, Boolean operations

- Unstaged interpreter returns Value
- Staged interpreter returns Code<Value>



Value Type and Subclass

```
public abstract class Value {
    public separable int intValue() { throw new Oops(); }
    public separable boolean booleanValue() { throw new Oops(); }
}

public class IntValue extends Value {
    private int _data;
    public IntValue(int data) { _data = data; }
    public separable int intValue() { return _data; }
    public String toString() { return _data+":IntValue"; }
}
```

Staged Interpreter with Value Type

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

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

Cannot move the value from stage 0 to stage 1 this way (using cross-stage persistence, CSP).

The Value type is not guaranteed to be code-free, and CSP of code could break type-safety (see Westbrook 2010).

Lifting for the Value Type

```
public abstract class Value { /* … */
    public separable abstract Code<Value> lift();
}

public class IntValue extends Value { /* … */
    private int _data;
    public separable Code<Value> lift() {
        final int lfData = _data;
        return <| new IntValue(lfData) |>;
    }
}

public class Val implements Exp { /* … */
    private Code<Value> _value;
    public Val(final Value value) { _value = value.lift(); }
}
```

CSP of int,
instead of
CSP of Value.

Demo and Benchmarks: Interpreter with Multiple Data Types

Overhead from Boxing/Unboxing

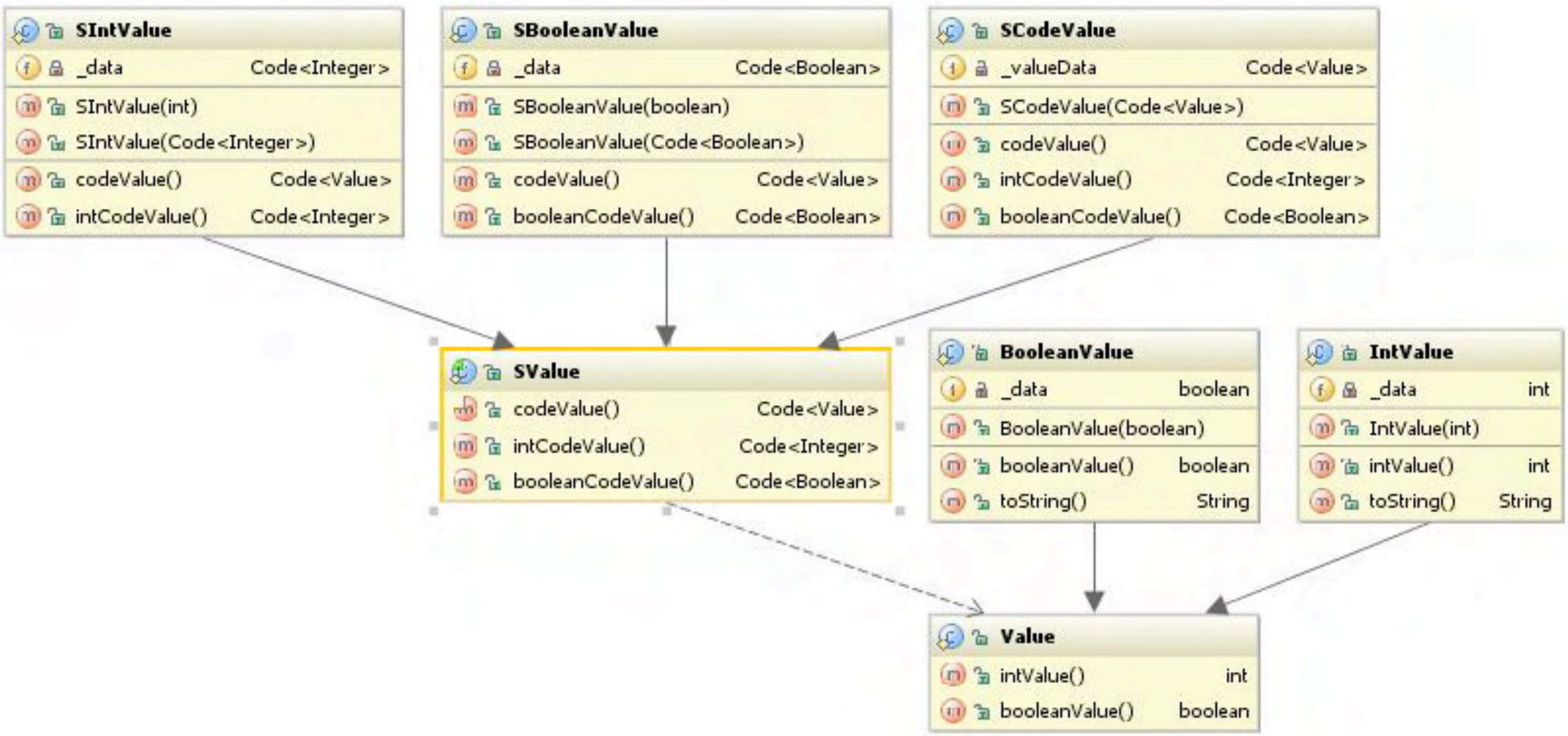
- eval() always returns Code<Value>:

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

Unnecessary in man cases!

Boxing/Unboxing Demo

Solution: “Staging the Value Type”



Solution: "Staging the Value Type"

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

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

Demo and Benchmarks: Dynamic Type Inference

Let Removes Code Duplication

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

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

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

Loss of type information!

```
public static class L
    // …

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

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

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);
    }
}
```

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);
    }
}
```

Automatic Loop Parallelization

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

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

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

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] = ...
```

$$\{ \text{Reads } R \} \cap \{ \text{Writes } W \} = \varnothing$$

Implementation: rwSet Method

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

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 ();
        }
    }
}

// similar for ASet
```

Demo and Benchmarks: Automatic Loop Parallelization

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);

    // …
}
```

Conclusion

- MSP for DSL implementation
  - Agile: as simple as writing an interpreters
  - Efficient: performance of compiled code
- Many compiler optimizations
- Framework for implementing DSLs
- Available for download:

  http://www.javamint.org