Xtext/TS - a typesystem framwork for Xtext

Version 2.0, May, 2010

Markus Völter
(voelter@acm.org)

Introduction

Starting with version 1.0, it has become absolutely feasible to build complex languages of Xtext. One aspect of a complex language is usually support for expressions. Expressions require recursive grammar definitions, for which assignment actions in Xtext provide reasonable support. However, once you have expressions, it typically also need a type system. While it can be argued that type system checks are nothing more than constraints, building a reasonable type system is a lot of work that can use additional support over plain constraint checks. This paper describes a first cut at a framework for specifying type systems for (expression) languages built with Xtext.

History

1.4	Type Root Classes Improved Error Reporting for Tests
1.5	Documentation improved
2.0	Ported to XText 2.0, added the DSL

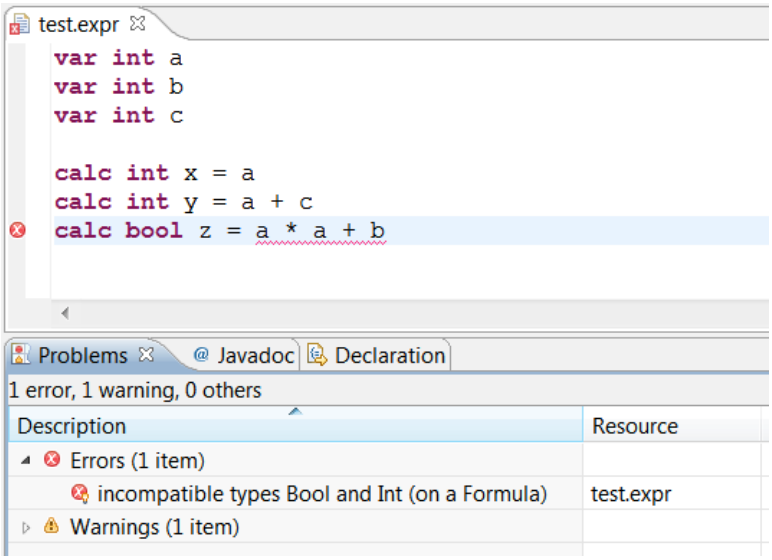
What is a Type System

Here is a definition from Wikipedia:

> *In computer science, a type system may be defined as a tractable syntactic framework for classifying phrases according to the kinds of values they compute. A type system associates types with each computed value. By examining the flow of these values, a type system attempts to prove that no type errors can occur. The type system in question determines what constitutes a type error, but a type system generally seeks to guarantee that operations expecting a certain kind of value are not used with values for which that operation makes no sense.*

Let us show and intuitive example. If in the above program that type of the *z* calc have (last line) would have been *bool*, then something is wrong. You cannot assign an integer value (the result of the calculation on the right side of the equals) to a variable of type *bool*. Note that the program is

structurally and syntactically correct, but the types don't compute. It is the job of a type system to notice this problem and report it to the user, as shown in the following screenshot.



A type system generally consists of the following building blocks:

- **type assignments:** certain language elements, such as the *int* and *bool* keywords had a fixed type.
- **type calculation rules:** for all the language elements, the type can be computed, typically from the types of their constituent elements.
- **typing constraints:** checks, that verify that the types of certain elements confirm to expectations defined by the language designer.

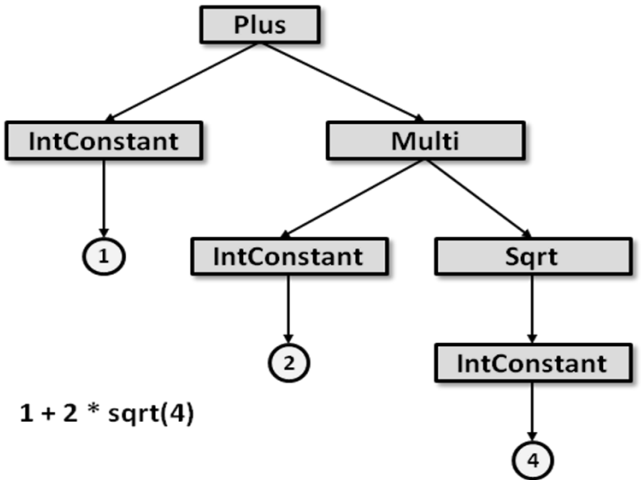
The type system framework explained in this paper allows the efficient implementation of all of these building blocks, and integrates with the Xtext validation framework.

When to use a Typesystem

You could (and should) ask the question when to use a typesystem, and when to simply use "normal" constraints.

First of all, the two aren't mutually exclusive. You can easily mix both. After all, the type system constraints are checked as part of Xtext's normal constraint checks.

Typesystems are especially useful for languages that contain expressions, assignments, function calls and - most importantly - many different runtime variable types, possibly with a type hierarchy. For example, you can use the type system to check that in a (Java-style) local variable declaration, the type of the init expression is compatible with the declared type of the variable. Most obviously, however, type systems are useful if you have the typical expression trees in your languages, as illustrated in the following diagram:



In these expressions, the type of the root (*Plus*) is called by recursively calculating the types of the child expressions. In most expression languages, expressions are trees where each node has zero, one or two children and types are calculated recursively.

The typesystem framework is optimized for these kinds of languages.

