

Implementing Java-like languages in Xtext with Xsemantics

Lorenzo Bettini

Dipartimento di Informatica, Università di Torino, Italy

PISA, CINA Kick-off Meeting, 5 Feb 2013.

XTEXT Eclipse framework

- XTEXT provides a higher-level framework that generates most of the typical and recurrent artifacts necessary for a fully-fledged IDE on top of Eclipse.
 - Really quick and easy to have a working implementation
 - Implement while designing and formalizing

XTEXT Eclipse framework

- XTEXT provides a higher-level framework that generates most of the typical and recurrent artifacts necessary for a fully-fledged IDE on top of Eclipse.
 - Really quick and easy to have a working implementation
 - Implement while designing and formalizing
- Type system and reduction rules still implemented in Java
- Gap between the formalization and implementation

FEATHERWEIGHT JAVA

- a lightweight functional version of Java:
 - mutually recursive class definitions,
 - class inheritance,
 - object creation,
 - method invocation,
 - method recursion through `this`,
 - subtyping and
 - field access.

A. Igarashi, B. Pierce, and P. Wadler. **Featherweight Java: a minimal core calculus for Java and GJ**. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

XTEXT Eclipse framework

- Write the grammar of the language using an EBNF-like syntax
- XTEXT generates an ANTLR parser.
- During parsing, the AST is generated in the shape of an EMF model

XTEXT Eclipse framework

- Write the grammar of the language using an EBNF-like syntax
- XTEXT generates an ANTLR parser.
- During parsing, the AST is generated in the shape of an EMF model

Selection: receiver=Expression ' .' message=[Member]
(' (' (args+=Expression (' , ' args+=Expression) *) ? ') ') ? ;

XTEXT Eclipse framework

- Write the grammar of the language using an EBNF-like syntax
- XTEXT generates an ANTLR parser.
- During parsing, the AST is generated in the shape of an EMF model

Selection: receiver=Expression ' .' message=[Member
(' (' (args+=Expression (' , ' args+=Expression)*)? ') ')?)? ;

```
interface Selection extends Expression {  
    Expression getReceiver();  
    Member getMessage();  
    EList<Expression> getArgs();  
}
```

FEATHERWEIGHT JAVA grammar in XTEXT

```
Program: (classes += Class)* (main = Expression)?;  
Class: 'class' name=ID ('extends' superclass=[Class])? '{'  
    (members += Member)* '}' ;  
Member: Field | Method;  
Field: type=[Class] name=ID ';' ;  
Method: type=[Class] name=ID  
    '(' (params += Parameter (',' params += Parameter)*)? ')'  
    '{' body=MethodBody '}' ;  
Parameter: type=[Class] name=ID;  
TypedElement: Member | Parameter;  
MethodBody: 'return' expression=Expression ';' ;
```


XTEXT FJ grammar (part II)

Expression: Selection | TerminalExpression ;

Selection: receiver=Expression '.' message=[Member]
('(' (args+=Expression (',' args+=Expression)*)? ')')? ;

TerminalExpression: This | ParamRef | New | Cast | Paren ;

This: variable='this';

ParamRef: parameter=[Parameter];

New: 'new' type=ClassType

' (' (args+=Expression (',' args+=Expression)*)? ')');

Cast: '(' type=[Class] ')' expression=TerminalExpression;

Paren: '(' Expression ')';

