Tom Manual

April 28, 2006

This manual contains information for Tom version 2.3.
This manual also exists in pdf.

Tom is a language extension which adds new matching primitives to languages like C, Java, and Caml. Although rich and complex, Tom is not a stand-alone language: like a preprocessor, it strongly relies on the underlying language (C, Java, or Caml), called *host-language* in the following. To this language, Tom adds several constructs.

The main construct, `%match`, is similar to the match primitive found in functional languages: given an object (called subject) and a list of patterns-actions, the match primitive selects the first pattern that matches the subject and performs the associated action. The subject against which we match can be any object, but in practice, this object is usually a tree-based data-structure, also called *term* in the algebraic programming community.

The match construct may be seen as an extension of the classical switch/case construct. The main difference is that the discrimination occurs on a term and not on atomic values like characters or integers: the patterns are used to discriminate and retrieve informations from an algebraic data structure. Therefore, Tom is a good language for programming by pattern matching, and it is particularly well-suited for programming various transformations on trees/terms or XML data-structures.

Information on Tom is available at the Tom web page tom.loria.fr.

Writing a good documentation is a difficult task. To help new users, as well as confirmed Tom developers, we have split the documentation into four complementary documents:

- the *tutorial* part introduces, using examples, many concepts related to Tom, from simple to complex ones. The goal is to illustrate concepts and ideas introduced by the reference and the user manual.

- the *language* part is a reference manual which describes the constructs, gives their precise syntax and informal semantics. To support the intuition, some examples may be given, but it is by no means a tutorial introduction to the language. This part also introduces the runtime library, which simplifies the use of Tom.

- the *tools* part describes how to use the Tom system in a production environment. This document explains both, how to install the software, and how to use is.

- the *cookbook* is a collection of recipes that we use, and that can be adapted to solve frequently encountered problems.

Contents

Part I

Tutorial

Chapter 1

Language Basics

1.1 Defining a data-structure

One of the most simple Tom program is the following:

```
import main.peano.types.*;

public class Main {
  %gom {
    module Peano
    abstract syntax
    Nat = zero()
        | suc(pred:Nat)
        | plus(x1:Nat, x2:Nat)
  }

  public final static void main(String[] args) {
    Nat z   = `zero();
    Nat one = `suc(z);
    System.out.println(z);
    System.out.println(one);
  }
}
```

The `%gom{ ...}` constructs defines a data-structure, also called algebraic data-type. This data-structure declares a sort (`Nat`) that has three operators (`zero`, `suc`, and `plus`). These operators are called constructors, because they are used to construct the data-structure. `zero` is a constant (arity 0), whereas the arity of `suc` and `plus` is respectively 1 and 2. In Gom, a name has to be given for each slot (`pred` for suc, and `x1`, `x2` for plus).

A second construct provided by Tom is the ' (backquote), which builds an object. The expression `'zero()` builds the object zero() of sort `Nat`. Therefore, the variable `z` is a Java object that can be used to build the object `suc(zero())`. In other words, an algebraic data-structure definition can be seen as a grammar. The ' construct allows to build trees (Abstract Syntax Trees) over this grammar.

The implementation generated by Gom is such that objects can be converted into readable strings, using the `toString()` method.

We assume that Tom is installed, as explained in chapter 9. First we compile the file `Main.t` using the `tom` command, then we compile the generated Java file Main.java using `javac` and we run it with `java`, as shown below:

```
$ tom Main.t
$ javac Main.java
$ java Main
zero
suc(zero)
```

1.2 Retrieving information in a data-structure

In addition to `%gom` and `'`, Tom provides a third construct: `%match`. Using the same program, we add a method `evaluate(Nat n)` and we modify the method `run()`:

```
public static Nat evaluate(Nat n) {
  %match(Nat n) {
    plus(x, zero()) -> { return 'x; }
    plus(x, suc(y)) -> { return 'suc(plus(x,y)); }
  }
  return n;
}

public final static void main(String[] args) {
  Nat two = 'plus(suc(zero()),suc(zero()));
  System.out.println(two);
  two = evaluate(two);
  System.out.println(two);
}
```

The `%match` construct is similar to `switch/case` in the sense that given a tree (generally called subject), the `%match` selects the first pattern that *matches* the tree, and executes the associated action. By *matching*, we mean: finding the instances of variables that make the pattern and the subject identical. In our example, the first pattern matches the subject `n` when `n` is a tree rooted by `plus`, and whose second subterm corresponds to the tree `zero()`. When it is the case, the variable `x` takes the first subterm as value. We say that a tree is *rooted* by a symbol when its top-node corresponds to this symbol. When a pattern matches, we say that its variables are *instantiated*. They can then be used in the action part (also called right-hand side). Similarly, the second rules can be applied when the second subterm is rooted by a `suc`. In that case, `x` and `y` are instantiated.

The method `evaluate` defines the addition of two integers represented by Peano successors (i.e. `zero` and `suc`). The first case says that the addition of an integer with `zero` is evaluated into the integer itself. The second case builds a new term (rooted by `suc`) whose subterm (`plus(x,y)`) denotes the addition of `x` and `y`.

When executing the `run()` method, we obtain:

```
plus(suc(zero),suc(zero))
suc(plus(suc(zero),zero))
```

In a match construct, the variables should not be declared by the programmer. They are written without `()`, and their type is inferred. A variable introduced should be *fresh*: it should not be declared earlier in a wrapping block, either in Java, either in a `%match` construct. A Tom variable can be used in the corresponding action, but has to be wrapped in a `'` construct. Note that the scope of a back-quote expression is delimited by well-parenthesis sentences (`'zero()`, `'suc(zero())`, `'suc(x)`). The special case `'x`, corresponding to a variable alone, is also correct. To write complex expressions, extra parenthesis can be added: `'(x)`, `'(x == y)` for example.

1.3 Combining Tom and Java

As illustrated below, Tom offers a simple way to retrieve information stored in subterms. The previous example describes a single-step when adding two integers. To get a canonical form (a term that can no longer be reduced), we need to encode a loop, to reach a fix-point. We also have to allow reductions in sub-trees (i.e. a `plus` which is not at a top-position):

```
public static Nat evaluate(Nat n) {
  %match(Nat n) {
    plus(x, zero()) -> { return 'x; }
    plus(x, suc(y)) -> { return 'suc(plus(x,y)); }
    // congruence rules
```

```
        suc(x)            -> { return 'suc(evaluate(x)); }
        plus(x,y)         -> { return 'plus(evaluate(x),evaluate(y)); }
      }
      return n;
    }

    public final static void main(String[] args) {
      Nat t = 'plus(suc(zero()),suc(zero()));
      System.out.println(t + " -> " + evaluate(t));
      t = 'plus(suc(zero()),plus(suc(zero()),suc(zero())));
      System.out.println(t + " -> " + evaluate(t));
    }
```

By adding two new congruence rules, a term that cannot be reduced by the first two rule will be recursively inspected by the two last rules. Therefore, if a transformation can be performed (at top position, or in sub-trees), it will be done:

```
plus(suc(zero),suc(zero)) -> suc(plus(suc(zero),zero))
plus(suc(zero),plus(suc(zero),suc(zero))) -> plus(suc(zero),suc(plus(suc(zero),zero)))
```

We use Java (while-loop or recursion) to compute a fix-point:

```
    public static Nat loop(Nat n) {
      Nat tmp = 'evaluate(n);
      if(tmp==n) {
        return n;
      }
      return loop(tmp);
    }

    public final static void main(String[] args) {
      Nat t = 'plus(suc(zero()),suc(zero()));
      System.out.println(t + " -> " + loop(t));
      t = 'plus(suc(zero()),plus(suc(zero()),suc(zero())));
      System.out.println(t + " -> " + loop(t));
    }
```

And we get the expected results:

```
plus(suc(zero),suc(zero)) -> suc(suc(zero))
plus(suc(zero),plus(suc(zero),suc(zero))) -> suc(suc(suc(zero)))
```

1.4   Programming in Tom

When using Gom, the generated data-structure is particular: data are maximally shared. This technique, also known as *hash-consing* ensures that two identical subterms are implemented by the same objects. Therefore, the memory footprint is minimal and the equality-check can be done is constant time: two terms are identical if the pointers are equal.

As a consequence, a term (implemented using Gom) is not mutable: once created, it cannot be modified. One can note that the generated API offers a `setx1(Nat t)` and a `setx2(Nat t)` method. These methods can be used to "modify" the first or the second child or a term rooted by a `plus`. Note that the term `t` given in argument is *never* modified: a copy where the first or the second child is modified, is return instead.

By combining Tom, Java, and Gom, it is quite easy to quickly develop applications which manipulates trees. As an example, let us consider a tiny language composed of boolean expressions (`true`, `false`, `and`, `or`, and `not`), expressions (`constant`, `variable`, `plus`, `mult`, and `mod`), and instructions (`skip`, `print`, `;`, `if`, and `while`). The abstract syntax of this language can be described using Gom:

```
import pico1.term.types.*;
import java.util.*;

class Pico1 {
  %gom {
    module Term
    imports int String
    abstract syntax
    Bool = True()
         | False()
         | Not(b:Bool)
         | Or(b1:Bool, b2:Bool)
         | And(b1:Bool, b2:Bool)
         | Eq(e1:Expr, e2:Expr)

    Expr = Var(name:String)
         | Cst(val:int)
         | Plus(e1:Expr, e2:Expr)
         | Mult(e1:Expr, e2:Expr)
         | Mod(e1:Expr, e2:Expr)

    Inst = Skip()
         | Assign(name:String, e:Expr)
         | Seq(i1:Inst, i2:Inst)
         | If(cond:Bool, i1:Inst, i2:Inst)
         | While(cond:Bool, i:Inst)
         | Print(e:Expr)
  }
  ...
}
```

Assume that we want to write an interpreter for this language, we need a notion of *environment* to store the value assigned to a variable. A simple solution is to use a `Map` which associate an expression (of sort `Expr`) to a name of variable (of sort `String`). Given this environment, the evaluation of an expression can be implemented as follows:

```
public static Expr evalExpr(Map env,Expr expr) {
  %match(Expr expr) {
    Var(n)                 -> { return (Expr)env.get('n); }
    Plus(Cst(v1),Cst(v2))  -> { return 'Cst(v1 + v2); }
    Mult(Cst(v1),Cst(v2))  -> { return 'Cst(v1 * v2); }
    Mod(Cst(v1),Cst(v2))   -> { return 'Cst(v1 % v2); }
    // congruence rules
    Plus(e1,e2) -> {
      return 'evalExpr(env,Plus(evalExpr(env,e1),evalExpr(env,e2)));
    }
    Mult(e1,e2) -> {
      return 'evalExpr(env,Mult(evalExpr(env,e1),evalExpr(env,e2)));
    }
    Mod(e1,e2)  -> {
      return 'evalExpr(env,Mod(evalExpr(env,e1),evalExpr(env,e2)));
    }

    x -> { return 'x; }
  }
  throw new RuntimeException("strange term: " + expr);
}
```

