XTypeS User Guide

Lorenzo Bettini

Chapter 1. XTypeS

1.1. What is XTypeS

XTypeS (Xtext TypeSystem) is a DSL for writing the type system of an Xtext based programming language (XTypeS itself is implemented in Xtext). A type system definition in XTypeS is a set of rules which have a conclusion and a set of premises; these rules act on the elements of the language we write the type system for. Then, XTypeS will generate the corresponding Java code that can be used to implement the scoping and the validator for our language.

A type system definition in XTYPES requires the ecore file of the EMF model for the AST produced by XTEXT. The ecore file contains an EMF meta model, i.e., the description of an EMF model. In XTEXT this is used to represent the AST of the program of an XTEXT language. Thus, the first thing to do in a XTYPES program is to refer to the ecore file of the language we want to write the type system for; this is done by specifying the path using a platform resource URI (see EMF documentation), e.g., for FJ (see Section Featherweight Java):

```
grammar "platform:/resource/<path to>/FJ.ecore"
```

Then, we can start writing the rules for the type system of that language; since we are now connected to the ecore, i.e., the definition of the elements of our language, we can write expressions to access the structure of the nodes of the AST using the dotted notation (see the following example). In particular, in the rules we define variables to refer to elements of AST, and we specify the type for those variables using the classes of the EMF model.

For instance, in the following

```
var Class C
var List[Field] F
$C.extends.name = 'Object'
$F := $C.fields
$F[0].type.basic = 'int'
```

we declare a variable for referring to an FJ Class and one to refer to a list (predefined collection in XTYPES) of FJ Fields. Then we can perform some operations on the structure of these elements (we refer to Listing 1 and 2 for the understanding the structure of the AST for FJ). Note that all these expressions in XTYPES are statically typed; it is easy to verify that the assignment to $F is correct (since fields in a Class is a sequence of Field), and that $F[] is correct (since we use the index operator [] on a list).

1.2. What are *types* in XTypeS?

It is crucial to understand what is a “type” in XTypeS. Well, there is no primitive concept of “type” in the XTYPES DSL: it is up to the programmer to have the concept of type in his language, and to give a meaning to the rules in the type system. This means that the grammar of the language for which we want to write the type system in XTypeS must have grammar rules for elements which will be considered as “types”.

1.3. Code generation

Given a type system specification XTypeS will generate some Java classes

1. a Java class for each rule defined in the type system
2. a Java class for the type system
3. a Java class for the validator (in case some ‘OK’ rules are specified, see OK Rules)

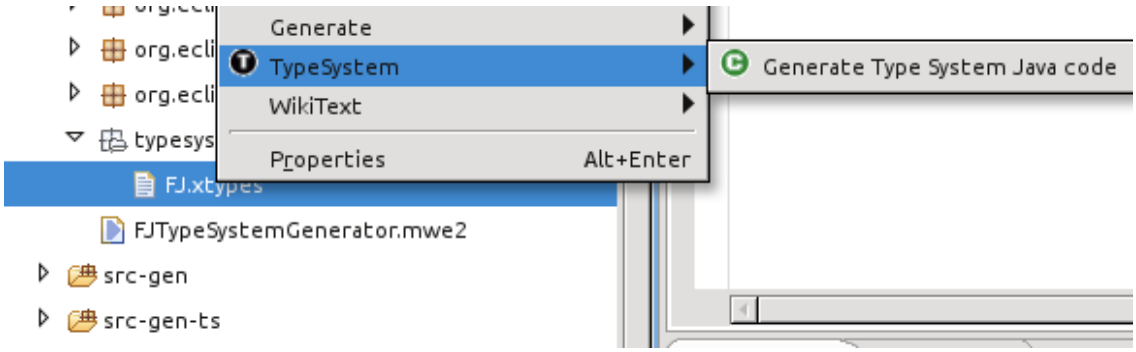
Typically, the user will never use the rule classes directly, but it will use the type system class and the validator class.

IMPORTANT: the generated classes should be “Injected” in your code!

Examples of how to use the generated code can be found in Using the generated code for FJ and in Using the generated code for Lambda.

In order to have the generator to generate meaningful methods to invoke, the several judgment kinds in the type system should be give a description (see Judgment Descriptions).

Code generation is performed by the pop action on files with extension **xtypes** (you can select multiple files also). Code generation using mwe2 workflow is currently under consideration. The Java files will be generated into the directory **src-gen-ts** (in order not to mix them with Xtext *src-gen* directory); thus, you have to make **src-gen-ts** a source folder. Please, also make sure that **src-gen-ts** is in your *build.properties*, if you plan to export the plugin.



1.4. Judgment Descriptions

Judgment descriptions comes before rule definitions in a XTypeS file. They associate some information to the kinds of judgments (identified by a judgment symbol and a typing relation, i.e., strings). Here's an example

```
judgments
    '|-' ':'
        kind='type'
        success='has type'
        nomatch='has not type'
        fail='cannot type'

    '|-' '<:'
        kind='subtype'
        binary
        success='is subtype of'
        nomatch='is not subtype of'
end
```

The specified strings will be used in code generation.

- **kind** will be used for generating method names in the generated Java type system;
- **binary** option specifies that both parameters of a rule belonging to that kind of judgment are to be considered input parameters;

In case you have a rule of the shape

```
rule Foo
derives
    G |- var MyGrammarElement e1 : var MyOtherGrammarElement e2
from...
```

since this rule belongs to judgments of kind `"|-", ":"`, in the generated Java type system we will have the methods

```
TypeSystemResult<MyOtherGrammarElement>
    typeAsMyOtherGrammarElement(MyGrammarElement left);
TypeSystemResult<MyGrammarElement,MyOtherGrammarElement>
    type(MyGrammarElement left, MyOtherGrammarElement right);
TypeSystemResult<Boolean>
    checkType(MyGrammarElement left, MyOtherGrammarElement right);
```

By convention, the right side of a rule is typically the result (output parameter), thus the first method can be used to get the right output argument of a rule given the left input argument. The second method can be used to pass both arguments as input parameters and after a succesful rule application get both of them as the output (remember that a rule can change both arguments during its application). The last one can be used simply to check that given the two input arguments the rule successfully applies.

```
rule Foo
derives
    G |- var MyGrammarElement e1 <: var MyOtherGrammarElement e2
from...
```

since this rule belongs to judgments of kind `"|-", "<:"`, where we specified the **binary** option, in the generated Java type system we will have only this method

```
TypeSystemResult<Boolean>
    checkSubtype(MyGrammarElement left, MyOtherGrammarElement right);
```

Since, for **binary** judgments we request that both arguments to the rules are input parameters.

Actually in any case we will have an overloaded version of each generated method taking also a *TypingJudgmentEnvironment* as parameter (in the generated methods without this parameter, the typing judgment is implicitly considered empty).

1.5. *OK* Rules

When rules (or axioms) have the string ‘OK’ as the right element, they are treated differently during code generation: XTYPES will generate a derived class from *AbstractDeclarativeValidator* with a *@Check* method for the left element’s type. Remember that XTEXT calls automatically such *@Check* methods according to the runtime type of the model element to be validated. Thus, XTYPES will generate a validator ready to be used.

1.6. Typing Judgment Environment

Each rule receives a typing judgment environment where some mappings can be stored (see also environment operators). A mapping has the shape

```
<key> -> <value>
```

where both `key` and `value` can be any object of your language (including strings).

1.7. StringProvider

The run-time system of XTypeS relies on the class *StringProvider* to get a string representation of the elements of the language, e.g., to print errors due to failed rules of the type system. You can think of *StringProvider* as the non-UI part of Xtext *LabelProvider*. The default implementation tries to get a usable string representation of the EObjects of the EMF model (AST), e.g., by using the feature *name* if available, and by also showing the EClass of the elements.

However, it is likely that the default string representation may not be what you like; thus you can derive from this class and provide custom representations for specific elements of your language.

This class relies on the *PolymorphicDispatcher* of Xtext, thus, you only need to define a method *stringRep* (returning a *String*) for a specific class of your DSL.

You then need to use Google Guice to inject such string provider. An example can be found in Using the generated code for Lambda.

