Overture – Open-source Tools for Formal Modelling TR-2011-11
June 2011

Restructuring of AST in Overture components: Overture-II

by

Kenneth Lausdahl
Augusto Ribeiro

Contents

1 Introduction **3**
- 1.1 Goals . . . 3
- 1.2 Plan . . . 4
- 1.3 Analysis support . . . 5
  - 1.3.1 Visitor . . . 5
  - 1.3.2 Supporting the VDMJ Type Checker through a `typeCheck` method 5
  - 1.3.3 Delegation through explicit method invocation . . . 6
  - 1.3.4 Plug-in use and derived trees . . . 8
  - 1.3.5 Plug-in access . . . 9
  - 1.3.6 Derived trees/custom plug-in data . . . 9
- 1.4 Illustration with a simple AST structure . . . 10

2 How to move forward **11**
- 2.1 AST Features . . . 11
  - 2.1.1 Structure . . . 11
  - 2.1.2 Enumerated node types . . . 12
  - 2.1.3 Copying . . . 12
  - 2.1.4 Copy-extend . . . 12
  - 2.1.5 Navigation . . . 13

Document History

Revision	Notes	Date
1	Initial proposed version	5.2011

Chapter 1

Introduction

This internal Overture memo aims to provide an overview of the alternatives we have for restructuring the AST inside Overture such that we only have one common AST and VDMJ becomes divided in more manageable components in the overall plug-in architecture. In order to determine the optimal way to get this working small experiments have been carried out on a very small subset of VDM to see the pros and cons with different alternatives.

1.1 Goals

Our goals are to have:

1. A single VDM AST shared by all Overture development. e.g. Parser, Type Checker, interpreter, IDE and other kinds of analysis.

2. An AST which only holds the VDM abstract syntax thus no code for analysis directly inside the AST itself.

3. An AST which is well structured and supports analysis through pre defined tree visitors which should be able to visit the complete tree in a pre defined way.

   (a) Depth first ..

   (b) ...

4. An AST which can be used both in VDM models for interpretation, pure Java code and code generated VDM models. This requirement is necessary because we would like the ability to decide for each component we develop on the Overture platform

if we shall develop it directly using Java or take our own medicine and model its functionality in VDM and then automatically code generate it to Java fitting into the new AST (note that this is currently the approach that was used for the UML mapping).

5. **The AST must support reuse of VDMJ both: Parser, Type Checker and interpreter. This means that the general way VDMJ works must be taken into account but not dictate the structure but a way to reshape the current code to fit the new AST must exist. "This excludes complete redevelopment".**

6. The ability to develop new plug-ins (including ones not developed by Overture) which can do complete analysis of the AST without requiring any changes to the AST.

7. The ability to either share an instance of the Overture AST among plugins or provide a clone working copy which a plug-in can manipulate.

1.2 Plan

1. Decide which features the AST should have. Which methods do we require by `Node` to enable the different kinds of analysis we want. And what is needed to hold the kind of information we require.

2. Create a tool which generates such an AST for Java.

3. Define the structure of the Overture AST in the format required by the new tool.

4. Split up the tasks of replacing the old AST with the new one:

   (a) Parser

   (b) Type Checker

   (c) Interpreter

   (d) Proof obligation generator

   (e) All IDE plug-ins

5. During the development of the new projects where the new AST is used test cases must be created. This means for e.g. the parser that one positive and one negative test must be written for each expression implemented.

1.3 Analysis support

A number of questions has been raised about how the AST should be structured: In particular the need for having a visitor to the AST and the dislike of visitors in general for some kinds of analysis. A few of the needed features are:

- Link between: Parent $\leftrightarrow$ Child
- Search method; allowing any parent of a type X to be found to be found for a node
- Visitors

The following sections will try to address the problem with integration of the current type checker and interpreter with the AST which is implemented inside the current AST.

1.3.1 Visitor

A straitforward visitor would then have a visit method for all types of nodes like shown in listing **??** and there is no return value.

```
public abstract class Node
{
  /**
   * Applies this node to the {@link Analysis} visitor {@code analysis}.
   * @param analysis the {@link Analysis} to which this node is applied
   */
  public abstract void apply(Analysis analysis);
}
```

```
public class AnalysisAdaptor implements Analysis
{
  /**
  * Called by the {@link ABinopExp} node from {@link ABinopExp#apply(Switch)}.
  * @param node the calling {@link ABinopExp} node
  */
  public void caseABinopExp(ABinopExp node)
  {

  }
}
```

Invocation

```
node.accept(new AnalysisAdaptor());
```

1.3.2 Supporting the VDMJ Type Checker through a `typeCheck` method

This idea have previously been discussed over email among a core group of core developers. It was discussed how the code from VDMJ which builds on the fact that each node knows how to type check it self and that the general feeling was that visitors is difficult to use.

