

# Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax

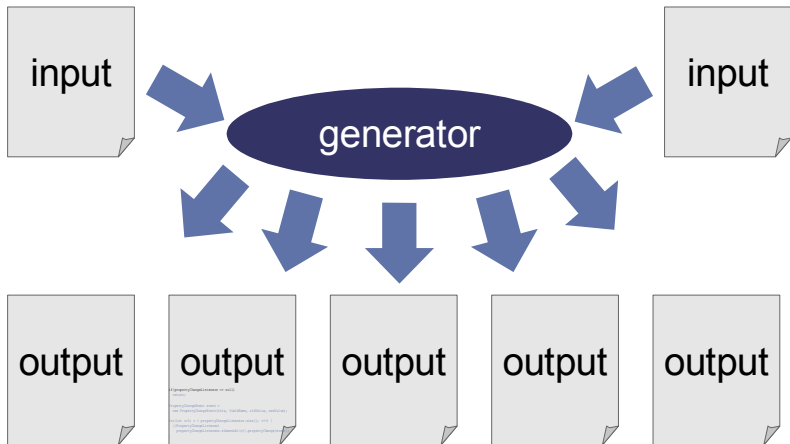
GPCE 2005

Martin Bravenboer, Rob Vermaas, Jurgen Vinju and Eelco Visser

Department of Information & Computing Sciences  
Universiteit Utrecht,  
The Netherlands

September 30, 2005

# Meta Programming: Implementing Code Generators



# Meta Programming: Implementing Code Generators



```
if(propertyChangeListeners == null)
    return;

PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

for(int c=0; c < propertyChangeListeners.size(); c++) {
    ((PropertyChangeListener)
        propertyChangeListeners.elementAt(c)).propertyChange(event);
}
```

# Meta Programming: Implementing Code Generators



```
if(propertyChangeListeners == null)
    return;

PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

for(int c=0; c < propertyChangeListeners.size(); c++) {
    ((PropertyChangeListener)
        propertyChangeListeners.elementAt(c)).propertyChange(event);
}
```

## Meta Programming with String Literals

```
String vName = "propertyChangeListeners";
jsc.add("if (");
jsc.append(vName);
jsc.append(" == null) return;");
jsc.add("PropertyChangeEvent event = new ");
jsc.append("PropertyChangeEvent");
jsc.append("(this, fieldName, oldValue, newValue);");
jsc.add("for (int i = 0; i < ");
jsc.append(vName);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(vName);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
jsc.unindent();
jsc.add("}");
```

## Meta Programming with String Literals

```
String vName = "propertyChangeListeners";
jsc.add("if (");
jsc.append(vName);
jsc.append(" == null) return;");
jsc.add("PropertyChangeEvent event = new");
jsc.append("PropertyChangeEvent");
jsc.append("(this, fieldName, oldValue,");
jsc.add("for (int i = 0; i < ");
jsc.append(vName);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(vName);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
jsc.unindent();
jsc.add("}");
```

Uses the Java syntax: the syntax of the domain.

No syntactic checks of the generated code.

Escaping to the meta language is difficult.

Code generator tries to do some pretty printing.

Further processing of the code is impossible.

## Meta Programming with Abstract Object Syntax

```
VariableDeclarationFragment fragment =
    _ast.newVariableDeclarationFragment();
fragment.setName(_ast.newSimpleName("event"));
ClassInstanceCreation newi = _ast.newClassInstanceCreation();
newi.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
List<Expression> args = newi.arguments();
args.add(_ast.newThisExpression());
args.add(_ast.newSimpleName("fieldName"));
args.add(_ast.newSimpleName("oldValue"));
args.add(_ast.newSimpleName("newValue"));
fragment.setInitializer(newi);
VariableDeclarationStatement vardec =
    _ast.newVariableDeclarationStatement(fragment);
vardec.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
```

# Meta Programming with Abstract Object Syntax

Extremely verbose and unclear: 90 lines of code!

Does not correspond to the structure of the code to be generated.

```
VariableDeclarationStatement newi =  
    _ast.newVariableDeclarationStatement(  
        fragment.setName(_ast.newSimpleName("event")),  
        ClassInstanceCreation newi = _ast.newClassInstanceCreation(  
            newi.setType(_ast.newSimpleType(  
                _ast.newSimpleName("PropertyChangeEvent"))));  
List<Expression> args = newi.arguments();  
args.add(_ast.newThisExpression());  
args.add(_ast.newSimpleName("fieldName"));  
args.add(_ast.newSimpleName("oldValue"));  
args.add(_ast.newSimpleName("newValue"));  
fragment.setInitializer(newi);  
VariableDeclarationStatement vardec =  
    _ast.newVariableDeclarationStatement(fragment),  
vardec.setType(_ast.newSimpleType(  
    _ast.newSimpleName("PropertyChangeEvent"))));
```

Syntactically checked by meta compiler and further processing is possible.

