Backstage Java
Making a Difference in Metaprogramming

Zachary Palmer and Scott F. Smith
The Johns Hopkins University

October 27, 2011
Difference-Based Metaprogramming

• Introduction to Metaprogramming
Difference-Based Metaprogramming

- Introduction to Metaprogramming
- Compile-Time Metaprogramming in Java
Difference-Based Metaprogramming

- Introduction to Metaprogramming
- Compile-Time Metaprogramming in Java
- Traditional Metaprogramming Model
Difference-Based Metaprogramming

- Introduction to Metaprogramming
- Compile-Time Metaprogramming in Java
- Traditional Metaprogramming Model
- Difference-Based Metaprogramming Model
What is metaprogramming?
Metaprogramming

- Programs input data and output data.

![Diagram](image-url)
Metaprogramming

- Programs input data and output data.

- Metaprograms input programs (or program fragments) and output the same.
Examples of Metaprogramming

- C Macros
- C++ Templates
- LISP Macros
- Template Haskell
- MetaOCaml
- Stratego
- Groovy
- etc. etc.
Classifying Metaprogramming

- When is it run?
  - Compile-time (static)
  - Runtime (dynamic)

- How are programs represented?
  - Textually (strings)
  - Lexically (tokens)
  - Structurally (ASTs)
  - Semantically (various structures)

- Which language is used to metaprogram?
  - Same language (homogenous)
  - Different language (heterogeneous)

from Accomplishments and Research Challenges in Metaprogramming (Tim Sheard)
Classifying Metaprogramming

• When is it run?
  - Compile-time (static)
  - Runtime (dynamic)

• How are programs represented?
  - Textually (strings)
  - Lexically (tokens)
  - Structurally (ASTs)
  - Semantically (various structures)

• Which language is used to metaprogram?
  - Same language (homogenous)
  - Different language (heterogeneous)

from Accomplishments and Research Challenges in Metaprogramming (Tim Sheard)
Template Haskell Example

$(
    let mkExp n v =
        if n == 0
            then [|| 1 ||]
            else [|| $(v) * $(mkExp (n-1) v) ||]
    in
    let f n =
        let funNm = mkName ("exp" ++ (show n)) in
        let params = [varP (mkName "x")]
            in
        funD funNm $ [clause params
                        (normalB $ mkExp n (varE $ mkName "x")) []]
        in
    mapM f [2..50]
)$
Template Haskell Example

\[\text{exp2 } x = x \times x\]
\[\text{exp3 } x = x \times x \times x\]
\[\text{exp4 } x = x \times x \times x \times x\]
\[\text{exp5 } x = x \times x \times x \times x \times x\]
\[\vdots\]
\[\text{exp50 } x = x \times x \times \ldots \times x \times x\]
Template Haskell

• Programmatic code generation
Template Haskell

- Programmatic code generation
- Literal syntax for AST construction
Template Haskell

- Programmatic code generation
- Literal syntax for AST construction
- Functional programming style
Template Haskell

- Programmatic code generation
- Literal syntax for AST construction
- Functional programming style
- Very limited ability to inspect environment
Why not Template Java?
public class Location {
    private int x;
    public int getX() { return this.x; }
    public void setX(int x) { this.x = x; }
    private int y;
    public int getY() { return this.y; }
    public void setY(int y) { this.y = y; }
    public Location(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public String toString() {
        return "("+this.x+","+this.y+")";
    }
}
public class Location {
    private int x;
    public int getX() { return this.x; }
    public void setX(int x) { this.x = x; }
    private int y;
    public int getY() { return this.y; }
    public void setY(int y) { this.y = y; }
    public Location(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public String toString() {
        return "("+this.x+","+this.y+")";
    }
}
public class Location {

    $( property( [private int x] ) )

    private int y;
    public int getY() { return this.y; }
    public void setY(int y) { this.y = y; }
    public Location(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public String toString() {
        return "("+this.x+","+this.y+")";
    }
}
public class Location {

  $( property( [private int x] ) )

  private int y;
  public int getY() { return this.y; }
  public void setY(int y) { this.y = y; }
  public Location(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public String toString() {
    return "("+this.x+","+this.y+")";
  }
}

Template Java?
public class Location {

   $( property( [|private int x|] )

   $( property( [|private int y|] )

   public Location(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public String toString() {
      return "("+this.x+","+this.y+")";
   }
}
}
public class Location {

    $( property( [ |private int x| ] ) )

    $( property( [ |private int y| ] ) )

    public Location(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "(" + this.x + "," + this.y + ")";
    }

}
public class Location {

    $( property( [private int x] )

    $( property( [private int y] )

    $( makePropertyConstructor(
        [int x], [int y] )

    public String toString() {
        return "("+this.x+","+this.y+");
    }
}
}
public class Location {
    $( property( [private int x|] )
    $( property( [private int y|] )
    $( makePropertyConstructor(
        [int x|], [int y|]) )
    public String toString() {
        return "("+this.x+","+this.y+")";
    }
}
Template Java?

```java
public class Location {
    $( property( [|private int x|] )
    $( property( [|private int y|] )
    $( makePropertyConstructor(  
        [|int x|], [|int y|] ) )
    public String toString() {
        return "(" + this.x + "," + this.y + ")";
    }
}
```

- Metaprograms can’t react to surrounding code
Template Java?

```java
public class Location {
    $( property( [private int x] ) )
    $( property( [private int y] ) )
    $( makePropertyConstructor(
        [int x], [int y]) )
    public String toString() {
        return "("+this.x+","+this.y+")";
    }
}
```

- Metaprograms can’t react to surrounding code
- Metaprogrammers compensate by duplicating information
Template Java?

```java
public class Location {
    $( property( [private int x] ) )
    $( property( [private int y] ) )
    $( makePropertyConstructor(
        [int x], [int y] ) )
    public String toString() {
        return "("+this.x+","+this.y+")";
    }
}
```

- Metaprograms can’t react to surrounding code
- Metaprogrammers compensate by duplicating information
- Functional metaprogramming in a declarative object-oriented language
What Do We Want?

