The Nosica implementation book

David Jobet

The Nosica implementation book

David Jobet

Table of Contents

List of Examples

Chapter 1. General architecture

The compiler is a multi pass compiler. First pass parses Nosica source code and build an Abstract Syntax Tree. (AST) Second pass extract symbols from Nosica source file to gather the symbol table (named SymbolEnvironment). Third pass type checks the source to find errors and produce an Intermediate Representation of the code (IR). Fourth pass is dedicated to various optimisations. Fifth pass produce c code.

Parsing

Parsing is performed using a generated parser. The grammar is defined in a file called nosica_parser.jj We use a tool named JavaCC to build our Nosica parser in java. (we will use Ant or SableCC when we have time in the future) Each grammar rule builds a new node in the AST. The nodes are defined in net.nosica.parser.node.* Each nodes are Visitable. We have chosen not to use a tool that builds an AST for you. This is cumbersome to do (write all classes by hand) but this is a little more memory efficient, and we were able to define some inheritance properties an automated tool would not have guessed. The parser is available in a class named NosicaParser. All you have to do is to init() NosicaParser with a file and call the compilationUnit() method to retrieve the AST.

Symbol extraction

Symbol extraction is performed by visiting only top level nodes of the AST. Namely :

- CompilationUnit
- ImportDeclaration
- PackageDeclaration
- ClassDeclaration
- InterfaceDeclaration
- NestedClassDeclaration
- NestedInterfaceDeclaration
- InitializerDeclaration
- DeinitializerDeclaration
- AliasDeclaration
- ConstructorDeclaration
- DestructorDeclaration
- OperatorDeclaration
- MethodDeclaration
- PropertyDeclaration

Those nodes are visited and the Symbol Table is built (the symbol table is called SymbolEnvironment). At this point, as we don't know yet if a type is generic or not, we state the fact that each type is potentially generic by calling them GenericType.

The SymbolEnvironment contains only GenericType. GenericType contain GenericMethod and

Field.

Type checking

Like symbol extraction, type checking is performed by a Visitor. Remaining nodes are analysed recursively till all code of a given method is visited. The type checker ensures types are compatible in operations like comparison, assignation, method call and so on. In fact, in Nosica everything is type checked using method call as all operations are described in source code as a method call. As such, a reference assignment is a call to Object.operator=. A primitive assignment is legal only when defined on the referenced primitive type. Identicaly, comparisons are legal only if the operations exist on the type. The task of finding matching methods in method call is left to the MethodResolver. Basically, a signature is created using the instance, the name of the method to call, and the type of the arguments. Then the MethodResolver returns a list of matching methods. If there is only one, there is no errors. If there is none this is an error as no methods were resolved. If there are more than one, then we have an ambiguity we need to report to user. In parallel, the type checker produces a simplified representation of the code : the Intermediate Representation (IR). This representation is composed of fewer nodes (~20 nodes) of very low levels that are very easy to analyse. This is this representation that is used later to perform analysis of the code for optimisations and code production.

Optimisations

Optimisations are performed on IR. This simple representation can be analysed in algorithms like

- constant propagation
- typename analysis

The IR can also be modified in algorithms like :

- Basic Block building/rearranging
- SSA building (Simple Statement Assignment)
- Loop rearranging/unrolling

Of course, we can imagine a lot of algorithm to optimise code, but for now, we have only typename analysis. We will need to perform SSA building (and unSSA at the end), because it simplifies other algorithms. We have talked of implementing constant propagation too. This algorithm will allow to optimise further typename analysis, code branching ...

Code production

Code production is actually performed by emitting c code. For now, we emit a large chunk of c source code. We have a class named CConverter that is responsible for name mangling, by value/adress argument passing, etc... In the future we will want to emit source code in smaller chunk. This will enable us to compare them with older versions of a previous compilation, thus enabling to recompil only the changed portion of the code. It will also be possible to perform code production in other programming languages : Java or Python or whatever and even assembly. But I really think assembly is a bad idea as we don't have the resources to produce code optimised for several architecture. C compiler like gcc already do that.

Chapter 2. Parsing

Current process

First step

The first step uses 3 files to create the JavaCC parser.

Tool used. jjkeywords.pl (in tools/)

Directory. compiler/sources/net/nosica/parser

Input.