Chapter 2. Examples

In this chapter we show two examples of use of XTypeS for writing the type system of two (toy) programming languages; we will write the type system for Featherweight Java (a light version of Java) and we will write the type system for inferring generic types (with unification) of a simple lambda calculus (i.e., a functional language).

2.1. Featherweight Java (FJ)

Featherweight Java is a lightweight functional version of Java, which focuses on a few basic features. It is not intended to be used as a programming language, but as a formal framework for studying properties of Java (A. Igarashi, B. Pierce, and P.Wadler. Featherweight Java: a minimal core calculus for Java and GJ. ACM TOPLAS, 23(3):396–450, 2001.). In this section we will see how to write the type system of FJ using XTypeS.

2.1.1. FJ in a nutshell

FJ focuses on the following features: mutually recursive class definitions, inheritance, object creation, method invocation, method recursion through this, subtyping and field access. In particular, a FJ program is a list of class definitions and a single main expression.

Here's an example of an FJ program:

```
class A extends Object { }
class B extends Object { }

class Pair extends Object {
    Object fst;
    Object snd;

    Pair setfst(Object newfst) {
        return new Pair(newfst, this.snd);
    }

    Pair setsnd(Object newscd) {
        return new Pair(this.fst, newscd);
    }
}

new Pair(new A(), new B()).setfst(new A()).fst
```

Since in FJ the class constructor has a fixed shape, we consider a simplified version of the language by assuming constructors as implicit; in particular when invoking `new` we should pass an argument for each field in the class, incuding inherited fields, in the same order of the hierarchy. Thus, if we have the following classes

```
class A { int i; boolean b; }
class B extends A { String s; }
```

we must create an instance of `B2` as follows: `new B(10, true, "foo")`.

2.1.2. FJ implemented in Xtext

We have already implemented FJ in Xtext, and its implementation can be found at http://fj-eclipse.sourceforge.net. In that implementation, the type system was implemented directly in Java. In XTypeS we ship another implementation of FJ where the type system is written using XTypeS instead of manually written Java code. We will use also a different namespace URI (namely, `http://www.xtypes.it/example/fj/FJ` instead of `http://example.xtext.org/FJ`).

The Xtext grammar of FJ is as follows (Since in FJ the class constructor has a fixed shape, we simplified the language by assuming constructors as implicit):

```
grammar it.xtypes.example.fj.FJ with org.eclipse.xtext.common.Terminals

generate fj "http://www.xtypes.it/example/fj/FJ"

Program :   (classes += Class)* (main = Expression)? ;

Type: BasicType | ClassType;
BasicType : basic=('int' | 'boolean' | 'String');
ClassType : classref=[Class];

TypedElement: Field | Parameter;

Class:
    'class' name=ID ('extends' extends=[Class])? '{'
        (fields += Field)*
        (methods += Method)*
    '}';

Field: type=Type name=ID ';' ;
Parameter: type=Type name=ID ;
Method:
    returntype=Type name=ID '(' (params+=Parameter
            (',' params+=Parameter)*)? ')' '{'
        body=MethodBody
    '}';
MethodBody: 'return' expression=Expression ';' ;

Expression: TerminalExpression
    ({Selection.receiver=current} '.' message=Message )*;
Message: MethodCall | FieldSelection;
MethodCall: name=[Method] '(' (args+=Argument (',' args+=Argument)*)? ')';
FieldSelection: name=[Field];

TerminalExpression returns Expression:
   This | Variable | New | Cast | Constant | Paren ;

This: variable='this';
Variable: paramref=[Parameter];
New: 'new' type=ClassType '(' (args+=Argument (',' args+=Argument)*)? ')';
Cast: '(' type=ClassType ')' object=TerminalExpression;
Paren returns Expression: '(' Expression ')';

Constant: IntConstant | BoolConstant | StringConstant;
StringConstant: constant=STRING;
IntConstant: constant=INT;
BoolConstant: constant = ('true' | 'false');
Argument:   Expression;
```

Given the grammar, and thus the generated ecore, we can write the type system in XTypes: first of all, we give our type system a name (though it is not strictly required, it will help code generation with better names), and, most importantly, we specify the platform URI to the ecore file; thus, assuming we have the sources of the FJ plugin in our workspace we write (we split here the platform URI only for formatting convenience):

```
typesystem FJ

grammar "platform:/resource/it.xtypes.example.fj/ \
    src-gen/it/xtypes/example/fj/FJ.ecore"
```

we give some judgment descriptions which help the generator generating meaningful names in the Java code

```
judgments
    '|-' ':'
        kind='type'
        success='has type'
        nomatch='has not type'
        fail='cannot type'

    '|-' '<:'
        kind='subtype'
        binary
        success='is subtype of'
        nomatch='is not subtype of'

    '|-' '~'
        kind='override'
        binary
        success='overrides'
        nomatch='does not override'
end
```

Now we can start writing rules.

2.1.3. Getting types of FJ expressions

Let's start with subclass rule: we define a rule, named *SubClass*, which takes two arguments both of type *Class* (defined after the *derives* keyword) and states that the relation `<:` holds if one of the following judgments (or premises) holds: C2 is the class Object (each class is a subclass of *Object*), the two classes are the same (subclassing is reflexive), recursively the superclass of C1 is subclass of C2 (subclassing is transitive). This rule also uses the functionality of specifying a custom error for judgments when they fail (see error specification).

```
rule SubClass
derives
    G |- var Class C1 <: var Class C2
    error=$C1 + ' is not a subclass of ' + $C2
from
    (
        $C2.name = 'Object'
        or
        $C1 = $C2
        or
        G |- $C1.extends <: $C2
    )
```

Each rule defines a *typing judgment* which consists of: a typing judgment environment (which we describe later and corresponds to the environments of the type theory, usually denoted by the greek letter Gamma), a judgment symbol (like |- in the example) and a typing statement, which in turns consists of the left and right elements and a relation (<: in the example). Note that these rules are written in the other direction of standard natural deduction rules, i.e., we first write the conclusion and then the judgments that must hold. This is to make XTYPES programs more similar to standard programs: a rule definition can be seen as a function, with parameters (the derives part) and the body (the from part); lastly, this order also makes the writing of rules easier and the programming IDE can detect code completion proposals better. Rules also have a typing judgment environment as parameter; an interesting use of this environment to pass information among rules is shown later in Section Lambda.

The *from* part is basically a list of judgments, the premises, that must hold in order to make the rule succeed; these judgments can be standard expression statements like checking for equality (with =) or variable assignment (with :=) or "invocation" of other rules (like the last one in the above rule); the latter can be told from the former by the fact that they have the shape of a typing judgment as defined above.

All the judgments in the rule body are to be considered in *logical and relation*; thus they all must hold, and they are checked in the same order they are defined. If a judgment fails, the whole group of judgments in and relation fails. If we need to evaluate judgments in *logical or relation* (as in the above rule), we need to use the or keyword. In this case the generated code will try to evaluate the judgments in the first branch of the or, and pass to the other branches only if the previous or branches fail; thus, also these judgments in or relation are evaluated according to the order they are defined.

As we said in Section What are types, there is no primitive concept of "type" in the XTYPES DSL: it is up to the programmer to have the concept of type in his language, and to give a meaning to the rules in the type system.

This means that the grammar of the language for which we want to write the type system in XTypeS must have grammar rules for elements which will be considered as “types”. Remember that the rules can use any element of the ecore of the language. For instance, in FJ the types are not classes (see how fields and parameters are defined in the grammar), but generic *Type* elements; then we have specializations for types: *BasicType* with an attribute *basic* to represent basic types, and *ClassType* with an attribute *classref* (i.e., a reference to a *Class*) to represent class types.

