RICE

Mint:
A Multi-stage Extension of Java

COMP 600

Mathias Ricken
Rice University

February 8, 2010

Multi-stage Programming

- Multi-stage programming (MSP) languages
  - Provide constructs for program generation
  - Statically typed: do not delay error checking until runtime
- Useful for program specialization
  - Optimizing a program for certain values
- Abstractions without the cost

MSP in Java

- Brackets delay computation, yield a value of type `Code<?>`
  `<| e |>` or `<| { s; s; } |>`

- Escapes stitch together pieces of code
  `` `c ``

- Run executes a piece of code, returning result
  `c.run()`

- Ex:

```
Code<Integer> x = <| 2 + 3 |>;
Code<Integer> y = <| 1 + `x |>;
int z = y.run(); // == 6
```

Unstaged power Function

```
double power(double x, int n) {
  double acc = 1;
  for(int i=0; i<n; ++i) acc = acc * x;
  return acc;
}
double d = power(2, 4);
```

Result: 16

- Overhead due to loop
  - Faster way to calculate x^4: `x*x*x*x`
  - Don't want to write x^2, x^3, x^4… by hand

Unstaged/Staged Comparison

```
double power(double x, int n) {
  double acc = 1;
  for(int i=0; i<n; ++i)
    acc = acc * x;
  return acc;
}
```

```
Code<Double> spower(Code<Double> x,int n) {
  Code<Double> acc = <|1|>;
  for(int i=0; i<n; ++i)
    acc = <| `acc * `x |>;
  return acc;
}
```

Staged power Function

```
Code<Double> spower(Code<Double> x,int n) {
  Code<Double> acc = <|1|>;
  for(int i=0; i<n; ++i)
    acc = <| `acc * `x |>;
  return acc; }
```

```
Code<Double> c = spower(<|2|>, 4);
```

Result: `<| (((1 * 2) * 2) * 2) * 2 |>`

```
Double d = c.run();
```

Result: 16

Staged power Function

```
Code<? extends Lambda> codePower4 = <|
  new Lambda() {
    public Double apply(final Double x) {
      return `(spower(<|x|>, 4));
//    return `(<| (((1*x)*x)*x)*x |>);
//    return       (((1*x)*x)*x)*x;
    }
} |>;
Lambda power4 = codePower4.run();

Double d = power4.apply(2);
```

Result: 16

Effects: Assignment

- Imperative languages allow side effects
- Example: Assignment

`<| y |>`

```
Code<Integer> x;
<| {
   Integer y = foo();
   `(x = <| y |>);
} |>.run();
Integer i = x.run();
```

`y`

y used out of scope!

Effects: Exceptions

```
Code<Integer> foo(Code<Integer> c) {
  throw new CodeContainerException(c);
}

try {
<| { Integer y; `(foo(<|y|>)); } |>.run();
}
catch(CodeContainerException e) {
  Code<Integer> c = e.getCode();
  Integer i = c.run();
}
```

<| y |>

y

y used out of scope!

Scope Extrusion

- Side effects involving code
  - Can move a variable access outside the scope where it is defined
  - Executing that code would cause an error
- Causes
  - Assignment of code values
  - Exceptions containing code
  - Cross-stage persistence (CSP) of code [1]

Weak Separability

- Prohibits side effects involving code in an escape to be visible from the outside

- Restricts code generators, not generated code
  - Escapes must be weakly separable
  - Generated code can freely use side effects

Weak vs. Strong Separability

- (Strong) separability condition in Kameyama'08,'09
  - Did not allow any side effects in an escape to be visible outside
- Weak separability is more expressive
  - Allow code-free side effects visible outside
  - Useful in imperative languages like Java

Weakly Separable Terms

- A term is weakly separable if…
  - Assignment only to code-free variables [2]
  - Exceptions thrown do not have constructors taking code [3]
  - CSP only for code-free types

  - Only weakly separable methods and constructors called (`separable` modifier)
  - Only weakly separable code is stitched in (`SafeCode` as opposed to `Code`)

Evaluation

- Formalism
  - Prove safety
- Implementation
  - Evaluate expressivity
  - Benchmarks to compare staging benefits to known results from functional languages

Lightweight Mint

- Developed a formalism based on Lightweight Java (Strniša'07)
  - Proves that weak separability prevents scope extrusion
- Fairly large to model safety issues
  - Models assignment, staging constructs, anonymous inner classes

- Many other imperative MSP systems do not have formalisms

Implementation

- Based on the OpenJDK compiler
  - Java 6 compatible
  - Cross-platform (needs SoyLatte on Mac)
- Modified compiler to support staging annotations
- Invoke compiler at runtime

