

Dryad & Java Front

Infrastructure for Java Transformation Systems

Sixth Stratego User Days (SUD 2005)

Martin Bravenboer

Department of Information & Computing Sciences
Utrecht University,
The Netherlands

May 3, 2005

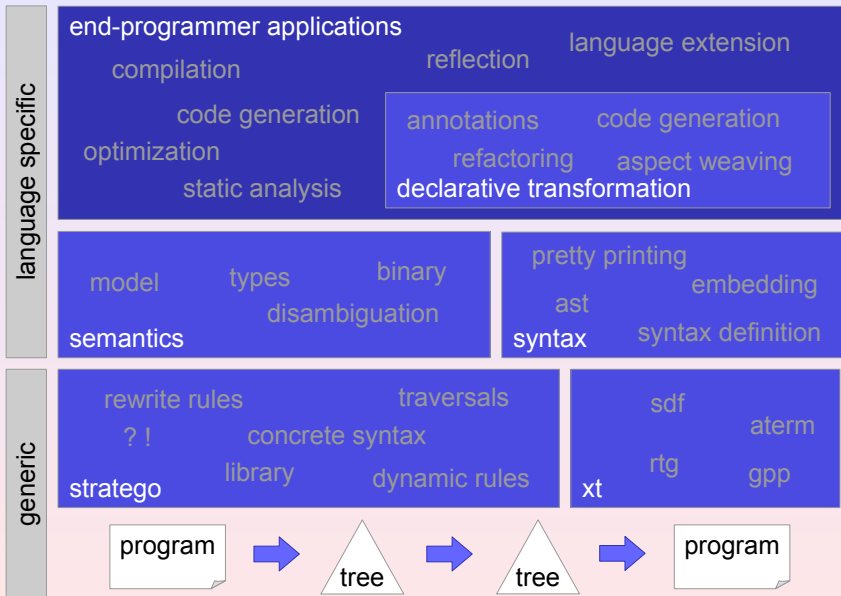
- ▶ **Stratego** – Generic language for program transformation
- ▶ **XT** – Generic infrastructure for transformation systems
- ▶ **XT Orbit** – Language specific tools

Satellite Java

- ▶ **Java Front**: Syntax related infrastructure for Java
- ▶ **Dryad**: Components for Java transformation systems
- ▶ **AspectJ Front**: extension of Java Front

Goals

- ▶ Complete tool set for Java meta-programming systems
- ▶ Compatible and up to date
- ▶ Validation: apply Stratego/XT to real-life language
- ▶ World domination



- ▶ **Syntax Definition**
- ▶ **Parser**
- ▶ **Pretty-Printer**
- ▶ **Embedding of Java**

SDF grammar for Java 2 version 1.5
(*i.e. generics, enums, annotations, ...*)

- ▶ Modular
- ▶ Structure of Java Language Specification, 3rd Edition
- ▶ Declarative disambiguation
 - ▶ Single expression non-terminal
 - ▶ Context-free priorities
 - ▶ Lexical restrictions
 - ▶ Rejections
- ▶ Integrated lexical and context-free syntax
 - ▶ Important for language extension (AspectJ)

Syntax Definition: Ambiguities

lexical restrictions

"+" -/- [\+]

ID -/- [a-zA-Z0-9_\\\$]

FloatNumeral -/- [fFdD]

lexical syntax

Keyword -> ID {reject}

context-free priorities ...

```
> {left: Expr "*" Expr -> Expr
      Expr "/" Expr -> Expr }
> {left: Expr "+" Expr -> Expr
      Expr "-" Expr -> Expr }
> {left: Expr "<<" Expr -> Expr
      Expr ">>" Expr -> Expr }
> {left: Expr "instanceof" RefType -> Expr
      Expr "<" Expr -> Expr
      Expr ">" Expr -> Expr }
> {left: Expr "==" Expr -> Expr
      Expr "!=" Expr -> Expr }
> Expr "&" Expr -> Expr
> Expr "^" Expr -> Expr
> Expr "|" Expr -> Expr ...
```