Similarly, the evaluation of boolean expressions can be implemented as follows:

```
public static Bool evalBool(Map env,Bool bool) {
  %match(Bool bool) {
    Not(True())      -> { return `False(); }
    Not(False())     -> { return `True(); }
    Not(b)           -> { return `evalBool(env,Not(evalBool(env,b))); }

    Or(True(),b2)    -> { return `True(); }
    Or(b1,True())    -> { return `True(); }
    Or(False(),b2)   -> { return `b2; }
    Or(b1,False())   -> { return `b1; }
    And(True(),b2)   -> { return `b2; }
    And(b1,True())   -> { return `b1; }
    And(False(),b2)  -> { return `False(); }
    And(b1,False())  -> { return `False(); }

    Eq(e1,e2) -> {
      Expr x=`evalExpr(env,e1);
      Expr y=`evalExpr(env,e2);
      return (x==y)?`True():`False();
    }

    x -> { return `x; }
  }
  throw new RuntimeException("strange term: " + bool);
}
```

Once defined the methods `evalExpr` and `evalBool`, it becomes easy to define the interpreter:

```
public static void eval(Map env, Inst inst) {
  %match(Inst inst) {
    Skip() -> {
      return;
    }

    Assign(name,e) -> {
      env.put(`name,evalExpr(env,`e));
      return;
    }

    Seq(i1,i2) -> {
      eval(env,`i1);
      eval(env,`i2);
      return;
    }

    Print(e) -> {
      System.out.println(evalExpr(env,`e));
      return;
    }

    If(b,i1,i2) -> {
      if(evalBool(env,`b)==`True()) {
        eval(env,`i1);
      } else {
        eval(env,`i2);
```

```
      }
      return;
    }

    w@While(b,i) -> {
      Bool cond = evalBool(env,'b);
      if(cond=='True()) {
        eval(env,'i);
        eval(env,'w);
      }
      return;
    }
  }
  throw new RuntimeException("strange term: " + inst);
}
```

Note: the last two rules use the conditional test `if` from `Java` to implement a conditional rule. Note also that terms are compared using the `==` operator, thanks to maximal sharing provided by Gom. To play with the `Pico` language, we just have to initialize the environment (`env`), create programs (`p1` and `p3`), and evaluate them (`eval`):

```
public final static void main(String[] args) {
  Map env = new HashMap();

  Inst p1 = 'Seq(Assign("a",Cst(1)) , Print(Var("a")));
  System.out.println("p1: " + p1);
  eval(env,p1);

  Inst p3 = 'Seq(Assign("i",Cst(0)),
              While(Not(Eq(Var("i"),Cst(10))),
                    Seq(Print(Var("i")),
                        Assign("i",Plus(Var("i"),Cst(1))))));
  System.out.println("p3: " + p3);
  eval(env,p3);
}
```

Exercise: write a Pico program that computes prime numbers up to 100.

Chapter 2

Advanced features

2.1 The "Hello World" example

One of the most simple Tom program is the following:

```
public class HelloWorld {
  %include { string.tom }

  public final static void main(String[] args) {
    String who = "World";
    %match(String who) {
      "World" -> { System.out.println("Hello " + who); }
      _        -> { System.out.println("Don't panic"); }
    }
  }
}
```

The `%include { string.tom }` construct imports the predefined (`Java`) mapping which defines the Tom `String` algebraic sort. Thus, `String` can be used in the `%match(String t)` construct.

This examples shows that pattern matching can be performed against built-in data-types. In this example, the `%match` construct is equivalent to a `switch/case` construct. '`_`' denotes an anonymous variable that corresponds to a default case in this example.

2.1.1 "Hello World" revisited

One particularity of Tom is to support list-matching (also known as associative-matching).

When considering that a string is a list of characters, it becomes natural to describe pattern-matching using an associative operator which corresponds to the concatenation of two characters. Thus, the string ``Hello'' is equivalent to the list of characters ('H','e','l','l','o').

Given a string, to check if the string contains the character 'e', we can use the following matching construct:

```
  %match(String t) {
    (_*,'e',_*) -> { System.out.println("we have found a 'e'"); }
  }
```

In this example, '`_*`' corresponds to an anonymous variable which can be instantiated by any list of characters (possibly reduced to the empty list).

In order to capture the context, it is possible to use a named variable:

```
  %match(String t) {
    (before*,'e',after*) -> {
      System.out.println("we have found a 'e'" +
                          " after " + `before* +
```

```
                    " but before " + ‘after*);
   }
 }
```

In this example, we have introduced two new variables (`before*` and `after*`), called "variable-star", which are instantiated by a list of characters. This is '`*`' that denoted a list variable.

To use their value in a `Java` expression, Tom provides the "`‘`" (backquote) mechanism. This construct can be use to retrieve the value of a Tom variable (instantiated by pattern-matching). As illustrated by the previous examples, ‘`before*` and ‘`after*` are replaced by expressions of sort `String`.

Suppose now that we look for a word whose last letter is a 'o'. A possible Tom pattern is:

```
%match(String t) {
  (before*,'o') -> { ... }
}
```

Using this mechanism, it also becomes possible to look for a word which contains an 'e' somewhere and an 'o' in last position:

```
%match(String t) {
  (before*,'e',_*,'o') -> { ... }
}
```

Note that some pattern could provide several outcomes:

```
%match(String t) {
  (before*,'l',after*) -> {
    System.out.println("we have found " + ‘before* +
                       " before 'l' and " + ‘after* + " after");
  }
}
```

It will match twice:

```
we have found he before 'l' and lo after
we have found hel before 'l' and o after
```

Let us suppose that we look for two consecutive 'l' anywhere in the matched expression. This could be expressed by:

```
%match(String t) {
  (_*,'l','l',_*) -> { ... }
}
```

Since this syntax is error prone when looking for long/complex substrings, Tom provides an abbreviated notation: '`ll`':

```
%match(String t) {
  (_*,'ll',_*) -> { ... }
}
```

This notation is fully equivalent to the previous one.

2.1.2 "Hello World" re-revisited

As illustrated previously, we can use variables to capture contexts. In fact, this mechanism is more general and we can use a variable anywhere to match something which is not statically know. Thus, it is possible to use variable (without a star) to explicitly say that a character should be there:

```
%match(String t) {
  (x,_*,'ll',_*,y) -> { ... }
}
```

This example matches the string `t` if this string contains two consecutive '`l`' and at least two other characters. In this case, the first letter of the string is stored in the variable '`x`' (respectively the last letter in '`y`').

The variable '`x`' and '`y`' are local to the given pattern. Thus, when considering several patterns in a same match construct, there is no interference between patterns. However, in a same pattern, we can use the same variable two times to indicate to we want to match two identical elements. This feature is known as non-linear matching:

```
%match(String t) {
  (x,_*,'ll',_*,y) -> { ... }
  (x,y,y,x)        -> { /* we have found a palindrome */ }
}
```

When looking for several instances of a same character, it may be interesting to use a variable to describe this repetition:

```
%match(String t) {
  (_*,'a',_*,'a',_*,'a',_*) -> { /* look for 3 'a' in a string */ }
  (_*,x@'a',_*,x,_*,x,_*)   -> { /* look for 3 'a' in a string */ }
}
```

Here, the first '`a`' is bound to '`x`', and then, all remaining occurrences of '`x`' are compared to this first instance. '`x`' is an alias for ''`a`''. **Note:** in the previous section, we have said that (`_*,'ll',_*`) is equivalent to (`_*,'l','l',_*`). This is still true, but the reader should also note that (`_*,x@'ab',_*`) becomes (after expansion) equivalent to (`_*,x@'a',x@'b',_*`), which can never match any string.

2.2 Associative matching

As illustrated previously, Tom supports a generalized form of string matching, called *associative matching*, also known as *list matching*. In Tom, some operators are special and do not have a fixed number of arguments. This intuitively corresponds to the idea of list where elements are stored in a given order. A list of elements may be written (`a,b,c`) or [`a,b,c`]. In Tom, it is written `f(a(),b(),c())`, where '`f`' is this special operator. In some sense, '`f`' is the name of the list. This allows to distinguish list of apples, from list of oranges for examples.

The definition of such operators can be done using Gom:

```
import list1.list.types.*;
public class List1 {
  %gom {
    module List
    abstract syntax
    E = a()
      | b()
      | c()
    L = f( E* )
  }
  ...
}
```

Once this signature defined, it becomes possible to define patterns and ' expressions. For example, the function that removes identical consecutive elements can be expressed as follows:

```
public static L removeDouble(L l) {
  %match(L l) {
    f(X1*,x,x,X2*) -> {
      return removeDouble('f(X1*,x,X2*));
    }
  }
  return l;
}
```

This example is interesting since it express a complex operation in a concise way: given a list `l`, the `%match` construct looks for two identical consecutive elements (`f(X1*,x,x,X2*)` is called a non-linear pattern since `x` appears twice). If there exists such two elements, the term `f(X1*,x,X2*)` is built (one `x` has been removed), and the `removeDouble` function is called recursively. When there is no such two elements, the pattern does not match, this means that the list does not contain any consecutive identical elements. Therefore, the instruction `return l` is executed and the function returns.

Similarly, a sorting algorithm can be implemented: if a list contains two elements in the wrong order, just swap them. This can be expressed as follows:

```
public static L swapSort(L l) {
  %match(L l) {
    f(X*,e1,Z*,e2,Y*) -> {
      if('gt(e1,e2)) {
        return 'swapSort(f(X*,e2,Z*,e1,Y*));
      }
    }
  }
  return l;
}


private static boolean gt(E e1, E e2) {
  return e1.toString().compareTo(e2.toString()) > 0;
}
```

In this example, the order is implemented by the `gt` function, using the lexicographical ordering provided by `Java`. **Note:** given a list `l = f(b(),b(),a(),a())`, there exists several ways to match the pattern. In this case, there are 6 possibilities:

```
1. X*=f(),        e1=b(), Z*=f(),        e2=b(), Y*=f(a(),a())
2. X*=f(),        e1=b(), Z*=f(b()),     e2=a(), Y*=f(a())
3. X*=f(),        e1=b(), Z*=f(b(),a()), e2=a(), Y*=f()
4. X*=f(b()),     e1=b(), Z*=f(),        e2=a(), Y*=f(a)
5. X*=f(b()),     e1=b(), Z*=f(a()),     e2=a(), Y*=f()
6. X*=f(b(),b()), e1=a,   Z*=f(),        e2=a(), Y*=f()
```

Assuming that `a < b`, there are only 4 solutions that can be used to apply the rule: 2, 3, 4, and 5 (`e1` must be greater than `e2`).