Don't worry about the layout.

# Meta Programming with Concrete Object Syntax

```
String x = "propertyChangeListeners";

List<Statement> stmts = stmt* [|
    if(#(id)[x] == null)
        return;

    PropertyChangeEvent event =
        new PropertyChangeEvent(this, fieldName, oldValue, newValue);

    for(int c=0; c < #(id)[x].size(); c++) {
        ((PropertyChangeListener)
            #(id)[x].elementAt(c)).propertyChange(event);
    }
|];
```

# Meta Programming with Concrete Object Syntax

Quotation:

`[[ Java (Object) ]]`

```
String x = "propertyChangeListeners";
```

```
List<Statement> stmts = stmt* [[
```

```
  if(#(id)[x] == null)
```

```
    return;
```

```
  PropertyChangeEvent event =
```

```
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);
```

```
  for(int c=0; c < #(id)[x].size(); c++) {
```

```
    ((PropertyChangeListener)
```

```
      #(id)[x].elementAt(c)).propertyChange(event);
```

```
  }
```

```
]];
```

# Meta Programming with Concrete Object Syntax

Anti-Quotation:

`#[ Java (Meta) ]`

```
String x = "propertyChangeListeners";
```

```
List<Statement> stmts = stmt* [[
```

```
    if( #(id)[x] == null)
```

```
        return;
```

```
    PropertyChangeEvent event =
```

```
        new PropertyChangeEvent(this, fieldName, oldValue, newValue);
```

```
    for(int c=0; c < #(id)[x].size(); c++) {
```

```
        ((PropertyChangeListener)
```

```
            #(id)[x].elementAt(c)).propertyChange(event);
```

```
    }
```

```
]];
```

# Meta Programming with Concrete Object Syntax

```
String x = "propertyChangeListeners";
```

Uses the syntax of the domain: Java.

```
List<Statement> stmts = stmt* [|  
    if( #(id)[x] == null)  
        return;
```

Syntax of the generated code is checked and further processing is possible.

```
    PropertyChangeEvent event =  
        new PropertyChangeEvent(this, fieldName, oldValue, newValue);
```

```
    for(int c=0; c < #(id)[x].size(); c++) {  
        ((PropertyChangeListener)  
            #(id)[x].elementAt(c)).propertyChange(event);
```

```
    }  
    ];
```

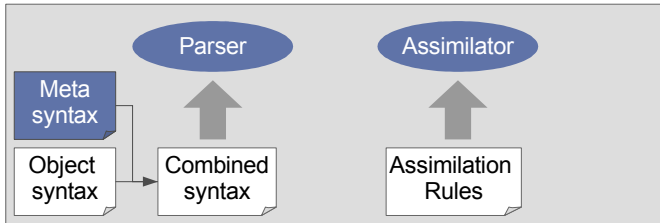
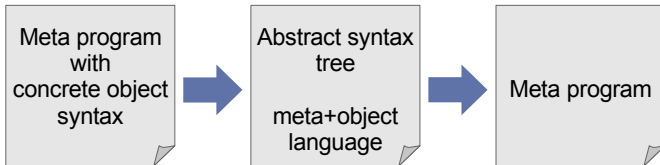
Separate pretty-printer:  
don't worry about the layout.

Support for interaction between the generated code and the meta language.

# Architecture for Realizing Concrete Object Syntax

## Parsing

## Assimilation



# Ambiguities in Concrete Object Syntax

```
String x = "propertyChangeListeners";
```

Syntactic clutter of explicit typing.

```
List<Statement> stmts = stmt* [|  
  if(#(id)[x] == null)  
    return;
```

Intimate knowledge of syntactic structure required.

```
PropertyChangeEvent event =  
  new PropertyChangeEvent(this, fieldName, oldValue, newValue);
```

```
for(int c=0; c < #(id)[x].size(); c++) {  
  ((PropertyChangeListener)  
    #(id)[x].elementAt(c)).propertyChange(event);  
}
```

```
];
```

Limited number of quotations and anti-quotations are supported.