- reserved.cfg
- keywords.cfg
- nosica_parser.jj.in

Output. nosica_parser.jj

Note

Historically, we wanted to automatically generate a tex documentation with files keyword.cfg and reserved.cfg. We never did it though. In the future we will modify the way we generate the parser by removing the first step. We will have the possibility to generate automatically documentation by extracting the keywords and reserved words instead of generating them. This will be a lot simpler. If you want to do this, feel free !

Second step

The second step uses the generated JavaCC parser to create the Java Parser

Tool used. javacc

Directory. compiler/sources/net/nosica/parser

Input. nosica_parser.jj

output.

- NosicaParser.java
- NosicaParserConstants.java
- NosicaParserTokenManager.java
- ParseException.java
- Token.java
- TokenMgrError.java

Modifying the parser

The parser

As you can see, the real input file is in fact the file nosica_parser.jj.in. If you want to look at the file I'd suggest strongly to read first the JavaCC documentation. Basically, let's say we have the following BNF rule :

Example 2.1. A simple BNF rule

```
tuple ::=
  '(' ')'
|
  TypeName
|
  '(' TypeName (',' TypeName)* ')'
```

We will want to modify the JavaCC parser by adding a new rule having the following syntax :

Example 2.2. The BNF rule in JavaCC

```
void Tuple() :
{
}
{
  '(' ')'
|
  TypeName()
|
  '(' TypeName() (',' TypeName())* ')'
}
```

Then we will add some Java code to retrieve data returned by rule TypeName() like this :

Example 2.3. Implementing the JavaCC rule

```
Tuple Tuple() :
{
  // we add here local variables
  Vector typeNames = new Vector(); // Vector&lt;TypeName&gt;
  TypeName typeName = null;
}
{
  '(' ')'
|
  typeName = TypeName()
  {  // we can add java code anywhere by opening a block with a brace
    typeNames.add(typeName);
  }
|
  '('
    typeName = TypeName() { typeNames.add(typeName); }
    (',' typeName = TypeName() { typeNames.add(typeName); }
    )*
```

```
  ')'
  {
    return new TupleImpl(typeNames);
  }
}
```

Of course, we will have to modify the existing BNF to add a 'call' to the new added BNF rule. In this case, we will have to modify the MethodDeclaration() rule (for example) so that our new rule is taken into account.

Note

The Nosica BNF is derived from the Java BNF. We have added BNF rules for

- constructors
- destructors
- static initializer
- static deinitializer
- genericity
- alias declaration (method aliasing and package aliasing)
- properties (to be removed in a future version)
- operators

We have removed rules like

- bitwise operators (because we want them to be performed in library)
- postfix increment operators (we think they are both inefficient and unneeded. In a high level language, we only need one operation to perform an incrementation)

Of course we have modified other rules as well to introduce

- constness
- primitive types
- native types
- inner types

Adding missing classes

In the former example, we have added code to parse and extract informations from the source code for a new BNF rule : the tuple rule.

We will now want to add the missing classes and interfaces we have referenced in our Java code : Tuple and TupleImpl

Example 2.4. Creating the Java Tuple interface

```
package net.nosica.parser.node.util.var;

import net.nosica.parser.node.Node;

interface Tuple extends Node
{
 /**
  * \brief return elements of this tuple
  *
  * \return Vector<TypeName>
  */
 Vector getTypeNames();
}
```

And

Example 2.5. Implementing the TupleImpl class

```
package net.nosica.parser.node.util.var;

class TupleImpl implements Tuple
{
 private Vector typeNames;  // Vector<TypeName>

 public TupleImpl(Vector typeNames)
 {
  this.typeNames = typeNames;
 }

 public Vector getTypeNames()
 {
  return typeNames;
 }

 // implementation of Node.accept for the visitor pattern
 public void accept(Visitor visitor)
 {
  ((NodeVisitor)visitor).visit(this);
 }
}
```

> **Note**
>
> Node is the base class of all nodes of the Abstract Syntax Tree (AST) used in Nosica. A Node is Visitable and Localizable. The first interface allows one to visit the AST, the latter allows one to assign information to localize a particular node in the source code. The information consists of a file name, a line number and a column number. Currently all information is contained in a single string. In the future it is likely we want to split it into several different fields.

We have now added the missing files for the parser to compil : net.nosica.parser.node.util.var.Tuple, and net.nosica.node.util.var.TupleImpl. Keep in mind this is only an example. This is very unlikely we implement tuples this way.

