RICE

Mint:
A Multi-stage Extension of Java

Purdue University - Computer Science Colloquia

Mathias Ricken
Rice University
March 15, 2010

Abstractions are Expensive

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

```
public static
int power17(int x) {
    return x * … * x;
}
```

power(2,17) : 41 ns

power17(2) : 9 ns

- Multi-stage programming (MSP) languages
  - Provide constructs for runtime code generation
  - 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
```

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|>, 17);
```

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

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

Result: `131072`

Staged power Function

```
Code<? extends Lambda> codePower17 = <|
  new Lambda() {
    public Double apply(final Double x) {
      return `(spower(<|x|>, 17));
//    return `(<| (((1*x)*x)*x) … *x |>);
//    return       (((1*x)*x)*x) … *x;
    }
} |>;
Lambda power17 = {codePower17.run()};

Double d = power17.apply(2);
```

Result: 131072

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]

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();
}
```

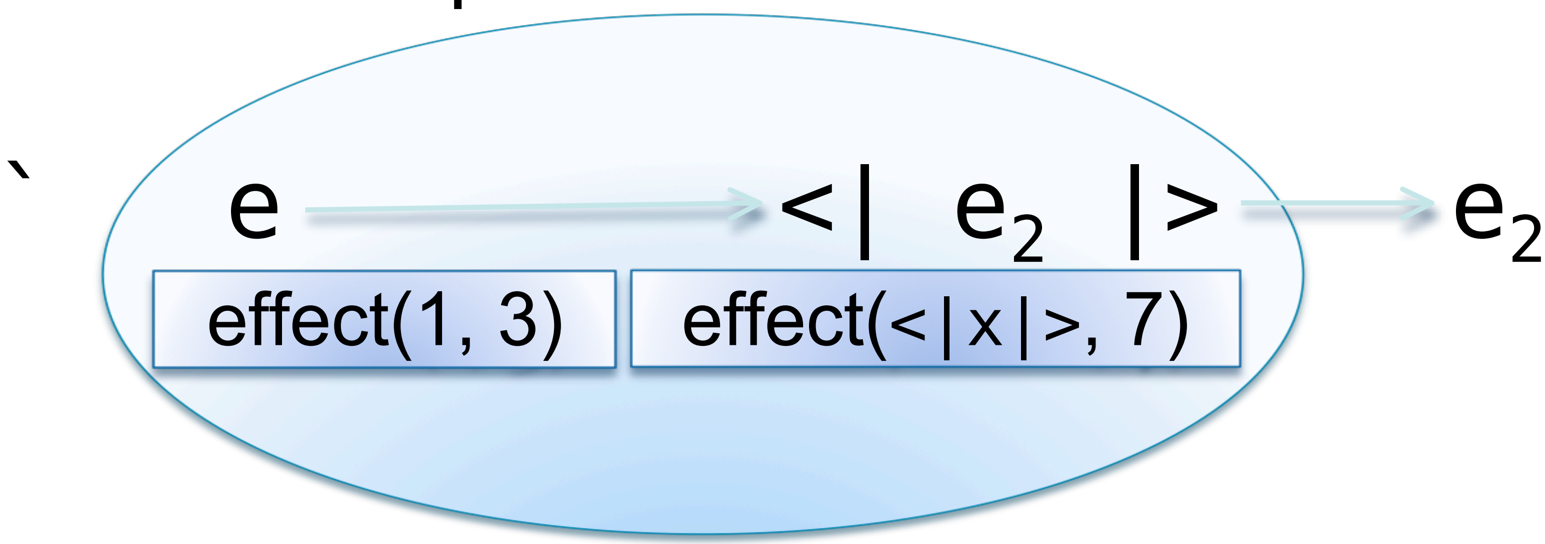
<| y |>

y

y used out of scope!

Solution: Weak Separability

- No effects containing code may be seen outside of escapes



- Restricts only escapes, not generated code
  - 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

Definition: Code-Free

A type T is *code-free* iff

- T not a subtype of `Code<A>` for some `A`
- All field types of T are code-free
- All method return types of T are code-free
- T is `final`

<u>Not code-free:</u>

```
class C {
  Code<Integer> c;
  Object x;
  Code<Integer> foo() { … }
}
```

Not `final`

Field not code-free

Field not code-free

Return not code-free

Definition: Weakly Separable

A term is *weakly separable* iff

- 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)

Expressivity of Weak Separability

- Build code with accumulators

```
public static separable
Code<Void> genCode(final int i) {
  return <| { System.out.println(i); } |>; }

Code<Void> accum = <| { } |>;
for(int i = 0; i < n; ++i)
   accum = <| { `accum; `(genCode(i)); } |>;
```

- Throw exceptions out of code generators

```
<| `(malformed(data)?
        throw new BadData(data):data); …) |>
```

- Update global counters, arrays…

4

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
  Code<Integer> eval(Env e, FEnv f);
}
class Int implements Exp { /* ... */
  public separable
  Code<Integer> eval(Env e, FEnv f) {
    final int v = _v; return <| v |>;
  }
}
class App implements Exp { /* ... */
  public separable
  Code<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 Code<Integer> v) {
  return new Env() {
    public separable
    Code<Integer> get(String y) {
      if (x==y) return v;
      else return env.get(y);
    }
  };
}

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

Throw an exception.

Can’t be done safely in other MSP systems.

Expressivity

- Staged interpreter
  - lint interpeter (Taha'04)
  - 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.

Performance Results

Benchmark	speedup	unstaged μs	staged μs
power	9.2	0.060	0.0065
fib	8.8	0.058	0.0065
mmult	4.7	13	2.7
eval-fact	20	0.83	0.042
eval-fib	24	18	0.73
serialize	26	1.5	0.057
av-mmult	65	20	0.30
av-mtrans	14	1.0	0.071

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/

Conclusion: Mint = Java + MSP

- MSP reduces the cost of abstractions
- Mint brings MSP to the mainstream
- Key insight: weak separability
  - Only code-free effects can be seen outside of escapes
- Can do MSP with common Java idioms
  - Build code with an accumulator
  - Throw exceptions out of generators

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
- Thanks to Jan Vitek, Lukasz Ziarek and the Purdue CS department for hosting this talk

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.

Footnotes

4. Since `throw` is not an expression in Java, use this code instead: ←