1. Use a visitor to visit all nodes through a generated visit structure
2. Implement a method in each node which delegates the method to an instance outside the node.

1.3.3 Delegation through explicit method invocation

To allow a node to behaviour like in the current AST a method could be added to the node itself to enable type check and eval. Listing **??** shows this for the base class `Node`. It is important to understand that this approach requires the AST to be updated if a new analysis should be done also the type argument can not be made type safe do to the lack of knowledge of the type between `Node` and the return type of the concrete analysis.

This was generally the preferred idea. However see section **??** *for a more general approach.*

```
public abstract class Node
{
  public <Typ extends Node> Typ typeCheck(TypeChecker tc, Environment env, NameScope scope, TypeList qualifiers)
  {
    return null;
  }

  public IValue eval(Eval evaluator, Context ctxt)
  {
    return null;
  }
}
```

An `ABinopExp` would then look like in listing **??** where `TypeChecker` and `Evaluator` is two container classes for all type checkers: expression, pattern etc. And the same goes for eval.

```
public class ABinopExp extends PExp
{
  @Override
  public <Typ extends Node> Typ typeCheck(TypeChecker tc, Environment env, NameScope scope, TypeList  qualifiers)
  {
    return tc.getPExp().caseABinopExp(this, env, scope, qualifiers);
  }


  @Override
  public IValue eval(Eval evaluator, Context ctxt)
  {
    return evaluator.getPExp().caseABinopExp(this, ctxt);
  }
}
```

The idea presented in listing **??** would then for the type checker be implemented as shown below in listing **??** for expressions. It looks very similar to whats inside VDMJ and it is almost a direct copy/paste job. The same applies to eval.

```
public static class ExpressionTc extends PExpTypeChecker
  {
  @Override
  public <Typ extends Node> Typ caseABinopExp(ABinopExp source,
      Environment env, NameScope scope, TypeList qualifiers)
  {
```

```
    PType expected = null;

    if(source.getBinop() instanceof APlusBinop || source.getBinop() instanceof AMinusBinop)
    {
      expected = new AIntType();
    }else if( source.getBinop() instanceof ALazyAndBinop ||source.getBinop() instanceof ALazyOrBinop)
    {
      expected = new ABoolType();
    }

    Node ltype = source.getLeft().typeCheck(this.parent(), env, scope, null);
    Node rtype = source.getRight().typeCheck(this.parent(), env, scope, null);

    if (!expected.getClass().isInstance(ltype))
    {
      report(3065, "Left hand of " + source.getBinop() + " is not "
          + expected);
    }

    if (!expected.getClass().isInstance(rtype))
    {
      report(3066, "Right hand of " + source.getBinop() + " is not "
          + expected);
    }
    source.setType((PType) expected);
    return (Typ) expected;
  }
}
```

```
PExp exp = new ABinopExp(new AIntConstExp(new TNumbersLiteral("2")),
new APlusBinop(new TPlus()),
new AIntConstExp(new TNumbersLiteral("5")));
//Type Check we get the type back
PType n = exp.typeCheck(typeChecker, new Environment(), new NameScope(), null);
//Eval we get a value back
IValue n = exp.eval(new Eval(new PBinopEval(), new PUnopEval(), new CustomPExpEval(), new PBooleanEval(), new
      PTypeEval()), null);
```

Generalization

If we take a closer look at the idea behind typecheck and eval from the node class we can see a pattern. This is just a visitor pattern wrapped as an QuestionAnswer visitor, so we could do this generally like:

```
public abstract class Node
{
  /**
   * Returns the answer for {@code answer} by applying this node with the
   * {@code question} to the {@link QuestionAnswer} visitor.
   * @param caller the {@link QuestionAnswer} to which this node is applied
   * @param question the question provided to {@code answer}
   * @return the answer as returned from {@code answer}
   */
  public abstract <Q,A> A apply(QuestionAnswer<Q,A> caller, Q question);
}
```

```
public class ABinopExp extends PExp
{
  /**
  * Calls the {@link QuestionAnswer<Q, A>#caseABinopExp(ABinopExp)} of the {@link QuestionAnswer<Q, A>} {@code
       caller}.
  * @param caller the {@link QuestionAnswer<Q, A>} to which this {@link ABinopExp} node is applied
  * @param question the question provided to {@code caller}
  */
  @Override
  public <Q, A> A apply(QuestionAnswer<Q, A> caller, Q question)
  {
    return caller.caseABinopExp(this, question);
  }
```

```
}
```