We will now have to modify the NodeVisitor visitors. To make things only compil, we have only to modify the net.nosica.parser.node.AbstractNodeVisitor and add the missing method AbstractNode-Visitor.visit(net.nosica.parser.node.util.var.Tuple tuple)

From this point on, the parser will be able to recognize tuples. We will now have to modify the symbol extraction pass and the type checker to handle them.

Note

Currently, this is how we choose the namespaces of the parser classes :

- filestructure : in this package we have classes related to package and import declarations : these are the declaration placed before the classes and interfaces declaration.
- util : in this package we have miscellaneous classes : classes related to genericity, typename, literal, operators ...
- typedeclaration : in this package we have classes related to declaration of types : class declaration, interface declaration, methode declaration, field declaration ...
- typedeclaration.expression : in this package we have classes related to expressions in methods. Expressions are pieces of code with side effect but with no modifications of the flow of the program. Things like AdditionExpression, PrimaryExpression ...
- typedeclaration.statement : in this package we have classes related to statements. These are pieces of code that possibly have side effects *and* control the flow of the program : things like Block, IfStatement, ForStatement ...

Chapter 3. Symbol extraction

The process

All classes related to symbol extraction to feed the symbol table are located in the package net.nosica.symbol

As we need only to create symbol information (ie type description), we need only to visit the upper nodes of the AST. Therefore, the algorithm is quite simple to do and to understand.

Basically, when we visit a ClassDeclaration, we create a new type (a GenericType). Then recursively we visit all sub nodes of a ClassDeclaration (all of type ClassBodyDeclaration or InterfaceMemberDeclaration). This way we will read all methods and fields and sub types of a particular types.

This process is implemented in class net.nosica.symbol.FeedSymbolVisitor. As the name of the class indicates, the purpose is to visit the AST and to extract relevant information to feed the symbol table.

The datas

Our symbol table is called SymbolEnvironment. It contains only the types gathered during the symbol extraction. You can see it as a Map of TypeName to GenericType. In fact this is a little more complicated as several generic types can have the same name. (in the case some are specialisations of another one)

Example 3.1. Multiple generic types having the same name

```
package some.package;

class MyTraits<T implements Numeric>
{
  word limit() { return T.max(); }
}

class MyTraits<int8>
{
  word limit() { return 255; }
}
```

In this example, the FeedSymbolVisitor will create two instances of GenericType. First one will be named as some.package.MyTraits<T>, second one will be named some.package.MyTraits<int8>

At this stage (is symbol extraction), we can't resolve yet types like int8. This is because we can't know if we have parsed int8 yet. Because of that, we cannot know if int8 is a real concrete type. In this case, we handle int8 and T the same way : as generic types.

Later, we will be able to understand int8 is a real concrete type, hence some.package.MyTraits<int8> is not a generic type but a real concrete type. But right now we don't know yet.

Instead of storing GenericType directly as a map TypeName->GenericType, we prefer to store them by their non generic typename.

A non generic typename being just the typename without any generic information.

Example 3.2. Non generic typename

Complete typename (a generic one) : some.package.MyTraits<T>

Non generic one : some.package.MyTraits

As specialised version of generic types may exist, this means the key we use is not unique. For example, the types from Example 3.1, “Multiple generic types having the same name” [8] will have the same non generic counterparts.

Hence, the first key gives access to a secondary map indexed by the real complete generic typename.

Hence to access GenericType 'GT' having 'tn' as TypeName, the symbolEnvironment must follow the steps :

1. create 'ngtn' (non generic typename) as the non generic counterpart of 'tn'
2. access the value (which is a map) indexed by key 'ngtn'
3. access the value in the secondary map by key 'tn'
4. access 'GT'

Chapter 4. Type checking

A Nosica class is type checked method by method. The method's complete AST is passed to the type checker and is visited by the type checker algorithm.

Currently, the type checker is made of two separate Java class. The first one, called StatementVisitor, visits and typecheck statements. The second one, TypeCheckVisitor, visits and typecheck expressions. The reason for splitting the type checking in two different class is only to have smaller algorithm to manage. Another reason is that type checking statements and expressions are really two different jobs.

The type checker has another job : it must translate on the fly the Nosica code into a lower level repreentation we call Intermediate Representation (IR). The IR being composed of fewer different nodes, this simplifies the other stage of the compiler like analysis, optimisations and target code production.