- Object-oriented, declarative metaprogramming style
What Do We Want?

- Object-oriented, declarative metaprogramming style
- Awareness of surrounding code
What Do We Want?

- Object-oriented, declarative metaprogramming style
- Awareness of surrounding code
- Modular, independent metaprograms
Backstage Java

How about some of this?

```java
@GenerateConstructorFromProperties
class Location {
    @Property private int x;
    @Property private int y;
    public String toString() {
        return "("+this.x+"","+this.y+")");
    }
}
```
Backstage Java

How about some of this?

```java
@@GenerateConstructorFromProperties
public class Location {
    @@Property private int x;
    @@Property private int y;
    public String toString() {
        return "("+this.x+","+this.y+")";
    }
}
```

Harder than it looks...
Traditional Metaprogramming
Traditional Metaprogramming

- Metaprograms are a series of program transformations
Traditional Metaprogramming

- Metaprogams are a series of program transformations
- Each available transformation occurs exactly once
Traditional Metaprogramming

- Metaprograms are a series of program transformations
- Each available transformation occurs exactly once
- True even for embedded syntax
Embedded Metaprogram Semantics

How do we pick?
Embedded Metaprogram Semantics

How do we pick?
Embedded Metaprogram Semantics

$\phi_1$  $\phi_2$

How do we pick?
Embedded Metaprogram Semantics

\[ \phi_1 \left( \phi_2 \left( \right) \right) \]
Embedded Metaprogram Semantics

\[ \phi_2(\phi_1) \]
Embedded Metaprogram Semantics

$\phi_2(\phi_1)$

How do we pick?
Ambiguity in Metaprogramming

Assuming three transformations $\phi_1, \phi_2, \phi_3 \ldots$
Ambiguity in Metaprogramming

Assuming three transformations $\phi_1, \phi_2, \phi_3...$

OpenJava, Groovy
Total Ordering Solution

Declaring $\phi_2$ before $\phi_1$, which is before $\phi_3$. 
Total Ordering Solution

Declaring $\phi_2$ before $\phi_1$, which is before $\phi_3$. 
Necessary Commutation Solution

Requiring that $\phi_i \circ \phi_j = \phi_j \circ \phi_i$ for all $i$ and $j$
Necessary Commutation Solution

Requiring that $\phi_i \circ \phi_j = \phi_j \circ \phi_i$ for all $i$ and $j$

Template Haskell, MetaOCaml, LISP, ...
Hybrid Solution

Suppose that $\phi_1$ and $\phi_2$ commute. Suppose that $\phi_3$ must occur after them.

$\rho$
Hybrid Solution

Suppose that $\phi_1$ and $\phi_2$ commute.
Hybrid Solution

Suppose that $\phi_1$ and $\phi_2$ commute.
Suppose that $\phi_1$ and $\phi_2$ commute. Suppose that $\phi_3$ must occur after them.
Hybrid Solution

Suppose that $\phi_1$ and $\phi_2$ commute. Suppose that $\phi_3$ must occur after them.
Hybrid Solution

Suppose that $\phi_1$ and $\phi_2$ commute. Suppose that $\phi_3$ must occur after them.
Hybrid Solution

Suppose that $\phi_1$ and $\phi_2$ commute. Suppose that $\phi_3$ must occur after them.
Hybrid Solution

Suppose that $\phi_1$ and $\phi_2$ commute. Suppose that $\phi_3$ must occur after them.

Backstage Java*
Commuting Transformations

How do we tell if $\phi_1$ and $\phi_2$ commute?
How do we tell if $\phi_1$ and $\phi_2$ commute?
Commuting Transformations

How do we tell if $\phi_1$ and $\phi_2$ commute?

Determining whether or not two arbitrary transformations commute is \textit{undecidable}!
Difference-Based Metaprogramming
Difference-Based Metaprogramming

Treat metaprograms as transformation *generators*:
Difference-Based Metaprogramming

Treat metaprograms as transformation *generators*:

\[ \phi(\rho) = \bar{\delta} \]
Difference-Based Metaprogramming

Treat metaprograms as transformation *generators*: 

\[ \phi(\rho) = \bar{\delta} \]

\[ [\bar{\delta}](\rho) = \rho' \]
Difference-Based Metaprogramming

Treat metaprograms as transformation *generators*:

\[
\phi(\rho) = \bar{\delta} \\
[\bar{\delta}](\rho) = \rho'
\]

- Language of \(\bar{\delta}\) is not Turing-complete
Difference-Based Metaprogramming

Treat metaprograms as transformation *generators*:

\[ \phi(\rho) = \bar{\delta} \]
\[ [\bar{\delta}](\rho) = \rho' \]

- Language of \( \bar{\delta} \) is not Turing-complete
- Each \( \bar{\delta} \) is generated on a case-by-case basis
Difference-Based Metaprogramming

Treat metaprograms as transformation *generators*:

\[ \phi(\rho) = \bar{\delta} \]

\[ \llbracket \bar{\delta} \rrbracket(\rho) = \rho' \]

- Language of \( \bar{\delta} \) is not Turing-complete
- Each \( \bar{\delta} \) is generated on a case-by-case basis
- No practically significant loss of expressiveness
Difference-Based Metaprogramming

Treat metaprograms as transformation *generators*:

\[ \phi(\rho) = \delta \]

\[ [\delta](\rho) = \rho' \]

\(\delta\) can express:

- Creation of a node
Difference-Based Metaprogramming

Treat metaprograms as transformation *generators*:

\[ \phi(\rho) = \bar{\delta} \]
\[ \llbracket \bar{\delta} \rrbracket(\rho) = \rho' \]

\( \bar{\delta} \) can express:

- Creation of a node
- Assignment to a node property
- Additions before or after an element in a list
- Removal of an element from a list
Difference-Based Metaprogramming

Treat metaprograms as transformation *generators*:

\[
\phi(\rho) = \bar{\delta} \\
\overline{\delta}(\rho) = \rho'
\]