Example Grammar

We start out with a simple language for defining variables and expressions for calculating values. For the time being, we support integer and Boolean types. Here is an example program/model.

```
var int a
var int b
var int c

calc int x = a
calc int y = a + c
calc int z = a * a + b
```

The grammar to define this language should look familiar to Xtext users. Note that it is not the goal of this paper is playing how grammars for (recursive) expression languages are defined. Here is the grammar:

```
grammar expr.ExprDemo with org.eclipse.xtext.common.Terminals

generate exprDemo "http://www.ExprDemo.expr"

import "http://www.eclipse.org/emf/2002/Ecore" as ecore


Model:
 elements+=Element*;

Element:
 VarDecl | Formula;


VarDecl returns Symbol:
 {VarDecl} "var" type=Type name=ID ";";

Type:
```

```
 IntType | BoolType | FloatType;

IntType:
  {IntType} "int";

BoolType:
 {BoolType} "bool";

FloatType:
 {FloatType} "float";

Formula:
 "calc" type=Type name=ID "=" expr=Expr ";";

Expr:
 Addition;

Addition returns Expression:
 Multiplication ({Plus.left=current}"+" right=Multiplication)*;

Multiplication returns Expression:
 Atomic ( {Multi.left=current} "*" right=Atomic)*;

Atomic returns Expression:
 {SymbolRef} symbol=[Symbol|QID] |
 {NumberLiteral} value=NUMBER;

terminal NUMBER returns ecore::EBigDecimal:
 ('0'..'9')* ('.' ('0'..'9')+)?;

terminal INT returns ecore::EInt:
 "$$$don't use this anymore$$$";

QID:
 ID ("." ID)*;
```

Note how we use a general concept *Symbol* for stuff that can be referenced; because we want to be able to define other kinds of referencable things later, and because of limitations in Xtext's linking mechanism, we have to do this. For some references we will later need qualified (dotted) names, hence the QID.

Setting Up

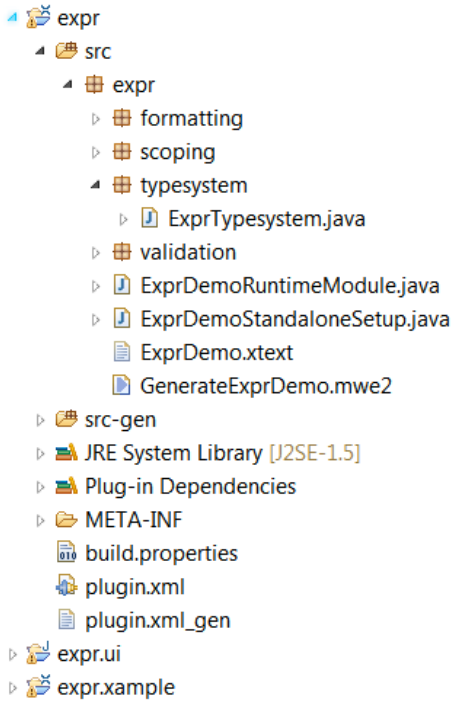
The framework comes with a Java API for describing type system rules, as well as with a textual DSL to provide nicer syntax and static checks for the API. The documentation first explains the use of the API. A later section looks at the DSL. The semantics of the rules and specifications is the same in both cases; I recommend you start reading the API documentation.

Please make sure the *de.itemis.xtext.typesystem* plug-in is available in your Eclipse installation or in the workspace. This plug-in, in turn, has dependencies on Xtext, so at this time it cannot be used without Xtext installed. With this plugin only, you can use the Java API to describe the typing rules. If you want to use the textual DSLs, make sure the *de.itemis.xtext.typesystem.dsl.* plugins are available as well.*

Then make sure that your language project (*expr* in the example) has a dependency on the type system plug-in.

The Typesystem Class

The first programming task is to implement a class that implements the type system itself. In sticking with Xtext project structure, we create a class called *ExprTypesystem*, which we put at the appropriate place into the expr language project.



In theory, this class has to implement the *ITypesystem* interface, but in the vast majority of cases you want to directly inherit from *DefaultTypesystem*. Doing this, will require you to implement its initialize method, which will do further down.

```
public class ExprTypesystem extends DefaultTypesystem {

 @Override
 protected void initialize() {

 }

}
```

Integration with the Validator

As mentioned above, one aspect of the type system is support for type checks, which obviously have to be integrated with the Xtext validation framework. Please insert the following code into your validator:

```
public class ExprDemoJavaValidator
     extends AbstractExprDemoJavaValidator {

 @Inject
```

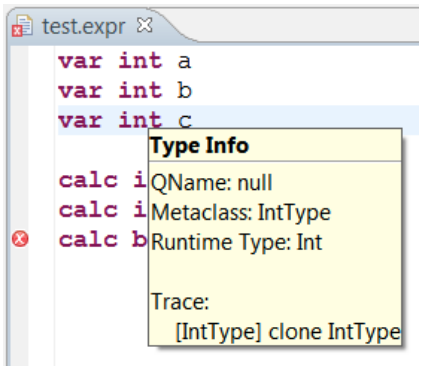
```
 private ITypesystem ts;

 @Check
 public void checkTypesystemRules( EObject x ) {
  ts.checkTypesystemConstraints(x, this);
 }

}
```

As you can see, it calls the type checking method for every object for which the validator is invoked. Notice, how we do is Xtext style Google juice injection to get a hold of the type system. To make this work, we have to implement binder method in the runtime module (for details take a look at the Xtext documentation) that associates the interface with our implementation class:

```
public class ExprDemoRuntimeModule
    extends expr.AbstractExprDemoRuntimeModule {

 public Class<? extends ITypesystem> bindITypestem() {
       return ExprTypesystem.class;
 }


}
```

The Info Popup

It is important to be able to trace the types in a program. Ideally, you want to select any program element press some kind of key combination and get information about the run-time type, as shown in the following screenshot:



The info pop-up shows the qualified name of the element, is metaclass, its type, as well as a trace that shows how the type was calculated. In the example above, it is trivial, since the *int* concept has a type directly associated with it. If more complex type calculation rules are involved, this calculation is shown as a tree structure in the pop-up, allowing you to understand and trace your own typing rules (which can be nontrivial sometimes, because of their recursive nature).