Type checking, or ensuring types are compatible

The main job of a type checker is ensuring types are compatible. For example, let's say I have a method whose full signature is :`intf`*`int i`*Then calling f() with a string must lead to the detection of an error. To detect there is an error, we must be able to deduce the type of the parameter we pass to method f, as well as which method is f.

Knowing the type of an expression

To know the type of an expression, we have to type check each component of the expression till we end up with a terminal expression whose type is known. We know the types of literals, like int or string. We also know the types of variable, because in Nosica, you have to define a variable (with its type) before using it. So, if we encounter a variable like say 'var', either we have registered it before and we know its type, either we do not know it and that's a type check error. In an expression like :

Example 4.1. A simple expression

int var = 3; 1 + var

We know the type of '1' : that's net.nosica.lang.int32, and we know the expression of var because we have found it in the VariableEnvironment. And that's net.nosica.lang.int32 too.

When typechecking this expression we will end up in method visit(AdditiveExpression) of class TypeCheckVisitor. The node contains several information :

the left expression (lhs) : a literal (type net.nosica.parser.node.util.literal.Literal) the right expression (rhs) : a PrimaryExpression (type net.nosica.parser.node.typedeclaration.expression.PrimaryExpression) the operator : either '+' or '-'.
The type checker will first visit the left expression and will return with the appropriate type. Then, the type checker will visit the right expression and return with the appropriate type too. Then the typechecker will have to determine if int32 and int32 can be added via the '+' operator. To do do, it will have to look into lhs and see if it exists a '+' operator. It will have to see if this operator accept a int32 as parameter. If this is the case, then the call can be done, else this is a type check error.

Now that we have found int32 indeed allows a '+' operation with another int32 as rhs, we can take the result of this call (of type int32) and returns it as the result of the type checking of an AdditiveExpression.

Compatible rules

When type checking the general rule "Expression1 Op ExpressionArguments", we will end up with the following type checked arguments : TypeName1 Op TypeNameArguments. With TypeNameArguments a list of TypeName. This signature will have to be searched on type TypeName1, and as result, a list of methods matching operation name "Op" will be returned. We now have to select a subset of those methods by looking if type name of arguments are compatible.

a primitive is compatible with another primitive if and only if they are of same type, or if it exists a cast operator between the source and the target.
a reference is compatible with another reference if and only if they are of same type, or if the source type derives from the target type.

Storing variable : the VariableEnvironment

Variables are stored in a VariableEnvironment. Basically, this is a table indexed by the name of a variable.

The VariableEnvironment has another job : it must remember the scope of variables. This enables the type checker to only say : "begin a scope", "add this variable", "end the scope (and returns the list of terminated variables)".

Scope is important because in Nosica, destructor must be called as soon as the variable reaches end of scope.

For a primitive type, this is easy : this is at the end of the scope. So we have to get the list of terminated variables and insert code (in the IR) to call the appropriate destructors.

Feor a reference type, this is a little more complicated as reference variable must be destroyed as soon as the instance is no more reachable from the code. As we can register an instance (the instance is being pointed to by the variable) in a global structure like a container, a field of a class, or in a variable with a longer scope, reference variable may escape the scope where they have been declared. To handle it, we have chosen just to insert a special node in the IR which is called SCOPELEAVE and just indicates that the variable is no longer live. This node will later be handled by a garbage collector (implementation dependant) and/or an analysis algorithm. In the current implementation, the garbage collector being just a reference couting algorithm (thus that do not handle circular references), we just add where needed couple of INCREF/DECREF IR node.

```
 VariableEnvironmentImplimplements VariableEnvironment {

  void save();

  Vector<Variable> restore();

  void getNumberOfActiveScope();

  Vector<Variable> variableInScope(int nbScope);

  add(Variable variable);

  void get(string name);

}
```

This interface is located in package net.nosica.compiler.typechecker. It is used mainly by StatementVisitor which create a scope (by calling method enter()) each time a block is entered.

When the block is ended, the method restore() is called, thus retrieving the list of terminated variables, and the appropriate code is inserted into IR.

LabelRepository : how to handle jumps

In Nosica, you can use loop to repeat repetitive tasks. Loops like for, do...while, while, and in the future foreach must be handled. We can prematuraly resume a loop or escape it using the keywords continue/break. That means we handle labels.