# Ambiguities in Concrete Object Syntax

```
String x = "propertyChangeListeners";

List<Statement> stms = |[
    if(#[x] == null)
        return;

    PropertyChangeEvent event =
        new PropertyChangeEvent(this, fieldName, oldValue, newValue);

    for(int c=0; c < #[x].size(); c++) {
        ((PropertyChangeListener)
            #[x].elementAt(c)).propertyChange(event);
    }
];
```

# Ambiguities: Quotations

- Single quotation symbol
- Multiple object non-terminals

Example:

- `[[ class Foo {} ]]`

## TypeDeclaration

```
package bar;
```

```
class Foo {}
```

## CompilationUnit

```
class Foo {}
```

## Statement

```
class Bar {  
    void fred() {  
        class Foo {}  
    }  
}
```

## Ambiguities: Anti-Quotations

- Single anti-quotation symbol
- Multiple object non-terminals
- Syntactical category of meta code unknown

### Examples:

- `[[ return #[x]; ]]`
  - `x` can be `Expression`
  - `x` can be `SimpleName`
  - `x` can be `String` (identifier)
- `[[ foo(#[xs]) ]]`
  - `xs` can be `Expression`
  - `xs` can be `List<Expression>`
  - ...

# Ambiguities: Current Solutions

## Use Different (Anti-)Quotation Symbols

- JTS, DMS, Stratego, Template Haskell, Jumbo
- Redundant
- Support for (anti-)quotations irregular

## Type-based Disambiguation by Context-Sensitive Parsing

- Meta-AspectJ, Aasa (ML)
- Excellent usability
- Parsing and type-checking tangled
- Object language specific, no multiple object languages

# Generalized Type-based Disambiguation

We need a *general* architecture for type-based disambiguation.

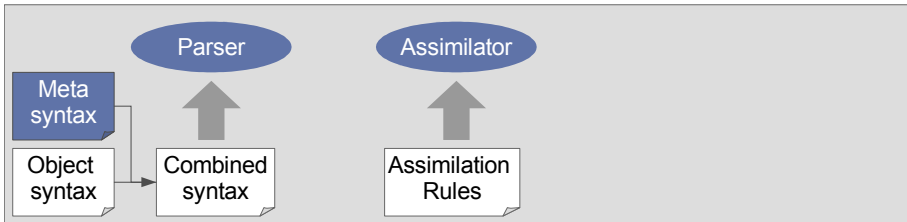
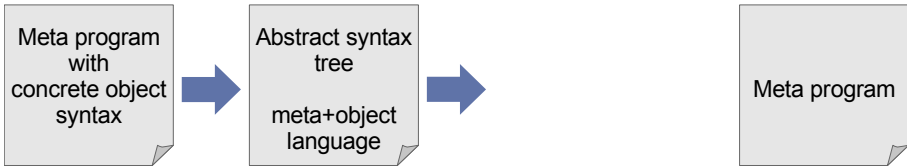
- **Introduce type-based disambiguation**
  - Single quotation and anti-quotation symbol
- **Preserve modular syntax definition**
  - Fully automatic parser generation
- **Preserve modular assimilation**
  - Embedding multiple object languages

Define only syntactical embedding and assimilation rules

# Generalized Type-based Disambiguation

## Parsing

## Assimilation

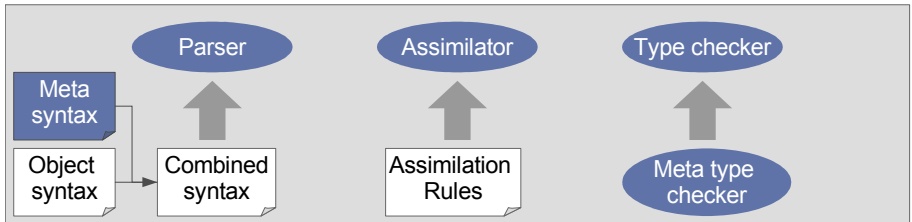
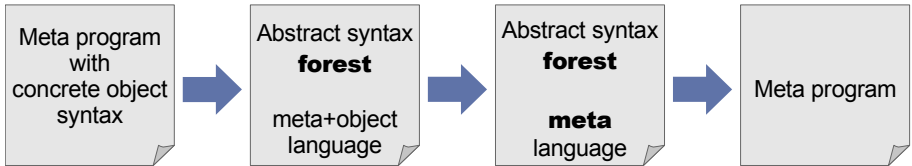


# Generalized Type-based Disambiguation

## Parsing

## Assimilation

## Type checking



# Example: Syntax Definition and Parsing

## context-free syntax

```
"[" CompilationUnit "]" -> MetaExpr {cons("ToMetaExpr")}  
"[" TypeDec          "]" -> MetaExpr {cons("ToMetaExpr")}  
"[" ClassBodyDec*   "]" -> MetaExpr {cons("ToMetaExpr")}
```

## context-free syntax

```
"#[" MetaExpr "]" -> ID   {cons("FromMetaExpr")}  
"#[" MetaExpr "]" -> Expr {cons("FromMetaExpr")}
```

```
[ package bar; class Foo {} ]
```

## ToMetaExpr( CompilationUnit( Some(PackageDec([], PackageName([Id("bar")])), [], , [ ClassDec( ClassDecHead([], Id("Foo"), None, None, None) , ClassBody([]) ) ) ]))