Thus, we need to define a subtyping rule in general for types, for basic types and for class types. We can use the polymorphic mechanism for rule selection: we write a rule for each subclass of *Type*, i.e., *ClassType* and *BasicType*. The rule to be applied is selected by the runtime system of XTYPES not only according to the judgment symbol and typing relation, but also according to the actual type of the elements; thus it uses *polymorphic dispatch*.

```
rule SubType
derives
    G |- var Type t1 <: var Type t2
from
    fail
    error=$t1 + ' is not a subtype of ' + $t2

rule SubTypeClass
derives
    G |- var ClassType c1 <: var ClassType c2
from
    G |- $c1.classref <: $c2.classref

rule SubTypeBasic
derives
    G |- var BasicType b1 <: var BasicType b2
    error=$b1 + ' is not a subtype of ' + $b2
from
    $b1.basic = $b2.basic
```

Furthermore, statically, XTYPES checks that there is a rule for any judgment used in the rules. When we write the rules and we refer to objects or one of their attributes, XTYPES uses the static type of that attribute to search for rules; for instance, in the case of FJ types, this means that when we use a judgment in a premise of a rule that uses the subtyping judgment (as shown later), the objects involved, statically, will be *Type* instances (though at runtime they will be either *BasicType* or *ClassType*). Thus, we need to write a rule for the subtyping judgment also for *Type* (the first rule *SubType* above), otherwise XTYPES will reject the judgment involving subtyping. This “base” case for the subtyping judgment is not only useful for keeping XTYPES from complaining. It will also be used actually when we try to check subtyping between a *ClassType* and a BasicType. In this case the subtyping must not hold, and for this reason we implemented this case using the special judgment **fail**, which always fails (see fail).

Then we have one for checking subtyping on list of arguments against list of parameters (as you might guess, we will use it later to check that we pass the right arguments to a method):

```
rule SubTypeSequences
derives
    G |- var List[Argument] args <: var List[Parameter] params
from
    len($args) = len($params)
    forall i in len($args) {
        var Type argType
        G |- $args[$i] : $argType
        G |- $argType <: $params[$i].type
    }
```

Thus, we check that the number of passed arguments is the same as the number of parameters (using the function **len**, see len), and then we check that each argument is a subtype of the corresponding parameter (note that we use the rules for subtyping defined earlier).

Let’s see now how we can define equality among types. Note that using the standard operator = would not help since it has the same semantics of Java `equals` (see equal), and the *Type* instances are not the same at runtime. Thus we can define specific judgments to check equality:

```
rule TTypeEquals
derives
    G |- var Type t1 == var Type t2
from
    G |- $t1 <: $t2
    G |- $t2 <: $t1

rule TParamsEquals
derives
    G |- var List[Parameter] params1 == var List[Parameter] params2
from
    len($params1) = len($params2)
    forall i in len($params1) {
        G |- $params1[$i].type == $params2[$i].type
    }
```

Note that we also define the equality judgment for list of parameters; we will need this later.

Up to now, for writing subtyping and equality rules, we used both parameters of rules as input parameters; The parameters of rules, however, are not necessarily always input parameters: one of them can be considered as an output parameter (in principle they can be both input and output parameters); for instance, if we use the rules to infer the type of expressions we can consider the left parameter as the input parameter and the right one as the output parameter. Also the typing judgment environment can be used both for input (as in FJ) and for output (as we will see in Lambda).

In our case we adopt this convention for judgments of the shape `|-` and `:` that we use to infer the type, i.e., the FJ *Type*, of our expressions. Thus, we have a rule for each FJ expression, where the right element is the output, i.e., the *Type* we compute for the expression (the left element); let's start with the simple ones: the types of constants:

```
rule TIntConstant
derives
    G |- var IntConstant i : var Type t
from
    var BasicType bt
    $bt.basic := 'int'
    $t := $bt

rule StringConstant
derives
    G |- var StringConstant s : var Type t
from
    var BasicType bt
    $bt.basic := 'String'
    $t := $bt

rule BoolConstant
derives
    G |- var BoolConstant b : var Type t
from
    var BasicType bt
    $bt.basic := 'boolean'
    $t := $bt
```

These rules basically create a *BasicType* and set the *basic* attribute with a string representing the basic type.

Now let's see the rules for inferring the *Type* of more interesting FJ expressions:

```
axiom TParam
    G |- var Parameter p : $p.type

axiom TVariable
    G |- var Variable v : $v.paramref.type

axiom TField
    G |- var Field f : $f.type
```

these rules introduce another kind of rules: *Axioms* i.e., rules without premises. Note also that the parameters of rules (and axioms) need not be a variable declaration (though at least one of the two must be): they can also be dotted expressions. In the case of a parameter we say that the type of a *Parameter* is the *Type* contained in the field *type* since parameters are explicitly typed. The rules (axioms) for *Field* and *Variable* are similar to the one for *Parameter*; actually, since the two grammar elements are both a *TypedElement* we might have written only one rule for *TypedElement*.

```
axiom TThis
    G |- var This t : (Type) env(G, 'this')
    error="'this' can only be used in method bodies"
```

For the type of `this` we use the predefined function **env** (see env) which read the value contained in the environment (first parameter of env) which corresponds to the passed key (the second parameter of env). The function **env** returns a generic Java `Object` so you need to cast it. The function **env** fails (and thus makes the rule fail) if no mapping for the specified key is found in the environment. Who puts the mapping for `this` in the environment? We'll see that later. However, we assume that if this rule fails it is because we are trying to type `this` from outside a method body (e.g., in the program's main expression). Note also that this rule might also fail in case the cast fails, though this will never happen due to the way we write the rules in the FJ type system.

```
rule TMethodBody
derives
    G |- var MethodBody m : var Type t
from
    G |- $m.expression : $t
```

Now, the rule for *MethodBody* should be clear.

The rule for *New* says that the type of a *New* expression is the type of the attribute *type* (see the FJ grammar); however, this rule also checks that the *New* expression is correct:

```
rule TNew
derives
    G |- var New e : var Type t
from
    // we must retrieve all the fields, also the inherited ones
    var List[Field] fields := getall($e.type.classref, fields, extends)
    len($e.args) = len($fields)
        error='argument number ' +
            len($e.args) + ' is not equal to field number ' +
            len($fields)
    forall i in len($e.args) {
        var Type argType
        var Type fieldType
        G |- $e.args[$i] : $argType
        G |- $fields[$i] : $fieldType
        G |- $argType <: $fieldType
        error='argument type (' + $argType +
            ') is not a subtype of field type (' +
            $fieldType + ')'
    }
    error='failure in checking arguments ' + $e.args
    $t := $e.type
```

A *New* expression in FJ is correct if we pass all the arguments for all the fields (included the inherited ones). To get all the fields we use the predefined function **getall** (see getall). Thus, we check that the number of passed arguments is the same as the number of fields (using the function **len**, see len), and then we check that each argument is a subtype of the corresponding field (note that we use the rules for subtyping defined earlier). Note the functionality of specifying a custom error for judgments when they fail (see error specification). We also use **forall** (see forall). Basically `forall i in n` corresponds to a loop where `i` goes from `0` to `n-1`.

Now let's pass to the rule for typing a *Selection* on an expression. As you can see from the grammar the *Message* on a *Selection* can be either a method invocation (*MethodCall*) or a field selection (*FieldSelection*). We can use the polymorphic mechanism for rule selection: the *Type* of a *Selection* is the *Type* of the *Message*; then we write a rule for each subclass of *Message*. Remember that the rule to be applied is selected by the runtime system of XTYPES not only according to the judgment symbol and typing relation, but also according to the actual type of the elements.

```
// never used
axiom TMessage
    G |- var Message m : var Type T

rule TMethodCall
derives
    G |- var MethodCall m : $m.name.returntype
from
    G |- $m.args  <: $m.name.params

axiom TFieldSelection
    G |- var FieldSelection f : $f.name.type

rule TSelection
derives
    G |- var Selection e : var Type t
from
    var Type receiverType
    G |- $e.receiver : $receiverType
    G |- $e.message : $t
```

Checking a *Selection* means checking that the receiver is well-typed, i.e., it has a type, and, in case of a method invocation, that we pass all the arguments and that the arguments are subtypes of the method's parameters (remember the rule `SubTypeSequences`).