Labels are mainly computed and created as necessary by the compiler. Sometimes, the user want to specify himself a label because it may want to break or resume several outer loops.

It is important to properly handles labels as, while we leave/enter a block, we have to properly destroy live variables present accross the boundaries of this block.

As for now, the LabelRepository is splitted into two parts. First one is externalised in net.nosica.compiler.application (and should be in net.nosica.compiler.typechecker), second part is handled directly by StatementVisitor.

The LabelRepository has nearly the same interface as the VariableEnvironment. Except that what is being stored are Label instead of Variable.

Each time a Statement is type checked, a pair of Labels named the begin label and the end label are generated. Those labels will be used in case the statement that is going to be type checked is a loop. They are generated in advance because we don't know yet if the labels will have to be computer generated, or user defined. If the statement we're about to type check is in fact a LabeledStatement, then all we have to do is take the user defined label, verify it is not already defined using the LabelReporitory, add it to the LabelRepository and replace the computer generated beginLabel/endLabel with the couple userDefinedLabel/endLabel.

If we encounter a loop, this couple will have to be pushed into a structure handled by StatementVisitor (that should be merged with LabelRepository). This structure has one role : make a correspondance between label's scope and variable's scope so that we can properly defined the latter when we use break/continue keywords. Entering/leaving a block is handled by global methods enterBlock()/endBlock().

Label handling apart, the statement will just translate Nosica loop code into the equivalent IR code. Proper comments are placed in each block describing how is translated the control flow.

ExceptionEnvironment : how to handle type checking of exceptions

To be done

IR translation

IR translation is driven by the type checker. The TypecheckVisitor, handling only exceptions, is quite straightforward. The StatementVisitor handling the control flow is more complicated. Both TypeCheckVisitor and StatementVisitor do not know the IR. They just manipulate an interface called IRTranslator (located in net.nosica.compiler.application).

The IRTranslator is responsible for proper translation between high level Nosica expression to low level IR expressions.

Chapter 5. C Code Production

Nosica can emit code with different backends thanks to its Intermediate Representation stage. For now, only a "c" code production back end has been implemented.

Parameter passing

We're emitting C, so basically, we cannot apply constructors, copy constructors or assign operators directly on the variables stored in the frame.

Passing reference variables is quite straightforward, while passing primitive variables (by value semantic) is a little bit tricky.

Primitive arguments

Const primitive passing

Const primitive arguments have a 'by value' semantic. That means we have to duplicate them when we want to pass them 'const'.

```
string s;
T.f(s);

class T {
  public static void f(const string s) {
    // use s
  }
}
```

is translated into

```
net_nosica_lang_string s;
net_nosica_lang_string_constructor(&s);
T_f(s);

void T_f(net_nosica_lang_string temp) {
  net_nosica_lang_string s;
  net_nosica_lang_string_constructor(&s);
  net_nosica_lang_string_operator_iassign(&s, temp);
  // use s
  net_nosica_lang_string_destructor(&s);
}
```

The rules are simple :

- From the caller, there's nothing special to do
- From the callee, and for an incoming const primitive argument A
  - creates a temporary 'temp'
  - constructs 'temp'
  - calls the assign operator between 'temp' and 'A' : 'temp' <- 'A'
  - modify the IR so that the code uses 'temp' instead of 'A' everywhere.

- destroy 'temp'

Rationale :

- if A contains mutable reference arguments, we want to be able to modify them without the caller's copy to be affected, so we have to duplicate the variables, just as c++ would do.

  In an optimised version, perhaps can we remove duplication when A does not contain mutable fields.

Var primitive passing

Var primitive passing have a 'by pointer' semantic.

```
string s;
T.f(s);

class T {
  static public sub f(var string s) {
    // use s
  }
}
```

is translated into

```
net_nosica_lang_string s;
net_nosica_lang_string_constructor(&s);
T_f(&s);

void T_f(net_nosica_lang_string *s) {
  // use s
}
```

So that's quite straightforward.

