

Infrastructure for Java Program Transformation

Master Course Program Transformation 2005-2006

Martin Bravenboer

Department of Information & Computing Sciences
Utrecht University,
The Netherlands

February 9, 2006

- **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**: Semantic analysis for Java transformation systems
- **AspectJ Front**: extension of Java Front

Applications

- **JavaBorg**: Swul, Regular Expressions, XML, JavaJava, XPath
- **Lutin**: decoupling constraints, architectural patterns
- **Prototypes**: obfuscation, refactorings, generators, compiler
- *<your application here>*

Java-front

- Syntax
- Parser
- Pretty-printer

Dryad

- Reclassification and qualification of names
- Type checker
- Library for introspection unifying source and bytecode
- Library for JLS definitions: subtyping, conversions, etc
- Bytecode disassembler and assembler

Syntax Definition

SDF grammar for Java 5.0

(*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)
- Parser generated from syntax definition

Parsing Java: CompilationUnit

```
$ echo "class Foo {}" | parse-java | pp-aterm
CompilationUnit(
  None
, []
, [ ClassDec(
      ClassDecHead([], Id("Foo"), None, None, None)
    , ClassBody([])
  )
]
)
```

```
$ echo "package foo; class Foo" | parse-java | pp-aterm
CompilationUnit(
  Some(PackageDec([], PackageName([Id("foo")])))
, []
, [ ClassDec( ... ) ]
)
```

Parsing Java: Expressions

```
$ echo "1 + x + xs[4]" | parse-java -s Expr | pp-aterm  
Plus(  
  Plus(Lit(Deci("1")), ExprName(Id("x")))  
  , ArrayAccess(ExprName(Id("xs")), Lit(Deci("4")))  
)
```

```
$ echo "this.y" | parse-java -s Expr | pp-aterm  
Field(This, Id("y"))
```

```
$ echo "x.y" | parse-java -s Expr | pp-aterm  
ExprName(AmbName(Id("x")), Id("y"))
```

Parsing Java: Method Invocations

```
$ echo "f()" | parse-java -s Expr | pp-aterm
Invoke(Method(MethodName(Id("f"))), [])
```

```
$ echo "this.f()" | parse-java -s Expr | pp-aterm
Invoke(Method(This, None, Id("f")), [])
```

```
$ echo "f().g()" | parse-java -s Expr | pp-aterm
Invoke(
  Method(
    Invoke(Method(MethodName(Id("f"))), [])
    , None
    , Id("g")
  )
, []
)
```

```
$ echo "super.f()" | parse-java -s Expr | pp-aterm
Invoke(SuperMethod(None, Id("f")), [])
```

Parsing Java: Types

```
$ echo "int" | parse-java -s Type | pp-aterm
Int
```

```
$ echo "int[] []" | parse-java -s Type | pp-aterm
ArrayType(ArrayType(Int))
```

```
$ echo "String" | parse-java -s Type | pp-aterm
ClassOrInterfaceType(TypeName(Id("String")), None)
```

```
$ echo "List<String>" | parse-java -s Type | pp-aterm
ClassOrInterfaceType(
  TypeName(Id("List"))
, Some(TypeArgs([
  ClassOrInterfaceType(TypeName(Id("String")), None)
])))
)
```

Parsing Java: Class Declaration

```
$ echo "public class Foo<E> extends Bar implements Fred {}"  
CompilationUnit(  
  None  
, []  
, [ ClassDec(  
  ClassDecHead(  
    [Public]  
  , Id("Foo")  
  , Some(TypeParams([TypeParam(Id("E"), None)]))  
  , Some(SuperDec(ClassType(TypeName(Id("Bar")), None)))  
  , Some(ImplementsDec([  
    InterfaceType(TypeName(Id("Fred")), None)  
  ]))  
  )  
  , ClassBody([])  
  )  
  ]  
)
```

Syntax Definition: Disambiguation of Expressions

<code>e++ e--</code>	PostfixExpr
<code>++e --e + - ~ ! (t)e</code>	UnaryExpr
<code>* / %</code>	MultiplicativeExpr
<code>+ -</code>	AdditiveExpr
<code><< >> >>></code>	ShiftExpr
<code>instanceof < > <= >=</code>	RelationalExpr
<code>== !=</code>	EqualityExpr
<code>&</code>	AndExpr
<code>^</code>	ExclusiveOrExpr
<code> </code>	InclusiveOrExpr
<code>&&</code>	ConditionalAndExpr
<code> </code>	ConditionalOrExpr
<code>? :</code>	ConditionalExpr
<code>= *= /= \% = += -= <<= ...</code>	AssignmentExpr

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?