Remember that XTYPES itself statically checks that the rules in the type system are correct, in particular, if we write a judgment in a premise of a rule, there must be a corresponding rule for the types of the involved objects in the type system. When we write the rules we refer to attributes of an object, and XTYPES uses the static type of that attribute to search for rules; for instance, `$e.message` has static type *Message*. Thus, in the type system there must be a rule for a judgment of that kind also for *Message*. You may want to implement these base cases either as axioms which always succeed, or as rules which always fail (using the special judgment **fail**, see fail), depending on your needs. The same holds for the base class *Argument* (which is requested by the rule `TNew` above) and so we define this base case:

```
// never used
axiom TArgument
    G |- var Argument a : var Type t
```

We could have written the rule for *Selection* in an alternative way, using casts and or branches; this way we could have written only one rule (remember that if a judgment in an or branch fails, we directly go to the other branch; also take a look at the FJ grammar, to know what that `name` is in both cases):

```
rule TSelection // alternative way
derives G |- var Selection e : var Type t
from
    var Type receiverType
    G |- $e.receiver : receiverType
    (
        var MethodCall m := (MethodCall) $e.message
        G |- $m.args  <: $m.name.params
        $t := $m.name.returntype
    or
        var FieldSelection f := (FieldSelection) $e.message
        $t := $f.name.type
    )
```

As for the type of FJ cast, statically, we know that a cast expression has the type of the type we cast to, thus we write:

```
rule TCast
derives
    G |- var Cast cast : var Type t
    error='invalid cast'
from
    var Type objectType
    G |- $cast.object : $objectType
    (
        G |- $cast.type <: $objectType
        or
        G |- $objectType <: $cast.type
    )
    error=
        $objectType + ' and ' +
        $cast.type + ' are incompatible types'
    $t := $cast.type
```

A cast expression, statically, is well-typed if the type of the object and the type we cast to are related.

2.1.4. Type checking FJ expressions

We then have a second set of rules, a set of 'OK' rules (see OK Rules). Remember that when rules (or axioms) have the string 'OK' as the right element, they are treated differently during code generation: XTYPES will generate a derived class from *AbstractDeclarativeValidator* with a *@Check* method for the left element's type. Remember that XTEXT calls automatically such *@Check* methods according to the runtime type of the model element to be validated. Thus, it might make little sense to write an 'OK' rule for each kind of FJ expression: most expressions will contain subexpressions and the generated validator would end up calling a check method for each subexpression, besides the one for the containing expression. For instance, if you have `new A().f.m(5)`, which is a *Selection* expression and you have an 'OK' rule for *New*, *FieldSelection* and *MethodInvocation* the validator would call a method for each of them; however, the 'OK' rule for *Selection* would itself check subexpressions, so there would be too much overhead. But most of all, there are expressions, like `this`, which could not be type-checked, not even given a type, without something in the typing environment (see the rule `TThis`).

For this reason, we have 'OK' rules only for FJ elements which define "something", i.e., classes, field declarations and method definitions. Checking whether classes and field declarations are correct basically boils down to checking whether there are no duplicates (and some other things as we will see in the following). As for methods, as we will see, we need to check something more.

A field definition is correct if there are no duplicate fields (neither in the class nor in the hierarchy):

```
rule TFieldOk
derives
    G |- var Field f : 'OK'
from
    // checks that there are no duplicate field in the hierarchy
    var Class C := (Class) container($f)
    !exists inheritedField in getall($C.extends, fields, extends) {
        $inheritedField.name = $f.name
    } error='duplicate field in base class'
    !exists otherField in $C.fields {
        $otherField.name = $f.name
        $otherField != $f
    } error='duplicate field in the same class'
```

Here we use some predefined functions: **container** (see container) which gets the container of a given object (here we know that fields are declared only in classes); then we use **exists**, in particular **!exists**, i.e., "not exists" (see exists and not exists) which checks that there is no element in a collection which satisfies the given judgments. Thus we check that there is no field with the same name in the inherited fields, and that there is no field with the same name in the fields of the current class. Note that in the latter case, we check that the field with the same name is NOT the field we are currently checking. This time using equality makes sense since we're checking that the field with the same name is not the same instance of the one we're checking (see equal operator).

The rule for checking that a class definition is well-typed is as follows:

```
rule TClassOk
derives
    G |- var Class C : 'OK'
    error='class hierarchy is not acyclic for ' + $C
from
    !exists programClass in getall($C, extends, extends) {
        $programClass.name = $C.name
    }
```

That is we check that the class hierarchy of the given class is acyclic. Remember that using **getall** is safe, since it computes the closure of classes in the *extends* relation, without getting into infinite loops.

Shouldn't we also check that there are no duplicate classes in the program? Yes we should, but for this example, we decided not to do that in the type system definition: we will implement this check manually in Java, in order to show how the generated validator can be mixed with a manually implemented validator (see later, using the generated code for FJ).

The rule for checking method definition is as follows:

```
rule TMethodOverride
derives
    G |- var Method m1 ~ var Method m2
from
    G |- $m1.params == $m2.params
    G |- $m1.returntype == $m2.returntype

rule TMethodOk
derives
    G |- var Method m : 'OK'
from
    var ClassType C
    var Type bodyType
    $C.classref := (Class) container($m)

    // check the 'override' predicate
    foreach getall($C.classref.extends, methods, extends) as inhMethod {
        (
            // either the method is different
            $inhMethod.name != $m.name
        or
            // otherwise they must have the same signature
            G |- $m ~ $inhMethod
        )
    }

    // check no duplicate method names in the same class
    !exists otherMethod in $C.classref.methods {
        $otherMethod.name = $m.name
        $otherMethod != $m
    } error='duplicate method in the same class'

    G, 'this' -> $C |- $m.body.expression : $bodyType
    G |- $bodyType <: $m.returntype
        error="body type '" + $bodyType +
            "' is not a subtype of "
            + "return type '" + $m.returntype + "'"
```

Again we use **container** knowing that a *Method* can be defined only in a *Class*. This time a method with the same name can be defined in the hierarchy (but not in the same class): in fact this would be *method overriding*. However, we must check whether this overriding is correct, i.e., the method signature must be the same if there's a method with the same name in the class hierarchy (FJ does not consider method overloading, neither covariant return types). This is checked by the rule `TMethodOverride` which defines a judgment with relation `~`. Finally we check that the body of the method is well-typed and that the type of the body is a subtype of the return type of the method. This time we insert in the typing environment used for checking well-typedness of the body and for getting the type of the body a mapping for `this`. Here we use also the loop functionality **foreach** (see foreach).

Finally, we check that the *main* expression of an FJ program (if one is defined) is well-typed (and can be given a type).

```
rule TProgram
derives
    G |- var Program p : 'OK'
from
    (
     $p.main = null
     or
     var Type mainType
     G |- $p.main : $mainType
    )
    error='main expression ' + $p.main + ' is not welltyped'
```

2.1.5. Using the generated code for FJ

Now let's see how we can use the generated code, after we generated Java code (see Code generation).

As for the scoping in the *FjScopeProvider* we write a utility method which, using the generated *FJTypeSystemDefinition*, by injection, gets the *Type* of an expression and, if the typing succeeds, takes the corresponding *Class*; remember that in our FJ implementation a *Type* either contains the string of a basic type (in the attribute *basic*) or the reference to a class (in the attribute *classref*). Since in the scoping we are interested in provide the possible methods to be selected on a receiver expression (or the fields), we are interested only in the case where the receiver of a *Selection* is of a *Class* type.

```
public class FJScopeProvider extends AbstractDeclarativeScopeProvider {

@Inject
protected FJTypeSystemDefinition fjTypeSystem;

protected TypingJudgmentEnvironment environmentForThis(EObject object) {
    return containingClassFinder.environmentForThis(object);
}

protected Class getExpressionClass(Expression expression) {
    TypeSystemResult<Type> result = fjTypeSystem.typeAsType(
            environmentForThis(expression), expression);
    Type type = result.getValue();
    if (type instanceof ClassType) {
        return ((ClassType) type).getClassref();
    }
    return null;
}
```

Remember that for typing `this`, a mapping for the string `'this'` must be present in the typing judgment environment, and so we must provide it; this is taken care of `containingClassFinder`, a utility class which simply gets the containing class of an *Expression* (which might also be null for the *Main* expression). Note that we use the generated method `typeAsType` (see Code generation and Judgment Descriptions, for understaning the names of the generated methods) passing the `expression` as the left argument of the typing judgment and get the right argument (a *Type* object) as the result (for the scoping we ignore possible failures in typing, since in that case we simply fail by returning an empty scope).