\(\bar{\delta}\) can express:

- Creation of a node
- Assignment to a node property
- Additions before or after an element in a list
Difference-Based Metaprogramming

Treat metaprogams as transformation *generators*:

\[ \phi(\rho) = \bar{\delta} \]
\[ [\bar{\delta}](\rho) = \rho' \]

\( \bar{\delta} \) can express:

- Creation of a node
- Assignment to a node property
- Additions before or after an element in a list
- Removal of an element from a list
Difference-Based Metaprogramming

Treat metaprograms as transformation *generators*:

\[ \phi(\rho) = \bar{\delta} \]
\[ \lbrack \bar{\delta} \rbrack(\rho) = \rho' \]

Now, prove commutation over pairs of \( \lbrack \bar{\delta} \rbrack \).
Difference-Based Metaprogramming

Treat metaprograms as transformation generators:

\[ \phi(\rho) = \bar{\delta} \]

\[ \llbracket \bar{\delta} \rrbracket(\rho) = \rho' \]

Now, prove commutation over pairs of \( \llbracket \bar{\delta} \rrbracket \).
public class Example {
    public static void main(String[] arg) {
        BlockStatementListNode list = context.getAnchor().
            getNearestAncestorOfType(
                BlockStatementListNode.class);
        list.addFirst(
            System.out.println("Hello, world!"););
    }
    BlockStatementListNode list = context.getAnchor().
        getNearestAncestorOfType(
            BlockStatementListNode.class);
    list.addLast(
        System.out.println("How are you?"););
}
A Simple Example

```java
public class Example {
    public static void main(String[] arg) {

        M1
        M2

    }
}
```

- Replace metaprograms with anchors
A Simple Example

```java
public class Example {
    public static void main(String[] arg) {

        M_1
        M_2

    }
}
```

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
A Simple Example

public class Example {
    public static void main(String[] arg) {
        M1
        M2
    }
}

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - $\phi_1(\rho) = \bar{\delta}_1$ (“Hello, world!” first)
A Simple Example

```java
public class Example {
    public static void main(String[] arg) {

        M1
        M2

    }
}
```

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - $\phi_1(\rho) = \delta_1$ ("Hello, world!" first)
  - $\phi_2(\rho) = \delta_2$ ("How are you?" last)
A Simple Example

public class Example {
    public static void main(String[] arg) {
        System.out.println("Hello, world!");
        System.out.println("How are you?");
    }
}

• Replace metaprograms with anchors
• Run each metaprogram to collect its changes
  • $\phi_1(\rho) = \delta_1$ ("Hello, world!" first)
  • $\phi_2(\rho) = \delta_2$ ("How are you?" last)
• Prove that $\delta_1$ and $\delta_2$ commute.
A Simple Example

```java
public class Example {
    public static void main(String[] arg) {
        M1
        M2
    }
}
```

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - \( \phi_1(\rho) = \overline{\delta}_1 \) ("Hello, world!" first)
  - \( \phi_2(\rho) = \overline{\delta}_2 \) ("How are you?" last)
- Prove that \( \overline{\delta}_1 \) and \( \overline{\delta}_2 \) commute.
- Execute \( \overline{\delta}_1 \) and \( \overline{\delta}_2 \) in some order.
A Simple Example

```java
public class Example {
    public static void main(String[] arg) {
        System.out.println("Hello, world!");
        M1
        M2
    }
}
```

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - $\phi_1(\rho) = \delta_1$ ("Hello, world!" first)
  - $\phi_2(\rho) = \delta_2$ ("How are you?" last)
- Prove that $\delta_1$ and $\delta_2$ commute.
- Execute $\delta_1$ and $\delta_2$ in some order.
A Simple Example

```java
public class Example {
    public static void main(String[] arg) {
        System.out.println("Hello, world!");
        System.out.println("How are you?");
    }
}
```

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - $\phi_1(\rho) = \bar{\delta}_1$ (“Hello, world!” first)
  - $\phi_2(\rho) = \bar{\delta}_2$ (“How are you?” last)
- Prove that $\bar{\delta}_1$ and $\bar{\delta}_2$ commute.
- Execute $\bar{\delta}_1$ and $\bar{\delta}_2$ in some order.
A Simple Example

```java
public class Example {
    public static void main(String[] arg) {
        
        M1
        M2

    }
}
```

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - $\phi_1(\rho) = \delta_1$ (“Hello, world!” first)
  - $\phi_2(\rho) = \delta_2$ (“How are you?” last)
- Prove that $\delta_1$ and $\delta_2$ commute.
- Execute $\delta_1$ and $\delta_2$ in some order.
A Simple Example

```java
public class Example {
    public static void main(String[] arg) {
        System.out.println("Hello, world!");
        System.out.println("How are you?");
    }
}
```

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - $\phi_1(\rho) = \overline{\delta_1}$ ("Hello, world!" first)
  - $\phi_2(\rho) = \overline{\delta_2}$ ("How are you?" last)
- Prove that $\overline{\delta_1}$ and $\overline{\delta_2}$ commute.
- Execute $\overline{\delta_1}$ and $\overline{\delta_2}$ in some order.
A Simple Example

```java
public class Example {
    public static void main(String[] arg) {
        System.out.println("Hello, world!");
        M1
        M2
        System.out.println("How are you?");
    }
}
```

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - $\phi_1(\rho) = \delta_1$ ("Hello, world!" first)
  - $\phi_2(\rho) = \delta_2$ ("How are you?" last)
- Prove that $\delta_1$ and $\delta_2$ commute.
- Execute $\delta_1$ and $\delta_2$ in some order.
public class Example {
    public static void main(String[] arg) {
        System.out.println("Hello, world!");
        System.out.println("How are you?");
    }
}