Java Front

- Non-ambiguous (ambiguities encoded in grammar)
- Generalize syntactic sorts

context-free syntax

```
Id                -> AmbName {cons("AmbName")}
```

```
AmbName "." Id -> AmbName {cons("AmbName")}
```

```
Id                -> ExprName {cons("ExprName")}
```

```
AmbName "." Id -> ExprName {cons("ExprName")}
```

```
Id                -> TypeName {cons("TypeName")}
```

```
PackageOrTypeName "." Id -> TypeName {cons("TypeName")}
```

```
Id                -> PackageOrTypeName {...}
```

```
PackageOrTypeName "." Id -> PackageOrTypeName {...}
```

Syntax Definition: How to Handle Ambiguities?

Alternative: Ambiguous Syntax Definition

- Generalized LR and parse forest (a la Transformers)
- Declarative syntax definition
- Performance?

context-free syntax

```
{Id "."}+ -> PackageName {cons("PackageName")}
```

```
Id -> TypeName {cons("TypeName")}
```

```
PackageName "." Id -> TypeName {cons("TypeName")}
```

```
TypeName "." Id -> TypeName {cons("TypeName")}
```

```
Id -> ExprName {cons("ExprName")}
```

```
TypeName "." Id -> ExprName {cons("ExprName")}
```

```
ExprName "." Id -> ExprName {cons("ExprName")}
```

Syntax Definition: How to Handle Ambiguities?

```
System.out.println("Hello world")
```

Generalize syntactic sorts

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

Ambiguous

```
MethodName(  
  amb(  
    [ ExprName(TypeName(Id("System")), Id("out"))  
      , ExprName(ExprName(Id("System")), Id("out"))  
      , TypeName(PackageName([Id("System")]), Id("out"))  
      , TypeName(TypeName(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*

Java Pretty Printer

```
$ cat Foo.java
public class Foo {
    public void bar() {
        if(true) {
            System.out.println("Stratego Rules!");
        }
    }
}
```

```
$ parse-java -i Foo.java | pp-java
public class Foo
{
    public void bar()
    {
        if(true)
        {
            System.out.println("Stratego Rules!");
        }
    }
}
```

Java Pretty Printer

```
Mul(  
  Lit(Deci("1"))  
  , Plus(Lit(Deci("2")), Lit(Deci("3")))  
)  
$ pp-java -i Foo.jtree  
1 * (2 + 3)
```

```
CastRef(  
  ClassOrInterfaceType(TypeName(Id("Integer")), None)  
  , Minus(Lit(Deci("2")))  
)  
$ pp-java -i Foo.aterm  
(Integer)(-2)
```

```
CastPrim(Int, Minus(Lit(Deci("2"))))  
$ pp-java -i Foo.aterm  
(int)-2
```

Java Pretty Printer

```
public class Foo {  
    /**  
     * This method reports the universal truth.  
     */  
    public void bar() {  
        // What an understatement!  
        System.out.println("Stratego Rules!");  
    }  
}
```

```
$ parse-java --preserve-comments -i Foo.java | pp-java
```

```
public class Foo  
{  
    /**  
     * This method reports the universal truth.  
     */  
    public void bar()  
    {  
        // What an understatement!  
        System.out.println("Stratego Rules!");  
    }  
}
```

Parsing Java usually does not provide enough information for performing a program transformation.

- Ambiguous names and constructs
 - Type, package, or expression?
 - `java.awt.List` or `java.util.List`?
- Type information
 - Required for many transformations
- Basic definitions
 - Subtyping, conversions, method resolution, access control, ...
- Environment and program representation
 - Class hierarchies, unify source and bytecode
 - Access Java bytecode

Dryad, The Tree Nymph

Parsing Java usually does not provide enough information for performing a program transformation.

- Ambiguous names and constructs
 - Type, package, or expression?
 - `java.awt.List` or `java.util.List`?
- Type information
 - Required for many transformations
- Basic definitions
 - Subtyping, conversions, method resolution
- Environment and program representation
 - Class hierarchies, unify source and bytecode
 - Access Java bytecode

Dryad: reclassification and qualification of names

Dryad: type checking and annotation of types and definitions

Dryad: library for basic definitions and code model.

Dryad: Reclassification and Qualification

`dryad-front`

- Reclassification (names, types)
 - Contextually dependent names
- Qualification (types)
 - Unqualified names are hard too handle
 - Use canonical 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: Example

```
$ cat Foo.java
import java.util.List;

class Foo {
    List bar() {
        return null;
    }
}
```

Dryad R&Q: Example