```
public static <T> T throwBadData(T d) {
  throw new BadData("bad data: "+d);
}

<| `(malformed(data)?
        throwBadData(data):
        …); …) |>
```

Extra Slides

Unstaged `power` in MetaOCaml

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

power(2, 17);;
```

Result: `131072`

- Overhead due to recursion
  - Faster way to calculate x^{17}: `x*x*x*…*x`
  - Don't want to write x^2, x^3, …, x^{17}… 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>., 17);;
```

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

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

Result: 131072

Staged power in MetaOCaml

```
let codePower17 =
  .< fun x -> .~(spower(.<x>., 17)) >.;;
//.< fun x -> .~(.< x*(x*…*(x*1)…) >.) >.;;
//.< fun x -> x*(x*…*(x*1)…) >.;;


let power17 = .! codePower17;;

power17(2);
```

Result: 131072

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) |>
```

Expressivity

- Staged interpreter
- Staged array views
- Simple staged serializer
  - Removes reflection and recursion overhead

Staged Reflection Primitives

Standard Primitives	Staged Primitives
`Class<A>`	`ClassCode<A>`
`Field<A>`	`FieldCode<A,B>`
`Field[]` `Class<A>.getFields()`	`FieldCode<A,?>[]` `ClassCode<A>.getFields()`
`Object Field.get(Object)`	`B FieldCode<A,B>.get(A)`

Simple Staged Serializer

```
public static <A>
Code<Void> sserialize(ClassCode<A> t, Code<A> o) {

  // handle base types
  if (t.getCodeClass() == Integer.class)
    return <| { writeInt(‘((Code<Integer>)o)); } |>;

  // handle defined classes
  Code <Void> result = <| { } |>;
  for (FieldCode <A,?> fc: t.getFields()) {
    result = <| { `result;
                  `(serializeField(fc, o)); } |>;
  }
  return result;
}
```

Typing for Weak Separability

```
<| { let Integer y = foo();
      `(e) } |>
```

e can see heap values with <| y |>

Should not see <| y |> outside

Consider a Small-Step Trace

```
<| { Integer y = foo();
     return `(e1); } |>
```

heap:
$l_1 = 0$
$l_2 = <|1|>$
$l_3 = B(f=l_1)$

e_1

heap:
$l_1 = 0$
$l_2 = <|1|>$
$l_3 = B(f=l_1)$

$\rightarrow^{*}$ $<|\ e_2\ |>$

heap:
$l_1 = 2$
$l_2 = <|1|>$
$l_3 = B(f=l_4)$
$l_4 = 7$
$l_5 = <|y|>$

```
<| { Integer y = foo();
     return e2; } |>
```

heap:
$l_1 = 2$
$l_2 = <|1|>$
$l_3 = B(f=l_4)$
$l_4 = 7$
$l_5 = <|1|>$

Bad!

Solution: Stack of Heap Typings

```
<| { Integer y = foo();
     return `(e1); } |>
```

```
<| { Integer y = foo();
     return e2; } |>
```

typing:
l_1 : Integer
l_2 : Code<Integer>
l_3 : B

e_1 →* <| e_2 |>

typing:
l_1 : Integer
l_2 : Code<Integer>
l_3 : B

typing:

typing:
l_1 : Integer
l_2 : Code<Integer>
l_3 : B

l_4 : Integer
l_5 : Code<Integer>

typing:
l_1 : Integer
l_2 : Code<Integer>
l_3 : B
l_4 : Integer

Smashing lemma

Typing in Symbols

$$\Sigma_1; \ldots; \Sigma_n; \Gamma \mid\!- \ (H, e) : T$$

One heap typing for each dynamic binding

Type heaps and expressions together

Typing in Symbols

$$\Sigma_1; \ldots; \Sigma_n; \Sigma; \Gamma, \vdash H$$

H typed with Γ

H constrains Σ

$$\Sigma_1; \ldots; \Sigma_n; \Sigma; \Gamma, y{:}\text{Integer} \vdash e : \text{Integer}$$

$\Sigma_1; \ldots; \Sigma_n; \Gamma \vdash$ (H,
```
<| { Integer y = foo();
     return `(e); } |>
```
)

Smashing Lemma (approx)

- If
  - $\Sigma_1; \ldots; \Sigma_n; \Gamma \vdash H_1$
  - $\Sigma_1; \ldots; \Sigma_n; \Sigma; \Gamma \vdash H_2$
  - $H_1 |_L = H_2 |_L$ for $L = dom(\cup_i \Sigma_i) - dom(cf(\cup_i \Sigma_i))$
- Then
  - $\Sigma_1; \ldots; \Sigma_{n-1}; \Sigma_n \cup cf(\Sigma); \Gamma \vdash H_2$