## Example: Syntax Definition and Parsing

```
dec = [ [ class Foo {} ] ]
```

```
Assign(ExprName(Id("dec")),  
  amb([  
    ToMetaExpr( CompilationUnit(... ClassDec(... Id("Foo"))... ) )  
  
    , ToMetaExpr( ClassDec(... Id("Foo") ... ) )  
  
    , ToMetaExpr([ ClassDec(... Id("Foo") ... ) ] )  
  
    , ToMetaExpr( ClassDecStm(ClassDec(... Id("Foo") ... ) ) )  
  
    , ToMetaExpr([ ClassDecStm(ClassDec(... Id("Foo") ... ) ) ] )  
  
  ]))
```

## Example: Assimilation

```
dec = [ [ class Foo {} ] ]
```

```
Assign(ExprName(Id("dec")),
  amb([
    { | CompilationUnit cu_0 = _ast.newCompilationUnit(); ...
      TypeDeclaration class_0 = _ast.newTypeDeclaration();
      class_0.setName(_ast.newSimpleName("Foo")); ... | cu_0 | }
    , ToMetaExpr( ClassDec(... Id("Foo") ...) )
    , ToMetaExpr([ ClassDec(... Id("Foo") ...) ] )
    , ToMetaExpr( ClassDecStm(ClassDec(... Id("Foo") ...) ) )
    , ToMetaExpr([ ClassDecStm(ClassDec(... Id("Foo") ...) ) ] )
  ]))
```

## Example: Assimilation

```
dec = [ [ class Foo {} ] ]
```

```
Assign(ExprName(Id("dec")),
  amb([
    { | CompilationUnit cu_0 = _ast.newCompilationUnit(); ...
      TypeDeclaration class_0 = _ast.newTypeDeclaration();
      class_0.setName(_ast.newSimpleName("Foo")); ... | cu_0 | }
    , { | TypeDeclaration class_1 = _ast.newTypeDeclaration();
      class_1.setName(_ast.newSimpleName("Foo")); ... | class_1 | }
    , ToMetaExpr([ ClassDec(... Id("Foo") ...) ] )
    , ToMetaExpr( ClassDecStm(ClassDec(... Id("Foo") ...) ))
    , ToMetaExpr([ ClassDecStm(ClassDec(... Id("Foo") ...) )])
  ]))
```

## Example: Assimilation

```
dec = [ class Foo {} ]
```

```
Assign(ExprName(Id("dec")),  
  amb([  
    { | CompilationUnit cu_0 = _ast.newCompilationUnit(); ...  
      TypeDeclaration class_0 = _ast.newTypeDeclaration();  
      class_0.setName(_ast.newSimpleName("Foo")); ... | cu_0 | }  
  
    , { | TypeDeclaration class_1 = _ast.newTypeDeclaration();  
      class_1.setName(_ast.newSimpleName("Foo")); ... | class_1 | }  
  
    , { | List<BodyDeclaration> decs_0 = new ArrayList<BodyDeclaration>();  
      decs_0.add( ... ); ... | decs_0 | }  
  
    , ToMetaExpr( ClassDecStm(ClassDec(... Id("Foo") ... ) ) )  
  
    , ToMetaExpr([ ClassDecStm(ClassDec(... Id("Foo") ... ) ) ] )  
  
  ]))
```

## Example: Assimilation

```
dec = [ [ class Foo {} ] ]
```

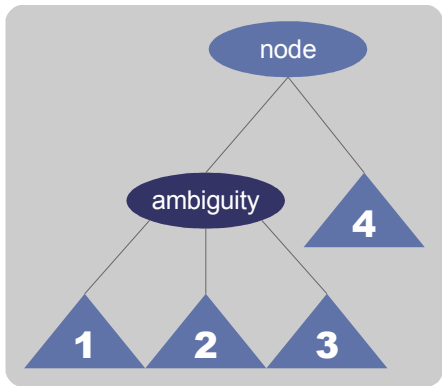
```
Assign(ExprName(Id("dec")),  
  amb([  
    { | CompilationUnit cu_0 = _ast.newCompilationUnit(); ...  
      TypeDeclaration class_0 = _ast.newTypeDeclaration();  
      class_0.setName(_ast.newSimpleName("Foo")); ... | cu_0 | }  
  
    , { | TypeDeclaration class_1 = _ast.newTypeDeclaration();  
      class_1.setName(_ast.newSimpleName("Foo")); ... | class_1 | }  
  
    , { | List<BodyDeclaration> decs_0 = new ArrayList<BodyDeclaration>();  
      decs_0.add( ... ); ... | decs_0 | }  
  
    , _ast.newTypeDeclaratonStatement(...);  
  
    , ToMetaExpr([ ClassDecStm(ClassDec(... Id("Foo") ... ) ) ] )  
  
  ]))
```