It becomes important to remind you how a rule is applied in Tom:

1. a rule whose pattern match is selected (in our case, there is only one rule),

2. then, for each solution of the matching problem (there are 6 solutions in our case), the right part is executed (i.e. the `Java` code).

   Let us suppose that Tom computes the solution `1.` first (the order in which solutions are computed is not fixed and depends on the implementation). In that case, the test `if('gt(e1,e2))` is evaluated. Since `b()` is not greater than `b()`, the `return swapSort(...)` is not performed.

   As specified in Section 5.2.2, another solution is computed. Let us say `2.`. In that case, the test becomes true and the function `swapSort` is called recursively.

3. when there is no more solution (i.e. this means that the list is already sorted), the control is transfered to the next pattern. Since there is no more pattern in our example, the `return l` is executed, which returns the sorted list.

The following code shows how to build a list and call the function defined previously. This results in sorted lists where elements only occur once.

```
public final static void main(String[] args) {
  L l = 'f(a(),b(),c(),a(),b(),c(),a());
  L res1 = swapSort(l);
```

```
    L res2 = removeDouble(res1);
    System.out.println(" l       = " + l);
    System.out.println("sorted l = " + res1);
    System.out.println("single l = " + res2);
  }
```

2.3   Another example using Gom

On one side Tom offers customized pattern matching facilities to support the development of XML and
string based applications. On the other side, Tom is a rule based engine which allows to solve complex
pattern matching problems over algebraic data-types.

In order to describe user defined abstract data-types, Tom provides a signature definition mechanism
(`%typeterm`, `%typelist`, `%op`, *etc.* c.f. section 2.4). However, Tom does not provide any support to
implement these abstract data-types. This is why, in addition to Tom, we have developed the system
Gom. This tool provides a human readable syntax definition formalism (inspired from SDF) as well as
an efficient implementation for the data-type. The implementation generated by Gom is based on the
`SharedObject` library (which is the code of the ATerm library), using maximal sharing in an efficient
way.

In the following, we consider a simple abstract data-type which defines three sorts (`Date`, `Person`,
and `PersonList`), as well as three constructors (`date`, `person`, and `concPerson`). In this signature, `Int`
and `String` are predefined builtin sorts:

```
%gom {
  module person
  imports int String
  abstract syntax
  Date = date(day:int, month:int, year:int)
  Person = person(firstname:String, lastname:String, birthdate:Date)
  PersonList = concPerson(Person*)
}
```

The notation `concPerson(Person*)` means that `concPerson` is a variadic associative operator which
takes a list of `Person` in arguments, and returns a `PersonList`.

By using the `%gom{ ... }` construct, a Tom signature and an implementation are automatically
generated. In order to connect these generated files, it is only necessary to import the corresponding
packages:

```
import addressbookgom.person.types.*;

public class AddressBookGom {
  %gom { ... }

  public final static void main(String[] args) {
    ...
  }
  ...
}
```

Once importation and definition are performed, it becomes very easy to use Tom: the constructors can
be immediately used in a backquote or a matching constructs.

2.3.1   Using the backquote mechanism

In this section, we consider a tiny address-book application: given a date and a list of persons, the
problem consists in finding out the persons whose birthday corresponds to the date.

The first sub-problem consists in building a list of persons. For this, we use the backquote notation
and the `concPerson` operator:

```
public static PersonList generateBook() {
  return `concPerson(
    person("John","Smith",date(1965,3,27)),
    person("Marie","Muller",date(1986,3,26)),
    person("Paul","Muller",date(2000,1,27))
  );
}
```

2.3.2 Using list-matching facilities

The `main(String[] args)` method generates a list of persons, builds a date object and calls the `happyBirthday()` method whose goal is to determine persons whose birthday corresponds to the date:

```
public final static void main(String[] args) {
  PersonList book = generateBook();
  Date today = `date(2003,3,27);
  happyBirthday(book,today);
}
```

The following method is interesting because it uses pattern matching to encode a searching procedure. As illustrated below, the use of associative operator, combined with non-linearity, allows us to solve the problem in a nice and concise way:

```
public static void happyBirthday(PersonList book, Date date) {
  %match(PersonList book, Date date) {
    concPerson(_*,person(firstname,_,date(year,month,day)),_*),date(_,month,day) -> {
      System.out.println("Happy birthday " + `firstname);
    }
  }
}
```

Let us mention that the `println` statement is executed for all persons whose birthdate corresponds to the given date. It also interesting to note that all instantiations of `firstname`, `year`, `month` and `day` are tried before exiting the `%match` construct.

2.4 Hand-written mappings

Up to now, we have considered examples where the implementation of data-types is either predefined (XML case), or generated by a tool (Gom case). One interesting contribution of Tom is to be customizable. By defining a mapping between the signature formalism (`%typeterm`, `%op`, *etc.*) and the concrete implementation, it becomes possible to perform pattern matching against any data-structure.

2.4.1 Defining a simple mapping

In this section, we consider that we have a `Java` library for representing tree based data structures. On another side, to be able to use Tom, we consider the following abstract data-type:

```
%typeterm Nat
%op Nat zero()
%op Nat suc(Nat)
```

By doing so, we have defined a sort (`Nat`), and two operators (`zero` and `suc`). When defining a mapping, the goal consists in explaining to Tom how the considered abstract data-type is implemented, and how a term (over this signature) can be de-constructed.

For expository reasons, the ATerm library is the `Java` library used to implement terms. However, any other tree/term based library could have been used instead.

In order to define a correct mapping, we have to describe how the algebraic sort (`Nat` in our example) is implemented (by the `ATermAppl` class). This is done via the `implement` construct:

```
%typeterm Nat {
  implement { ATermAppl }
}
```

The second part of the mapping definition consists in defining how the symbols (`zero` and `suc`) are represented in memory. This is done via the `is_fsym` construct:

```
%op Nat zero() {
  is_fsym(t) { t.getName().equals("zero") }
}

%op Nat suc(pred:Nat) {
  is_fsym(t)       { t.getName().equals("suc") }
  get_slot(pred,t) { (ATermAppl)t.getArgument(0) }
}
```

In addition, `get_slot` describes how to retrieve a subterm of a term.

Given a term built over the ATerm library, it becomes possible to perform pattern matching as previously explained.

2.4.2   Using backquote constructs

When using the backquote construct (`'suc(suc(zero))` for example), Tom has to know how to build a term in memory. For this purpose, we consider an extension of the signature definition formalism: the `make` construct.

```
%op Nat zero() {
  is_fsym(t) { t.getName().equals("zero") }
  make       { SingletonFactory.getInstance().makeAppl(
                 SingletonFactory.getInstance().makeAFun("zero",0,false)) }
}

%op Nat suc(pred:Nat) {
  is_fsym(t)       { t.getName().equals("suc") }
  get_slot(pred,t) { (ATermAppl)t.getArgument(0) }
  make(t)          { SingletonFactory.getInstance().makeAppl(
                       SingletonFactory.getInstance().makeAFun("suc",1,false),t) }
}
```

The `makeAFun` function has three arguments: the function name, the number of arguments and a boolean that indicates if the quotes are included in the function name or not.

Given this mapping, Tom can be used as previously: the following function implements the addition of two Peano integers:

```
public ATermAppl plus(ATermAppl t1, ATermAppl t2) {
  %match(Nat t1, Nat t2) {
    x,zero() -> { return 'x; }
    x,suc(y) -> { return 'suc(plus(x,y)); }
  }
  return null;
}
```

Please, note the use of `ATermAppl` in Java code, and the use of `Nat` in Tom construct.

2.4.3   Advanced examples

Let us suppose that we have a Java class `Person` with 2 getters (`getFirstname` and `getLastname`). In order to illustrate the signature definition formalism, we try to redefine (without using Gom) the abstract data type for the sort `Person`.

The first thing to do consists in defining the Tom sort `Person`:

```
%typeterm Person {
  implement { Person }
}
```

To avoid any confusion, we use the same name twice: the Tom sort `Person` is implemented by the `Java` class `Person`. When declaring an operator, we defined the behavior as shown in the previous example:

```
%op Person person(firstname:String, lastname:String) {
  is_fsym(t) { t instanceof Person }
  get_slot(firstname,t) { t.getFirstname() }
  get_slot(lastname,t) { t.getLastname() }
  make(t0, t1) {  new Person(t0, t1) }
}
```

In this example, we illustrate another possibility offered by Tom: being able to know whether a term is rooted by a symbol without explicitly representing this symbol. The is_fsym(t) construct should return `true` when the term `t` is rooted by the algebraic operator we are currently defining. In this example, we say that the term `t` is rooted by the symbol `person` when the object `t` is implemented by an instance of the class `Person`. By doing so, we do not explicitly represent the symbol `person`, even if it could have been done via the reflective capabilities of `Java` (by using `Person.getClass()` for example).

2.4.4   Using list-matching

In this section, we show how to describe a mapping for associative operators.

```
%typeterm TomList {
  implement { ArrayList }
  equals(l1,l2)     { l1.equals(l2) }
}
```

Assuming that the sort `Element` is already defined, we can use the `%oparray` construct to define an associative operator. We also have to explain to Tom how to compute the size of a list, and how to access a given element. This is done via get_size and get_element constructs:

```
%oparray TomList conc( Element* ) {
  is_fsym(t)         { t instanceof ArrayList }
  get_element(l,n) { (ATermAppl)l.get(n) }
  get_size(l)        { l.size() }
  make_empty(n)      { myEmpty(n) }
  make_append(e,l) { myAdd(e,(ArrayList)l) }
}
```

This construct is similar to `%op` except that additional information have to be given: how to build an empty list (make_empty), and how to add an element to a given list (make_append). The auxiliary `Java` functions are defined as follows:

```
private static ArrayList myAdd(Object e,ArrayList l) {
  l.add(e);
  return l;
}

private static ArrayList myEmpty(int n) {
  ArrayList res = new ArrayList(n);
  return res;
}
```

Usually, we use an associative operator to represent (in a abstract way) a list data structure. There are many ways to implement a list, but the two most well-known are the use of array based list, and the use of linked-list. The previously described mapping shows how to map an abstract list to an array based implementation. Tom offers another similar construct `%oplist` to map an associative operator to a linked-list based implementation.