To make this pop up work, you basically have to implement a normal Eclipse text editor pop-up. Either you know how this works, or you copy it from the example code ☺. The important bit is the code that assembles the text content for the pop-up:

```
private String getDescription(final int offset,
    final XtextResource resource) {
 IParseResult parseResult = resource.getParseResult();
 CompositeNode rootNode = parseResult.getRootNode();
 AbstractNode currentNode =
    ParseTreeUtil.getLastCompleteNodeByOffset(rootNode, offset);
 EObject semanticObject = NodeUtil.getNearestSemanticObject(currentNode);
 StringBuffer bf = new StringBuffer();
 bf.append( "QName: "+qfnp.getQualifiedName(semanticObject )+"\n" );
 bf.append( "Metaclass: "+semanticObject.eClass().getName()+"\n" );
 TypeCalculationTrace trace = new TypeCalculationTrace();
 EObject type = ts.typeof(semanticObject, trace);
 if ( type != null ) {
        bf.append("Runtime Type: "+ts.typeString(type));
 } else {
        bf.append("Runtime Type: <no type>");
 }
 bf.append("\n\nTrace: ");
 for (String s: trace.toStringArray()) {
        bf.append("\n  "+s);
 }
 return bf.toString();
}
```

We are now ready to implement our first typing rules.

Basic Typing Rules

Typing rules and type checks all two categories: declarative and procedural. For the time being stick with declarative ones. Currently, the declarative ones are implemented in Java method calls from within initialize method. Subsequent versions of this framework may use an Xtext DSL for that part.

Clones as Types

Let us start by defining that the type of an *int* , as well as of a *bool* is a clone of itself. This is the simplest kind of typing rule. It also showcases, that any EObject can be used as a type. Here all the rules:

```
@Override
protected void initialize() {
        ExprDemoPackage lang = ExprDemoPackage.eINSTANCE;

        try {

                useCloneAsType(lang.getIntType());
                useCloneAsType(lang.getBoolType());
                useCloneAsType(lang.getFloatType());

        } catch (TypesystemConfigurationException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

}
```

The above code basically states that whenever someone or something wants to know the type of the *IntType* (*int*) or *BoolType* (*bool*) concept, clone of the notes themselves is returned.

We can try this out by running the editor and pressing ctrl-shift-I on one of these *int* or *bool* types. The pop-up will show us the types.

Derive Type from Feature

Let us know about the type of the *var* and the *calc*. Their type can be derived from their *type* child, to which we already assigned types in the previous section (sorry for the heavy overloading of the word "type", can't be avoided here!). We add the following two lines to type system specification:

```
useTypeOfFeature(lang.getVarDecl(), lang.getElement_Type());
useTypeOfFeature(lang.getFormula(), lang.getElement_Type());
```

Notice how this is already the first example of a type calculation rule, since we don't prescribe a fixed type, but rather derive the type from feature of the respective element.

We can implement a similar typing rule for variable (i.e. symbol) references. The type of a symbol reference is the type of the variable it references:

```
useTypeOfFeature(lang.getSymbolRef(),
             lang.getSymbolRef_Symbol());
```

Using Fixed Type

For the time being, we can also prescribe the type of the *Plus* and *Multi* expressions. These have to be ints.

```
useFixedType(lang.getPlus(), lang.getIntType() );
useFixedType(lang.getMulti(), lang.getIntType() );
```

The *useFixedType* associates class (second argument) with a language construct. When asked for the type of the concept, the class is instantiated. It is also possible to pass in an object (maybe with certain properties set to certain values) as the second argument, notice the different name of the method:

```
usePrototypeAsType( <concept class>, <an EObject instance> );
```

For the time being, these are all the typing rules we need. Every element of the program should now have a type associated, which can be validated through the pop-up (automated testing is covered later). Let us not cover type checks.

Simple Type Checks

Of course you can use standard Xtext validation to implement type checks. However, for common checks, there are declarative shortcuts.

In our language, we have to make sure that the types of the arguments for *Plus* and *Multi* are *int* and not *bool*. This can be specified as follows:

```
ensureFeatureType(lang.getPlus(),
        lang.getPlus_Left(), lang.getIntType());
ensureFeatureType(lang.getPlus(),
       lang.getPlus_Right(), lang.getIntType());

ensureFeatureType(lang.getMulti(),
       lang.getMulti_Left(), lang.getIntType());
ensureFeatureType(lang.getMulti(),
       lang.getMulti_Right(), lang.getIntType());
```

Notice that you can pass in any number of types this function, to specify alternatives. So if you wanted to allow the right side last to be an int or bool, you can write the following:

```
ensureFeatureType(lang.getPlus(), lang.getPlus_Right(),
       lang.getIntType(), lang.getBoolType() );
```

Alternatively, you can also pass in one or more instances of *CustomTypeChecker*, implementing its *isValid* method in any way you like.

```
public abstract class CustomTypeChecker {

 private String info;

 public CustomTypeChecker( String info ) {
       this.info = info;
 }

 @Override
 public String toString() {
       return info;
 }

 public abstract boolean isValid( ITypesystem ts,
       EObject type, TypeCalculationTrace trace );

}
```