# Example: Assimilation

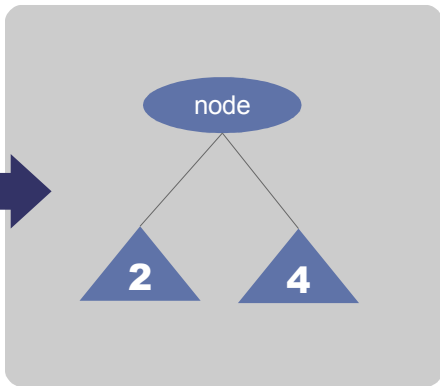
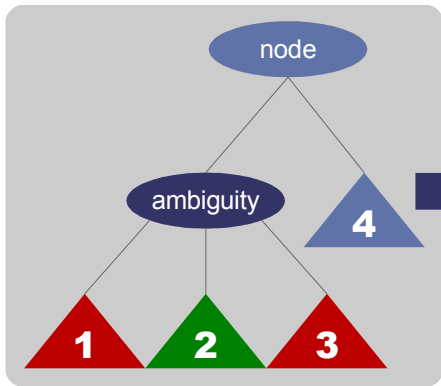
```
dec = [ [ class Foo {} ] ]
```

```
Assign(ExprName(Id("dec")),  
  amb(  
    { | CompilationUnit cu_0 = _ast.newCompilationUnit(); ...  
      TypeDeclaration class_0 = _ast.newTypeDeclaration();  
      class_0.setName(_ast.newSimpleName("Foo")); ... | cu_0 | }  
    , { | TypeDeclaration class_1 = _ast.newTypeDeclaration();  
        class_1.setName(_ast.newSimpleName("Foo")); ... | class_1 | }  
    , { | List<BodyDeclaration> decs_0 = new ArrayList<BodyDeclaration>();  
        decs_0.add( ... ); ... | decs_0 | }  
    , _ast.newTypeDeclaratonStatement(...);  
    , { | List<Statement> stms_0 = new ArrayList<Statement>();  
        stms_0.add(_ast.newTypeDeclaratonStatement(...)); ... | stms_0 | }  
  ))
```

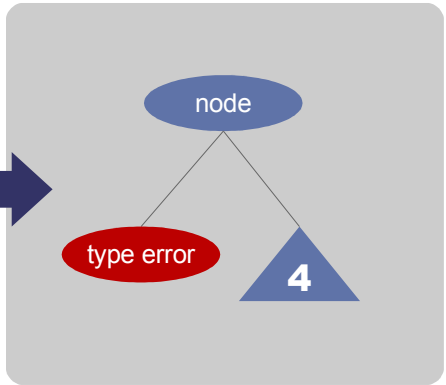
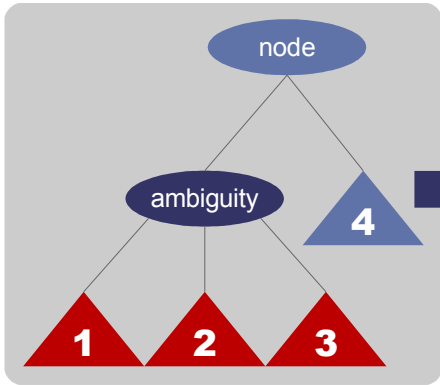
# Type-based Disambiguation