Syntax Definition: Ambiguities: Cast Expressions

JLS: Cast Expressions

```
( ReferenceType ) UnaryExpressionNotPlusMinus  
( PrimitiveType ) UnaryExpression
```

JLS: Different Priorities

```
$ echo "(Integer) - 2" | parse-java -s Expr | pp-aterm  
Minus(ExprName(Id("Integer")), Lit(Deci("2")))
```

```
$ echo "(Integer) (- 2)" | parse-java -s Expr | pp-aterm  
CastRef(  
  ClassOrInterfaceType(TypeName(Id("Integer")), None)  
  , Minus(Lit(Deci("2")))  
)
```

```
$ echo "(int) - 2" | parse-java -s Expr | pp-aterm  
CastPrim(Int, Minus(Lit(Deci("2"))))
```

Compare to C: syntactical ambiguity (see Transformers)

Syntax Definition: Ambiguities: Cast Expressions

Java Front: Cast Expressions

```
"(" PrimType ")" Expr -> Expr {cons("CastPrim")}  
"(" RefType ")" Expr -> Expr {cons("CastRef")}  
PrimType -> Type  
RefType -> Type
```

Java Front: Cast Priorities

context-free priorities

```
"(" RefType ")" Expr -> Expr  
> { "++" Expr -> Expr  
    "--" Expr -> Expr  
    "+" Expr -> Expr  
    "-" Expr -> Expr }
```

context-free priorities

```
"(" PrimType ")" Expr -> Expr  
> {left:  
    Expr "*" Expr -> Expr  
    Expr "/" Expr -> Expr  
    Expr "%" Expr -> Expr }
```

Syntax Definition: Context-Dependent Ambiguities

Java is an ambiguous language

▶ `import java.util.ArrayList`

Package, typename

```
TypeImportDec(  
  TypeName(  
    PackageOrTypeName(PackageOrTypeName(Id("java")), Id("util"))  
  , Id("ArrayList")  
)  
)
```

▶ `System.out.println("Hello world")`

Package, typename, field, local variable

```
MethodName(  
  AmbName(AmbName(Id("System")), Id("out"))  
  , Id("println")  
)
```

Syntax Definition: How to Handle Ambiguities?

- ▶ Preserve ambiguities: parse forest (GLR)
- ▶ Generalize syntactic sorts: `PackageOrTypeName`, `AmbiguousName`, `ClassOrInterfaceType`

JLS: Mixture

ReferenceType:

ClassOrInterfaceType

TypeVariable

ClassOrInterfaceType:

ClassType

InterfaceType

PackageOrTypeName

Identifier

PackageOrTypeName . Identifier

ExpressionName:

Identifier

AmbiguousName . Identifier

Syntax Definition: Context-Dependent Ambiguities

Java Front

- ▶ Non-ambiguous (ambiguities encoded in grammar)
- ▶ e.g. `PackageOrTypeName`, `AmbName`,
`ClassOrInterfaceType`

Alternative: Preserve

- ▶ Use Generalized LR and parse forest (a la Transformers)
- ▶ Declarative syntax definition
- ▶ Performance? Embedding?
- ▶ Would like to experiment with this: already have disambiguating type checker.

Alternative parsing technology resulted in various fixes in the JLS itself!