In the final step for the situation, we should make sure that the right side of the equals sign in a *Formula* is compatible with the left side. In simple type systems, "compatible" means "the same". In more complex type systems, various kinds of subtype relationships have to be supported (we will cover this below). Here is the code we need:

```
ensureOrderedCompatibility(lang.getFormula(),
       lang.getElement_Type(), lang.getFormula_Expr());
```

This specifies that for *Formulas*, the type of the *expr* must be "compatible" with the type of the *type* (notice how the *type* property has been pulled up to *Element* because both *VarDecl* and *Formula* have that property). You could also write the following:

```
ensureOrderedCompatibility(lang.getFormula(),
       lang.getFormula_Expr());
```

By passing and only one feature, the type of the element itself is used as the left side of comparison. In our case this would also work, because of type inference rule specified above says:

```
useTypeOfFeature(lang.getFormula(), lang.getElement_Type());
```

Let me add two more details. There are actually two related methods: *ensureOrderedCompatibility* and *ensure**Un**orderedCompatibility*.

When calling

```
ensureOrderedCompatibility(c, f1, f2 );
```

then the constraint requires the types of *f1* and *f2* to be the same, or *f2* to be a subtype of *f1*. In our formula example, if *int* were a subtype of *float* (which makes sense because *ints* can be seen as a special case of *float*) then a program

```
calc float x = <something with int type>
```

would be ok. *ensureOrderedCompatibility* will check for this.

In contrast, when calling

```
ensureUnorderedCompatibility(c, f1, f2 );
```

then the constraint requires the types of *f1* and *f2* to be the same, or *f2* to be a subtype of *f1*, or vice versa. This is useful for our *Plus*, for example, where either *left* or *right* could be *ints* or *floats*, and it would still be valid.

Subtyping

Let us introduce number literals. Here is the necessary change to the grammar:

```
Atomic returns Expression:
 {SymbolRef} var=[Symbol] |
 {NumberLiteral} value=NUMBER;

terminal NUMBER returns ecore::EBigDecimal:
 ('0'..'9')* ('.' ('0'..'9')+)?;

terminal INT returns ecore::EInt:
 "$$$don't use this anymore$$$";
```

Notice that the decimal dot, and the digits behind it, are optional. In other words, if there is a dot, the we have a *float;* if not, it's an *int*. We have to implement this typing rule. Let us first introduce *float* as a type:

```
Type:
 IntType | BoolType | FloatType;

IntType:
  {IntType} "int";

BoolType:
 {BoolType} "bool";

FloatType:
 {FloatType} "float";
```

This typing rule is an example of where a declarative approach does not work, because the type of the element depends on the content (dot or not).

Here is the code; it is implemented in a *type* method in the type system class that is called via Xtext's polymorphic dispatcher:

```
public EObject type( NumberLiteral l,
                          TypeCalculationTrace trace ) {
```

```
        if ( l.getValue().toString().indexOf(".") > 0 ) {
                return Utils.create(lang.getFloatType());
        } else {
                return Utils.create(lang.getIntType());
        }
}
```

You verify that it works by using the pop-up in the editor. Of course we have to do more changes to our type system to make the typing work. Here are the changes.

First, the *left* and *right* properties of the *Plus* and *Multi* should also be able to be *floats*, not just *ints*. We simply pass in the additional type to the *ensureFeatureType* method (currently we cannot do this using subtyping - missing feature, will come!)

```
ensureFeatureType(lang.getPlus(), lang.getPlus_Left(),
        lang.getIntType(), lang.getFloatType());
ensureFeatureType(lang.getPlus(), lang.getPlus_Right(),
        lang.getIntType(), lang.getFloatType());

ensureFeatureType(lang.getMulti(), lang.getMulti_Left(),
        lang.getIntType(), lang.getFloatType());
ensureFeatureType(lang.getMulti(), lang.getMulti_Right(),
        lang.getIntType(), lang.getFloatType());
```

Also, the type of the *Plus* and Multi *itself* is now not simply a fixed type (int), but rather a computation that should calculate the common (i.e. more general) type of the two:

```
computeCommonType(lang.getPlus(),
        lang.getPlus_Left(), lang.getPlus_Right() );
computeCommonType(lang.getMulti(),
        lang.getMulti_Left(), lang.getMulti_Right() );
```

To make this work, we have to declare *int* a subtype of *float* (in a mathematical sense, *float* is more general!)

```
declareSubtype(lang.getIntType(), lang.getFloatType());
```

Then we can write code like this, and we should get the respective error markers.

```
calc int t1 = 2;        // works
calc int t2 = 2.2;      // error: float cannot be assigned to int
calc float t3 = 2.2;    // works
calc float t4 = 2;      // works; int can be assigned to float
calc int t5 = 2 + 2;    // works; type of plus is int
calc int t6 = 2 + 2.3; // error:common type of 2 and 2.3 is float
                        //        which cannot be assigned to int
```

Structured Types

Until now, types had been opaque objects. An *int* is an *int* is an *int*. Let us now consider structured types, where the properties of the type objects are relevant for the type checks.