Chapter 3

Strategies

3.1 Introduction to strategies

3.1.1 Elementary strategy

There exists three kinds of elementary strategy: `Fail`, which always fails, `Identity`, which always succeeds, and transformation rules of the form $l \to r$. Therefore, if we consider the elementary strategy a $\Rightarrow$ b (which replaces a by b), we have the following results:

```
(a -> b)[a]      = b
(a -> b)[b]      = failure
(a -> b)[f(a)]   = failure
(Identity)[a]    = a
(Identity)[b]    = b
(Identity)[f(a)] = f(a)
(Fail)[a]        = failure
```

3.1.2 Basic combinators

Composition

The sequential operator, `Sequence(S1,S2)`, applies the strategy `S1`, and then the strategy `S2`. It fails if either `S1` fails, or `S2` fails.

```
(Sequence(a -> b, b -> c))[a] = c
(Sequence(a -> b, c -> d))[a] = failure
(Sequence(b -> c, a -> b))[a] = failure
```

Choice

The choice operator, `Choice(S1,S2)`, applies the strategy `S1`. If the application `S1` fails, it applies the strategy `S2`. Therefore, `Choice(S1,S2)` fails if both `S1` and `S2` fail.

```
(Choice(a -> b, b -> c))[a]   = b
(Choice(b -> c, a -> b))[a]   = b
(Choice(b -> c, c -> d))[a]   = failure
(Choice(b -> c, Identity))[a] = a
```

Not

The strategy `Not(S)`, applies the strategy and fails when `S` succeeds. Otherwise, it succeeds and corresponds to the `Identity`.

```
(Not(a -> b))[a]   = failure
(Not(b -> c))[a]   = a
```

3.1.3   Parameterized strategies

By combining basic combinators, more complex strategies can be defined. To make the definitions generic, parameters can be used. For example, we can define the two following strategies:

- `Try(S) = Choice(S, Identity)`, which tries to apply `S`, but never fails

- `Repeat(S) = Try(S, Repeat(S))`, which applies recursively `S` until it fails, and then returns the last unfailing result

```
(Try(b -> c))[a]                = a
(Repeat(a -> b))[a]             = b
(Repeat(Choice(b -> c, a -> b)))[a] = c
(Repeat(b -> c))[a]             = a
```

3.1.4   Traversal strategies

We consider two kinds of traversal strategy (`All(S)` and `One(S)`). The first one applies `S` to all subterms, whereas the second one applies `S` to only one subterm.

All

The application of the strategy `All(S)` to a term `t` applies `S` on each immediate subterm of `t`. The strategy `All(S)` fails if `S` fails on at least one immediate subterm.

```
(All(a -> b))[f(a)]        = f(b)
(All(a -> b))[g(a,a)]      = g(b,b)
(All(a -> b))[g(a,b)]      = failure
(All(a -> b))[a]           = a
(All(Try(a -> b)))[g(a,c)] = g(b,c)
```

Note: the application of `All(S)` to a constant *never* fails: it returns the constant itself.

One

The application of the strategy `One(S)` to a term `t` tries to apply `S` on an immediate subterm of `t`. The strategy `One(S)` succeeds if there is a subterm such that `S` can be applied. The subterms are tried from left to right.

```
(One(a -> b))[f(a)]   = f(b)
(One(a -> b))[g(a,a)] = g(a,b)
(One(a -> b))[g(b,a)] = g(b,b)
(One(a -> b))[a]      = failure
```

Note: the application of `One(S)` to a constant *always* fails: there is no subterm such that `S` can be applied.

3.1.5   High level strategies

By combining the previously mentioned constructs, it becomes possible to define well know strategies:

```
BottomUp(S)     = Sequence(All(BottomUp(S)), S)
TopDown(S)      = Sequence(S, All(TopDown(S)))
OnceBottomUp(S) = Choice(One(OnceBottomUp(S)), S)
OnceTopDown(S)  = Choice(S, One(OnceTopDown(S)))
Innermost(S)    = Repeat(OnceBottomUp(S))
Outermost(S)    = Repeat(OnceTopDown(S))
```

3.2 Strategies in practice

Let us consider again a Pico language whose syntax is a bit simpler than the one seen in section 1.4.

```
import pico2.term.types.*;
import java.util.*;
import jjtraveler.VisitFailure;
import jjtraveler.reflective.VisitableVisitor;

class Pico2 {
  %include { mutraveler.tom }

  %gom {
    module Term
    imports int String

    abstract syntax
    Bool = True()
         | False()
         | Neg(b:Bool)
         | Or(b1:Bool, b2:Bool)
         | And(b1:Bool, b2:Bool)
         | Eq(e1:Expr, e2:Expr)

    Expr = Var(name:String)
         | Cst(val:int)
         | Let(name:String, e:Expr, body:Expr)
         | Seq( Expr* )
         | If(cond:Bool, e1:Expr, e2:Expr)
         | Print(e:Expr)
         | Plus(e1:Expr, e2:Expr)
  }
  ...
}
```

As an exercise, we want to write an optimization function that replaces an instruction of the form `If(Neg(b),i1,i2)` by a simpler one: `If(b,i2,i1)`. A possible implementation is:

```
public static Expr opti(Expr expr) {
  %match(Expr expr) {
    If(Neg(b),i1,i2) -> { return `opti(If(b,i2,i1)); }
    x -> { return `x; }
  }
  throw new RuntimeException("strange term: " + expr);
}

public final static void main(String[] args) {
  Expr p4 = `Let("i",Cst(0),
                If(Neg(Eq(Var("i"),Cst(10))),
                  Seq(Print(Var("i")), Let("i",Plus(Var("i"),Cst(1)),Var("i"))),
                  Seq()));

  System.out.println("p4      = " + p4);
  System.out.println("opti(p4) = " + opti(p4));
}
```

When executing this program, we obtain:

```
p4        = Let("i",Cst(0),If(Neg(Eq(Var("i"),Cst(10))),
            ConsSeq(Print(Var("i")),ConsSeq(Let("i",
            Plus(Var("i"),Cst(1)),Var("i")),EmptySeq)),EmptySeq))
opti(p4) = Let("i",Cst(0),If(Neg(Eq(Var("i"),Cst(10))),
            ConsSeq(Print(Var("i")),ConsSeq(Let("i",
            Plus(Var("i"),Cst(1)),Var("i")),EmptySeq)),EmptySeq))
```

This does not correspond to the expected result, simply because the `opti` function performs an optimization when the expression starts with an `If` instruction. To get the expected behavior, we have to add congruence rules that will allow to apply the rule in subterms (one rule for each constructor):

```
public static Expr opti(Expr expr) {
  %match(Expr expr) {
    If(Neg(b),i1,i2) -> { return `opti(If(b,i2,i1)); }
    // congruence rules
    Let(n,e1,e2)     -> { return `Let(n,opti(e1),opti(e2)); }
    Seq(head,tail*)  -> { return `Seq(opti(head),opti(tail*)); }
    If(b,i1,i2)      -> { return `If(b,opti(i1),opti(i2)); }
    Print(e)         -> { return `Print(e); }
    Plus(e1,e2)      -> { return `Plus(e1,e2); }
    x -> { return `x; }
  }
  throw new RuntimeException("strange term: " + expr);
}
```

Since this is not very convenient, we will show how the use of strategies can simplify this task.

3.2.1 Printing constants using a strategy

Let us start with a very simple task which consists in printing all the nodes that corresponds to a constant (`Cst(_)`). To do that, we have to define an elementary strategy that is successful when it is applied on a node `Cst(_)`:

```
%strategy stratPrintCst() extends `Fail() {
  visit Expr {
    Cst(x) -> { System.out.println("cst: " + `x); }
  }
}
```

Note: this strategy extends `Fail`. This means that its application leads to a failure when it cannot be applied. In our case, the strategy succeeds on nodes of the form `Cst(x)`, and fails on all the others. To traverse the program and print all `Cst` nodes, a `TopDown` strategy can be applied:

```
public static void printCst(Expr expr) {
  try {
    `TopDown(Try(stratPrintCst())).visit(expr);
  } catch (VisitFailure e) {
    System.out.println("strategy failed");
  }
}

public final static void main(String[] args) {
  ...
  System.out.println("p4 = " + p4);
  printCst(p4);
}
```

Note: the strategy given as argument of a `TopDown` should not fail. Otherwise, the `TopDown` will also fail. This is why the `stratPrint` is wrapped into a `Try`, which makes the strategy always successful. This results in:

```
p4 = Let("i",Cst(0),If(Neg(Eq(Var("i"),Cst(10))),
     ConsSeq(Print(Var("i")),ConsSeq(Let("i",
     Plus(Var("i"),Cst(1)),Var("i")),EmptySeq)),EmptySeq))
cst: 0
cst: 10
cst: 1
```

3.2.2   Combining elementary strategies

As a second exercise, we will try to write another strategy that performs the same task, but we will try to separate the strategy that looks for a constant from the strategy that prints a node. So, let us define these two strategies:

```
%strategy FindCst() extends 'Fail() {
  visit Expr {
    c@Cst(x) -> { return 'c; }
  }
}

%strategy PrintTree() extends 'Identity() {
  visit Expr {
    x -> { System.out.println('x); }
  }
}
```

Similarly to `stratPrintCst`, the strategy `FindCst` extends `Fail`. The goal of the `PrintTree` strategy is to print a node of sort `Expr`. By extending `Identity`, we specify the default behavior when the strategy is applied on a term of a different sort. **Note:** we could have extended `Fail` and used `Try(PrintTree())` instead. To print the node `Cst`, we have to look for a `Cst` and print this node. This can be done by combining, using a `Sequence`, the two strategies `FindCst` and `PrintTree`:

```
public static void printCst(Expr expr) {
  try {
    'TopDown(Try(stratPrintCst())).visit(expr);
    'TopDown(Try(Sequence(FindCst(),PrintTree()))).visit(expr);
  } catch (VisitFailure e) {
    System.out.println("strategy failed");
  }
}
```

This results in:

```
cst: 0
cst: 10
cst: 1
Cst(0)
Cst(10)
Cst(1)
```

Note: in the second case, the nodes are printed using the `toString()` method generated by Gom.

3.2.3   Modifying a subterm

Here, we will try to rename all the variables from a given program: the name should be modified into _name.

To achieve this task, you can define a primitive strategy that performs the modification, and apply it using a strategy such as `TopDown`:

```
%strategy stratRenameVar() extends `Fail() {
  visit Expr {
    Var(name) -> { return `Var("_"+name); }
  }
}
```

```
public static void optimize(Expr expr) {
  try {
    `Sequence(TopDown(Try(stratRenameVar())),PrintTree()).visit(expr);
  } catch (VisitFailure e) {
    System.out.println("strategy failed");
  }
}
```

Note: to print the resulting term, the `TopDown` application of `stratRenameVar` (wrapped by a `Try`) is combined, using a `Sequence`, with the strategy `PrintTree`. The application of `optimize` to `p4` results in:

```
Let("i",Cst(0),If(Neg(Eq(Var("_i"),Cst(10))),
  ConsSeq(Print(Var("_i")),ConsSeq(Let("i",
  Plus(Var("_i"),Cst(1)),Var("_i")),EmptySeq)),EmptySeq))
```

Suppose now that we want to print the intermediate steps: we do not want to perform all the replacements in one step, but for debugging purpose, we want to print the intermediate term after each application of the renaming rule.