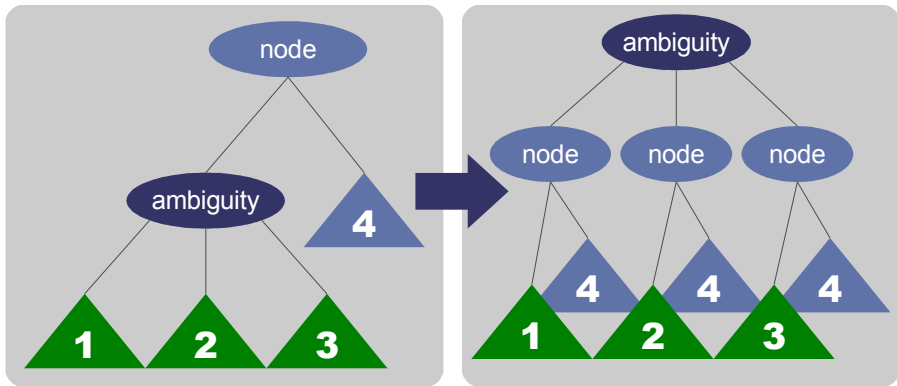
# Type-based Disambiguation



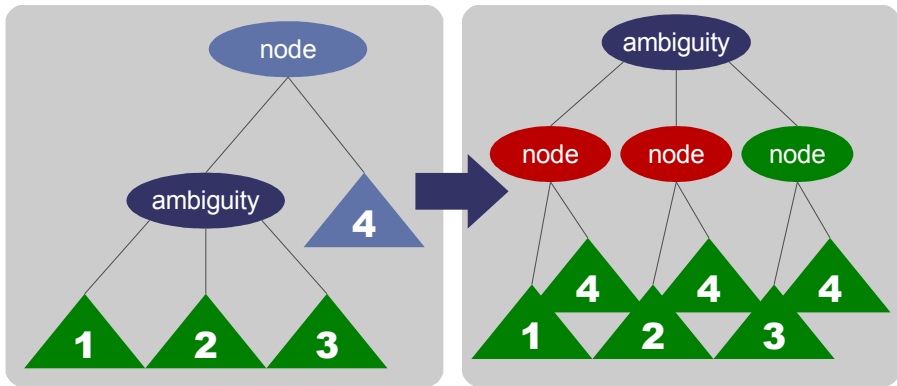
# Type-based Disambiguation



# Type-based Disambiguation



# Type-based Disambiguation



## Example: Type-based Disambiguation

```
dec = [ class Foo {} ]
```

```
Assign(ExprName(Id("dec")), amb([  
    CompilationUnit  
    , TypeDeclaration  
    , List<BodyDeclaration>  
    , TypeDeclaratonStatement  
    , List<Statement>  
]))
```

```
amb([  
    Assign(ExprName(Id("dec")), CompilationUnit)  
    , Assign(ExprName(Id("dec")), TypeDeclaration)  
    , Assign(ExprName(Id("dec")), List<BodyDeclaration>)  
    , Assign(ExprName(Id("dec")), TypeDeclaratonStatement)  
    , Assign(ExprName(Id("dec")), List<Statement>)  
])
```

## Example: Type-based Disambiguation

```
dec = [ class Foo {} ]
```

```
Assign(ExprName(Id("dec")), amb([  
    CompilationUnit  
    , TypeDeclaration  
    , List<BodyDeclaration>  
    , TypeDeclaratonStatement  
    , List<Statement>  
]))
```

```
amb([  
    Assign(ExprName(Id("dec")), CompilationUnit)  
    , Assign(ExprName(Id("dec")), TypeDeclaration)  
    , Assign(ExprName(Id("dec")), List<BodyDeclaration>)  
    , Assign(ExprName(Id("dec")), TypeDeclaratonStatement)  
    , Assign(ExprName(Id("dec")), List<Statement>)  
])
```

## Example: Anti-Quotation

```
Expression expr = ...  
Statement stmt = [| return #[expr]; |]
```

```
Return(  
  Some(  
    amb([  
      FromMetaExpr(ExprName(Id("expr")))  
      , ExprName([ FromMetaExpr(ExprName(Id("expr"))) ])  
      , ExprName([ Id(FromMetaExpr(ExprName(Id("expr")))) ])  
    ])))
```

```
Expression expr = ...  
Statement stmt = { |  
  ReturnStatement return_1 = _ast.newReturnStatement();  
  return_1.setExpression(amb([expr, _ast.newSimpleName(expr)]));  
  | return_1 |};
```

## Example: Anti-Quotation

```
Expression expr = ...  
Statement stmt = [| return #[expr]; |]
```

```
Return(  
  Some(  
    amb([  
      FromMetaExpr(ExprName(Id("expr")))  
      , ExprName([ FromMetaExpr(ExprName(Id("expr"))) ])  
      , ExprName([ Id(FromMetaExpr(ExprName(Id("expr")))) ])  
    ])))
```

```
Expression expr = ...  
Statement stmt = { |  
  ReturnStatement return_1 = _ast.newReturnStatement();  
  return_1.setExpression(amb([expr, _ast.newSimpleName(expr)]));  
  | return_1 |};
```

## Example: Anti-Quotation

```
Expression expr = ...  
Statement stmt = [| return #[expr]; |]
```

```
Return(  
  Some(  
    amb([  
      FromMetaExpr(ExprName(Id("expr")))  
      , ExprName([ FromMetaExpr(ExprName(Id("expr"))) ])  
      , ExprName([ Id(FromMetaExpr(ExprName(Id("expr")))) ])  
    ])))
```

```
Expression expr = ...  
Statement stmt = { |  
  ReturnStatement return_1 = _ast.newReturnStatement();  
  return_1.setExpression(expr);  
  | return_1 |};
```

# Explicit Disambiguation

Suppose you still want to indicate the type of a quotation.

- Some expressions can stay ambiguous
- Some systems support disambiguation syntax (e.g. Meta-AspectJ, Stratego)

```
'(CompUnit) [ class Foo {} ]  
'(ClassDec) [ class Foo {} ]
```

Type-based disambiguation supports *casting* for disambiguation.

```
(CompilationUnit) [| class Foo {} |]  
(TypeDeclaration) [| class Foo {} |]  
(List<Statement>) [| class Foo {} |]
```

Any object language construct can be disambiguated.