Type Comparison Features

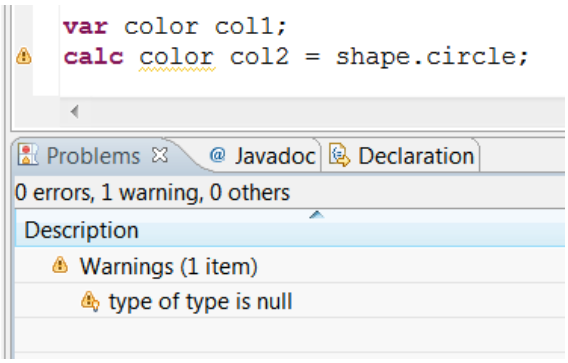
We use *enums* as the example. Here is some example code:

```
enum color {
  red green blue
}

enum shape {
  rect triangle circle
}

var color col1;
calc color col2 = shape.circle;
```

The changes necessary to the grammar are shown in the following code:

```
Element:
 VarDecl | Formula | EnumDecl;

EnumDecl:
 "enum" name=ID "{"
        (literals+=EnumLiteral)*
 "}";

EnumLiteral returns Symbol:
 {EnumLiteral} name=ID;

// …

Type:
 IntType | BoolType | FloatType | EnumType;

EnumType:
   enumRef=[EnumDecl];

// …
```

If we don't make any changes to the type system, we'll get a warning: as the system tries to ensure that the types of the calc are compatible, it notices that the type of the color symbol reference is *null*. This is because *EnumTypes* don't have a type yet.



Let us first define the type of the *EnumDecl*. It should be an *EnumType* whose *enumRef* reference points to the enum it declares. Here is where the "structured" comes in. It is not enough (usually) to say that something is an enum. We have to say *which* enum it is. This is why the *EnumType* has a pointer to the enum it represents. The custom *type* function below builds this structure for the *EnumDecl*.

```
public EObject type( EnumDecl l, TypeCalculationTrace trace ) {
       EnumType t = (EnumType) Utils.create(lang.getEnumType());
       t.setEnumRef(l);
```

```
        trace.add(1, "enum, type is "+typeString(t));
        return t;
 }
```

We also define that the type of an EnumType is the type of the enum it references - i.e. the thing we created in the above method:

```
useTypeOfFeature(lang.getEnumType(), lang.getEnumType_EnumRef());
```

This makes the warning go away, but it makes a new one show up. Now the system complains that the enum literal reference (on the right side of the calc's equals sign) has no type. If we look at the trace we see that the type of the reference is the type of the references object, but the type of that object - the enum literal - is null. Let's fix this; the *useTypeOfAncestor* uses the type of the ancestor of the given type, here: the *EnumDecl*, as the type of the element in question.

```
useTypeOfAncestor(lang.getEnumLiteral(), lang.getEnumDecl());
```

This makes all warnings go away but doesn't solve the obvious issue: we should not be able to assign a shape enum literal to a variable of type color. Although they are both *EnumTypes*, they are *different enums*, which makes them different types. We have to make one last specification:

```
 declareTypeComparisonFeature(lang.getEnumType(),
        lang.getEnumType_EnumRef());
```

This makes sure that when types are compared, and both types are *EnumTypes*, the system then compares the values of the *enumRef* reference. Note that this is really just an equals comparison; no subtyping rules etc. are considered. We'll address this in the next example.

Type Recursion Features

Let us introduce arrays to demonstrate this. Here is the code we want to be able to write:

```
var int i;
var array[int] anIntArray;
var array[float] aFloatArray;

// works: assign two int arrays
calc array[int] anotherOne = anIntArray;

// error: cannot assign float array to int array
calc array[int] anotherOne2 = aFloatArray;

// works: int array is a "subtype" of float array
calc array[float] arr3 = anIntArray;

// works: array access makes it an int
calc int atest = anIntArray[i+1];

// works :-)
calc float atest2 = anIntArray[i+1] + 3.7;
```

Let us first extend the grammar accordingly. Here are the relevant parts:

```
Type:
 PrimitiveType | ArrayType;

PrimitiveType:
```

```
 IntType | BoolType | FloatType | EnumType;

ArrayType:
 {ArrayType} "array" "[" baseType=Type "]";

Multiplication returns Expression:
 PostfixOperators ( {Multi.left=current} "*"
    right=PostfixOperators)*;

PostfixOperators returns Expression:
 Atomic ({ArrayAccess.expr=current} "[" index=Expr "]")?;

Atomic returns Expression:
 {SymbolRef} symbol=[Symbol|QID] |
 {NumberLiteral} value=NUMBER;
```

Let us adress typing now. We first have to assemble the structured type of the *ArrayType* as in

```
var array[int] anIntArray;
```

Here is the code that does it:

```
public EObject type( ArrayType a, TypeCalculationTrace trace ) {
       ArrayType arraytype =
           (ArrayType) Utils.create(lang.getArrayType());
       EObject basetype = typeof( a.getBaseType(), trace );
       arraytype.setBaseType( (Type) basetype );
       trace.add(a, "base type is "+typeString(basetype));
       return arraytype;
}
```

Doing this we will be able to assign any array to each other, because we have not yet declared that the base type of an *ArrayType* should be taken into account when comparing types. However, in contrast to the enum example, we want to be able to assign an array of ints to an array of floats, as in this example:

```
// works: int array is a "subtype" of float array
calc array[float] arr3 = anIntArray;
```

So we have to make sure that the subtype relationship of the base type inside the *ArrayType* is considered. Here is how we do that (note the *Recursion* as opposed to the *Comparison* above!):

```
declareTypeRecursionFeature(lang.getArrayType(),
    lang.getArrayType_BaseType());
```

This allows us to work with array-type variables and make the type system work. However, we still need to address the *ArrayAccess* thing as in

```
// works: array access makes it an int
calc int atest = anIntArray[i+1];
```