• Replace metaprograms with anchors
• Run each metaprogram to collect its changes
  • $\phi_1(\rho) = \bar{\delta}_1$ ("Hello, world!" first)
  • $\phi_2(\rho) = \bar{\delta}_2$ ("How are you?" last)
• Prove that $\bar{\delta}_1$ and $\bar{\delta}_2$ commute.
• Execute $\bar{\delta}_1$ and $\bar{\delta}_2$ in some order.
An Example of Conflict

```java
public class Example {
    public static void main(String[] arg) {
        BlockStatementListNode list = context.getAnchor().
            getNearestAncestorOfType(
                BlockStatementListNode.class);
        list.addFirst(
            System.out.println("Hello, world!"));
    }
    BlockStatementListNode list = context.getAnchor().
        getNearestAncestorOfType(
            BlockStatementListNode.class);
    list.addFirst(
        System.out.println("How are you?")];
    }
```
public class Example {
    public static void main(String[] arg) {
        System.out.println("Hello, world!");
        System.out.println("Hello, world!");
    }
}

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - $\phi_1(\rho) = \bar{\delta}_1$ ("Hello, world!" first)
  - $\phi_2(\rho) = \bar{\delta}_2$ ("How are you?" first)
An Example of Conflict

public class Example {
    public static void main(String[] arg) {

        \( M_1 \)
        \( M_2 \)
    }
}

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - \( \phi_1(\rho) = \bar{\delta}_1 \) ("Hello, world!" first)
  - \( \phi_2(\rho) = \bar{\delta}_2 \) ("How are you?" first)
- Now, \( \bar{\delta}_1 \) and \( \bar{\delta}_2 \) do not commute!
An Example of Conflict

class Example {
    public static void main(String[] arg) {
        System.out.println("Hello, world!");
        System.out.println("How are you?");
    }
}

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - $\phi_1(\rho) = \overline{\delta}_1$ ("Hello, world!" first)
  - $\phi_2(\rho) = \overline{\delta}_2$ ("How are you?" first)
- Now, $\overline{\delta}_1$ and $\overline{\delta}_2$ do not commute!
An Example of Conflict

```java
public class Example {
    public static void main(String[] arg) {
        System.out.println("Hello, world!");
        System.out.println("How are you?");
        \[ M_1 \]
        \[ M_2 \]
    }
}
```

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - \( \phi_1(\rho) = \bar{\delta}_1 \) ("Hello, world!" first)
  - \( \phi_2(\rho) = \bar{\delta}_2 \) ("How are you?" first)
- Now, \( \bar{\delta}_1 \) and \( \bar{\delta}_2 \) do not commute!
An Example of Conflict

public class Example {
    public static void main(String[] arg) {

        $M_1$
        $M_2$
    }
}

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - $\phi_1(\rho) = \overline{\delta}_1$ (“Hello, world!” first)
  - $\phi_2(\rho) = \overline{\delta}_2$ (“How are you?” first)
- Now, $\overline{\delta}_1$ and $\overline{\delta}_2$ do not commute!
An Example of Conflict

public class Example {
    public static void main(String[] arg) {
        System.out.println("Hello, world!".EXPAND dash M_1 EXPAND dash M_2
        System.out.println("Hello, world!");
        M_1
        M_2
    }
}

• Replace metaprograms with anchors
• Run each metaprogram to collect its changes
  • $\phi_1(\rho) = \overline{\delta_1}$ ("Hello, world!" first)
  • $\phi_2(\rho) = \overline{\delta_2}$ ("How are you?" first)
• Now, $\overline{\delta_1}$ and $\overline{\delta_2}$ do not commute!
An Example of Conflict

```java
public class Example {
    public static void main(String[] arg) {
        System.out.println("How are you?");
        System.out.println("Hello, world!");
        M1
        M2
    }
}
```

- Replace metaprograms with anchors
- Run each metaprogram to collect its changes
  - $\phi_1(\rho) = \overline{\delta}_1$ ("Hello, world!" first)
  - $\phi_2(\rho) = \overline{\delta}_2$ ("How are you?" first)
- Now, $\overline{\delta}_1$ and $\overline{\delta}_2$ do not commute!
public class Example {
    public static void main(String[] arg) {
        System.out.println("How are you?");
        System.out.println("Hello, world!");
    }
}

• Replace metaprograms with anchors
• Run each metaprogram to collect its changes
  • $\phi_1(\rho) = \overline{\delta}_1$ ("Hello, world!" first)
  • $\phi_2(\rho) = \overline{\delta}_2$ ("How are you?" first)
• Now, $\overline{\delta}_1$ and $\overline{\delta}_2$ do not commute!
An Example of Conflict

```java
public class Example {
    public static void main(String[] arg) {
        System.out.println("How are you?");
        System.out.println("Hello, world!");
    }
}
```

- Replace metaprogams with anchors
- Run each metaprogram to collect its changes
  - \( \phi_1(\rho) = \overline{\delta}_1 \) ("Hello, world!" first)
  - \( \phi_2(\rho) = \overline{\delta}_2 \) ("How are you?" first)
- Now, \( \overline{\delta}_1 \) and \( \overline{\delta}_2 \) do not commute!
- But we can detect this!
Conflict Detection

**Record Node Creation Rule**

\[
\eta \mapsto \hat{\nu} \notin \rho \\
\rho[\eta \mapsto \{l \mapsto \hat{\nu}\}] \Rightarrow \rho'
\]

\[
(\underbrace{\overline{\eta}(l = \hat{\nu})}_{\text{If } \eta \mapsto \hat{\nu}}) \rho \Rightarrow \rho'
\]

**List Node Creation Rule**

\[
\eta \mapsto \hat{\nu} \notin \rho \\
\rho[\eta \mapsto [(>, M, \emptyset), (<, M, \emptyset)]] \Rightarrow \rho'
\]

\[
(M > L \eta) \rho \Rightarrow \rho'
\]

**Record Assignment Rule**

\[
\eta \mapsto \mathcal{R} \in \rho \\
\rho[\eta \mapsto \mathcal{R}[l \mapsto \hat{\nu}]] \Rightarrow \rho'
\]

\[
(\eta.l \leftarrow \hat{\nu}) \rho \Rightarrow \rho'
\]