# Experience

## JavaJava

- Embedding of Java in Java
- Assimilation to Eclipse Java AST (JDT Core DOM)
- Eclipse Java AST is not yet parameterized

## Meta-AspectJ

- Partial implementation of Meta-AspectJ
- Assimilation to Meta-AspectJ AST.
- Support for more complex typing situations
- No object language specific conversions (reification)
- Distinct syntactical categories implement common interfaces.
- No implementation of `kwinfer`

## Assumptions and Possible Extensions

- Manifest typing (types explicitly given in declarations)
  - Other statically typed languages?
  - Disambiguation and type inferencing?
- Sufficiently typed representation is very important
  - Distinct syntactical categories → different type
  - Parameterized collections
- Object language specific type checking
- Object language specific conversions (reification)
- Explicit ambiguities in (formal) type systems

# Conclusion

Parse first, solve ambiguities later.

## Enablers

- Generalized-LR parsing produces all alternatives
- Separate type checking on ambiguous program

## Results

- Modular syntactical embedding and assimilation
- Object language independent disambiguation
- Extensible type checker
- General architecture extended with type-based disambiguation

We would like to work with you to embed  
object languages for your applications!

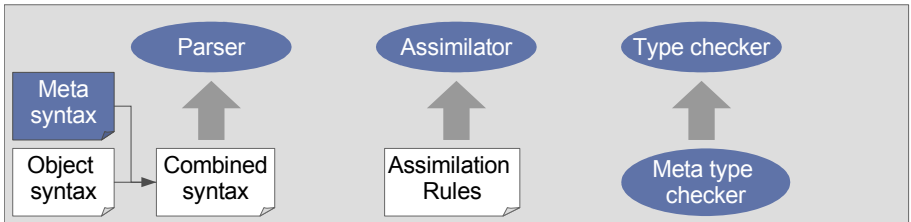
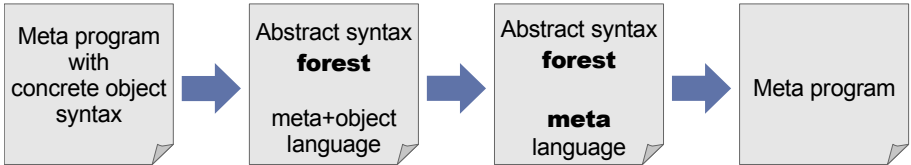
## Appendices

# Type Checking First?

## Parsing

## Assimilation

## Type checking

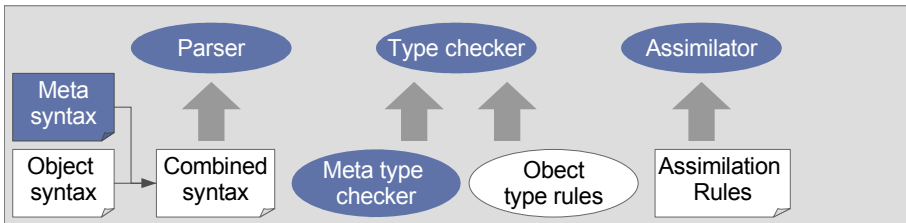
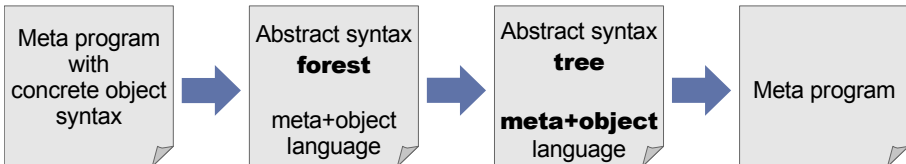


# Type Checking First?

**Parsing**

**Type checking**

**Assimilation**

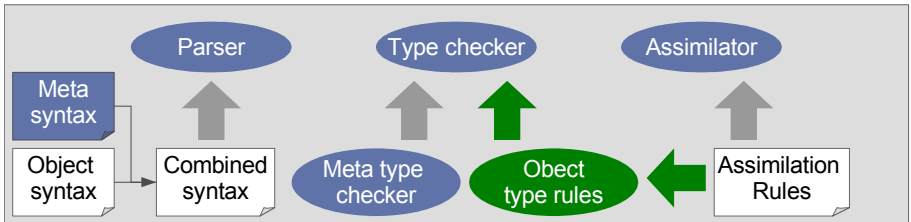
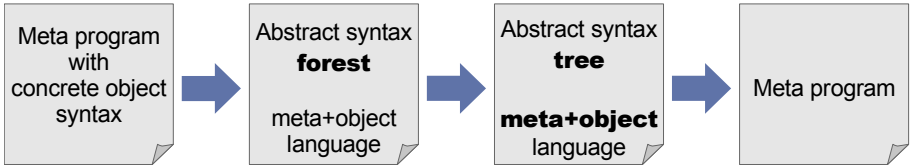