Compiler Stages

- Compile time
  - Generate bytecode to create ASTs for brackets
  - Safety checks enforcing weak separability
- Runtime
  - Create AST objects where brackets are found
  - Compile AST to class files when code is run
    - Serialize AST into a string in memory
    - Pass to javac compiler
    - Load classes using reflection

Expressivity

- Staged interpreter
  - **lint** interpeter (Taha'04)
  - Throws exception if environment lookup fails

- Staged array views

Unstaged Interpreter

```
interface Exp {
  public int eval(Env e, FEnv f);
}
class Int implements Exp {
  private int _v;
  public Int(int value ) { _v = v; }
  public int eval(Env e, FEnv f) { return _v; }
}
class App implements Exp {
  private String _s;
  private Exp _a; // argument
  public App(String s, Exp a) { _s = s; _a = a; }
  public int eval(Env e, FEnv f) {
    return f.get(_s).apply(_a.eval(e,f));
  }
}
```

Staged Interpreter

```
interface Exp {
  public separable
  SafeCode<Integer> eval(Env e, FEnv f);
}
class Int implements Exp { /* ... */
  public separable
  SafeCode<Integer> eval(Env e, FEnv f) {
    final int v = _v; return <| v |>;
  }
}
class App implements Exp { /* ... */
  public separable
  SafeCode<Integer> eval(Env e, FEnv f) {
    return
      <| `(f.get(_s)).apply(`(_a.eval(e,f))) |>;
  }
}
```

Staged Environment

```
static separable Env ext(final Env env,
    final String x, final SafeCode<Integer> v) {
  return new Env() {
    public separable
    SafeCode<Integer> get(String y) {
      if (x==y) return v;
      else return env.get(y);
    }
  };
}

static Env env0 = new Env() {
  public separable SafeCode<Integer> get(String y) {
    throw Yikes(y);
  }
}
```

Throw an exception.

Can’t be done safely in other MSP systems.

Expressivity

- Staged interpreter
  - Throws exception if environment lookup fails

- Staged array views
  - ➢ HJ's way of mapping multiple dimensions into a 1-dimensional array (Shirako'07)
  - Removal of index math
  - Loop unrolling
  - Side effects in arrays

Unstaged Array Views

```
class DoubleArrayView {
  double[] base;
  //...
  public double get(int i, int j) {
    return base[offset + (j-j0)
                     + jSize*(i-i0)];
  }
  public void set(double v, int i, int j) {
    base[offset + (j-j0)
              + jSize*(i-i0 )] = v;
  }
}
```

Staged Array Views

```
class SDoubleArrayView {
  Code<double[]> base;
  //...
  public separable
  Code<Double> get(final int i, final int j) {
    return <| `(base)[`offset + (j-`j0)
                               + `jSize*(i-`i0)] |>;
  }
  public separable
  Code<Void> set(final Code<Double> v,
                 final int i, final int j) {
    return <| {
      `(base)[`offset + (j-`j0) +
                          `jSize*(i-`i0)] = `v; } |>;
  }
}
```

Using Staged Array Views

Much more convenient in Java than previous MSP systems.