`parse-java` and `pp-java`

- ▶ Supports comment preservation (e.g. Javadoc)

`parse-java`

- ▶ Supports preserving position information

`pp-java`

- ▶ Hand-crafted in Stratego/Box
 - ▶ Good-case and worst-case formatting
 - ▶ Full Stratego pattern-matching
- ▶ Preserves priorities
 - ▶ Inserts parentheses where necessary
 - ▶ generated from SDF syntax definition

Functional tests: roundtrip

- ▶ GNU Classpath and J2SDK
- ▶ `ast = parse; ast' = parse | pp | parse; compare`
- ▶ Found various bugs in syntax definition and pretty-printer

Unit tests: parse unit

```
test always take longest match for --  
"1--2" fails
```

```
test multiplication has higher priority than addition  
"1 + 2 * 3" -> Plus(_, Mul(_, _))
```

```
test Cast operators 1  
"(int) -1" -> CastPrim(_, Minus(_))
```

```
test Cast operators 8  
"(Integer) -1 " -> Minus(ExprName(Id("Integer")),Lit(Deci("1")))
```

Concrete syntax for Java in meta language

► Stratego-Java-15

```
Explode(r) :
  bstm [[ for(lvdec; e1; e*) stm ]| -> [[
    ForStatement x1 = _ast.newForStatement();
    x1.setBody(e2);
    x1.setExpression(e3);
    x1.initializers().add(~e: <r> lvdec);
    java.util.List x2 = x1.updaters();
    bstm2*
  ]| where ...

insert-admin(|x_matcher, x_admin) :
  bstm [[ if( x_matcher.lookingAt() ) {
    bstm_inner*
  } else { bstm* }
  ]| ->
  bstm [[ if( x_matcher.lookingAt() ) {
    x_admin = x_matcher.end();
    bstm_inner*
  } else { bstm* }
  ]|
```

Embedding of Java

- ▶ Generic embedding: parameterized with expression sort of meta language
- ▶ Explicit disambiguation

```
module Embedded-Java-15[E]
imports Java-15-Prefixed
exports
  variables
    [ij] [0-9]* -> JavaDeciLiteral {prefer}
    [xyz] [0-9]* -> JavaID          {prefer}
    "e"   [0-9]* -> JavaExpr       {prefer}

  context-free syntax
    "[[" JavaBlockStm "]" " -> E {cons("ToMetaExpr")}
    "bstm" "[[" JavaBlockStm "]" " -> E {cons("ToMetaExpr")}
    "bstm*" "[[" JavaBlockStm*" "]" " -> E {cons("ToMetaListExpr")}

    "~" E -> JavaExpr {cons("FromMetaExpr")}
    "~*" E -> {JavaExpr " ," }* {cons("FromMetaExpr")}
```

Beyond support for Java syntax

- ▶ Java Bytecode \leftrightarrow ATerm bridge
- ▶ Dryad Library
 - ▶ Model for Java source and bytecode
 - ▶ Implementation of JLS definitions
- ▶ Reclassification (disambiguation) and qualification
- ▶ Type checker

Invaluable for the implementation of a Java transformation system!

- ▶ Access to bytecode is important for semantic analysis
 - ▶ Disambiguation and type-checking (used extensively)
- ▶ Bytecode represented in ATerm
 - ▶ Follows structure of the JVM Specification
 - ▶ Signature: `dryad/bytecode/signature`
- ▶ Disassembler: `class2aterm`
 - ▶ Generics Signatures
 - ▶ Local variables tables, line numbers, etc.
 - ▶ Code optional (`-c`)
- ▶ Assembler: `aterm2class`
- ▶ Implemented in Java, based on Apache's BCEL and Java ATerm Library

Java Bytecode ↔ ATerm Bridge

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Foo implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String msg = "Don't push me!";
        System.out.println(msg);
    }
}
```

Java Bytecode ↔ ATerm Bridge

```
ClassFile(  
  MinorVersion(0), MajorVersion(49)  
  , AccessFlags([Super, Public])  
  , ThisClass("Foo")  
  , SuperClass(Some("java.lang.Object"))  
  , Interfaces(["java.awt.event.ActionListener"])  
  , Fields([])  
  , Methods([  
    Method(  
      AccessFlags([Public])  
      , Name("<init>")  
      , MethodDescriptor([], Void)  
      , Attributes([])  
    )  
    , Method(  
      AccessFlags([Public])  
      , Name("actionPerformed")  
      , MethodDescriptor([ObjectType("java.awt.event.ActionEvent")], Void)  
      , Attributes([])  
    )  
  ])  
  , Attributes([SourceFile("Foo.java")])  
)
```

Java Bytecode ↔ ATerm Bridge