**List Add Before Rule**

\[
\hat{\eta}_3 \neq > \quad \eta_1 \mapsto \mathcal{L} \in \rho \\
\hat{\eta}_3 = \Sigma(\hat{\eta}_3, M, \mathcal{L}) \\
\mathcal{L} = \left[\overline{\eta'}, \hat{\nu}_3, \overline{\eta''}\right] \\
\mathcal{L}' = \left[\overline{\eta'}, \eta_2, M, \emptyset, \hat{\nu}_3, \overline{\eta''}\right] \\
\rho[\eta_1 \mapsto \mathcal{L}'] \Rightarrow \rho'
\]

\[
(M > \eta_1 : \eta_2 \mapsto \hat{\eta}_3) \rho \Rightarrow \rho'
\]

**List Add After Rule**

\[
\hat{\eta}_3 \neq < \quad \eta_1 \mapsto \mathcal{L} \in \rho \\
\hat{\eta}_3 = \Sigma(\hat{\eta}_3, M, \mathcal{L}) \\
\mathcal{L} = \left[\overline{\eta'}, \hat{\nu}_3, \overline{\eta''}\right] \\
\mathcal{L}' = \left[\overline{\eta'}, \hat{\nu}_3, (\eta_2, M, \emptyset), \hat{\nu}_3, \overline{\eta''}\right] \\
\rho[\eta_1 \mapsto \mathcal{L}'] \Rightarrow \rho'
\]

\[
(M > \eta_1 : \hat{\eta}_3 \mapsto \eta_2) \rho \Rightarrow \rho'
\]

**List Remove Rule**

\[
\eta_1 \mapsto \mathcal{L} \in \rho \\
\hat{\eta}_2 = (\eta_2, M', S) = \Sigma(\eta_2, M, \mathcal{L}) \\
\mathcal{L} = \left[\overline{\eta'}, \hat{\nu}_2, \overline{\eta''}\right] \\
\mathcal{L}' = \left[\overline{\eta'}, (\eta_2, M', S \cup \{M\}), \overline{\eta''}\right] \\
\rho[\eta_1 \mapsto \mathcal{L}'] \Rightarrow \rho'
\]

\[
(\mathcal{M} > \eta_1 : \downarrow \eta_2) \rho \Rightarrow \rho'
\]

**Recursive Application Rule**

\[
\delta' e \Rightarrow \rho \\
\delta \rho \Rightarrow \rho'
\]

\[
\delta (\delta' e) \Rightarrow \rho'
\]

**Value Rule**

\[
\hat{\nu} \Rightarrow \hat{\nu}
\]

**Record Assignment Conflict Rule**

\[
\hat{\nu} \neq \hat{\nu}' \\
\eta.l \leftarrow \hat{\nu} \leftrightarrow \eta.l \leftarrow \hat{\nu}'
\]

**Add Before Conflict Rule**

\[
\omega(\eta_2) \\
\omega(\eta'_2)
\]

\[
\eta_1 : \eta_2 \leftrightarrow \hat{\eta}_3 \leftrightarrow \eta_1 : \eta'_2 \leftrightarrow \hat{\eta}_3
\]

**Add After Conflict Rule**

\[
\omega(\eta_2) \\
\omega(\eta'_2)
\]

\[
\eta_1 : \hat{\eta}_3 \leftrightarrow \eta_2 \leftrightarrow \eta_1 : \hat{\eta}_3 \leftrightarrow \eta'_2
\]

**Unordered Creation Conflict Rule**

\[
\delta = \overline{\eta}(l \mapsto \hat{\nu}) \lor \delta = \overline{\eta} \eta \in \delta' \\
\delta \leftrightarrow \delta'
\]
Conflicts Detection

Record Node Creation Rule
\[ \eta \mapsto \hat{v} \notin \rho \quad \rho[\eta \mapsto \{ l \mapsto \hat{v} \}] \Rightarrow \rho' \]
\[ (\hat{v}, \eta(l = \hat{v})) \rho \Rightarrow \rho' \]

List Node Creation Rule
\[ \eta \mapsto \hat{v} \notin \rho \quad \rho[\eta \mapsto [(\triangleright, \mathcal{M}, \emptyset), (\triangleleft, \mathcal{M}, \emptyset)]] \Rightarrow \rho' \]
\[ (\mathcal{M}\triangleright \triangleleft \eta) \rho \Rightarrow \rho' \]

Record Assignment Rule
\[ \eta \mapsto R \in \rho \quad \rho[\eta \mapsto R[l \mapsto \hat{v}]] \Rightarrow \rho' \]
\[ (\eta.l \leftarrow \hat{v}) \rho \Rightarrow \rho' \]

List Add Before Rule
\[ \hat{\eta}_3 \neq \triangleright \quad \eta_1 \mapsto \mathcal{L} \in \rho \]
\[ \hat{\eta}_3 = \Sigma(\hat{\eta}_3, \mathcal{M}, \mathcal{L}) \quad \mathcal{L} = [\hat{\eta}', \hat{\eta}_3, \hat{\eta}'''] \]
\[ \mathcal{L}' = [\hat{\eta}', (\eta_2, \mathcal{M}, \emptyset), \hat{\eta}_3, \hat{\eta}'''] \]
\[ \rho[\eta_1 \mapsto \mathcal{L}'] \Rightarrow \rho' \]
\[ (\mathcal{M}\triangleright \eta_1 : \eta_2 \mapsto \hat{\eta}_3) \rho \Rightarrow \rho' \]

List Add After Rule
\[ \hat{\eta}_3 \neq \triangleleft \quad \eta_1 \mapsto \mathcal{L} \in \rho \]
\[ \hat{\eta}_3 = \Sigma(\hat{\eta}_3, \mathcal{M}, \mathcal{L}) \quad \mathcal{L} = [\hat{\eta}', \hat{\eta}_3, \hat{\eta}'''] \]
\[ \mathcal{L}' = [\hat{\eta}', \hat{\eta}_3, (\eta_2, \mathcal{M}, \emptyset), \hat{\eta}'''] \]
\[ \rho[\eta_1 \mapsto \mathcal{L}'] \Rightarrow \rho' \]
\[ (\mathcal{M}\triangleright \eta_1 : \eta_2 \mapsto \hat{\eta}_3) \rho \Rightarrow \rho' \]