Once we have the *Class* of the receiver, we can computer the scoping for *MethodInvocation* and *FieldSelection* by simply getting all the methods (fields, respectively) of a *Class*:

```
public IScope scope_FieldSelection_name(Selection sel, EReference ref) {
    return Scopes.scopeFor(auxiliaryFunctions
            .getFields(getExpressionClass(sel.getReceiver())));
}

public IScope scope_MethodCall_name(Selection sel, EReference ref) {
    return Scopes.scopeFor(auxiliaryFunctions
            .getMethods(getExpressionClass(sel.getReceiver())));
}
```

We're not showing here the `auxiliaryfunctions` class (you may take a look at FJ code), but it does what we expect (and also handle the case where the *Class* object is null, by returning an empty scope).

Now let's use the generated validator; Xtext has already a mechanism for composing validators, so we take our previous *FJJavaValidator* and we use the annotation *@ComposedChecks* specifying the generated validator, *FJTypeSystemValidator*:

```
@ComposedChecks(validators = { FJTypeSystemValidator.class })
public class FJJavaValidator extends AbstractFJJavaValidator {

    @Check
    public void checkNoDuplicateClasses
            (org.eclipse.xtext.example.fj.Class c) {
        ...
    }
}
```

Remember that, when we show the rule `TClassOk` we intentionally left out the check for duplicate classes in a FJ program; we do this check manually in Java, and rely on the generated validator for all the other checks.

That's all! Note how much code did we have to write in Java for the scoping and the validator... as for the rest, we rely on the Java code generated by XtypeS.

2.2. Type Infererence for a Lambda Calculus

From Wikipedia:

> In mathematical logic and computer science, lambda calculus is a formal system for function definition, function application and recursion. The portion of lambda calculus relevant to computation is now called the untyped lambda calculus. In both typed and untyped versions, ideas from lambda calculus have found application in the fields of logic, recursion theory (computability), and linguistics, and have played an important role in the development of the theory of programming languages (with untyped lambda calculus being the original inspiration for functional programming, in particular Lisp, and typed lambda calculi serving as the foundation for modern type systems). This article deals primarily with the untyped lambda calculus.

As another example of use of XTYPES we also developed a prototype implementation of a l-calculus in XTEXT (we'll show the grammar in the following); in this lambda-calculus we can specify the type of the parameter of the abstraction, but we can also leave it empty; we can then infer the type of each l term. In particular, we infer types using type variables when the type of a term can be generic. The types of this l-calculus can be basic types (in this example integer or string), arrow types, and type variables (denoted by identifiers).

The challenging part in writing a type system for this language is that we need to perform *unification* in order to infer the *most general type* (see, e.g., J. A. Robinson. Computational logic: The unification computation. Machine Intelligence, 6, 1971.).

Again, this is just a tutorial example, but this technique can be used to infer types in another language implemented in Xtext, especially for functional languages.

2.2.1. Lambda in a nutshell

You can think of lambda abstraction as a function definition (without a name), with a parameter (in this version we consider one single parameter) and a body, such as

```
lambda x. x
```

which is the identity function (given an argument it returns the same argument). Lambda application, which corresponds to function invocation, is denoted without the parenthesis, thus if we have a lambda abstraction `M` and an argument `N` we write `M N` to mean "invoke the function `M` passing `N` as the argument.

Both of the following definitions with an explicit type for the parameter are correct:

```
lambda x : string . x
lambda x : int . x
```

These two functions have types, respectively, `string -> string` (given a `string` it returns a `string`) and `int -> int`. Note that *arrow types* associate to the right, thus `a -> b -> c` is to be intended as `a -> (b -> c)`; otherwise, we must use parenthesis.

Indeed, we can be more general and say that the parameter `x` can be "any" type, using a type variable (similar to Java generics):

```
lambda x : a . x
```

This function then has type `a -> a`; note that since we return the argument as we received it, then the return type must be the same as the argument type, thus the type variable `a` must be the same in `a -> a`.

In other cases, we cannot be generic; consider that in our language we have the unary operator `-` which can be used on integers only. Then, the function

```
lambda x . -x
```

imposes `x` to be an integer, thus this function has type `int -> int`.

Other functions can be partially generic, like the following one (which makes a little sense, it's used only as an example)

```
lambda x . 10
```

which has type `a -> int`.

We might also let the system infer the type (and that's what we intend to do with our type system definition).

For non trivial cases the type inference is more interesting than the examples we saw so far; fo rinstance, consider this lambda abstraction

```
lambda x . lambda y. x y
```

which has type `(a -> b) -> a -> b`: how can this be inferred? Informally, `x` cannot be any type, since in the body we read `x y` then `x` must be a function; for the moment we give it a generic type `X1 -> X2`; what can the type of `y` be? It can be a generic type, say `X3`, but since we pass it to `x` then it must have the same type of the argument of `x`, thus we require `X1` to be the same as `X3`. The result of `x y` will have the same type as the return type of `x`, i.e., `X2`. Thus, the above function has the following type: it takes an argument `x` of type `X1 -> X2`, and it returns a function (the inner lambda abstraction) which takes an argument `y` of type `X1` and returns something of type `X2`. Thus, using different type variable names, `(a -> b) -> a -> b` (we used the parenthesis since by default arrow types associate to the right). Again, the type variables make the function generic, provided that the same type is used for all occurrences of `a` and the same type is used for all occurrences of `b`.

Here are some other non trivial examples, together with their inferred types

`lambda x .lambda y.y x`	**a** -> **(a** -> **b)** -> **b**
`lambda f .(lambda x.(f (f x)))`	**(a** -> **a)** -> **a** -> **a**
`lambda f .lambda g.lambda x.(f(g x))`	**(a** -> **b)** -> **(c** -> **a)** -> **c** -> **b**

Note that there are functions which cannot be typed (at least with simple type systems we're used to), e.g.,

```
lambda x . x x
```

cannot be typed, since `x` should be a function, say with type `a -> b`, but since we apply `x` to `x` it should also be of type `a`; however, `a -> b` and `a` cannot be unified, since `a` occurs in `a -> b`.

2.2.2. Lambda implemented in Xtext

This is the grammar for our simple lambda calculus in Xtext:

```
grammar it.xtypes.example.lambda.Lambda with org.eclipse.xtext.common.Terminals

generate lambda "http://www.xtypes.it/example/lambda/Lambda"

Program: term=Term;

// left associative
Term: TerminalTerm ({Application.fun=current} arg=TerminalTerm)*;

TerminalTerm returns Term:
    '(' Term ')' | StringConstant | IntConstant | Arithmetics | Variable | Abstraction;

StringConstant: string=STRING;

IntConstant: int=INT;

Arithmetics: '-' term=Term;

Variable: ref=[Parameter];

Abstraction: 'lambda' param=Parameter '.' term=Term;

Parameter: name=ID (':' type=Type)?;

// right associative
Type: TerminalType ({ArrowType.left = current} '->' right=Type)?;

TerminalType returns Type:
    '(' Type ')' | BasicType | TypeVariable;

BasicType: {IntType} 'int' | {StringType} 'string';

TypeVariable: typevarName=ID;
```

2.2.3. Type inference for Lambda in XTypeS

Thus, we want to write the type system definition in XTypeS for Lambda, also inferring types (performing unification for inferring the most general type). We start with the first parts of the type system, defining the name, the URI to the ecore of Lambda, and some judgment descriptions (note that we will use always `|-` as the judgment symbol, but we use different typing relations, thus we will have 5 kinds of judgments):

```
typesystem Lambda

grammar "platform:/resource/it.xtypes.example.lambda/src-gen/it/xtypes/example/lambda/Lambda.ecore"

judgments
    '|-' '=='
        kind="unify"

    '|-' '==>'
        kind="substitution"

    '|-' '~>'
        kind="mapping"

    '|-' '!-'
        kind="notoccur"
        binary

    '|-' ':'
        kind="type"
end
```

Let’s start by writing the rules which check that a type variable does NOT occur in another type term (denoted by `!-`):

```
// base case, always succeeds
axiom NotOccurType
    G |- var Type t1 !- var Type t2

rule NotOccurVar
derives
    G |- var TypeVariable t1 !- var TypeVariable t2
from
    $t1.typevarName != $t2.typevarName

rule NotOccurVarInArrow
derives
    G |- var TypeVariable t1 !- var ArrowType t2
    error='type variable ' + $t1.typevarName + ' occurs in (' +
        $t2.left + ') -> (' + $t2.right + ')'
from
    G |- $t1 !- $t2.left
    G |- $t1 !- $t2.right
```

The meaning should be straightforward: a type variable occurs in another one, if they both have the same name, and a type variable occurs in an arrow type if it occurs either on the left or on the right. Negate these properties, and you get the *not occurs* rules above (for any other type *not occurs* always holds).