FJ IDE

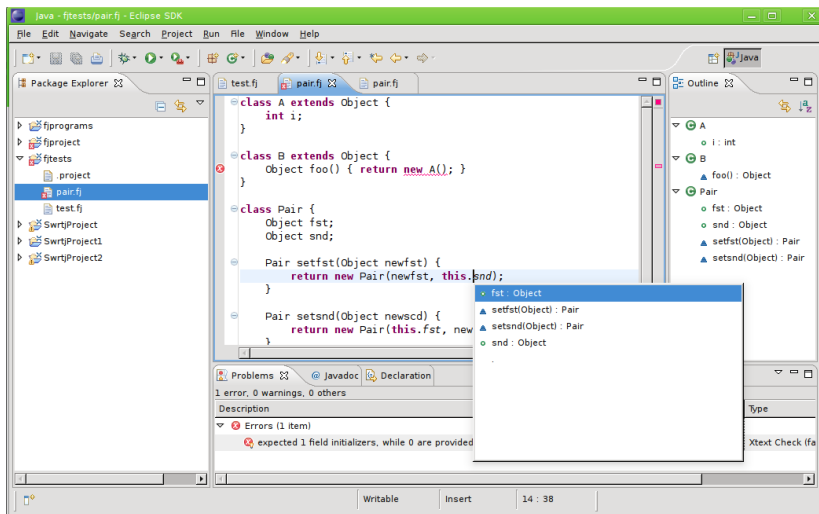


Figure: A screenshot of the FJ IDE.

Checking that a program is semantically correct in the XTEXT:

- *scoping*: cross references can be resolved, e.g., the binding of a variable to its definition in the program

```
class A {  
    String s;  
    String toString() {  
        return this.s;  
    }  
}
```

Checking that a program is semantically correct in the XTEXT:

- *scoping*: cross references can be resolved, e.g., the binding of a variable to its definition in the program

```
class A {  
    String s;  
    String toString() {  
        return this.s;  
    }  
}
```

- *validation*, the AST model is correct, e.g., checking that the return value of a method body is consistent with the method signature

```
class A {  
    String s;  
    String toString() {  
        return 10;  
    }  
}
```

Our aim

- Scoping and Validation usually rely on **types**;
- the programmer needs to implement a **type system** in Java.
- Instead, we would like to implement the type system functionalities with a DSL.
- This would allow us to have the typing rules in a compact form,
- and then have the corresponding Java code

Our proposal

We present **XSEMANTICS**, a DSL for writing rules for languages implemented in Xtext

- the static semantics (type system),
- the dynamic semantics (operational semantics) and
- relation rules (subtyping).

XSEMANTICS

- A system definition in XSEMANTICS is
 - a set of *judgments*
 - and a set of *rules* which have a conclusion and a set of premises (and a rule environment)

XSEMANTICS

- A system definition in XSEMANTICS is
 - a set of *judgments*
 - and a set of *rules* which have a conclusion and a set of premises (and a rule environment)

Starting from the definitions of judgments and rules, XSEMANTICS generates Java code that can be used for scoping and validation.

XSEMANTICS and Xbase

- an extensible and reusable statically typed expression language
- a Java with “less noise”
 - type inference
 - closures
- integrates completely with Java and Eclipse JDT
- full access to Java type system

XSEMANTICS and Xbase

- an extensible and reusable statically typed expression language
- a Java with “less noise”
 - type inference
 - closures
- integrates completely with Java and Eclipse JDT
- full access to Java type system

Example

```
val personList = newArrayList(  
    new Person("James", "Smith", 50),  
    new Person("John", "Smith", 40),  
    new Person("James", "Anderson", 40),  
    new Person("John", "Anderson", 30),  
    new Person("Paul", "Anderson", 30))  
personList.filter[firstname.startsWith("J")].  
    sortBy[age].take(3).map[surname + ", " + firstname].  
    join("; ")
```

FJ judgments in XSEMANTICS

```

judgments {
  type |- Expression expression : output Class
    error "cannot type " + expression
  subtype |- Class left <: Class right
    error left + " is not a subtype of " + right
  subtypesequences |-
    List<Expression> expressions << List<? extends TypedElement> elements
  reduce |- Expression exp ~> output Expression
}

```

XSEMANTICS rules

```
rule MyRule
  G |- Selection exp : Class type
from {
  // premises
  type = ... // assignment to output parameter
}
```

XSEMANTICS rules

```
rule MyRule
```

```
  G |- Selection exp : Class type
```

```
from {
```

```
  // premises
```

```
  type = ... // assignment to output parameter
```

```
}
```

```
rule MyRule
```

```
  G |- Selection exp : exp.message.type
```

```
from {
```

```
  // premises
```

```
}
```

XSEMANTICS rules

rule MyRule

G |- Selection exp : Class type

from {

// premises

type = ... *// assignment to output parameter*

}

rule MyRule

G |- Selection exp : exp.message.type

from {

// premises

}

must “respect” the judgment

judgments {

type |- Expression expression : **output** Class

error "cannot type " + expression

...