List Remove Rule
\[ \eta_1 \mapsto \mathcal{L} \in \rho \]
\[ \hat{\eta}_2 = (\eta_2, \mathcal{M}', \mathcal{S}) = \Sigma(\eta_2, \mathcal{M}, \mathcal{L}) \]
\[ \mathcal{L} = [\hat{\eta}', \hat{\eta}_2, \hat{\eta}'''] \]
\[ \mathcal{L}' = [\hat{\eta}', (\eta_2, \mathcal{M}', \mathcal{S} \cup \{ \mathcal{M} \}), \hat{\eta}'''] \]
\[ \rho[\eta_1 \mapsto \mathcal{L}'] \Rightarrow \rho' \]
\[ (\mathcal{M}\triangleright \eta_1 : \downarrow \eta_2) \rho \Rightarrow \rho' \]

Recursive Application Rule
\[ \delta' e \Rightarrow \rho \quad \delta \rho \Rightarrow \rho' \]
\[ \delta (\delta' e) \Rightarrow \rho' \]

Value Rule
\[ \hat{v} \Rightarrow \hat{v} \]

Record Assignment Conflict Rule
\[ \hat{v} \neq \hat{v}' \]
\[ \eta.l \leftarrow \hat{v} \leftrightarrow \eta.l \leftarrow \hat{v}' \]

Add Before Conflict Rule
\[ \omega(\eta_2) \quad \omega(\eta_2') \]
\[ \eta_1 : \eta_2 \mapsto \hat{\eta}_3 \leftrightarrow \eta_1 : \eta_2' \mapsto \hat{\eta}_3 \]

Add After Conflict Rule
\[ \omega(\eta_2) \quad \omega(\eta_2') \]
\[ \eta_1 : \hat{\eta}_3 \mapsto \eta_2 \leftrightarrow \eta_1 : \hat{\eta}_3' \mapsto \eta_2' \]

Unordered Creation Conflict Rule
\[ \delta = \cup_{\eta(l \mapsto \hat{v})} \delta \vee \delta = \cup_{\eta} \eta \in \delta' \]
\[ \delta \leftrightarrow \delta' \]
Huzzah!

- Metaprogram conflicts are detected at compile time
Huzzah!

- Metaprogram conflicts are detected at compile time
- Metaprograms are still aware of their surroundings
Huzzah!

- Metaprogram conflicts are detected at compile time
- Metaprograms are still aware of their surroundings

😊
So how do we resolve the conflict?
public static void main(String[] arg) {

    BlockStatementListNode list = context.getAnchor().
    getNearestAncestorOfType(
        BlockStatementListNode.class);
    list.addFirst(
        <:System.out.println("Hello, world!");:>);
}

::

    BlockStatementListNode list = context.getAnchor().
    getNearestAncestorOfType(
        BlockStatementListNode.class);
    list.addFirst(
        <:System.out.println("How are you?");:>);
}

Dependencies

```java
public static void main(String[] arg) {
    BlockStatementListNode list = context.getAnchor().
            getNearestAncestorOfType(
                BlockStatementListNode.class);
    list.addFirst(
        <:System.out.println("Hello, world!");:>
    );

    #target foo; ← Declare target membership
    BlockStatementListNode list = context.getAnchor().
            getNearestAncestorOfType(
                BlockStatementListNode.class);
    list.addFirst(
        <:System.out.println("How are you?");:>
    );
}
```
Dependencies

public static void main(String[] arg) {
    #: #depends foo; ← Declare target dependency
    BlockStatementListNode list = context.getAnchor().
    getNearestAncestorOfType(
        BlockStatementListNode.class);
    list.addFirst(
        <:System.out.println("Hello, world!");:>);

    ...

    #: #target foo;
    BlockStatementListNode list = context.getAnchor().
    getNearestAncestorOfType(
        BlockStatementListNode.class);
    list.addFirst(
        <:System.out.println("How are you?");:>);
}

Dependencies

```java
public static void main(String[] arg) {

  #depends foo;
  BlockStatementListNode list = context.getAnchor().
      getNearestAncestorOfType(
        BlockStatementListNode.class);
  list.addFirst(
      <:System.out.println("Hello, world!");:>
        :
      ]

  #target foo;
  BlockStatementListNode list = context.getAnchor().
      getNearestAncestorOfType(
        BlockStatementListNode.class);
  list.addFirst(
      <:System.out.println("How are you?");:>
        :
      ]
}
```
Dependency Graph

- One node per metaprogram
• One node per metaprogram
Dependency Graph

- One node per metaprogram
- $\mathcal{M}_2$ is a member of target “foo”
Dependency Graph

- One node per metaprogram
- $\mathcal{M}_2$ is a member of target “foo”
• One node per metaprogram
• $\mathcal{M}_2$ is a member of target “foo”
Dependency Graph

- One node per metaprogram
- $\mathcal{M}_2$ is a member of target “foo”
- $\mathcal{M}_1$ depends on the target “foo”
Dependency Graph

- One node per metaprogram
- $M_2$ is a member of target “foo”
- $M_1$ depends on the target “foo”
• One node per metaprogram
• $M_2$ is a member of target “foo”
• $M_1$ depends on the target “foo”
• Therefore, $M_1$ depends on $M_2$ ($M_1 \rightarrow M_2$)
Dependency Graph

- One node per metaprogram
- $\mathcal{M}_2$ is a member of target “foo”
- $\mathcal{M}_1$ depends on the target “foo”
- Therefore, $\mathcal{M}_1$ depends on $\mathcal{M}_2$ ($\mathcal{M}_1 \rightsquigarrow \mathcal{M}_2$)
- No more requirement for them to commute!
Dependency Graph