References arguments

References arguments are always passed by pointer, so that's easier to handle. The rules are the same, regardless of the 'constness' (or the 'varness') of the variable.

```
A a = new A();
T.f(a);
T.f2(a);

class T {
  static public sub f(A a) {
  }
  static public sub f2(const A a) {
  }
}
```

is translated into :

```
A a;
a = ALLOC();
```

```
A_constructor(a);
T_f(a);
T_f2(a);

void T_f(A *a) {
  INCREF(a);
  // use a
  DECREF(a);
}

void T_f2(A *a) {
  INCREF(a);
  // use a
  DECREF(a);
}
```

So the rules are, for each reference arguments,

- insert an INCREF at the beginning of the function
- insert a DECREF at the end of the function

Results

retrieving result from a function can be tricky in the primitive case

Primitive result

Just as the "const parameter passing", and because we don't emit assembly, we cannot directly modify the variables stored in the stack. Like const parameter passing, we have a "by value" semantic.

```
string s = T.f();

class T {
  static public string f() {
    return string(new char[0]);
  }
}
```

is translated into

```
// sring s;
net_nosica_lang_string s;
net_nosica_lang_string_constructor(&s);
// temp = T.f();
net_nosica_lang_string temp;
temp = T_f();
// s = temp;
net_nosica_lang_string_operator_iassign(&s, temp);
net_nosica_lang_string_destructor(&temp); // end of scope for temp : let's destroy it
net_nosica_lang_string_destructor(&s);

net_nosica_lang_string T_f() {
  // temp = new char[0];
  net_nosica_lang_char_array *temp;
  temp = ALLOC();
  net_nosica_lang_char_array_constructor(&temp, 0);
```

```
  INCREF(temp);
  // temp2 = string(temp);
  net_nosica_lang_string temp2;
  net_nosica_lang_string_constructor(&temp2, temp);
  DECREF(temp); // end of scope for temp : let's destroy it
  // return temp2; (equivalent to assigning it to a 'fake' variable result)
  net_nosica_lang_string result;
  net_nosica_lang_string_constructor(&result);
  net_nosica_lang_string_operator_iassign(&result, temp2);
  net_nosica_lang_string_destructor(&temp2); // end of scope for temp2 : let's destroy it

  return result;
}
```

That seems to be really complicated, but that's quite simple. We use the c compiler to generate a hard copy of the result (just like it does when passing it on the stack). Therefore, that copy must be constructed in the callee and destructed in the caller so that each instance is constructed once and destructed once. That's the hard part. So that's not that hard, is it ?

In the previous example, the result is stored into a temporary variable named "result". This variable is constructed but never destroyed. Of course, the previous code is not optimised. We can construct the result directly into temp2 instead of using another variable, but we're not talking about optimisations here.

The 'result' variable is passed into the 'temp' variable in the caller. And temp is properly destroyed so everything's safe.

The rules are simple :

- in the caller
    - retrieve the result of the call in a temporary variable (do not construct it)
    - at the end of the scope of this temporary variable, destroy it
- in the callee
    - construct a temporary variable 'result'
    - assign the result to the temporary variable 'result'
    - do not destroy 'result'

Reference result

The reference case is much simpler

```
A a = T.f();

class T {
  static public A f() {
    return new A();
  }
}
```

is translated into

```
A *a;
a = T_f();
DECREF(a);

A *T_f() {
  A *temp;
  temp = ALLOC();
  A_constructor(temp);
  INCREF(temp);
  return temp;
}
```

You can draw a parallel here between primitive result and reference result : in the callee, the result must use an INCREF, and should not be DECREFed. In the caller, the variable is received normally and then is DECREFed at the end of the scope.

Unwanted results

Sometime, we call a method that returns a result we're not interested in.

In order for the reference counting algorithm to succeed, each result must be properly assigned a temporary variable which is then destroyed in the usual way, by

- calling the proper destructor for a primitive variable
- applying a DECREF operation on a reference variable

The assign operator

The nosica assign operator has the following signature :

```
class T {
  sub operator~(T rhs) {
    // do the copy
  }
}
```

Originally, the operator's signature was T->T instead of T->(). Unfortunately, the terms of the equations were then :

- T->T signature (T operator~(T rhs))
- by value result

It appears that this equation has no solution. A by-value result needs the copy operator which has a T->T signature. Hence the copy operator needs to returns the result by value, so this leads to an infinite recursion.

As this problem is a language problem and not an implementation problem, we had to change the language on that point.

Definition of a TypeName

TypeName are the basic building block of Nosica's type system. TypeName is a very important basic block as it is created during parsing, and get transformed during all stages of compilation. This is the only type used from parsing to production. So chances are you will see it everywhere.