```
Code(  
  MaxStack(Some(2))  
  , MaxLocals(Some(3))  
  , Instructions([  
    LDC(String("Don't push me!"))  
    , ASTORE(2)  
    , GETSTATIC(  
      FieldRef(Class("java.lang.System"), Name("out")  
        , FieldDescriptor(ObjectType("java.io.PrintStream"))  
      )  
    , ALOAD(2)  
    , INVOKEVIRTUAL(  
      MethodRef(  
        Class("java.io.PrintStream"), Name("println")  
        , MethodDescriptor([ObjectType("java.lang.String")], Void)  
      )  
    , RETURN  
  ])  
  , ExceptionTable([])  
  , Attributes([])  
)
```

Dryad Model

- ▶ `dryad/model/-`: representation of source and bytecode
- ▶ Object oriented
- ▶ Global structure linked: `get-superclass`, etc.
- ▶ `get-methods`, `get-fields`, `get-formal-parameter-types`, ...
- ▶ Abstraction over source code or bytecode

JLS definitions

- ▶ Based on model
- ▶ Conversions and types
- ▶ For example: `is-assignment-convertable(|t)`, `supertypes`, `is-subtype(|type)`
- ▶ `<proper-supertypes> Int() => [Long, Float, Double]`
- ▶ Many of these definitions are non-trivial

Dryad: Reclassification and Qualification

`dryad-front (dryad-reclassify-ambnames)`

- ▶ Reclassification (names, types)
 - ▶ Contextually dependent names
- ▶ Qualification (types)
 - ▶ Unqualified names are hard too handle
 - ▶ Use fully qualified names in transformations

Complex

- ▶ Imports, on demand imports, static imports
- ▶ Inner classes, non-trivial rules for visibility and shadowing
- ▶ Complex scoping rules
 - ▶ Even bugs in Sun's Java compiler
 - ▶ ... and the JLS
- ▶ Transformation should not be bothered with this

Implementation

- ▶ Strategies and scoped dynamic rules

Dryad R&Q: TypeName versus PackageName

Java Source

```
import java.util.ArrayList;
```

Parse

```
TypeImportDec(  
  TypeName(  
    PackageOrTypeName(  
      PackageOrTypeName(Id("java")), Id("util")  
    )  
  , Id("ArrayList")  
))
```

Reclassify

```
TypeImportDec(  
  TypeName(  
    PackageName([Id("java"), Id("util")])  
  , Id("ArrayList")  
))
```

Java Source

```
System.out.println("Hello World!");
```

Parse

```
MethodName(  
  AmbName(AmbName(Id("System")), Id("out"))  
, Id("println"))
```

Reclassify

```
MethodName(  
  ExprName(  
    TypeName(PackageName([Id("java"), Id("lang")])  
      , Id("System"))  
    , Id("out")  
  )  
, Id("println"))
```

Dryad R&Q: ClassType, InterfaceType and Qualification

Java Source

```
import java.util.List;

public class Foo {
    List getFoo() {};
}
```

Parse

```
MethodDecHead(...,
  ClassOrInterfaceType(TypeName(Id("List")), None)
  ... )
```

Reclassify

```
MethodDecHead(...,
  InterfaceType(
    TypeName(PackageName([Id("java"), Id("util")])
      , Id("List")), None)
  ...)
```

`dryad-front --tc on (dryad-type-checker)`

- ▶ Annotates expressions with their types ... or not
- ▶ Type-aware transformations
 - ▶ e.g. extract method, inner class lifting
- ▶ Also available as a library
 - ▶ `dryad/type-check/-`
- ▶ Implementation: rewrite-rules and scoped dynamic rules

Status

- ▶ Basic operators and method resolution works
- ▶ But, not yet complete: no inner classes, no access modifiers
- ▶ No error reporting: only annotation
- ▶ `dryad-vis-tc-jtree`: show untyped expressions