First we have to make sure that the expression to which we apply the [] is actually an array:

```
ensureFeatureType(lang.getArrayAccess(),
       lang.getArrayAccess_Expr(), lang.getArrayType());
```

Then we have to make sure that the expression in the square brackets is of type int:

```
ensureFeatureType(lang.getArrayAccess(),
       lang.getArrayAccess_Index(), lang.getIntType());
```

Finally, we have to define the type of the *ArrayAccess;* there we need to extract that base type from the array:

```
 public EObject type( ArrayAccess a, TypeCalculationTrace trace )
{
        ArrayType arrayType =
               (ArrayType) typeof( a.getExpr(), trace );
        trace.add( a, "array type is "+typeString(arrayType));
        Type bt = arrayType.getBaseType();
        trace.add( a, "base type is is "+typeString(bt));
        return bt;
 }
```

That's it.

Type Characteristics

Types can be associated with so-called characteristics. These are a little bit like tags or marker interfaces. The example in this document doesn't lend itself to using characteristics, so here is a general explanation.

You can define a characteristic by instantiating the respective class:

```
TypeCharacteristic iterable = new TypeCharacteristic("iterable");
```

You can now associate any type with such as characteristic, like so:

```
declareCharacteristic( lang.getSomeClass(), iterable );
```

You can now use this as a type for which you can check:

```
ensureFeatureType( lang.getAnotherClass(),
 lang.getAnotherClass_Feature(), iterable );
```

Of course, the idea is to declare the same characteristic for several types, and then just check for this characteristic.

Type Coercion

A type coercion is an implicit type conversion. If, for example, an *int* is expected in a given context (i.e. an expression of type *int* would be valid), but we actually provide a *string*, then, without coercion, this would be a mistake:

```
var int v4 = "Hallo"; // error without coercion
```

Now assume we had the following coercion rule:

- if an *int* is expected,
- and we actually provide a *string*,
- and the element is a string literal,
- and it's value can be transformed to an *int* (i.e. the string contains just numbers)
- then consider it an *int*.

We could then write the following code:

```
var int v4 = "Hallo";  // error: string, but int expected
var int v5 = "100";    // works, because of custom coercion rule!
```

```
calc int cc = 10 + "100";   // works, coercion
```

While this is a somewhat contrived example, I have had to use coercion in many places in real world projects. Notice that a coercion rule is not the same as simply changing the type definition. We don't want number-only-string literals to be treated as *int* everywhere – we just want to consider them *ints* if the program was invalid otherwise.

Coercion rules are implemented as polymorphically dispatched methods in the type system class. Here is the implementation of our coercion rule:

```
public EObject typeCoerce( EObject candidateElement,
          StringType candidate, IntType expected,
          TypeCalculationTrace trace ) {
 if ( candidateElement instanceof StringLiteral ) {
    try {
      Integer.valueOf(
         ((StringLiteral) candidateElement).getValue());
      return create(lang.getIntType());
    } catch ( NumberFormatException fallthrough ) {}
  }
 return null;
}
```

The signature has to be:

- element whose type we want to coerce
- it's current type (polymorphic)
- the type we're trying to coerce to (polymorphic)
- and a TypeCalculationTrace, as usual.

The method has to return the calculated type, or *null* if no coercion is possible.

The typesystem DSL

The typesystem DSL is a facade around the API we have explained above. Specifically, the generator generates Java code that resembles the code we have written manually above.

The abstractions used in the DSL are similar to those in the API, but the syntax is much more concise. In addition, the DSL provides the following advantanges:

- Code Completion into the meta model for which the type system is specified
- Specification of polymorphic typing rules (i.e. specified for a super type, instead of specified for each subtype separately)
- Static error checks, particularly, warning hint at cases where a typing rule accesses the type of an element for which no typing rule has been defined yet (that would lead to a null type error at runtime)

- Navigational support: ctrl-clicking on a feature name does not navigate to the feature definition (in the Ecore file), but instead to the typing rule that calculates the type oft hat feature, if any.

Setting up

In the language project, create a .ts file. For the example, we use *expr.ts*. Add the following contents:

```
typesystem expr.typesys.ExprTypesystem
 ecore file "platform:/resource/expr/src-gen/expr/ExprDemo.ecore"
 language package expr.exprDemo.ExprDemoPackage
```

The first parameter is the name of the to-be-generated typesystem class. The generator appends the suffix *Generated*. This is because the DSL is not a full replacement for the API: those typing rules that cannot be specified declaratively still have to be implemented in Java. The second parameter is the platform URI of the ecore file for which we want to specify the typing rules. The third parameter is the fully qualified name of the *EPackage* class of the target language.

Code generation happens automatically after saving the .ts file. Take a look at the *src-gen* directory of your language, there should be a *expr.typesys.ExprTypesystemGenerated* class. It's structure should look familiar.

Sections and Rules

After the header explained above, .ts files consist of sections that have strings as headings. These are used primarily to structure the outline view and to provide code folding points.

Sections consist of typing rules and subtype specifications; if you have read the API documentation, the syntax should be mostly obvious.

```
// float is a subtype of string
subtype IntType base FloatType
// primitive types use clones of themselves as their type
typeof Type + -> clone
// string literals have fixed string type
typeof StringLiteral -> StringType
```

The plus sign behind the meta class name denotes that this typing rule also applies to all subtypes of that metaclass for which no specific typing rule has been specified.