# Type Checking First?

**Parsing**

**Type checking**

**Assimilation**



# Algorithm

```
disambiguate(node) =
  if node is ambiguous then
    return resolve(node)
  else if node has ambiguous child then
    return resolve(lift-ambiguity(node))
  else return node

resolve(node) =
  node' := remove from node all alternatives which are not type correct
  if #node' == 0 then report type error
  else if #node' == 1 then return node'
  else if #node' > 1 then
    if node' contains a meta statement or declaration then
      report ambiguity error
    else return node'

lift-ambiguity(node) =
  if node == c[1> node1 2> node2 ... j> nodej] then
    return 1> c[node1] 2> c[node2] ... j> c[nodej]
```

## Interesting Situations

```
void generate() {  
    TypeDeclaration dec = foo([| return; |]);  
}  
TypeDeclaration foo(List<Statement> list) { ... }  
TypeDeclaration foo(Statement list) { ... }
```

```
void generate() {  
    TypeDeclaration dec = foo([| return; |]);  
}  
List<TypeDeclaration> foo(List<Statement> list)  
TypeDeclaration      foo(Statement list)
```

```
void generate() {  
    foo([| return; |]);  
}  
<T> void foo(List<T> list)
```

## More Interesting Situations

```
[[
  class Foo() {
    #[type] getBar()
    ...
  }
]]
```

- *type* can be Type, Id
- *type* can be BodyDeclaration
- *type* can be List<BodyDeclaration>
- *type* can be Modifier
- *type* can be List<Modifier>

## More and More Interesting Situations

```
# [ [ ] ]
```

Is not identity.

```
# [ foo( [ ] ) ]
```

## Object Language Specific Conversions

- This is allowed:

```
String x;  
e = |[ #[x] ]|;
```

- This is not:

```
e = x
```

- list of identifiers to list of expressions
- reification (e.g. meta-level array to object-level array constant)

## Explosion of ambiguities?

- No lifting beyond the statement level
  - Statement is unit of disambiguation
  - Cannot be more context information
- Number of ambiguities
  - Depends on number of quotations and anti-quotations
  - Not on size of quoted object code fragments
- Ambiguities in anti-quotations are immediately resolved if there is no quotation in the meta code.
- n-ary 'object' operators:  $amb_0 * amb_1 * \dots * amb_n$

# Compositional Assimilation Rules

```
Assimilate(r) :  
  InterfaceDec(InterfaceDecHead(mod*, Id(y), typeparams, extends), decs)  
  -> e [|  
    {| TypeDeclaration x = _ast.newTypeDeclaration();  
      x.setInterface(true);  
      x.setName(e1);  
      bstm*  
    | x |}  
  ]|  
  where ...
```

```
Assimilate(r) :  
  InterfaceDec(InterfaceDecHead(mod*, y, typeparams, extends), decs)  
  -> e [|  
    {| TypeDeclaration x = _ast.newTypeDeclaration();  
      x.setInterface(true);  
      x.setName(e1);  
      bstm*  
    | x |}  
  ]|  
  where ...
```

## Example: Assimilation

- Embedded object fragments to meta language
- Results in an ambiguous meta program

### Example Assimilation Rules

```
Assimilate(rec) :  
  [[ return; ]] -> [[ _ast.newReturnStatement() ]]  
  
Assimilate(rec) :  
  [[ e.y(e*) ]] -> [[  
    { | MethodInvocation x = _ast.newMethodInvocation();  
      x.setName(~e:<AssimilateId(rec)> y);  
      x.setExpression(~e:<rec> e);  
      bstm* | x | } ]]  
  where  
    <newname> "inv" => x  
    ; <AssimilateArgs(rec | x)> e* => bstm*
```

# More Ambiguities in Concrete Object Syntax

## Lexical state: tokens depend on context

- Current solutions:
  - Union of lexical syntax (introduces new keywords)
  - Maintain lexical state, fixed quotation symbols.
  - Interaction between lexer and parser
- Not an issue in *scannerless* parsing.

## Experience: Meta-AspectJ

- Partial implementation of Meta-AspectJ
- Assimilation to Meta-AspectJ AST.

### Interesting issues

- Syntactical embedding of AspectJ in Java is trivial
  - AspectJ has a non-trivial lexical syntax.
  - No reserved keywords
- Assimilation takes most LOC
- Support for more complex typing situations
- No object language specific conversions (reification)
- Distinct syntactical categories implement common interfaces.
- No implementation of `infer`