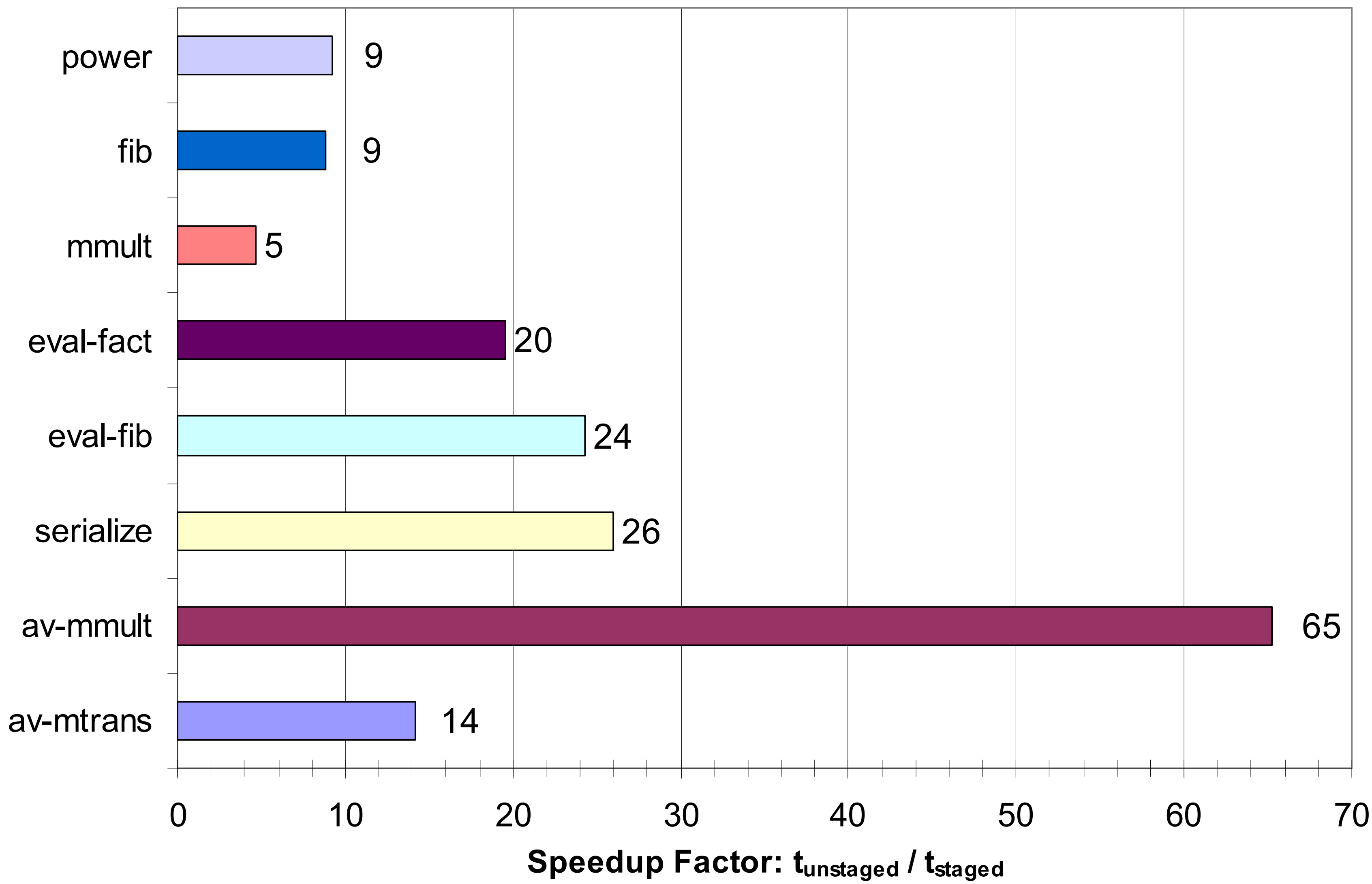
```
    final SDoubleArrayView input,
    final SDoubleArrayView output) {
  Code<Void> stats = <| { } |>;
  for (int i = 0; i < m; i ++)
  for (int j = 0; j < m; j ++)
    stats = <| {
      `stats;
      `(output.set(input.get(i,j),j,i));
    } |>;
  return stats;
}
Code<Void> c = stranspose(4, 4, a, b);

// Generates code like this
b [0+(0-0)+4*(0-0)] = a [0+(0-0)+4*(0-0)];
b [0+(0-0)+4*(1-0)] = a [0+(1-0)+4*(0-0)];//...
```

Loop unrolling using for-loop.

Side effects in arrays.

Can't be done in other MSP systems.

Benchmarks



Apple MacBook 2.0 GHz Intel Core Duo 2 MB L2 cache, 2 GB RAM, Mac OS 10.4

Future Work

- Speed up runtime compilation
  - Use NextGen template class technology (Sasitorn'06)
  - Compile snippets statically, link together at runtime
- Avoid 64 kB method size JVM limit
- Cooperation with Habanero Group
  - Integrate staged array views into HJ http://habanero.rice.edu/
- Integrate with Closures for Java? http://javac.info/

Conclusion

- Statically-typed safe MSP for imperative languages
- More expressive than previous systems
- Implementation based on OpenJDK
- Java benefits from staging as expected

Thank You

- Weak separability:

  safe, expressive multi-stage programming in imperative languages

- Download

  http://mint.concutest.org/

- Thanks to my co-authors Edwin Westbrook, Jun Inoue, Tamer Abdelatif, and Walid Taha, and to my advisor Corky Cartwright

Footnotes

Footnotes

1. Scope extrusion by CSP of code, see extra slide.
2. Assignment only to code-free variables, unless the variables are bound in the term.
3. Exceptions thrown may not have constructors taking code, unless the exception is caught in the term.

Extra Slides

Unstaged power in MetaOCaml

```
let rec power(x, n) = if n=0
  then 1 else x*power(x, n-1);;
```

```
power(2, 4);;
```

Result: 16

- Overhead due to recursion
  - Faster way to calculate x^4: `x*x*x*x`
  - Don't want to write x^2, x^3, x^4… by hand

Staged power in MetaOCaml

```
let rec spower(x, n) = if n=0
  then .<1>.
  else .< .~(x) * .~(power(x, n-1)) >.;;

let c = spower(.<2>., 4);;
```