Rules that are implemented in Java should still be declared in the model:

```
typeof NumberLiteral -> javacode
```

This leads to the generated class becoming abstract, with an abstract type(..) method for each *javacode* typing rule. You have to write a manual subclass and use that one instead of the generated one. Notice that to implement other code parts, the same approach is required, even if the generated class may not be abstract.

Type constraints are specified after the rule in curly braces:

```
typeof Plus -> common left right {
	ensureType left :<=: IntType, FloatType
	ensureType right :<=: IntType, FloatType
	ensureCompatibility left :<=>: right
}
```

In case of compatibility, :<=>: denotes unordered, :<=: denotes ordered compatibility.

For a full explanation of the syntax, take a look at */expr/src/expr/typesystem/expr.ts*. It specifies the complete (declarative part of the) type system for the expr language. It uses comments to explain the details. When working with the DSL, you can also always look at the generated code to see what the DSL code means.

Testing

Testing the language structure is simple: just write down the model in the syntax you expect, and see if it parses. By providing a reasonably large set of example models (coverage), you can make sure all the programs you want to write are possible. Running the parser over all these programs in a batch basically provides the necessary automation.

Constraint checks, and type checks specifically, are not as simple to test, especially because of regressions. Type system rules are non-trivial, and often recursive. Dedicated support is useful; the support should

- be able to load models from within a JUnit test,
- assert that the number of issues is a specific number (to make sure no issues creep in over time)
- and should be able to assert that certain errors are detected (i.e. issues with a specific test are "attached" to certain model elements.

Basics

The typesystem framework comes with a couple of utilities for testing constraints as part of a JUnit 4 test case. The project that contains the tests needs to have a dependency on the *de.itemis.xtext.typesystem.testing* plugin, which contains all the base classes and utilities.

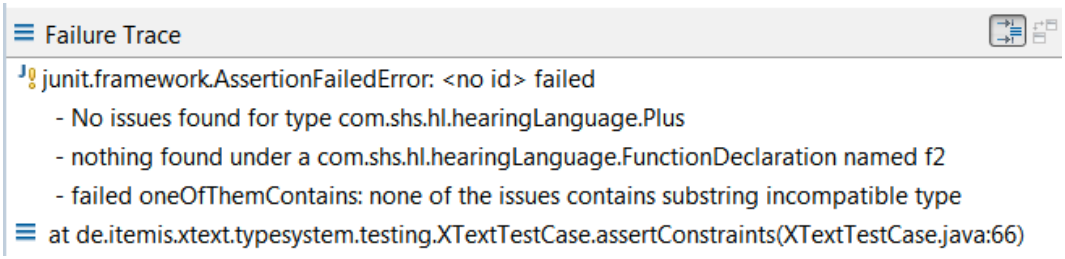
Here is a trivial test:

```
public class Basic extends XTextTestCase {

 @Test
 public void testTypesOfParams() throws Exception {
	EObject root =
		initializeAndGetRoot(new ExprDemoStandaloneSetup(),
			R.modelroot+"/basic.expr");
	assertConstraints( allIssues.errorsOnly().sizeIs(0) );
 }

}
```

The using the *initializeAndGetRoot* method you can read model files. You can pass in any number of files, they are all loaded into the same resource set, so cross-references will work. However, the *allIssues* collection only contains the issues from the first file (we call it the primary file).

Writing the actual tests is based on two main ingredients: the *assertConstraints* method, as well as a fluent-interface-based way of filtering issues. In the above example, you can see that we assert that the issues collection contains zero errors.

If an assertion fails, the exception error message contains a trace of which of the constraints or filters (see below) actually failed.



You can also pass in an additional ID to the *assertConstraints* method as the first argument. If you do so, this ID is output in the error message in the constraints and helps you track down the location of the problem.

Filtering Issues

In the following example (Subtyping.java) we assert that there are incompatible types on two of the calcs:

```
assertConstraints( allIssues.forType(Formula.class).named("t2").
      theOneAndOnlyContains("incompatible") );
assertConstraints( allIssues.forType(Formula.class).named("t6").
      theOneAndOnlyContains("incompatible") );
```

There are a number of methods available on the *IssueCollection* that are useful for filtering. Here is the list:

The method…	… returns a new IssueCollection that
forType(*t*)	contains only those issues that are attached to an instance of *t*
get(*index*)	are at position *index* in the *IssueCollection*
inLine(*line*)	are in line *line* in the model file
withStringFeatureValue(*n*, *v*)	whose feature named *n* has the value (tostring()) *v*
errorsOnly()	contains no warnings
named(*n*)	contains only those issues that are attached to an element with name

	property value *n*
forElement(*t*, *n*)	contains only those issues that are attached to elements of type *t* that have the name *n*
under(*t*)	contains only those issues whose element has an ancestor of type *t*
under(*t*, *n*)	contains only those issues whose element has an ancestor of type *t* named *n*

We also use a couple of assertion methods:

The method…	… asserts that
sizeIs(*s*)	the size of the current collection is *s*
oneOfThemContains(*t*)	the collection has any size, and one of the error messages contains the substring *t*
allOfThemContain(*t*)	the collection has any size, and all the error messages contains the substring *t*
theOneAndOnlyContains(*t*)	the collection is of size 1 and the message of the singe error contains the substring *t*

Finally, you can use *dumpIssues()* on any *IssueCollection* to output the issues to the console.