The solution consists in combining the `stratRenameVar` strategy with the `PrintTree` strategy. **Note:** you can try to implement it yourself before reading the solution A first solution consists in applying `stratRenameVar` using a `OnceBottomUp` strategy, and immediately apply `PrintTree` on the resulting term. This could be implemented as follows:

```
`Repeat(Sequence(OnceBottomUp(stratRenameVar()),PrintTree())).visit(expr);
```

Unfortunately, this results in:

```
Let("i",Cst(0),If(Neg(Eq(Var("_i"),Cst(10))),...
Let("i",Cst(0),If(Neg(Eq(Var("__i"),Cst(10))),...
Let("i",Cst(0),If(Neg(Eq(Var("___i"),Cst(10))),...
Let("i",Cst(0),If(Neg(Eq(Var("____i"),Cst(10))),...
Let("i",Cst(0),If(Neg(Eq(Var("_____i"),Cst(10))),...
Let("i",Cst(0),If(Neg(Eq(Var("_____i"),Cst(10))),...
Let("i",Cst(0),If(Neg(Eq(Var("_____i"),Cst(10))),...
Let("i",Cst(0),If(Neg(Eq(Var("_____i"),Cst(10))),...
Let("i",Cst(0),If(Neg(Eq(Var("_____i"),Cst(10))),...
Let("i",Cst(0),If(Neg(Eq(Var("_____i"),Cst(10))),...
Let("i",Cst(0),If(Neg(Eq(Var("_____i"),Cst(10))),...
Let("i",Cst(0),If(Neg(Eq(Var("_____i"),Cst(10))),...
...
```

This is not the expected behavior! Why?

Simply because the renaming rule can be applied several times on a same variable. To fix this problem, we have to apply the renaming rule only if the considered variable has not already be renamed.

To know if a variable has been renamed, you just have to define an elementary strategy, called `RenamedVar`, that succeeds when the name of the variable starts with an underscore. This can be easily implemented using string matching capabilities:

```
%strategy RenamedVar() extends `Fail() {
  visit Expr {
    v@Var(('_',name*)) -> { return `v; }
  }
}
```

To finish our implementation, it is sufficient to apply `stratRenameVar` only when `RenamedVar` fails, i.e., when `Not(RenamedVar)` succeeds.

```
'Repeat(Sequence(
     OnceBottomUp(Sequence(Not(RenamedVar()),stratRenameVar())),
     PrintTree())
   ).visit(expr);
```

This results in (layouts have been added to improve readability):

```
Let("i",Cst(0),If(Neg(Eq(Var("_i"),Cst(10))),
  ConsSeq(Print(Var("i")),ConsSeq(Let("i",
  Plus(Var("i"),Cst(1)),Var("i")),EmptySeq)),EmptySeq))
Let("i",Cst(0),If(Neg(Eq(Var("_i"),Cst(10))),
  ConsSeq(Print(Var("_i")),ConsSeq(Let("i",
  Plus(Var("i"),Cst(1)),Var("i")),EmptySeq)),EmptySeq))
Let("i",Cst(0),If(Neg(Eq(Var("_i"),Cst(10))),
  ConsSeq(Print(Var("_i")),ConsSeq(Let("i",
  Plus(Var("_i"),Cst(1)),Var("i")),EmptySeq)),EmptySeq))
Let("i",Cst(0),If(Neg(Eq(Var("_i"),Cst(10))),
  ConsSeq(Print(Var("_i")),ConsSeq(Let("i",
  Plus(Var("_i"),Cst(1)),Var("_i")),EmptySeq)),EmptySeq))
```

3.2.4   Re-implementing the tiny optimizer

Now that you know how to use strategies, it should be easy to implement the tiny optimizer seen in the beginning of section 3.2.

You just have to define the transformation rule and a strategy that will apply the rule in an innermost way:

```
%strategy OptIf() extends 'Fail() {
  visit Expr {
    If(Neg(b),i1,i2) -> { return 'If(b,i2,i1); }
  }
}

public void optimize(Expr expr) {
  try {
    'Sequence(Innermost(OptIf()),PrintTree()).visit(expr);
  } catch (VisitFailure e) {
    System.out.println("strategy failed");
  }
}
```

Applied to the program `p4`, as expected this results in:

```
Let("i",Cst(0),If(Eq(Var("i"),Cst(10)),EmptySeq,
  ConsSeq(Print(Var("i")),ConsSeq(Let("i",
  Plus(Var("i"),Cst(1)),Var("i")),EmptySeq))))
```

Note: when programming with strategies, it is no longer necessary to implement a congruence rule for each constructor of the signature.

3.3   Advanced usage

This section is not written yet. We just mention the titles that will be present in the next release.

3.3.1   Definition of recursive strategies

3.3.2   Definition of parameterized strategies

3.3.3   Getting a position

3.3.4   Getting or replacing a subterm

3.3.5   Probabilistic strategies

3.3.6   Programming non deterministic transition systems

See Section 16.2 in the CookBook for an example of such system implemented using strategies.

Chapter 4

XML

Even if this section mostly deals with XML features, every explained technique can be used with other data-structures as well.

Note: this section is not very up-to-date. In particular, the explanations about `genericTraversal` functions should be replaced by some strategies.

4.1   Manipulating Xml documents

This example is inspired from a practical study. It involves a simple modeling of the Platform for Privacy Preferences Project (P3P), developed by the World Wide Web Consortium, which is emerging as an industry standard providing a simple, automated way for users to gain more control over the use of personal information on Web sites they visit.

Given a client and a server, the problem consists in verifying that the client's policy is compliant with the server's policy. Both policies preferences are expressed in APPEL (A P3P Preference Exchange Language) and written in XML. The server's policy is the following file `server.xml`:

```
<POLICIES xmlns="http://www.w3.org/2002/01/P3Pv1">
  <POLICY name="mypolicy" discuri="http://www.ibm.com/privacy"
          opturi="http://www.ibm.com/privacy" xml:lang="en">
    <STATEMENT>
      <RECIPIENT> <delivery/> </RECIPIENT>
      <PURPOSE> <contact/> </PURPOSE>
      <DATA-GROUP>
        <DATA ref="#dynamic.clickstream"/>
        <DATA ref="#dynamic.http"/>
        <DATA ref="#dynamic.clientevents"/>
        <DATA ref="#user.home-info.postal.country"/>
        <DATA ref="#dynamic.cookies"> <CATEGORIES> <content/> </CATEGORIES> </DATA>
      </DATA-GROUP>
    </STATEMENT>
  </POLICY>
</POLICIES>
```

The client's policy is the following file `client.xml`:

```
<RULESET appel="http://www.w3.org/2002/04/APPELv1"
         p3p="http://www.w3.org/2000/12/P3Pv1"
         crtdby="W3C" crtdon="2001-02-19T16:21:21+01:00">
  ...
  <RULE behavior="limited1" prompt="yes">
    <POLICY>
      <STATEMENT>
        <PURPOSE connective="and-exact"> <current/> </PURPOSE>
```

```
        <RECIPIENT connective="or">
          <other-recipient/>
          <public/>
          <unrelated/>
        </RECIPIENT>
      </STATEMENT>
    </POLICY>
  </RULE>
  ...
  <RULE behavior="request" prompt="yes">
    <POLICY>
      <STATEMENT>
        <DATA-GROUP>
          <DATA ref="#dynamic.clientevents"> </DATA>
          <DATA ref="#dynamic.clickstream"/>
        </DATA-GROUP>
      </STATEMENT>
    </POLICY>
  </RULE>
  ...
</RULESET>
```

For expository reasons, we consider only a sub-problem in which we say that a client's policy is compatible with the server's policy if all `ref` attributes from a <DATA></DATA> node also appear on the server side. In the considered examples, the policies are compatible because

```
<DATA-GROUP>
  <DATA ref="#dynamic.clientevents"> </DATA>
  <DATA ref="#dynamic.clickstream"/>
</DATA-GROUP>
```

is included in:

```
<DATA-GROUP>
  <DATA ref="#dynamic.clickstream"/>
  <DATA ref="#dynamic.http"/>
  <DATA ref="#dynamic.clientevents"/>
  <DATA ref="#user.home-info.postal.country"/>
  <DATA ref="#dynamic.cookies"> <CATEGORIES> <content/> </CATEGORIES> </DATA>
</DATA-GROUP>
```

The problem consists in implementing such a verification tool in Tom and Java.

4.1.1 Loading Xml documents

The first part of the program declares imports, and defines the `main` and the `run` methods.

```
import tom.library.xml.*;
import tom.library.adt.tnode.*;
import tom.library.adt.tnode.types.*;
import aterm.*;
import java.util.*;

public class Evaluator {

  %include{ adt/tnode/TNode.tom }

  private XmlTools xtools;
```

```
  public static void main (String args[]) {
    xtools = new XmlTools();
    TNode server = (TNode)xtools.convertXMLToATerm("server.xml");
    TNode client = (TNode)xtools.convertXMLToATerm("client.xml");
    boolean compatible = compareDataGroup(getDataGroup(server.getDocElem()),
                                          getDataGroup(client.getDocElem()));
    System.out.println("result = " + compatible);
  }
  ...
}
```

As explained in Part II (Tom language description), to each XML document corresponds a DOM (Document Object Model) representation. In Tom, we have defined a mapping from DOM sorts to abstract algebraic sorts: `TNode` and `TNodeList`, which correspond respectively to `Node` and `NodeList`, defined by the `Java` DOM implementation.

This mapping has to be initialized in the following way:

- the import of `tom.library.adt.tnode.*` and `tom.library.adt.tnode.types.*` defines the sorts `TNode` and `TNodeList` and allows us to build objects over these two sorts.

- the `%include{ TNode.tom }` Tom construct is similar to the `#include` C preprocessor construct. In this case, it imports the definition of Tom algebraic constructors needed to manipulate XML documents.

In complement to Tom, we provide a runtime library which contains several methods to manipulate XML documents. In particular, we provide the `XmlTools` class (defined in the `tom.library.xml.*` package).

The `run` method takes two filenames as arguments, reads the associated files and convert their XML content into `TNode` terms (using the `convertXMLToATerm` function).

The `getDataGroup` function is used to retrieve a `<DATA></DATA>` node in the considered XML document. Note that `getDocElem` is first applied to XML documents in order to consider subterms of the `DocumentNode` element. Then, the `compareDataGroup` function is used to check that the client's policy is compatible with the server's policy. The content of these functions is detailed in the next sections.