/* M₁ */ [: #target a; :]
/* M₂ */ [: #target a; :]
/* M₃ */ [: #target a, b; :]
/* M₄ */ [: #target c; #depends a; :]
/* M₅ */ [: #target c; #depends b; :]
/* M₆ */ [: #depends c; :]
/* M₇ */ [: :]

Diagram:

```
  M₁ ──> a ──> M₄ ──> M₆
  ^       |       |
  |       v       |
M₂ ──> M₃ ──> b ──> M₅
  |       |       |
  v       v       v
M₇
```
• $M_6$ depends on $M_2$ - no obligation to commute
• \( M_6 \) depends on \( M_2 \) - no obligation to commute

• No path between \( M_5 \) and \( M_4 \) - must commute
• $M_6$ depends on $M_2$ - no obligation to commute
• No path between $M_5$ and $M_4$ - must commute
• No path to $M_7$ - must *always* commute
• $M_6$ depends on $M_2$ - no obligation to commute
• No path between $M_5$ and $M_4$ - must commute
• No path to $M_7$ - must *always* commute
• More paths means less obligation to prove commutativity
A more practical example...
A more practical example…

…but first, a new feature
Meta-Annotations

```java
@Property private int x;
```

- Declarative metaprogramming abstraction
Meta-Annotations

```java
@Property private int x;
```

- Declarative metaprogramming abstraction
- Specifies metaprogram code and dependencies
Meta-Annotations

```java
@Property private int x;
```

- Declarative metaprogramming abstraction
- Specifies metaprogram code and dependencies
- Can annotate any declaration or block statement
Meta-Annotations

```java
@Property private int x;
```

- Declarative metaprogramming abstraction
- Specifies metaprogram code and dependencies
- Can annotate any declaration or block statement
- Allows easy reuse of metaprogramming constructs
Meta-Annotations

```java
@Property private int x;
```

- Declarative metaprogramming abstraction
- Specifies metaprogram code and dependencies
- Can annotate any declaration or block statement
- Allows easy reuse of metaprogramming constructs
- Here defined by user class named Property
Meta-Annotation Dependencies

```java
// @@GenerateConstructorFromProperties
public class Location {
    // @@Property
    private int x;
    // @@Property
    private int y;
    public String toString() {
        return "(" + this.x + "," + this.y + ")";
    }
}
```
Meta-Annotation Dependencies

```java
@GenerateConstructorFromProperties
public class Location {
    @Property private int x;
    @Property private int y;
    public String toString() {
        return "("+this.x+","+this.y+"))";
    }
}
```

- Meta-annotation defns. include dependencies
Meta-Annotation Dependencies

@@GenerateConstructorFromProperties

public class Location {
    @@Property private int x;
    @@Property private int y;
    public String toString() {
        return "("+this.x+","+this.y+")";
    }
}

- Meta-annotation defns. include dependencies
- @@Property is a member of target “property”
Meta-Annotation Dependencies

```java
@GenerateConstructorFromProperties
public class Location {
    @@Property private int x;
    @@Property private int y;
    public String toString() {
        return "("+this.x+","+this.y+")";
    }
}
```

- Meta-annotation defns. include dependencies
- @@Property is a member of target “property”
- @@GenerateConstructorFromProperties depends on “property”
BSJ Dependency Graph

- One node per metaprogram
BSJ Dependency Graph

- @@Property
- @@GenerateConstructorFromProperties

- One node per metaprogram
BSJ Dependency Graph

- One node per metaprogram
- `@@Property` participates in “property” target
BSJ Dependency Graph

- One node per metaprogram
- `@@Property` participates in “property” target
BSJ Dependency Graph

- One node per metaprogram
- `@@Property` participates in “property” target
BSJ Dependency Graph

- One node per metaprogram
- `@@Property` participates in “property” target
- `@@GenerateConstructorFromProperties` depends on “property”
BSJ Dependency Graph

- One node per metaprogram
- `@@Property` participates in “property” target
- `@@GenerateConstructorFromProperties` depends on “property”
@GenerateConstructorFromProperties

\( \phi_1 \) @@Property \( \phi_2 \) @@Property

property

\( \phi_3 \)
• $\phi_3$ depends on $\phi_1$
BSJ Dependency Graph

- \( \phi_1 \) depends on \( \phi_3 \)
- \( \phi_3 \) depends on \( \phi_1 \)
- \( \phi_3 \) depends on \( \phi_2 \)
BSJ Dependency Graph

- $\phi_3$ depends on $\phi_1$
- $\phi_3$ depends on $\phi_2$
Execution Example

\[ \rho \]

- Execute \( \phi_1 \) obtaining \( \bar{\delta}_1 \).
- Execute \( \phi_2 \) obtaining \( \bar{\delta}_2 \).
- Prove \( J\bar{\delta}_1 K \) and \( J\bar{\delta}_2 K \) commute.
Execution Example

- Execute $\phi_1$ obtaining $\bar{\delta}_1$. 

```latex
\rho
```

$\rho \sim = J \bar{\delta}_1 K J \bar{\delta}_2 K J \bar{\delta}_2 K J \bar{\delta}_1 K \circ \bar{\delta}_3 \phi_3 \rho' J \bar{\delta}_3 K$
Execution Example

- Execute $\phi_1$ obtaining $\bar{\delta}_1$.
- Execute $\phi_2$ obtaining $\bar{\delta}_2$.
- Prove $J \bar{\delta}_1 K$ and $J \bar{\delta}_2 K$ commute.
Execution Example

- Execute $\phi_1$ obtaining $\bar{\delta}_1$.
- Execute $\phi_2$ obtaining $\bar{\delta}_2$. 
Execution Example

- Execute $\phi_1$ obtaining $\bar{\delta}_1$.
- Execute $\phi_2$ obtaining $\bar{\delta}_2$. 
Execution Example

• Execute $\phi_1$ obtaining $\bar{\delta}_1$.
• Execute $\phi_2$ obtaining $\bar{\delta}_2$.
• Prove $[[\bar{\delta}_1]]$ and $[[\bar{\delta}_2]]$ commute.
Execution Example