Miscellaneous

Special Type Comparison Functions

In case the default type comparison facilities including subtypes don't work for you, you can implement your own strategy by implementing a *compareTypes* polymorphic method:

```
protected Boolean compareTypes( EObject type1, EObject type2,
       CheckKind kind, TypeCalculationTrace trace ) {
       if ( kind == CheckKind.same ) {
          …
        } else ….
}
```

The *CheckKind* (same, unordered, ordered) determines which kind of comparison is expected. Don’t forget to put some information into the trace to help users understand how a type was calculated.

Type Strings

Sometimes the default string representation (as created by *typeString(t)*) is not very nice; the polymorphically dispatched *typeToString(t)* method can be used to make strings nicer in error messages, as the following example shows. The methods have to be defined in the typesystem class.

```
public String typeToString( EObject o ) {
	String cn = o.eClass().getName();
	if ( cn.toLowerCase().endsWith("type"))
		return cn.substring(0,cn.length()-4);
	return null;
}

public String typeToString( ArrayType a ) {
	return "array["+typeString(a.getBaseType())+"]";
}
```

Custom Error Messages

All the *ensure...* methods that express type system constraints are overloaded to take an additional string as the first argument. This string serves as a custom error message, as in

```
ensureFeatureType("array index must be Int, idiot :)",
	lang.getArrayAccess(),
	lang.getArrayAccess_Index(),
	lang.getIntType());
```

Type Root Classes

Sometimes the type system rules can be non trivial and it is hard to keep the overview of what's happening. The *TypeCalculationTrace* is one way of keeping track. Another approach is to declare which classes should be used as types. You can declare a set of valid (super) classes for your types like so:

```
declareTypeRootEClasses(EClass1, EClass2, …);
```

After this declaration, whenever you are using a type object that is not an instance of these classes (or subtypes of them), you'll get a *InvalidType* runtime exception.

The ITypesystem API

The *ITypesystem* interface is the primary API to interact with the typesystem from within a validator, if you don't want to use the declarative stuff available in *DefaultTypesystem*, or if you need to know the type of an element for other reasons.

Take a look at the JavaDoc of the class to learn how to use it. Based on the tutorial above it should be obvious what the methods do based on their names anyway.

Using the Typesystem in Scopes

Sometimes it is useful to query the typesystem in a scope provider, to restrict code completion to type-compatible proposals.

In principle, this is straight forward: inject the typesystem into the scope provider and use it to filter the proposals. In practice, it's not so simple.

In case your scope is local (i.e. within the same file), the approach might work. If the targets of your references are in other resources, the problem is that the target objects aren't yet loaded, and you have to make do with the *IEObjectDescriptions*. These, however, have no type, and hence cannot be used for filtering. The trick is to make sure the *IEObjectDescriptions* (as stored in the index) actually do contain some type information.

Pimping the IEObjectDescriptions

We have to provide our own implementation of *DefaultResourceDescription,* as shown in the following code. In essence, we calculate a type and store it, as a string, in a user data field of the *IEObjectDescription*.

```
public class MyResourceDescription
        extends DefaultResourceDescription {

 public static final String KEY_TYPE = "type";
 private IQualifiedNameProvider nameProv;
 private ITypesystem typesystem;

 public MyResourceDescription(Resource resource,
        IQualifiedNameProvider nameProvider, ITypesystem ts) {
   super(resource, nameProvider );
   this.nameProv = nameProvider;
   this.typesystem = ts;
 }

 @Override
 protected IEObjectDescription
             createIEObjectDescription(EObject from) {
   if (nameProv == null) return null;
   String qualifiedName = nameProv.getQualifiedName(from);
   if (qualifiedName != null) {
     if ( from instanceof WhatEverYouWantToIndex ) {
       EObject o = // … the element whose type you want to store
       EObject type = typesystem.typeof(o,
                      new TypeCalculationTrace());
       return createWithUserData(qualifiedName, from,
              KEY_TYPE, type.eClass().getName());
     }
   }
   return super.createIEObjectDescription(from);
 }

 private IEObjectDescription createWithUserData(String qname,
         EObject object, String key, String value) {
   Map<String, String> userData = new HashMap<String, String>();
   userData.put(key, value);
   return EObjectDescription.create(qname, object, userData);
 }
```

```
}
```

To make sure our new *ResourceDescription* is used, we also have to implement our own Manager:

```
public class MyManager
     extends DefaultResourceDescriptionManager {

 @Inject
 private ITypesystem ts;

 @Override
 protected IResourceDescription
       internalGetResourceDescription(Resource resource,
       IQualifiedNameProvider nameProvider) {
   return new MyResourceDescription(resource, nameProvider, ts);
 }

}
```

We also have to register this guy in the runtime module.

```
public Class<? extends IResourceDescription.Manager>
                        bindIResourceDescriptionManager() {
  return MyManager.class;
}
```

Using the information in the Scope Provider

We are now finally ready to enhance the scope provider. It will extract the user data from the *IEObjectDescriptions* and use it for type filtering:

```
public IScope scope_ WhatEverYouWantToIndex(
         final ContextType ctx, EReference ref ) {
  Map<String, IEObjectDescription> map =
              Maps.newLinkedHashMap();
 IScope all = delegateGetScope(ctx, ref);
 for (IEObjectDescription od: all.getContents()) {
    String odtype =
         od.getUserData(MyResourceDescription.KEY_TYPE);
    String contextType = ts.typeof(ctx, new
              TypeCalculationTrace()).eClass().getName();
    if ( odtype.equals(contextType) ) {
       String localName = // … od.getName();
       map.put(localName, new
             AliasedEObjectDescription(localName, od));
    }
  }
  return new MapBasedScope(IScope.NULLSCOPE, map);
}
```