XSEMANTICS axioms

axiom TThis

G |- This _this : **env**(G, 'this', Class)

XSEMANTICS axioms

axiom TThis

G |- This _this : **env**(G, 'this', Class)

must “respect” the judgment

judgments {

type |- Expression expression : **output** Class

error "cannot type " + expression

...

Examples

Rule for subtyping

rule Subclassing

G |- Class left <: Class right

from {

left == right **or**

right.name == "Object" **or**

G |- left.superclass <: right

}

Examples

Grammar for FJ

```
Cast: '(' type=[Class] ')' expression=Expression;
```

Examples

Grammar for FJ

```
Cast: '(' type=[Class] ')' expression=Expression;
```

Rule for cast

```
rule TCast
  G |- Cast cast : cast.type
  from {
    G |- cast.expression : var Class expType

    { G |- cast.type <: expType }
    or
    { G |- expType <: cast.type }
  }
```

Examples

```
rule SubtypeSequence
G |- List<Expression> expressions << List<TypedElement> elems
from {
  expressions.size == elems.size
  var i = 0
  for (exp : expressions) {
    G |- exp : var Class expType
    G |- expType <: elems.get(i++)
  }
}
```

Examples

rule SubtypeSequence

G |- List<Expression> expressions << List<TypedElement> elems

from {

expressions.size == elems.size

var i = 0

for (exp : expressions) {

G |- exp : **var** Class expType

G |- expType <: elems.get(i++)

}

}

Rule for “new”

rule TNew

G |- New newExp : newExp.type

from {

var f = fields(newExp.type)

G |- newExp.args << f

}

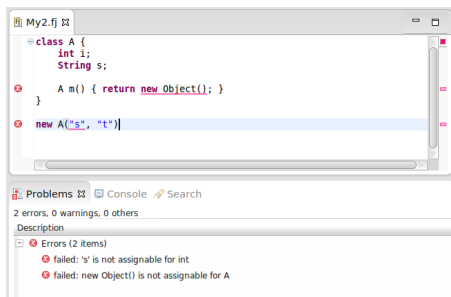
Checkrules

For generating the Validator

```
checkrule CheckMethodBody for
  Method method
from {
  val C = method.getContainerOfType(typeof(Class))
  'this' <- C |- method.body.expression : var Class bodyType
  empty |- bodyType <: method.type
}
```

Errors and Traces

- in case one of the premises fails the whole judgment fails
- the error trace will be used to automatically generate all the error markers
- in general the trace of the rules applied is available to the programmer for testing and debugging



Using traces: Example

Expressing *Subject Reduction*

```

judgments {
  subjred |= Expression e ~> output Expression : output Class <: output Class
}

```

rule SubjRed

```

  G |= Expression e ~> Expression e1 : Class C1 <: Class C

```

from {

```

  G |- e : C

```

```

  G |- e ~> e1

```

```

  G |- e1 : C1

```

```

  G |- C1 <: C

```

```

}

```

Using traces: Example

```
class A {  
    Object m() { return this.n(new B()); }  
    A n(A o) { return new A(); }  
}  
class B extends A {}  
new A().m()
```


Using traces: Example

```

class A {
  Object m() { return this.n(new B()); }
  A n(A o) { return new A(); }
}
class B extends A {}
new A().m()