Before entering the details of the unification rules, we must say where we collect the substitutions to apply to variables that we collect during the unification; in fact, the unification will not directly apply the substitutions to unify the terms, but it will only collect the substitutions (or fail in case two terms cannot be unified). We can use the *typing judgment environment* (see also Typing Judgment Environment and environment operators) that each rule receives as an argument to store all these substitutions. Thus, we must make sure to pass around always the same environment.

In the environment we will insert mappings where key is a *TypeVariable* name and the value is a Lambda *Type* that we map the type variable to. However, we cannot simply insert a mapping in the environment (this would override the previous mapping): we should check whether a mapping for the same variable is already there. If a variable `X` already maps (in the environment) to a type *t* , and we want to insert a new mapping for `X`, say the type *t'*, instead of inserting the new mapping we should try to unify `t'` with `t` (which will probably generate other mappings).

Thus, denoting all these operations with the relation `~>`, and by assuming that we perform them ONLY after checking that the occur check succeeded, we can write the following rule:

```
rule ExtendVariableMapping
derives
    G |- var TypeVariable v1 ~> var Type t1
from
    (var Type currentMappingForV1 := (Type) env(G, $v1.typevarName)
     G |- $currentMappingForV1 == $t1
     or
     G += $v1.typevarName -> $t1
    )
```

Remember that `==` represents the rules to perform the unification. Thus, as we said, if there's a mapping we invoke the unification with the value of the current mapping, otherwise we actually insert the mapping in the environment (the `->` in the last line is the "mapping" keyword operator, see Typing Judgment Environment, NOT the lambda *ArrowType* operator).

For instance, if we have the mapping `X` maps to `Y` and we are trying to add the mapping `X` maps to `int -> string`, we will actually try to unify `Y` with `int -> string`; in the end we will have, in the environment `X` maps to `Y` and `Y` maps to `int -> string`.

Now let's start writing the rules for the unification (denoted by `==`):

```
// base case
rule UnifyType
derives
    G |- var Type t1 == var Type t2
from
    $t1 = $t2

rule UnifyVar
derives
    G |- var TypeVariable v1 == var TypeVariable v2
from
    (
    $v1.typevarName = $v2.typevarName
    or
    G |- $v1 ~> $v2
    )
```

The first one is the base case; the second one says that two variables unify if they have the same name (without any further substitution); otherwise, we record the substitution.

```
axiom UnifyIntType
    G |- var IntType i1 == var IntType i2

axiom UnifyStringType
    G |- var StringType s1 == var StringType s2
```

Basic types in our implementation are simply instances of *IntType* or *StringType* which do not contain any field; thus the rules for basic types are trivial.

```
rule UnifyVariableBasicType
derives
    G |- var Type t1 == var BasicType b
from
    var TypeVariable v1 := (TypeVariable) $t1
    G |- $v1 ~> $b

rule UnifyBasicTypeVariable
derives
    G |- var BasicType b == var Type t1
from
    var TypeVariable v1 := (TypeVariable) $t1
    G |- $v1 ~> $b
```

A *Type* `t1` unifies with a basic type `b` (note that we use here the base class for basic types) only if it `t1` is a type variable; and we record the substitution (actually we use the rule `ExtendVariableMapping`, but from now on we will simply say “record the substitution”).

Now let’s see the rules for unification which concern an arrow type:

```
rule UnifyVariableArrow
derives
    G |- var Type t1 == var ArrowType a2
from
    G |- $t1 !- $a2
    var TypeVariable v1 := (TypeVariable) $t1
    G |- $v1 ~> $a2

rule UnifyArrowVariable
derives
    G |- var ArrowType a2 == var Type t1
from
    G |- $t1 == $a2

rule UnifyArrow
derives
    G |- var ArrowType a1 == var ArrowType a2
from
    G |- $a1.left == $a2.left
    G |- $a1.right == $a2.right
```

A type variable unifies with an arrow type (rule `UnifyVariableArrow`) if it does not occur in it; and then we record the substitution. The symmetric case trivially uses the previous one. Finally two arrow types unify if the single elements unify respectively (`UnifyArrow`). Remember that, at run-time XTypeS will select the rules according to the actual types of the elements.

During the actual rules which assign a type to lambda terms, we will check whether two type terms unify and if they do, we'll need to perform actual substitutions. Thus, we define the rules for the "substitution" judgment, denoted by `==>`. In these rules, the left parameter is intended as input and the right one as output.

```
rule SubstituteType
derives
    G |- var Type t ==> var Type result
from
    $result := $t

rule SubstituteTypeVariable
derives
    G |- var TypeVariable v ==> var Type result
from
    (
    $result := clone((Type) env(G, $v.typevarName))
    G |- $result ==> $result // recursive
    or
    $result := $v
    )
```

Note that when we substitute a type variable we create a new *Type* by cloning the value tha type variable maps to (see clone). However, we cannot stop here: a type variable can map to another type variable (see the discussion about the rule `ExtendVariableMapping`) with its own mapping in the same environment, and we must make sure we apply all the substitutions that concern a type variable, thus we apply the substitution judgment recursively on the result. For instance, if we have the mappings `X1` to `(int->X2)` and `X2` to `string` and we have the type X1->X2, after the substitution we must have `->string`.

```
rule SubstituteArrowType
derives
    G |- var ArrowType a ==> var Type result
from
    var Type newLeft
    var Type newRight
    G |- $a.left ==> $newLeft
    G |- $a.right ==> $newRight
    // substitution already clones
    $a.left := $newLeft
    $a.right := $newRight
    $result := $a
```

The substitution for an arrow type simply delegates it to its components.

Now we are ready to write the rules for the actual "type" judgments, which will infer the type of a lambda term. (The base case is to make the type system of XTypeS happy). These rules take a *Term* as the left input parameter, and return the *Type* as the right output parameter.

```
// base case not used
axiom TTerm
    G |- var Term term : var Type t

rule TParam
derives
    G |- var Parameter p : var Type t
from
    (
        // if the parameter has already a type, use that
        $p.type != null
        $t := $p.type
        or
        // otherwise we create a type variable with a fresh name
        var TypeVariable v
        $v.typevarName := newname('X')
        $t := $v
    )
```

Remember that a *Parameter* in our implementation can have an explicit type; in that case, the type of a parameter is the one written by the programmer, otherwise, its type will be a *TypeVariable* with a fresh name (using **newname**, see newname). Note that the type of parameters will be used by lambda *Variable* which refer to a *Parameter* (see the grammar). However, as we will see, we will not simply assign to a *Variable* the type of the referred *Parameter* (see `TVariable` in the following).

```
rule TIntConstant
derives
    G |- var IntConstant i : var Type int
from
    var IntType intType
    $int := $intType

rule TStringConstant
derives
    G |- var StringConstant s : var Type string
from
    var StringType stringType
    $string := $stringType
```

Assigning a *Type* to constants is trivial.

```
rule TArithmetics
derives
    G |- var Arithmetics ar : var Type int
from
    var Type termType
    G |- $ar.term : $termType
    var IntType intType
    G |- $termType == $intType
    G |- $termType ==> $int
```

As we said, if we write `-x` we require that the term is of type `int`; thus we try to unify the type of the term with an *IntType*; if the unification succeeds, the resulting type is the type of the term with substitutions. Note that the unification is important, since, if the type of the term is a type variable, this way we record the fact that that type variable must unify with the integer type.

```
rule TVariable
derives
    G |- var Variable v : var Type t
from
    $t := (Type) env(G, $v.ref)
    // perform possible substitutions
    G |- $t ==> $t
```

The type of a *Variable* relates to the referred *Parameter*, but not directly: we assume that during the typing of the abstraction (see the next rule) we put into the environment the type of the parameter of the abstraction, which will be available when typing the body. Remember that when typing a *Parameter* we may give it a type variable with a fresh name. Thus, it is crucial that all the occurrences of a *Variable* referring to the same parameter have the same type, and calling `TParam` rule several times would generate different type variables, which is not what we want. Putting the type of a parameter into the environment during the typing of a lambda abstraction solves this problem.