- Execute $\phi_1$ obtaining $\bar{\delta}_1$.
- Execute $\phi_2$ obtaining $\bar{\delta}_2$.
- Prove $\llbracket \bar{\delta}_1 \rrbracket$ and $\llbracket \bar{\delta}_2 \rrbracket$ commute.
Execution Example

- Execute $\phi_1$ obtaining $\bar{\delta}_1$.
- Execute $\phi_2$ obtaining $\bar{\delta}_2$.
- Prove $[\bar{\delta}_1]$ and $[\bar{\delta}_2]$ commute.
Execution Example

- Execute $\phi_1$ obtaining $\delta_1$.
- Execute $\phi_2$ obtaining $\delta_2$.
- Prove $[\delta_1]$ and $[\delta_2]$ commute.
Execution Example

- Apply $\lceil \bar{\delta}_1 \rceil$ and $\lceil \bar{\delta}_2 \rceil$ to $\rho$. 
Execution Example

- Apply $\lceil \bar{\delta}_1 \rceil$ and $\lceil \bar{\delta}_2 \rceil$ to $\rho$. 
Execution Example

- Apply $[[\delta_1]]$ and $[[\delta_2]]$ to $\rho$. 
Execution Example

- Apply $[[\bar{\delta}_1]]$ and $[[\bar{\delta}_2]]$ to $\rho$.
### Execution Example

- Apply $[[\bar{\delta}_1]]$ and $[[\bar{\delta}_2]]$ to $\rho$.
- Execute $\phi_3$ on the result to get $\bar{\delta}_3$. 
Execution Example

- Apply $\lbrack \bar{\delta}_1 \rbrack$ and $\lbrack \bar{\delta}_2 \rbrack$ to $\rho$.
- Execute $\phi_3$ on the result to get $\bar{\delta}_3$. 
Execution Example

- Apply $[[\bar{\delta}_1]]$ and $[[\bar{\delta}_2]]$ to $\rho$.
- Execute $\phi_3$ on the result to get $\bar{\delta}_3$.
- Apply $[[\bar{\delta}_3]]$ to get the final object program.
Execution Example

- Apply $[[\delta_1]]$ and $[[\delta_2]]$ to $\rho$.
- Execute $\phi_3$ on the result to get $\bar{\delta}_3$.
- Apply $[[\delta_3]]$ to get the final object program.
Execution Example

- Apply $[[\delta_1]]$ and $[[\delta_2]]$ to $\rho$.
- Execute $\phi_3$ on the result to get $\bar{\delta}_3$.
- Apply $[[\delta_3]]$ to get the final object program.
Difference-Based Metaprogramming

Summary

- Ambiguities detected at compile-time
Difference-Based Metaprogramming

Summary

• Ambiguities detected at compile-time
• Metaprograms can inspect their environments
Difference-Based Metaprogramming

Summary

• Ambiguities detected at compile-time
• Metaprograms can inspect their environments
• Modular, declarative metaprogramming style
Difference-Based Metaprogramming

Summary

- Ambiguities detected at compile-time
- Metaprograms can inspect their environments
- Modular, declarative metaprogramming style
- Suitable for OO languages like Java
Backstage Java Implementation

- Working reference implementation available
Backstage Java Implementation

- Working reference implementation available
- Includes source (~50k SLOC)
Backstage Java Implementation

- Working reference implementation available
- Includes source (≈50k SLOC)
- Full superset of Java 1.6
BSJ Standard Library – @@Memoized

@@Memoized
BSJ Standard Library – @@Memoized

@@Memoized

- Class for generating images
BSJ Standard Library – @@Memoized

@@Memoized

- Class for generating images
- Each image is generated from pair of Colors
BSJ Standard Library – @@Memoized

@@Memoized

- Class for generating images
- Each image is generated from pair of Colors
- Memoizing image generation routine for performance
BSJ Standard Library – @@Memoized

@@Memoized

- Class for generating images
- Each image is generated from pair of Colors
- Memoizing image generation routine for performance
- Store cached images in a private Map keyed by input to generation method
public class ImageGenerator {
    public Image gen(Color a, Color b) {⋯}
}
public class ImageGenerator {
    @Memoized
    public Image gen(Color a, Color b) { [...] }
public class ImageGenerator {

    public Image gen(Color a, Color b) {

    }

}
public class ImageGenerator {

  private static class Key {
    private Color a;
    private Color b;
  }

  private Map<Key, Image> cache = new ... 

  public Image gen(Color a, Color b) {
    /* return cache value, igen as needed */
  }

}
public class ImageGenerator {
    private static class Key {
        private Color a;
        private Color b;
        ...
    }
    private Map<Key, Image> cache = new ...
    public Image gen(Color a, Color b) {\}
}
public class ImageGenerator {
    private static class Key {
        private Color a;
        private Color b;
        ...
    }
    private Map<Key, Image> cache = new ...
    public Image gen(Color a, Color b) {⋯}
}
public class ImageGenerator {
    private static class Key {
        private Color a;
        private Color b;
        ...
    }
    private Map<Key, Image> cache = new ...
    private Image igen(Color a, Color b) {
    }
}
public class ImageGenerator {
    private static class Key {
        private Color a;
        private Color b;
        ...
    }
    private Map<Key, Image> cache = new ...
    private Image igen(Color a, Color b) {...}
    public Image gen(Color a, Color b) {
        /* return cache value, igen as needed */
    }
}
public class ImageGenerator {
    @Memoized
    public Image gen(Color a, Color b) {...}
}
Difference-Based Metaprogramming
Difference-Based Metaprogramming Questions?
Expressiveness

Difference-based metaprogramming separates analysis from modification.

```java
public class Example {
    private int x = 0;
    private int y = 0;
    @@LogAndCount
    public void foo() { ... }
    @@LogAndCount
    public void bar() { ... }
}
```
Injection Conflicts

$\mathcal{M}_1 \quad \mathcal{M}_2 \quad \mathcal{M}_3$

$t$
Injection Conflicts