```

```

SubjRed: [] |= new A().m() ~> new A().n(new B()) : A <: Object
TSelection: [] |- new A().m() : Object
TNew: [] |- new A() : A
  SubtypeSequence: [] |- [] << []
  SubtypeSequence: [] |- [] << []
RSelection: [] |- new A().m() ~> new A().n(new B())
TSelection: [] |- new A().n(new B()) : A
TNew: [] |- new A() : A
  SubtypeSequence: [] |- [] << []
  SubtypeSequence: [] |- [new B()] << [A o]
  TNew: [] |- new B() : B
    SubtypeSequence: [] |- [] << []
  Subclassing: [] |- B <: A
Subclassing: [] |- A <: Object

```

Case study: type inference for λ -calculus

- we also developed a prototype implementation of λ -calculus in XTEXT;
- we used XSEMANTICS to write a type system for inferring types with type variables (generic types);
- we implemented *unification* in order to infer the most general type.

Case study: type inference for λ -calculus

- we also developed a prototype implementation of λ -calculus in XTEXT;
- we used XSEMANTICS to write a type system for inferring types with type variables (generic types);
- we implemented *unification* in order to infer the most general type.

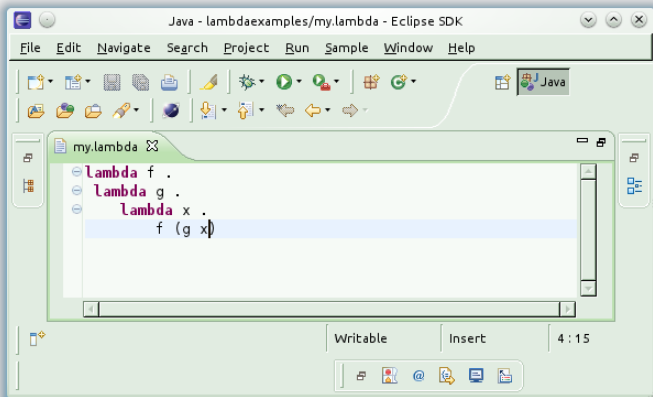
<code>lambda x . x</code>	<code>a -> a</code>
<code>lambda x . 10</code>	<code>a -> int</code>
<code>lambda x . -x</code>	<code>int -> int</code>
<code>(lambda x . lambda y . y) 10</code>	<code>a -> a</code>
<code>lambda x . lambda y . x y</code>	<code>(a -> b) -> a -> b</code>
<code>lambda x . lambda y . y x</code>	<code>a -> (a -> b) -> b</code>
<code>lambda f . (lambda x . (f (f x)))</code>	<code>(a -> a) -> a -> a</code>
<code>lambda f . lambda g . lambda x . (f (g x))</code>	<code>(a -> b) -> (c -> a) -> c -> b</code>

Inferring λ types

we can use the generated type system to write Eclipse editor actions for automatically inserting the inferred types of λ terms:

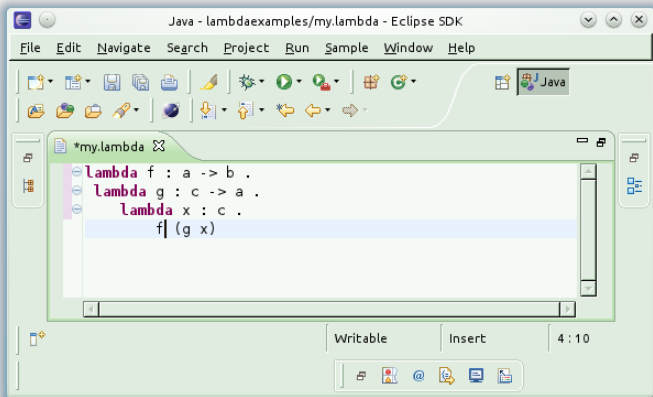
Inferring λ types

we can use the generated type system to write Eclipse editor actions for automatically inserting the inferred types of λ terms:



Inferring λ types

we can use the generated type system to write Eclipse editor pop-up actions for automatically inferring the types of terms:



Fully featured IDE

The screenshot displays the Eclipse IDE interface for a project named "it.xsemantics.example.fj". The main editor shows the source code for a file named "My.fj". The code defines a rule named "SubJed" and includes several annotations and checks.

```

rule SubJed
  G |- cast exp -> Expression expl
  = from {
    val cast = clone(exp)
    if (isValue(cast.expression)) {
      G |- cast.expression <| cast.type
      expl = cast.expression
    } else {
      G |- cast.expression -> var Expression expi
      cast.expression = expi
      expl = cast
    }
  }

rule SubJed
  G |= Expression e -> Expression e1 : Type T1 <: Type T
  = from {
    G |- e : T
    G |- e -> e1
    G |- e1 : T1
    G |- T1 <: T
  }

// checkrules (for the generated)

checkrule CheckMethodBody
  Method method
  = from {
    val typeForThis = fjT
    EcoreUtil2.getCo
  }

'this' <- typeForThis |- method.body.expression <| method.type
  
```

The Package Explorer on the left shows the project structure, including folders for "src" and "src-gen". The Outline view on the right lists the classes and methods in the project, such as "SubJed (subpred)", "CheckMethodBody", "CheckField", "CheckMethodOverride", "CheckMethodHierarchyNotCyclic", and "CheckMain".

The status bar at the bottom indicates the editor is in "Writable" mode, the cursor is in "Insert" mode, and the current line is 358 of 10.

Debugging rules

The screenshot shows the Eclipse IDE in a debug state. The main window title is "Debug - it.xsemantics.example.fj/src/it/xsemantics/example/fj/typing/fj.xsemantics - Eclipse Platform".

Debug Console: Shows the execution flow of the thread [main] (Suspended). The current line of execution is highlighted in green: "fj.xsemantics line: 362". Other lines shown include "NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: not available", "NativeMethodAccessorImpl.invoke(Object, Object[]) line: 57", "DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: 43", and "Method.invoke(Object, Object...) line: 601".

Variables View: Displays the state of variables at the current line. The table below summarizes the visible variables:

Name	Value
Trace	NoneApplicationTrace (id=368)
e	SelectionImpl (id=368)
args	EObjectContainmentEList<E> (id=41)
eContainer	ProgramImpl (id=423)

Source Editor: Shows the rule definition for 'G' in the file 'fj.xsemantics.g'. The rule is:

```

G | = Expression e -> Expression e1 : Type T1 <: Type T
from {
  G |- e : T
  G |- e -> e1
  G |- e1 : T1
  G |- T1 <: T
}
// checkrules (for the generated validator)
  
```

The line "G |- e -> e1" is highlighted in blue, indicating the current execution point.

Outline View: Lists the classes in the project, including TIntConstant (type), TBoolConstant (type), TSelection (type), TCast (type), TExpressionClassType (classtype), GeneralSubtyping (subtype), BasicSubtyping (subtype), and ClassSubtyping (subtype).

Conclusions

<http://xsemantics.sourceforge.net>



Lorenzo Bettini.

A DSL for Writing Type Systems for Xtext Languages.

In *PPPJ*, pages 31–40. ACM, 2011.



Lorenzo Bettini.

Implementing Java-like languages in Xtext with Xsemantics.

In *OOPS (SAC)*. ACM, 2013.

To appear.



Lorenzo Bettini, Dietmar Stoll, Markus Völter, and Serano Colameo.

Approaches and Tools for Implementing Type Systems in Xtext.

In *Software Language Engineering*, LNCS. Springer, 2012.

Conclusions

<http://xsemantics.sourceforge.net>



Lorenzo Bettini.

A DSL for Writing Type Systems for Xtext Languages.

In *PPPJ*, pages 31–40. ACM, 2011.



Lorenzo Bettini.

Implementing Java-like languages in Xtext with Xsemantics.

In *OOPS (SAC)*. ACM, 2013.

To appear.



Lorenzo Bettini, Dietmar Stoll, Markus Völter, and Serano Colameo.

Approaches and Tools for Implementing Type Systems in Xtext.

In *Software Language Engineering*, LNCS. Springer, 2012.

Thanks!