```
$ parse-java -i Foo.java | pp-aterm
CompilationUnit(
  None
, [ TypeImportDec(
    TypeName(
      PackageOrTypeName(PackageOrTypeName(Id("java")), Id("util"))
    , Id("List")
    )
  )
]
, [ ClassDec(
  ClassDecHead([], Id("Foo"), None, None, None)
, ClassBody(
  [ MethodDec(
    MethodDecHead(
      []
    , None
    , ClassOrInterfaceType(TypeName(Id("List")), None)
    , Id("bar")
    , []
    , None
    )
  )
  ...
```

Dryad R&Q: Example

```
$ dryad-front -i Foo.java | pp-aterm
CompilationUnit(
  None
, [ TypeImportDec(
    TypeName(
      PackageName([Id("java"), Id("util")])
    , Id("List")
    )
  )
]
, [ ClassDec(
  ClassDecHead([], Id("Foo"), None, None, None)
, ClassBody(
  [ MethodDec(
    MethodDecHead(
      []
    , None
    , InterfaceType(
      TypeName(
        PackageName([Id("java"), Id("util")])
      , Id("List")
      )
    , None
  )

```

Dryad R&Q: TypeName versus PackageName

```
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")  
))
```

Dryad R&Q: AmbName

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

Parse

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

Reclassify

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

Dryad R&Q: ClassType, InterfaceType and Qualification

```
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 R&Q: Fields and Local Variables

```
class Foo {  
  int x;  
  int getX() { return x; }  
  void setX(int x) { this.x = x; }  
}
```

Parse

```
Return(Some(ExprName(Id("x"))))  
ExprStm(  
  Assign(Field(This, Id("x")), ExprName(Id("x")))  
)
```

Reclassify

```
Return(Some(Field(Id("x"))))  
ExprStm(  
  Assign(Field(This, Id("x")), ExprName(Id("x")))  
)
```

Dryad Type Checker

`dryad-front --tc on`

- Annotates expressions with their types ... or not
- Annotates conversions
- Annotates declaring classes (future: signatures)

- Type-aware or API specific transformations
- Implementation: rewrite-rules and scoped dynamic rules
- No error reporting: only annotation
- `dryad-vis-tc-jtree`: show untyped expressions

Dryad Type Checker

1 + 5

```
Plus(  
  Lit(Deci("1")){ Type(Int) }  
  , Lit(Deci("5")){ Type(Int) }  
) { Type(Int) }
```

"test " + 123

```
Plus(  
  Lit(String([Chars("test")])) Type(String)  
  , Lit(Deci("123")){ Type(Int) }  
) { Type(String) }
```

this

```
This{  
  Type(ClassType(TypeName(PackageName([]), Id("Foo")), None))  
}
```

Dryad Type Checker

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

```
Field(  
  TypeName(  
    PackageName([Id("java"), Id("lang")])  
    , Id("System")  
  )  
  , Id("out")  
) { Type(  
  ClassType(  
    TypeName(  
      PackageName([Id("java"), Id("io")])  
      , Id("PrintStream")  
    )  
    , None  
  )  
  )  
  , DeclaringClass(  
    TypeName(PackageName([Id("java"), Id("lang")]), Id("System"))  
  )  
}
```

Dryad Type Checker

```
double d; d = 1;
```

```
Assign(...){ Type(Double), AssignmentConversion(  
  [WideningPrimitiveConversion(Int, Double)] ) }
```

```
Number n; n = 1;
```

```
Assign(...){ ..., AssignmentConversion(  
  [ BoxingConversion(Int, RefInteger)  
  , WideningReferenceConversion([RefNumber, RefInteger])  
  ])}  
])}
```

```
List<String> list; list = new ArrayList();
```

```
Assign(...){ ..., AssignmentConversion(  
  [ WideningReferenceConversion(  
    [ Raw List  
    , Raw AbstractList  
    , Raw ArrayList  
    ])  
  , UncheckedConversion(Raw List, List<String>)  
  ])}  
])}
```

Dryad Model

- Representation of source and bytecode classes
- repository of available classes
- Classes, methods, fields, packages: lookup by name
- Graph + tree (more on that later)
- For example:
 - `get-superclass`, `get-inherited-methods`, `get-methods`,
`get-fields` `get-declaring-class`,
`get-formal-parameter-types`, ...

JLS definitions

- Conversions, types, access-control
- For example:
 - `is-subtype(| type)`
 - `is-assignment-convertable(| t)`,
 - `is-accessible-from(| from)`
 - `supertypes`

Java Bytecode ↔ ATerm Bridge

- 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([])  
)
```

- Parsing, pretty-printing, reclassification, qualification, type-checking all available for immediate use.
- Cool program transformations without much effort.
- Might all sound complex, but complexity is mostly hidden.
- Bugs are not impossible:
 - **** parser
 - **** pretty printer
 - *** reclassification
 - *** model
 - ** type checker

See Also

- <http://www.stratego-language.org/Stratego/JavaFront>
- <http://www.stratego-language.org/Stratego/TheDryad>