Of course, after retrieving the type of the parameter from the environment, we must make sure to apply possibly recorded substitutions, before returning the type of the variable.

A possible alternative solution might be, when typing a parameter, in case we assign it a type variable, to also change the *type* field, as in the following:

```
rule TParam // DANGEROUS alternative
derives
    G |- var Parameter p : var Type t
from
    (
        // as above...
        or
        // otherwise we create a type variable with a fresh name
        var TypeVariable v
        $v.typevarName := newname('X')
        $p.type := $v   // assign the type variable to the parameter
        $t := $v
    )
```

However, remember that we are acting (at run-time, when the typing rules are evaluated) to the elements of the EMF model, the AST of your program. Xtext automatically synchronizes changes to the model with the text of the program. Thus, during the application of this rule, the program would change! This, almost surely, is not what you want.

Let us now concentrate on the rule for lambda abstraction:

```
rule TAbstraction
derives
    G |- var Abstraction fun : var Type t
from
    // type of the param
    var Type paramType
    G |- $fun.param : $paramType
    // type of the body with assumption for param
    var Type bodyType
    G += $fun.param -> $paramType
    G |- $fun.term : $bodyType
    G -= $fun.param
    // perform substitutions
    G |- $paramType ==> $paramType
    G |- $bodyType ==> $bodyType
    // build the result arrow type
    var ArrowType arrowType
    $arrowType.left := clone($paramType)
    $arrowType.right := clone($bodyType)
    $t := $arrowType
```

For typing a lambda abstraction, we take the type of the parameter, and we type the body of the abstraction after putting the mapping for the type of the parameter in the environment (consistently, after typing the body, we remove that mapping). Then we apply possible substitutions to the type of the parameter and the type of the body, and we return a brand new arrow type, using the two components.

```
rule TApplication
derives
    G |- var Application a : var Type t
from
    var Type funType
    G |- $a.fun : $funType

    // make sure $funType is an arrow type
    var ArrowType genericArrowType
    var TypeVariable Arg
    var TypeVariable Ret
    // create a generic arrowtype
    $Arg.typevarName := newname('X')
    $Ret.typevarName := newname('X')
    $genericArrowType.left := $Arg
    $genericArrowType.right := $Ret
    G |- $funType == $genericArrowType
    G |- $funType ==> $genericArrowType

    var Type argType
    G |- $a.arg : $argType

    // make sure that the left unifies with arg type
    G |- $genericArrowType.left == $argType
    G |- $genericArrowType ==> $genericArrowType
    $t := $genericArrowType.right
```

For typing a lambda application, we require the left part to be a function, thus we take its type, make sure it can be unified with a generic arrow type (remember that calling **newname** different times produces different names, see newname). If it can be unified we perform substitutions. then we take the type of the argument, and we try to unify it with the left part of the arrow type. If also this unification succeeds, then we perform the substitutions on the whole arrow type, and we return as the resulting type the right part of the arrow type.

Let’s see, informally, how these rules infer the type for

```
lambda x. lambda f. f -x
```

To avoid ambiguities with the `->` of *ArrowType* in the following example we use `-->` for environment mappings. We will also show the collected substitutions in `{ }`. Try not to confuse mappings for parameters and mappings representing type variable substitutions, which have strings as keys.

- TAbstraction
  - what is the type of x? a type variable X1
  - type the body (lambda f. f -x) with mapping x --> X1

- TAbstraction
  - what is the type of f? a type variable X2
  - type the body (f -x) with (additional) mapping f --> X2
    - TApplication
      - what is the type of the left part? a type variable X2
      - (we used TVariable)
      - but it must be an arrow type (X3 -> X4)
      - unify X2 with (X3 -> X4)
      - this will generate the substitution ‘X2’ --> (X3 -> X4)
      - {‘X2’ --> (X3 -> X4)}
      - thus now the type of f is (X3 -> X4)
      - what is the type of the right part? it’s int
      - (we used TArithmetics, and we unified type of x, X1 with int)
      - {‘X2’ --> (X3 -> X4), ‘X1’ --> int}
      - try to unify X3 with int
      - {‘X2’ --> (X3 -> X4), ‘X1’ --> int, ‘X3’ --> int}
      - thus now the type of f is (int -> X4)
      - the type of the application is X4
  - apply the substitution to the type of param and body
  - f of type X2, becomes f of type (int -> X4)
  - the type of body (f -x) becomes X4
  - the resulting type is then (int -> X4) -> X4
- apply the substitution to the type of param and body
- x of type X1 becomes x of type int
- the body has type (int -> X4) -> X4
- the type of our term is int -> (int -> X4) -> X4

This type, let’s write it for simplicity as `int -> (int -> a) -> a` says that we must pass to this lambda term a function which takes an integer, and a function which takes an integer and returns “any type”; as a result we will have that “any type”. Thus the following lambda application terms are well typed (remember that application associates to the left)

```
(lambda x. lambda f. f -x) (10) (lambda y. -y)
(lambda x. lambda f. f -x) (1) (lambda y. "foo")
```

The first one will return `10` (actually `--10`), the second one `"foo"`.

2.2.4. Type Checking for Lambda

Now that we wrote all the rules for inferring types of Lambda terms, we can write an ‘OK’ rule (see ‘OK’ Rules) that says that a Lambda *Program* is correct if we can assign a type to the term:

```
rule TProgramOk
derives
    G |- var Program p : 'ok'
from
    var Type t
    G |- $p.term : $t
```

2.2.5. Using the generated code for Lambda

We can use the validator generated by XTypeS in the default *LambdaJavaValidator* (the one generated by Xtext), similarly to the way we did for FJ (see Using the generated code for FJ).

Furthermore, we wrote a custom *StringProvider* (see StringProvider), namely *LambdaStringRepresentationWithTypeBeautifier*, so that we generate better string representations for our Lambda terms; in particular, we "beautify" types by assigning better names to inferred type variables (e.g., instead of `X1 -> X3 -> (X4 -> X3)` we will get `a -> b -> (c -> b)`) and by putting parenthesis around arrow types (remember that arrow types associate to the right, see Lambda in a nutshell). We are not showing here this class (but you can find in the sources of the Lambda example).

Thus, we inject our string provider by customizing *LambdaRuntimeModule*:

```
public class LambdaRuntimeModule extends
           it.xtypes.example.lambda.AbstractLambdaRuntimeModule {

    public Class<? extends it.xtypes.runtime.StringProvider> bindStringProvider() {
        return LambdaStringRepresentationWithTypeBeautifier.class;
    }
}
```

As for scoping, we do not need any custom implementation in Lambda.

We can also use the generated Java code for the inference type system to create an editor popup action (for the editor of Lambda) to automatically infer types and add the types in the program text, in particular we set the types of the parameters of abstractions. Most of the code we show here deals with Eclipse plugin development: the part which uses the code of the type system generated by XTypeS is minimal.

This is done in the class *LambdaTypeModifier* (you can see the complete code in the sources of the Lambda example); we show here just the main parts:

```
public class LambdaTypeModifier {

    @Inject
    protected LambdaTypeSystemDefinition typeSystem;

    public RuleFailedException setAllTypes(Term term) {
        TypeSystemResult<Type> result = typeSystem.typeAsType(term);
        if (result.getFailure() != null)
            return result.getFailure();
        Type inferredType = result.getValue();
        new LambdaTypeBeautifier().beautifyTypeVariables(inferredType);
        setAllTypes(term, inferredType);
        return null;
    }
    ...
```

Through the generated type system we infer the type of the term and then we update all the types of the parameters of the abstractions in the term. Note that before we "beautify" the type variable names.