This approach allows us to do any analysis in a type safe way like:

And we define our type checker and evaluator as:

```
public class TypeCheckInfo
{
  public Environment env;
  public NameScope scope;
  public TypeList qualifiers;
}

public class TypeCheckVisitor extends QuestionAnswerAdaptor<TypeCheckInfo, PType>
{
  @Override
  public PType caseABinopExp(ABinopExp node, TypeCheckInfo question)
  {
    PType expected = null;

    if (node.getBinop() instanceof APlusBinop
        || node.getBinop() instanceof AMinusBinop)
    {
      expected = new AIntType();
    } else if (node.getBinop() instanceof ALazyAndBinop
        || node.getBinop() instanceof ALazyOrBinop)
    {
      expected = new ABoolType();
    }

    Node ltype = node.getLeft().apply(this,question);
    Node rtype = node.getRight().apply(this,question);

    if (!expected.getClass().isInstance(ltype))
    {
      report(3065, "Left hand of " + node.getBinop() + " is not "
          + expected);
    }

    if (!expected.getClass().isInstance(rtype))
    {
      report(3066, "Right hand of " + node.getBinop() + " is not "
          + expected);
    }
    node.setType((PType) expected);
    return expected;
  }
}
```

and invoke this like

```
PType t = exp.apply(new TypeCheckVisitor(), new TypeCheckInfo());
```

This approach provides enables both a type checker and interpreter to be implemented and other things to be implemented through the same interface on the AST nodes.

1.3.4 Plug-in use and derived trees

There is generally three ways plug-ins might use the AST:

1. Read only. The tree is used as input in a read only style to do some internal processing.

2. Read/Write. The plug-in will use the tree for internal processing and possibly add plug-in specific data to the AST.

3. Read/Write. The plug-in will use the tree and possible rewrite the AST and add new information to it.

The three different usage kinds differ be either leaving the source AST intact possibly adding plug-in specific data or changing the AST in any way. Thus the latter will require a AST clone, however the two first types could be direct access to the AST used in our IDE.

When it comes to "derived trees" it might turn out to be very inefficient to copy and AST all the time especially for the parse/type check phases. Instead a visitor can potentially be implemented to instantiate a new AST-2 from a source AST, but this will add unnecessary complexity and break back tracking to the original AST.

1.3.5 Plug-in access

We can chose to either give access to the internal AST of the IDE or only allow access to a clone which a plug-in can obtain.

```
/**
 * Returns a deep clone of this {@link ARealType} node.
 * @return a deep clone of this {@link ARealType} node
 */
public ARealType clone()
{
  ...
}


/**
**
 * Creates a deep clone of this {@link ARealType} node while putting all
 * old node-new node relations in the map {@code oldToNewMap}.
 * @param oldToNewMap the map filled with the old node-new node relation
 * @return a deep clone of this {@link ARealType} node
 */
public ARealType clone(Map<Node,Node> oldToNewMap)
{
  ...
}
```

1.3.6 Derived trees/custom plug-in data

We should keep it simple. Since the plug-in which exists now only need a feature to provide simple tags on certain nodes a complete AST copy to new identical classes with a number of new fields might be overdoing it. However a simple map in the `Node` class could hold such information. But this does not exclude the other derived AST idea from being implemented later, it will just be an implementation of the source AST visitor.

```
String pluginId ="org.overture.ide.plugins.codegeneration";
String field="javaType";
exp.setCustomField(pluginId, field, String.class);
Class javaClass = exp.getCustomField(pluginId, field,Object.class);
```

1.4 Illustration with a simple AST structure

To be written:

```
Tokens

  plus = '+';
  int = 'int';
  real = 'real';
  bool = 'bool';
  true = 'true';
  false = 'false';
  and_and = '&&';
  or_or = '||';
  numbers_literal = 'some regex for numbers';

Abstract Syntax Tree

binop
    = {plus} [token]:plus
    | {minus} [token]:minus
    | {lazy_and} [token]:and_and
    | {lazy_or} [token]:or_or
    ;

unop
    = {negate} [token]:minus
    ;

exp
    = {binop} [left]:exp binop [right]:exp
    | {unop} unop exp
    | {int_const} numbers_literal
    | {boolean_const} boolean
    | {apply} [root]:exp [args]:exp*
    ;

boolean
    = {true}
    | {false}
    ;

type
    = {real} [token]:real
    | {int} [token]:int
    | {bool} [token]:bool
    ;

Aspect Declaration

exp
    = [type]:type
    ;
```