```
System.out.println("Hello World!")
```

```
Invoke(  
  Method(  
    MethodName(  
      ExprName(  
        TypeName(PackageName([Id("java"), Id("lang")]) , Id("System"))  
        , Id("out")  
      ){ ClassType(  
        TypeName(PackageName([Id("java"), Id("io")])  
          , Id("PrintStream"))  
        , None  
      )  
    }  
    , Id("println")  
  )  
)  
, [ Lit(String([Chars("Hello World!")]))]{  
  ClassType(TypeName(PackageName([Id("java"), Id("lang")])  
    , Id("String")), None)  
}  
]  
){Void}
```

Application: JavaJava

Embed Java syntax in Java

context-free syntax

```
"type" "[" Type "]" -> MetaExpr {cons("ToMetaExpr")}
```

variables

```
"e" [0-9]* -> Expr {prefer}
```

```
"e" [0-9]* "*" -> {Expr ", "*} {prefer}
```

Assimilation rules for Eclipse JDT Core API

Assimilate(r) :

```
type [ double ] -> [ ast.newPrimitiveType(PrimitiveType.DOUBLE) ]
```

Assimilate(r) :

```
[ y(e*) ] -> [
```

```
{ | MethodInvocation x = ast.newMethodInvocation();
```

```
  x.setName(ast.newSimpleName("~y"));
```

```
  bstm* | x |}
```

```
]
```

```
where <newname> "inv" => x
```

```
; <ExplodeArgs(r | x)> e* => bstm*
```

JavaJava: Ambiguities

Problem: explicit disambiguation in JavaJava

- ▶ Indicate syntactic sort in (anti-)quotations
- ▶ Naming convention for variables

```
String x = "Foo";
```

```
CompilationUnit dec = compilation-unit [[  
  public class x {  
    public static void main(String[] args) {  
      System.out.println("Hello world");  
    }  
  } ]];
```

Solution: type-based disambiguation

- ▶ No explicit disambiguation: parse forest
- ▶ Ambiguities eliminated in extension of Dryad type checker

JavaJava: Ambiguous Embedding

context-free syntax

```
"[" CompUnit "]" -> MetaExpr {cons("ToMetaExpr")}  
"[" TypeDec "]" -> MetaExpr {cons("ToMetaExpr")}  
"[" BlockStm "]" -> MetaExpr {cons("ToMetaExpr")}  
"[" BlockStm* "]" -> MetaExpr {cons("ToMetaExpr")}
```

context-free syntax

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

variables

```
MetaVarID -> ID  
MetaVarID -> Expr  
MetaVarID -> {Expr ", "}
```

lexical syntax

```
[A-Za-z\_\\$][A-Za-z0-9\_\\$]* -> MetaVarID
```

JavaJava: Disambiguation

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

Ambiguity Lifting

```
dec = 1> CompUnit 2> TypeDec 3> List<BodyDec>
```

```
1> dec = CompUnit 2> dec = TypeDec 3> dec = List<BodyDec>
```

```
f(1> CompUnit 2> TypeDec 3> List<BodyDec>)
```

```
1> f(CompUnit) 2> f(TypeDec) 3> f(List<BodyDec>)
```

Conclusion

- ▶ Working solid support for implementing Java transformations systems
- ▶ Some components are finished, some are work in progress

What's next?

- ▶ Full type checker
- ▶ Extensibility of components
- ▶ High-level transformation for Java
- ▶ JVM Bytecode back-end

See Also

- ▶ <http://dryad.stratego.org> – Dryad website
- ▶ <http://planet.stratego.org> – Weblog Karl, Rob & Martin