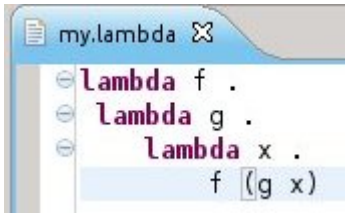
As explained in the Xtext documentation, since we need to modify the EMF model of the program in the open editor, we need to use the *IDocumentEditor* and its *process* method; this is done by the class *LambdaTermModifier*:

```
public class LambdaTermModifier {

    @Inject
    protected LambdaTypeModifier lambdaTypeModifier;

    public void modifyTermWithInferredType(IXtextDocument xtextDocument) {
        IDocumentEditor iDocumentEditor = LambdaUiUtil.getDocumentEditor();
        iDocumentEditor.process(new IUnitOfWork.Void<XtextResource>() {
            public void process(XtextResource resource) {
                Program program = (Program) resource.getContents().get(0);
                lambdaTypeModifier.setAllTypes(program.getTerm());
            }
        }, xtextDocument);
    }
```

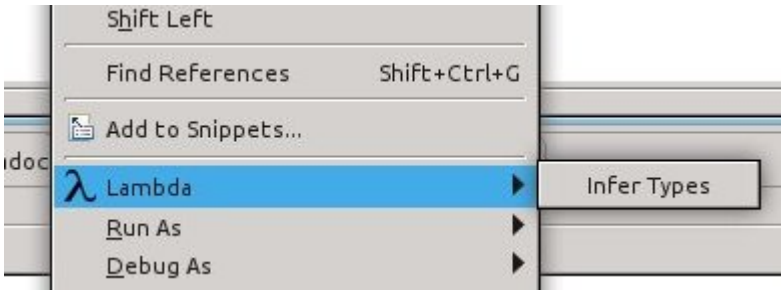
Then we have the code implementing the editor action:

```
public class InferTypesAction implements IEditorActionDelegate {
    protected IEditorPart editor;

    protected LambdaTermModifier lambdaTermModifier;

    public InferTypesAction() {
        lambdaTermModifier = LambdaUiUtil.getInjector().getInstance(
                LambdaTermModifier.class);
    }

    @Override
    public void run(IAction action) {
        IXtextDocument xtextDocument = ((XtextEditor) editor).getDocument();
        lambdaTermModifier.modifyTermWithInferredType(xtextDocument);
    }

    @Override
    public void setActiveEditor(IAction action, IEditorPart targetEditor) {
        this.editor = targetEditor;
    }
}
```

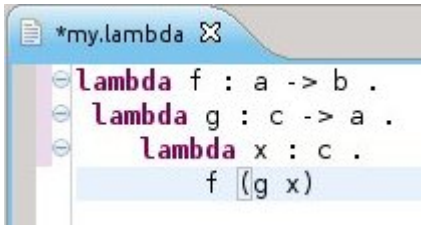
Let’s see this editor action in action: suppose you have the following opened lambda file (this is the abstraction for composing functions):



We can right click on the editor and select the infer type action.



And this is the modified program with the inferred types:



Chapter 3. Reference

3.1. error specification

You can specify a custom error in the rule definition (after the parameters) to provide a more meaningful error message. You can concatenate with operator + strings and objects. For instance,

```
rule SubClass
derives
    G |- var Class C1 <: var Class C2
    error=$C1 + ' is not a subclass of ' + $C2
from
    ...
```

You can also specify an error after judgments in the premises of a rule, e.g.,

```
rule TNewOk
derives
    G |- var New e : 'welltyped'
from
    var List[Field] fields := getall($e.type, fields, extends)
    len($e.args) = len($fields)
        error='argument number ' +
            len($e.args) + ' is not equal to field number ' +
            len($fields)
    ...
```

3.2. equal operator (=) and (!=)

The predefined equal operator = has the same semantics of Java `equals`; thus it can be used to compare strings and integers in the expected way. But when you apply to the objects of your language, remember that the equality succeeds only if the two operands are the same object. Of course we also have the *non equality* operator `!=`.

3.3. len

This predefined function requires a list of elements as the argument and returns its size.

3.4. env

With this function you can access the mappings contained in a typing judgment environment. This function has to be invoked as follows:

```
env(G, key)
```

where `G` is the environment, and `key` is any object. It returns an `Object` so you need to cast the result in your rule. It fails if no mapping is found in the environment.

3.5. clone

It returns a clone of the argument; it relies on cloning functionalities of EMF.

3.6. cast

Type casting has the same syntax and semantics as in Java; statically the types involved must be related. At run-time a cast can fail, making the containing group of judgments fail.

3.7. fail

This judgment always fails; you can also specify a custom error. This judgment can be useful for base case rules, to make sure that for base case the rule fails, e.g.,

```
rule BaseCase
derives
    G |- var Expression e : 'welltyped'
from
    fail='cannot type a generic expression'
```

3.8. success

On the contrary, this judgment always succeeds. This can be useful in or branches, where the judgments of the first branch are not crucial. For instance, in the case of FJ (see Featherweight Java), we can write a rule that computes all the subclasses in a program of a given class as follows:

```
rule FindSubClasses
derives
    G |- var Class C :~> var List[Class] subclasses
from
    var Program P := (Program) container($C)
    foreach $P.classes as class {
        (
            G |- $class <: $C
            $subclasses += $class
            or
            success
        )
    }
```

That is, we scan all the classes of a program, and if a class is a subclass of the given one we add it to the result list (see list operators) (the right parameter of the rule is an output parameter), otherwise we simply go on with the other classes.

3.9. getall

This function can be used to collect elements in the EMF model representing the AST of a program, it has three parameters:

```
getall(obj, feature, extendFeature)
```

where `obj` is an object of the model, `feature` is the feature name of (the class of) `obj` and `extendFeature` is the feature name of (the class of) `obj` to follow to keep collecting elements.

For instance, we've used this function in the type systme of FJ (see "Type checking FJ expressions"#FJTypeChecking) as follows:

```
var List[Field] fields := getall($e.type, fields, extends)
```

where `e.type` is a FJ *Class* object: we request all the *fields* in the object class and all the *fields* we find following the feature *extends*; we stop the collecting as soon as *extends* is null or when we encounter an element already examined. The last point is very important: this function avoids entering in infinite loops in case there are cycles in the model (in FJ this might happen in case of a cyclic class hierarchy).

Similarly we can get all the base classes in the hierarchy of a FJ *Class* `$C` as follows:

```
var List[Class] classes := getall($C, extends, extends)
```

The static type system of XTypeS checks the invocation is correct, and also assigns a static type to `getall` based on the type of the feature *feature*; since *extends* is a reference to a *Class* the result is a *List[Class]*; since *fields* is a list of *Field* the result is a *List[Field]* (pay attention on how the type system of XTypeS (and also the run-time system) computes the resulting type of `getall`.

3.10. container

This function, given an object, (actually an *EObject*) returns its (EMF) container.

3.11. foreach

This loop which has the shape

```
foreach <list> as <element> {
    <judgments>
}
```

requires that all the judgments in the body are satisfied by each element in the list. The variable represented by `element` is available only in the body of the `foreach`.

3.12. forall

This loop uses an index variable (which can be used in the body of the loop to access a specific element of a list, elements start with index `0`). All the judgments in the loop must be satisfied at each iteration

```
forall <index> in <number> {
    <judgments>
}
```

The loop makes `index` go from `0` to `number-1`.

3.13. exists and not exists

These loops require that there exist at last one element (no element, respectively) in the specified list which satisfies each (does not satisfy any, respectively) judgment in the body. The negative variant is expressed either as `!exists` or `not exists`.

```
exists <elem> in <list> {
    <judgments>
}

!exists <elem> in <list> {
    <judgments>
}
```

3.14. list operators

We provide simple operators for adding an element to a list with `+-` and remove an element from a list with `-=`. These operators never fail.

3.15. environment operators

3.16. newname

This function returns a string by appending an incremental number to the passed string argument; this way we can generate new names. Each invocation will generate a new name, e.g.,

```
newname('X') // generated X1
newname('X') // generated X2
```

Assumptions on the possible generated names cannot be made, but that it will be unique.