4.1.2   Retrieving information

Given an XML subtree, the problem consists in extracting a `<DATA-GROUP></DATA-GROUP>` node. For expository reasons, we first consider that such a `<DATA-GROUP></DATA-GROUP>` node can only appear at two different specific places: one for the client's definition and one for the server's definition.

Thus, in the Tom program we naturally consider two cases:

```
private static TNode getDataGroup(TNode doc) {
  %match(TNode doc) {
    <POLICIES>
      <POLICY>
        <STATEMENT>
          datagroup@<DATA-GROUP></DATA-GROUP>
        </STATEMENT>
      </POLICY>
    </POLICIES> -> { return datagroup; }

    <RULESET>
      <RULE>
        <POLICY>
          <STATEMENT>
            datagroup@<DATA-GROUP></DATA-GROUP>
          </STATEMENT>
        </POLICY>
```

```
        </RULE>
      </RULESET> -> { return datagroup; }
    }

    return `xml(<DATA-GROUP/>);
  }
```

The first pattern means that a `<DATA-GROUP></DATA-GROUP>` node has to be found under a `<STATEMENT></STATEMENT>` node, which should be under a `<POLICY></POLICY>` node, which should be under a `<POLICIES></POLICIES>` node. Once such a pattern is found, the mechanism allows us to give a name (`datagroup`) to this subterm and reuse it in the action part: `return datagroup;`.

The XML notation implicitly extends the given patterns by adding context variables. Thus, the `<DATA-GROUP></DATA-GROUP>` pattern means that a subterm those head symbol is `<DATA-GROUP></DATA-GROUP>` is searched. But this subterm can contain attributes and children even if it is not explicitly defined by the notation. To make the definition of patterns more precise, the user can use the explicit notation and define the following pattern: `<DATA-GROUP  (_*)>(_*)</DATA-GROUP>`. A more detailed explanation can be found in Part II (Tom language description).

4.1.3   Comparing two Xml subtrees

Given two `<DATA-GROUP></DATA-GROUP>` nodes, the problem consists in checking that all `<DATA></DATA>` nodes from the first tree also appear in the second one. This can be easily implemented by the following couple of functions:

```
  private static boolean compareDataGroup(TNode server, TNode client) {
    boolean res = true;
    %match(TNode client) {
      <DATA-GROUP><DATA ref=reftext></DATA></DATA-GROUP>
      -> { res = res && appearsIn(reftext,server); }
    }
    return res;
  }
```

Given a `<DATA-GROUP></DATA-GROUP>` tree, we look for a `<DATA></DATA>` subtree which contains an attribute named `ref`. When such an attribute exists, its content (a string) is stored into the Java `reftext` variable. The `appearsIn` function is then used to check that the attribute also appears on the server side.

```
  private static boolean appearsIn(String refclient, TNode server) {
    %match(TNode server) {
      <DATA-GROUP><DATA ref=reftext></DATA></DATA-GROUP>
      -> {
        if(reftext.equals(refclient)) {
          return true;
        }
      }
    }
    return false;
  }
```

This piece of code is interesting because it introduces what we call "conditional rules". Given the string `refclient` we look for a `<DATA></DATA>` subtree containing a `ref` attribute with exactly the same string. Since it is not possible to use an instantiated variable in a pattern (something like `<DATA ref=refclient>...</DATA>`, we have to introduce a fresh variable `reftext` and check in a condition that this variable is equal to `refclient`. This is done via a Java condition (the `if` clause): the action part (`return true;`) is executed only when the condition is satisfied, and clearly it will end the method. If the condition is not satisfied, then the next `<DATA></DATA>` subtree is searched. It is exactly the case of a switch statement, when the action part is not exited by a `return`, `break` or `goto`, and the control flow is transferred to the next matching solution or the next matching pattern.

If no such subtree exists, this means that the two policies are not compatible and `false` is returned.

4.1.4   Retrieving information using traversal functions

The Tom runtime library provides a set of generic functions in order to perform various kinds of traversal over a term. This is useful when searching a pattern somewhere in a term, at any position. As the XML documents are seen by Tom as terms, the different traversals on terms are also valid when dealing with XML.

In the current example, the `getDataGroup` functions looks for a <DATA-GROUP></DATA-GROUP> node. Instead of statically specifying the path (POLICIES, POLICY, STATEMENT, DATA-GROUP and RULESET, RULE, POLICY, STATEMENT, DATA-GROUP) we could have used the generic traversal mechanism provided by the package `tom.library.traversal.*`.

```
private static GenericTraversal traversal = new GenericTraversal();
```

In the following, we generalize the previous problem by considering that we can have more than one appearance of the <DATA-GROUP></DATA-GROUP> node per XML document.

We further expose the use of the generic traversal mechanism. The `collectDatagroup` method creates an inner class `Collect1` with one method ,`apply`, and then it calls the `genericCollect` method of the traversal object created previously with an object of this inner class as a parameter.

```
protected static void collectDatagroup(final Collection collection, TNode subject) {
  Collect1 collect = new Collect1() {
      public boolean apply(ATerm t) {
        if(t instanceof TNode) {
          %match(TNode t) {
            <DATA-GROUP> </DATA-GROUP> -> {
              collection.add(t);
              return false;
            }
          }
        }
        return true;
      } // end apply
    }; // end new

  traversal.genericCollect(subject, collect);
}
```

Let us now analyze the inner class: firstly, it extends the class `Collect1`, where `1` stands for the number of arguments of the `apply` method, and secondly it implements the method `apply`. The method call `traversal.genericCollect` will call this method for all subtrees of the `subject` term. Since `subject` may contain subterms of different sorts (integers or strings for example), the `instanceof` construct is used as a first filter to select only the subterms which correspond to XML nodes. When such a subterm is rooted by a <DATA-GROUP></DATA-GROUP>, the subterm is added to the collection (by a side-effect). The `return false;` statement indicates that it is no longer necessary to traverse the considered term (in our example, a <DATA-GROUP></DATA-GROUP> node cannot be found inside a <DATA-GROUP></DATA-GROUP> itself). Respectively, the `return true;` statement indicates that the `apply` function should be recursively applied to the current term in order to continue the traversal.

Given a collection, the `getDataGroup` method uses an iterator to select a <DATA-GROUP></DATA-GROUP> node.

```
private static TNode getDataGroup(TNode doc) {
  HashSet c = new HashSet();
  collectDatagroup(c,doc);
  Iterator it = c.iterator();
  while(it.hasNext()) {
    TNode datagroup = (TNode)it.next();
```

```
      return datagroup;
    }

    return `xml(<DATA-GROUP/>);
  }
```

4.2   Building and sorting Xml/DOM documents

In this section, we consider a DOM mapping for XML documents. This means that XML documents are still considered as algebraic terms (built over `TNode` and `TNodeList`), but their internal representation is backed by the W3C DOM library.

In the following, we consider a small data-base represented by an XML document `person.xml`:

```
<Persons>
  <Person Age="30"> <FirstName> Paul    </FirstName> </Person>
  <Person Age="28"> <FirstName> Mark    </FirstName> </Person>
  <Person Age="21"> <FirstName> Jurgen </FirstName> </Person>
  <Person Age="21"> <FirstName> Julien </FirstName> </Person>
  <Person Age="24"> <FirstName> Pierre-Etienne </FirstName> </Person>
</Persons>
```

The problem consists in sorting this document according to different criteria (age or first-name for example).

4.2.1   Loading Xml documents

The first part of the program declares imports, and defines the `main` method.

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.File;

public class PersonSort1 {
  private static Document dom;
  %include{ dom.tom }

  public static void main (String args[]) {
    try {
      dom = DocumentBuilderFactory.newInstance()
        .newDocumentBuilder().parse("person.xml");
      Element e = dom.getDocumentElement();
      dom.replaceChild(sort(e),e);

      Transformer transform = TransformerFactory.newInstance().newTransformer();
      StreamResult result = new StreamResult(new File("Sorted.xml"));
      transform.transform(new DOMSource(dom), result);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  ...
}
```

The mapping has to be initialized in the following ways:

- the import of `org.w3c.dom.*` and `javax.xml.*` packages in order to use the DOM library.

- the `%include{ dom.tom }` construct imports the definition of Tom algebraic constructors needed to manipulate DOM objects.

- the `dom` variable has to be instantiated. This variable corresponds to the notion of Document in the DOM terminology.

4.2.2 Comparing two nodes

In order to implement a sorting algorithm, we need to know how to compare two elements.

The following function takes two XML documents in argument and compares their attribute `Age`, using the string comparison operator `compareTo`.

```
private static int compare(Node t1, Node t2) {
  %match(TNode t1, TNode t2) {
    <Person Age=a1></Person>, <Person Age=a2></Person> -> {
      return 'a1.compareTo('a2);
    }
  }
  return 0;
}
```

In this example, it is interesting to note that an XML node is seen as an associative operator. This feature, combined with the implicit notation, is such that <Person Age=a1></Person> will match any XML node headed by `Person` which contains the attribute `Age`. This XML node may contain several sub-nodes, but they will not be stored in any variable.

4.2.3 Sorting a list of nodes

In this section we use a swap sort algorithm which consists in finding two elements that are not in the correct order. Once they are found, they are swapped before calling the algorithm recursively. When no two such elements exist, this means that the list is sorted.