A TypeName describes several things : package, class or interface name, array information and generic information. A TypeName has several predicats like :

```
boolean isArray();
;
boolean isScalar();
;
boolean isGeneric();
;
boolean isEmpty();
;
```

These predicat allows to analyse the content of a TypeName. If the TypeName is not generic nor an array, it is possible to iterate over its components. You will use methods :

```
Iterator iterateComponents();
;
TypeName getFirstComponents();
;
TypeNameComponent getLastComponent();
;
```

To respectively, iterate through components of TypeName, get the package part of a TypeName (ie, the first N-1 elements), or get last component (ie the last Nth element). Each component of a TypeName is a TypeNameComponent.

Definition of a TypeNameComponent

TypeNameComponent are the components of a TypeName. A TypeNameComponent can be generic or not. The predicat is :

```
boolean isGeneric();
;
```

And the method to iterate over generic parameters is :

```
Iterator iterateGenericExtension();
;
```

This method returns an iterator over all TypeName as the generic parameters. You can retrieve the package component name by calling the method

```
String getPackageElement();
;
```

AliasTable

An alias table is just a kind of map from a short name to a complete TypeName

Structure of an AliasTable

An alias table is composed of

- a parent AliasTable
- a map

During symbol extraction

AliasTable are created during symbol extraction. The top level AliasTable represent package net.nosica.lang. In this root AliasTable all components of package net.nosica.lang are listed. You will find for example

- int8 -> net.nosica.lang.int8
- string -> net.nosica.lang.string

An AliasTable is created for each CompilationUnit (and take net.nosica.lang's AliasTable as parent AliasTable) An AliasTable is created for each classes and interfaces (and take its CompilationUnit's AliasTable as parent AliasTable) An AliasTable is created for each nested classes and interfaces (and take its enclosing's AliasTable as parent AliasTable) The CompilationUnit AliasTable contains aliases for each import directive.

Example 1. CompilationUnit's example

```
package net.myorg;

import net.nosica.containers.Vector;

import net.myOrg.util.MyUtil
```

This will create an AliasTable with parent net.nosica.lang's AliasTable and with a map containing the following keys :

- Vector -> net.nosica.containers.Vector
- MyUtil -> net.myOrg.util.MyUtil

For each classes and interfaces (and nested one), the AliasTable will contain aliases for each import directive, and for each template's arguments

Example 2. Class/Interface example (relevant for nested versions)

```
class MyClass<T> {

  import net.package1.A A1;

  import net.package2.A A2;
}
```

This will create an AliasTable with parent CompilationUnit's AliasTable and with a map containing the following keys :

- T -> T
- A1 -> net.package1.A
- A2 -> net.package2.A

During genericity solving

During genericity solving, the GenericitySolver will transform a type coming from Symbol extraction to a fully resolved non generic (ie with qualified generic parameters) types. During this process, new AliasTable will be created. The resulting AliasTable will contain

- as parent : the AliasTable created during symbol extraction
- as map : the resolved generic parameters

Generic methods will have a custom AliasTable on the same model

- parent : enclosing type's AliasTable
- as map : the resolved generic parameters

Interface of AliasTable

An AliasTable must be seen as a map. As a result, it is possible to iterate through all aliases, to query for the existence of a particular alias, to get the resolved form of an alias.

As an AliasTable "reuse" its enclosing aliasTable (the parent), it is possible to query for an existing parent (true in most cases) and to retrieve the parent AliasTable. However those methods should be of no interest most of the time.

Of course, a query is performed on current AliasTable and sent to parent's AliasTable if the result can not be carried out in the current AliasTable.

Appendix A. GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sen

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in

We have designed this License in order to use it for manuals for free software, because free software needs free docu

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright hol

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied ve

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with th

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sect

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in th

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, La

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, leg

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains X

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Do

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a ma

It is requested, but not required, that you contact the authors of the Document well before redistributing any large nu

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, p

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of
* B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modificati

* C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
* D. Preserve all the copyright notices of the Document.
* E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
* F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Mod
* G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Docu
* H. Include an unaltered copy of this License.
* I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, n
* J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the D
* K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserv
* L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers
* M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
* N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Sect
* O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and con

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified V

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for public

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in secti

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be re

In the combination, you must combine any sections Entitled "History" in the various original documents, forming on

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replac

You may extract a single document from such a collection, and distribute it individually under this License, provide

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is les

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this Licen

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from tin

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numb