Result: `.< 2 * (2 * (2 * (2 * 1))) >.`

```
let d = .! c;;
```

Result: 16

Staged power in MetaOCaml

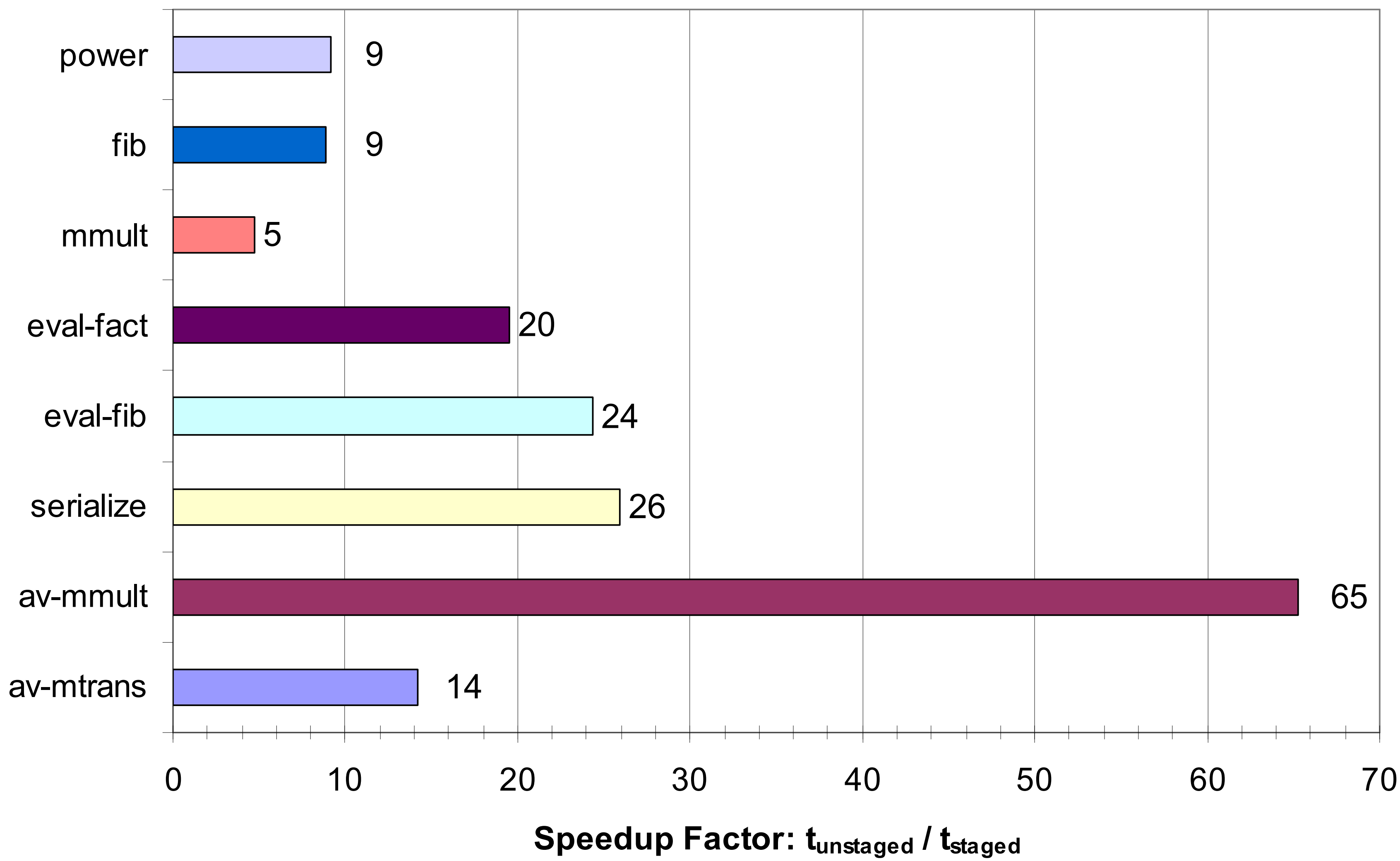
```
let codePower4 =
  .< fun x -> .~(spower(.<x>., 4)) >.;;
//.< fun x -> .~(.< x*(x*(x*(x*1))) >.) >.;;
//.< fun x -> x*(x*(x*(x*1))) >.;;


let power4 = .! codePower4;;

power4(2);
```

Result: 16

Scope Extrusion by CSP of Code

```
interface IntCodeFun {
  Code <Integer> apply(Integer y);
}
interface Thunk { Code<Integer> call(); }
Code<Code<Integer>> doCSP(Thunk t) {
  return <| t.call() |>;
}

<| new IntCodeFun() {
  Code<Integer> apply(Integer y) {
    return `(doCSP(new Thunk () {
      Code<Integer> call() {
        return <| y |>;
      }
    }));
  }
}.apply(1) |>
```

Benchmarks



Apple MacBook 2.0 GHz Intel Core Duo 2 MB L2 cache, 2 GB RAM, Mac OS 10.4