Chapter 2

How to move forward

2.1 AST Features

copy SL API OO structure - link to VDM PP classes

2.1.1 Structure

All nodes in the AST are linked by their parent. This means that any node knows who its parent is and can only be a child of a single node.

```
/**
 * Sets the parent node of this node.
 * @param parent the new parent node of this node
 */
protected void parent(Node parent) {
  this.parent = parent;
}
```

```
public void removeChild(Node child)
{
  //removes the child and sets the child of this node to null
}
```

```
//sets a new child of this node by first removing the current child from this node then removing the new one from
    its old parent and adding it
public void setLeft(PExp value)
{
  if (this._left != null) {
    this._left.parent(null);
  }
  if (value != null) {
    if (value.parent() != null) {
      value.parent().removeChild(value);
  }
    value.parent(this);
  }
  this._left = value;
}
```

2.1.2 Enumerated node types

All nodes must have return an element from a node enumeration identifying the specific node type. The same applies to derrived nodes which must return an element of an enumeration from the class it inherit from. e.g.:

```
public enum NodeEnum
{
  TOKEN,
  BINOP,
  UNOP,
  EXP,
  BOOLEAN,
  TYPE
}

public abstract class PExp extends Node
{
  public NodeEnum kindNode()
  {
    return NodeEnum.EXP;
  }
}
```

2.1.3 Copying

The AST must include the following clone methods, which either creates a single deep copy of the AST or creates a copy including a mapping from source to copy map.

```
  public @Override abstract Object clone();
  public abstract Node clone(Map<Node,Node> oldToNewMap);
```

2.1.4 Copy-extend

It should be possible to extend an AST with either new fields on nodes or by adding new subclasses the the AST, however this poses a challenge when such an extended tree should be constructed from from a smaller tree. $SourceAST \subset ExtendedAST$. The AST creator must be able to output a visitor which can instantiate any extended tree of its source tree.

```
@Override
public ABinopExpExtended caseABinopExp(ABinopExp node)
{//n.setLeft(node.getLeft().apply(this)) could also be an option
  ABinopExpExtended n = new ABinopExpExtended(
    node.getLeft().apply(this),
    node.getBinop().apply(this),
    node.getLeft().apply(this),
    null .....);
  return n;
}
```

2.1.5 Navigation

Get ancestor by type

Get an ancestor of a specific type from a node if it exists by searching up in the tree.

```
/**
 * Returns the nearest ancestor of this node (including itself)
 * which is a subclass of {@code classType}.
 * @param classType the superclass used
 * @return the nearest ancestor of this node
 */
public <T extends Node> T getAncestor(Class<T> classType) {
  Node n = this;
  while (!classType.isInstance(n)) {
    n = n.parent();
    if (n == null) return null;
  }
  return classType.cast(n);
}
```

Standard visitor

The standard visitor implemented for each node a call to analysis where the method called is specific the the current node.

```
/**
 * Applies this node to the {@link Analysis} visitor {@code analysis}.
 * @param analysis the {@link Analysis} to which this node is applied
 */
public abstract void apply(IAnalysis analysis);
```

Answer visitor

An answer visitor implements a call for each node to the caller with a specific method for this node and allows a value of A to be returned.

```
/**
 * Returns the answer for {@code caller} by applying this node to the
 * {@link Answer} visitor.
 * @param caller the {@link Answer} to which this node is applied
 * @return the answer as returned from {@code caller}
 */
public abstract <A> A apply(IAnswer<A> caller);
```

Question visitor

A question visitor implements a call for each node to the caller with a specific method for this node where a question of type Q can be parsed to the method.

```
/**
 * Applies this node to the {@link Question} visitor {@code caller}.
 * @param caller the {@link Question} to which this node is applied
 * @param question the question provided to {@code caller}
 */
public abstract <Q> void apply(IQuestion<Q> caller, Q question);
```

Question Answer visitor

A question visitor implements a call for each node to the caller with a specific method for this node where a question of type Q can be parsed to the method and a return value of A will be returned.

```
/**
 * Returns the answer for {@code answer} by applying this node with the
 * {@code question} to the {@link QuestionAnswer} visitor.
 * @param caller the {@link QuestionAnswer} to which this node is applied
 * @param question the question provided to {@code answer}
 * @return the answer as returned from {@code answer}
 */
public abstract <Q,A> A apply(IQuestionAnswer<Q,A> caller, Q question);
```

Bibliography