```
private static Node sort(Node subject) {
  %match(TNode subject) {
   <Persons>(X1*,p1,X2*,p2,X3*)</Persons> -> {
      if(compare('p1,'p2) > 0) {
        return sort('xml(dom,<Persons>X1* p2 X2* p1 X3*</Persons>));
      }
    }
  }
  return subject;
}
```

The pattern <Persons>(X1*,p1,X2*,p2,X3*)</Persons> looks for two elements `p1` and `p2` and stores the remaining contexts in `X1*`, `X2*`, and `X3*`. In order to give a name to contexts (and then retrieve them to build a new list), the pattern is written in explicit notation. This notation prevents Tom to add extra variables. Otherwise, the expanded form of <Persons>X1* p1 X2* p2 X3*</Persons> (written in explicit notation) would have been <Persons>(_*,X1*,_*,p1,_*,X2*,_*,p2,_*,X3*,_*)</Persons>.

Note that the action part is guarded by a condition (`if(compare(p1,p2) > 0)`). This means that `return sort(...)` is executed only when two bad-ordered elements are found. When `p1` and `p2` are correctly ordered, the matching procedure continues and extracts another couple `p1` and `p2`. As mentioned in Part II, when Tom cannot find two elements `p1` and `p2` such that the condition is satisfied (the list is sorted), the `return sort(...)` statement is not executed and the control flow is transferred to the following statement: `return subject;`

As mentioned previously, when two elements have to be swapped, a new list is built. This done via the ''' (backquote) construct. This construct, used before to retrieve instantiated variables, can also be use to build a new term. In order to build an XML document, the `xml(...)` function has to be used. The number of parameters depends on the XML mapping which is used. In our case, when manipulating DOM objects, a new subtree is built is the context of a main document. This extra-parameter is given in the first argument of the `xml` function. Thus, '`xml(dom,<Persons>...<Persons>`) builds a new `<Person></Person>` node, as a child of the `dom` document. When using the `TNode.tom` mapping, a library based on Apigen is used. In this case, it is no longer necessary (and even not correct) to use this extra argument.

4.2.4 Sorting by side effect

In the previous section, we considered a sorting algorithm expressed in a pure functional programming style: when two elements have to be swapped, a new list is built and returned.

In the following example, we exploit OO-style of the DOM library and perform a swap in place: the list is updated by swapping two elements (with side effect):

```
private static void sort(Node subject) {
  %match(TNode subject) {
    r @ <_>p1@<_ Age=a1></_> p2@<_ Age=a2></_></_> -> {
      if('a1.compareTo('a2) > 0) {
        'r.replaceChild('p2.cloneNode(true),'p1);
        'r.replaceChild('p1,'p2);
        'sort(r);
        return;
      }
    }
  }
}
```

In this example, we use the notion of anonymous XML node (`<_>...</_>`) which allows to describe more generic algorithms.

Part II

Language

Chapter 5

Tom

5.1 Notations

The syntax of the language is given in BNF-like notation. Terminal symbols are set in typewriter font (‘`like this`’). Non-terminal symbols are set in italic font (like that). Square brackets [...] denote optional components. Parentheses with a trailing star sign (...)* denotes zero, one or several repetitions of the enclosed components. Parentheses with a trailing plus sign (...)+ denote one or several repetitions of the enclosed components. Parentheses (...) denote grouping.

5.1.1 Lexical conventions

Identifier	::=	Letter (Letter | Digit | ‘_’ | ‘-’)*
Integer	::=	(Digit)+
Double	::=	(Digit)+ [‘.’] (Digit)* | ‘.’ (Digit)+
String	::=	‘"’ (Letter | (‘\’ (‘n’ | ‘t’ | ‘b’ | ‘r’ | ‘f’ | ‘\’ | ‘’’ | ‘"’)))* ‘"’
Letter	::=	‘A’ ... ‘Z’ | ‘a’ ... ‘z’
Digit	::=	‘0’ ... ‘9’
Char	::=	‘’’ (Letter |Digit) ‘’’

5.1.2 Names

SubjectName	::=	Identifier
Type	::=	Identifier
SlotName	::=	Identifier
HeadSymbol	::=	Identifier
	|	Integer
	|	Double
	|	String
	|	Char
VariableName	::=	Identifier
AnnotedName	::=	Identifier
LabelName	::=	Identifier
FileName	::=	Identifier
AttributeName	::=	Identifier
XMLName	::=	Identifier
Name	::=	Identifier

5.2 Tom constructs

A Tom program is a host language program (namely C, Java, or Caml) extended by several new constructs such as `%match`, `%strategy`, `%rule`, `%include`, `%gom`, or `backquote`. Tom is a multi-languages compiler, so, its syntax depends on the host language syntax. But for simplicity, we only present the syntax of its constructs and explain how they can be integrated into the host language.

Using Java as *host-language*, the following Tom program is correct:

```
public class HelloWorld {
  %include { string.tom }

  public final static void main(String[] args) {
    String who = "World";
    %match(String who) {
      "World" -> { System.out.println("Hello " + who); }
      _        -> { System.out.println("Don't panic"); }
    }
  }
}
```

5.2.1 Tom program

Basically, a Tom program is list of blocks, where each block is either a Tom construct, or a sequence of characters. The idea is that after transformation, the sequence of characters merged with the compiled Tom constructs should be a valid host language program. In the previous example, `%include` and `%match` constructs are replaced by function definitions and Java instructions, making the resulting program a correct Java program.

Syntax of a Tom program:

Tom	::=	BlockList
BlockList	::=	
	(	MatchConstruct
	|	StrategyConstruct
	|	RuleConstruct
	|	BackQuoteTerm
	|	IncludeConstruct
	|	GomConstruct
	|	TypeTerm
	|	Operator
	|	OperatorList
	|	OperatorArray
	|	'{' BlockList '}'
	)*	

- **MatchConstruct** is translated into a list of instructions. This construct may appear anywhere a list of instructions is valid in the host language.

- **StrategyConstruct** is translated into a class definition. Since it is translated into a class, this construct is valid only for Java.

- **RuleConstruct** is translated into a function definition. This construct may appear anywhere a function declaration is valid in the host language.

- **BackQuoteTerm** is translated into a function call.

- **IncludeConstruct** is replaced by the content of the file referenced by the construct. Tom looks for include files in: `./packageName/`, `$TOM_HOME/share/jtom/` and `<path>`, where `<path>` is specified by option: `--import <path>`. If the file contains some Tom constructs, they are expanded.

- **Gom** allows to define a Gom grammar. This construct is replaced by the content of the generated mapping. See the Chapter 6 for more details.

- **TypeTerm**, as well as **Operator**, **OperatorList**, and **OperatorArray** are replaced by some functions definitions.

5.2.2   Match construct

The %match construct (MatchConstruct) is one of the main contributions of Tom. This construct can be seen as an extension of the SwitchCase construct in C or Java, except that patterns are no longer restricted to constants (chars or integers). Given an object (the subject) and a list of patterns, our goal is to find the first pattern that *matches* the subjects (i.e. that have a *compatible shape*). More formally, a pattern is a term built over variables and constructors. The latter ones describe the *shape* of the pattern, whereas the former ones are *holes* that can be instantiated to capture a value. When we consider the term $f(a(), g(b()))$, this has to be viewed as a tree based data-structure where $a()$ is the first child of the root, labeled by f. Similarly, $b()$ is the unique child of g, which is the second child of the root. We say that the pattern $f(x, y)$ matches this term (called subject), because we can give values to x and y such that the pattern and the subject become equal: we just have to assign $a()$ to x and $g(b())$ to y. Finding this assignment is called matching and instantiating. This is exactly what Tom is supposed to do. A pattern may of course contains subterms. Therefore, $f(x, g(b()))$ or $f(a(), g(y))$ are valid patterns which match against the subject.

Assuming that s is a Java variable of sort T, referencing a term (the tree based object $f(a(), g(b()))$ for example), the following Tom construct is valid:

```
%match(T s) {
  f(a(),g(y)) -> { /* code that uses y */ }
  f(x,g(b())) -> { /* code that uses x */ }
  f(x,y)      -> { /* code that uses x and y */ }
}
```

A MatchConstruct is composed of two parts:

- a list of host language variables (called *subjects*). These variables should reference the objects to be matched.

- a list of PatternAction: this is a list of pairs (pattern,action), where an action is a set of host language instructions which is executed each time a pattern matches the subjects.

The construct is defined as follow:

MatchConstruct	::=	'%match' '(' MatchArguments ')' '{' (PatternAction)* '}'
MatchArguments	::=	Type SubjectName (',' Type SubjectName)*
PatternAction	::=	[LabelName':'] TermList ('|' TermList)* '->' '{' BlockList '}'
TermList	::=	Term (',' Term)*

For expository reasons, we consider that a %match construct is evaluated in the following way:

- given a list of host language variables (they corresponds to objects only composed of constructors, called ground terms), the execution control is transferred to the first PatternAction whose patterns match the list of ground terms.

- given such a PatternAction, the list of variables is instantiated and the associated semantic action is executed. The instantiated variable are bound in the underlying host-language, and thus can be used in the action part.

- if the execution control is transferred outside the %match instruction (by a goto, break or return for example), the matching process is finished. Otherwise, the execution control is transferred to the next PatternAction whose patterns match the list of ground terms.

- when there is no more PatternAction whose patterns match the list of subjects, the %match instruction is finished, and the execution control is transferred to the next instruction.

The semantics of a match constructs may remind the switch/case construct, however, a big difference exists. In Tom, a theory like associativity may be attached to a constructor. Thus, given a subject and a pattern, there may exist several ways to match the pattern against the subject. Informally, when considering the subject $conc(a(), g(b()), c(), g(d()))$, and the pattern $conc(\_^*, g(x), \_^*)$, there are

two possible match for this problem: either $x = b$, either $x = d()$. Note that $\_^*$ is a special *hole* which can capture any sublist of $conc(\ldots)$.

When taking this new possibility into account, the evaluation of a %match construct is a bit more complex:

- given a PatternAction whose patterns match the list of ground terms, the list of variables is instantiated and the associated semantic action is executed.

- if the execution control is not transferred outside the %match instruction, in addition to the previous explanations, if the considered matching theory may return several matches, for each match, the free variables are instantiated and the associated semantic action is executed. This means that a same action may be executed several times, but in a different context: i.e. the variable have different instantiations.

- when all matches have been computed (there is at most one match in the syntactic theory), the execution control is transferred to the next PatternAction whose patterns match the list of ground terms.

- as before, when there is no more PatternAction whose patterns match the list of subject, the %match instruction is finished, and the execution control is transferred to the next instruction.

As mentioned in the BNF-syntax, a *label* may be attached to a pattern. In that case, in C and Java, it becomes possible to exit from the current PatternAction (using a goto or a break), without exiting from the whole %match construct. The control is transferred to the next pattern which matches the subjects. This feature is useful to exit from a complex associative-matching pattern which, for any reason, is no longer interesting.

Note: the behavior is not determined if a semantic action modifies a host language variable which is an argument of a %match instruction under evaluation.

Note: the ''|'' symbol can be used to introduce disjunction of pattern. This is a shortcut which avoid duplicating a PatternAction. However, this construct is **deprecated**. We recommend to use the *disjunction of symbols* construct, presented in the next section.

5.2.3 Tom pattern

As we can imagine, the behavior of a %match construct strongly depends on the patterns which are involved. The formalism which defines the syntax of a pattern is also an essential component of Tom. Unfortunately, its formal definition is complex, simply because there exist several ways to define patterns which have equivalent behaviors. On the other hand, the different shortcuts may help the programmer to simplify the definitions of patterns.

Informally, a pattern is a term that can be either a variable or an anonymous variable (x or _ for example). A pattern can also be composed of constructors (a(), f(a()), g(a(),x), or h(a(),_,x) for example). When a pattern matches a subject, it may be useful to keep a reference to a matched subterm. The annotation mechanism (z@g(y) for example) can be used for this purpose. Thus, considering the pattern $f(x, z@g(y))$ and the subject $f(a(), g(h(b())))$, y is instantiated by $h(b())$, and z is instantiated by $g(h(b()))$. This can be useful in C, to free the memory for example.

When identical actions have to be performed for a set of patterns which share a common structure, the *disjunction of symbols* may be used: pattern $(f \mid g)(a())$ is equivalent to the set $\{f(a()), g(a())\}$. The disjunction of symbols may also be used in subterm, like in $h((f \mid g)(x))$.

More formally, a Tom term has the following syntax:

Term	::=	[AnnotedName '@'] PlainTerm
PlainTerm	::=	VariableName ['*']
	|	HeadSymbolList [ExplicitTermList | ImplicitPairList]
	|	ExplicitTermList
	|	'_'
	|	'_*'
	|	XMLTerm
HeadSymbolList	::=	HeadSymbol
	|	'(' HeadSymbol ('|' HeadSymbol)+ ')'
ExplicitTermList	::=	'(' Term (',' Term)* ')'
ImplicitPairList	::=	'[' PairTerm (',' PairTerm)* ']'
PairTerm	::=	'[' SlotName '=' Term ']'

A pattern is a term which could contain variables. When matching a pattern against a subject (a ground term), these variables are instantiated by the matching procedure (generated by Tom). In Tom, the variables do not have to be declared: their type is inferred automatically, depending on the context in which they appear.

As described previously, Tom offers several mechanisms to simplify the definition of a pattern:

- standard notation: a pattern can be defined using a classical prefix term notation. To make a distinction between variables and constants, the latter have to be written with explicit parentheses, like `x()`. In this case, the corresponding Tom operator (`%op x()`) should have been declared. When omitting parentheses, like `x`, this denotes a variable.

- unnamed variable: the `_` notation denotes an anonymous variable. It can be used everywhere a variable name can be used. It is useful when the instance of the variable does not need to be used. Similarly, the `_*` notation can be used to denote an anonymous list-variable. This last notation can improve the efficiency of list-matching because the instances of anonymous list-variables do not need to be built.

- annoted variable: the `@` operator allows to give a variable name to a subterm. In `f(x@g(_))` for example, `x` is a variable that will be instantiated by the instance of the subterm `g(_)`. The variable `x` can then be used as any other variable.

- implicit notation: as explained below, the `%op` operator forces to give name to arguments. Assuming that the operator `f` has two arguments, named `arg1` and `arg2`, then we can write the pattern `f[arg1=a()]` which is equivalent to `f(a(),_)`. This notation can be interesting when using constructors with many subterms.

- unnamed list operator: it is often the case that given a list-sort, only one list-operator is defined. It this case, when there is no ambiguity, the name of the operator can be omitted. Considering the `conc` list-operator for example (see `%oplist` and `%oparray` below), to improve the readability, the pattern `conc(_*,x,_*)` can be written `(_*,x,_*)`. This feature is particularly useful in the XML notation introduced in the following.

- symbol disjunction notation: to factorize the definition of pattern which have common subterms, it is possible to describe a family of patterns using a disjunction of symbols. The pattern `(f|g)(a(),b())` corresponds to the disjunction `f(a(),b())` or `g(a(),b())`. To be allowed in a disjunction (in standard notation), the constructors should have the same signature (arity, domain and codomain).

  In practice, it is usually better to use the disjunction notation with the implicit notation Tom offers: `((f|g)[arg1=a()])`. In that case, the signatures of symbols do not have to be identical: only involved slots have to be common (same names and types). Thus, the pattern `(f|g)[arg1=a()]` is correct, even if `g` has more slots than `f`: it only has to have the slot `arg1`, with the same sort.

  Note: the disjunction of symbol can also be used in XML notation: `<A|B>...</A|B>`.

5.2.4 Rule construct

In Tom, we can also define a set of rewrite rules. All the left-hand sides should begin with the same root symbol:

RuleConstruct	::=	'%rule' '{' (Rule)* '}'
Rule	::=	RuleBody (RuleCondition)*
RuleBody	::=	Term '->' Term
RuleCondition	::=	'where' Term ':=' Term
	|	'if' Term '==' Term

The `%rule` construct is composed of a list of conditional rewrite rules (the left-hand side is a term and the right-hand side is a term). All these rules (enclosed into a `%rule { ... }` construct) should begin with the same root symbol. The Tom compiler will generate a function (with a number of arguments equals to the arity of this root symbol) whose name corresponds to the name of this unique root symbol. Given a ground term, applying this function returns the instantiated and normalized right-hand side of the first rule (from top to bottom) whose pattern matches the considered subject and whose conditions are satisfied. When no rule can be applied (i.e. no pattern matches the subject, or no condition is satisfied), the given ground term, rooted by the root symbol of the rewrite system is returned.

In Tom, we consider two kinds of conditions:

- an equality condition ($t_1 == t_2$) is a pair of ground terms that belong to the same type. The condition is satisfied when the normal forms of the two terms (t_1 and t_2) are equal modulo the `equals` predicate defined in the definition of the type associated to t_1 and t_2.

- a matching condition ($p := t$) is a pair of terms where p may contain free variables. The condition is satisfied if the pattern p can be matched against the normal form of t. In this case, the free variables of p are instantiated and can be used in other conditions or the right-hand side of the rule.

5.2.5 Backquote construct

Another construct of Tom is the backquote ('').

This construct can be used to build an algebraic term or to retrieve the value of a Tom variable (a variable instantiated by pattern-matching). The syntax of this operator is not fixed since it depends on the underlying language.

However, a backquote term should be of the following form:

- 'Name: to denote a Tom variable

- 'Name'*': to denote a Tom list-variable

- 'Name(...): to build a prefix term

- '(...): to build an expression

- 'xml(...): to build an XML term

In general, it is sufficient to add a backquote before the term you want to build to have the wanted behavior. The execution of `'f(g(a()))` will build the term $f(g(a))$, assuming that f, g, and a are Tom operators. Suppose now that g is no longer a constructor but a function of the host-language. The construction `'f(g(a()))` is still valid, but the semantic is the following: the constant a is built, then the function g is called, and the returned is put under the constructor f. Therefore, the result of g must be a correct term, which belongs to the right type (i.e. the domain of f).

To simplify the interaction with the host-language, it is also possible to use "unknown symbols" like `f(x.g())` or `f(1+x)`. The scope of the backquote construct is determined by the scope of the most external enclosing braces, except in two case: `'x` and `'x*` which allows you to use variables instantiated by the pattern part. In that case the scope is limited to the length of the variable name, eventually extended by the `'*'`. Sometime, when writing complex expression like `if('x=='y || 'x=='z)`, it can be useful to introduce extra braces (`if( '(x==y || x==z) )`) to extend the scope of the backquote.

5.2.6 *Meta-quote* construct

Tom provide a construct '`%[`' … '`]%`' allowing to build formatted strings without the need to encode special characters such as tabulations and carriage return as usually done in `Java`. For example, to build a string containing the `HelloWorld` program, one can simply use:

```
String hello = %[
  public class Hello {
    public static void main(String[] args) {
      System.out.println("Hello\n\tWorld !");
    }
  }
]%
```

Additionally, it is possible to insert in this string the content of a `String` variable, or the result of a function call (if it is a `String`) using the '`@`' as escape character: the code contained between '`@`' in this construct will be evaluated and the result inserted in the surrounding formatted string. Then, to add to the `HelloWorld` example a version string, we can use:

```
String version = "v12";
String hello2=%[
  public class Hello {
    public static void main(String[] args) {
      System.out.println("Hello\n\tWorld   @version@");
    }
  }
]%;
```

Even if the contents of the *meta-quote* construct is a formatted string, it is required that this string contains correctly balanced braces. **Note:** the expression between two '`@`' can contain Tom constructs, like `backquote` expressions for example.

5.3 Tom signature constructs

5.3.1 Sort definition constructs

To define the mapping between the algebraic constructors and their concrete implementation, Tom provides a signature-mapping mechanism composed of several constructs. In addition to predefined mapping for usual builtin sorts (`int`, `long`, `double`, `boolean`, `string`, and `char`), all other algebraic sorts have to be declared using the `%typeterm` construct.

To use predefined sorts (in a `%match` construct or in the definition of a new operator), it is sufficient to use the `%include` construct (`%include { int.tom }` for example).

When defining a new type with the `%typeterm` construct, at least two extra informations have to be provided:

- the `implement` construct describes how the new type is implemented. The host language part written between braces ('`{`' and '`}`') is never parsed. It is used by the compiler to declare some functions and variables.

- the `visitor_fwd` represents a `Java` class. In order to use strategies, sorts must define a `Java` class. This class must extend `VisitorFwd` and must implement the interface `VisitableVisitor`. The goal of the `VisitorFwd` is to provide a base class for building strategies. Thus, it has to provide typed visitors for each defined sort, and implement its `visit` method to call the typed `visit_sort` functions depending on the dynamic type of the subject. More information about `visitor_fwd` can be found in 7.5.

- the `equals(t1,t2)` construct corresponds to a predicate (parametrized by two term variables). This predicate should return `true` if the terms are "equal". The `true` value should correspond to the builtin `true` value of the considered host language. This last optional predicate is used to compare builtin values and to compile non-linear left-hand sides.

Given a `Java` class `Person` we can define an algebraic mapping for this class:

```
%typeterm TomPerson {
  implement { Person }
  equals(t1,t2) { t1.equals(t2) }
}
```

Here, we assume that the method `equals` implement a comparison function over instances of `Person`. Note also that we have used `TomPerson` to make a clear distinction between algebraic sorts (defined in Tom) and implementation sorts (defined in `Java`, via the use of classes). In practice, we usually use the same name to denote both the algebraic sort and the implementation sort.

The grammar is the following:

IncludeConstruct	::=	'%include' '{' FileName '}'
GomConstruct	::=	'%gom' '{' GomGrammar '}'
GoalLanguageBlock	::=	'{' BlockList '}'
TypeTerm	::=	'%typeterm' Type '{'
		KeywordImplement [KeywordEquals] [KeywordVisitorFwd]
		'}'
KeywordImplement	::=	'implement' GoalLanguageBlock
KeywordEquals	::=	'equals' '(' Name ',' Name ')' GoalLanguageBlock
KeywordVisitorFwd	::=	'visitor_fwd' GoalLanguageVisitorFwd

Note: in order to use strategy constructs (`%strategy`), an extra information have to be defined using the `visitor_fwd` construct.

This sort information should correspond to a `Java` class. In order to use strategies, sorts must define a `Java` class. This class must extend `VisitorFwd` and must implement the interface `VisitableVisitor`. The goal of the `VisitorFwd` is to provide a base class for building strategies. Thus, it has to provide typed visitors for each defined sort, and implement its `visit` method to call the typed `visit_sort` functions depending on the dynamic type of the subject.

See Section 7.5 for a detailed example of strategy support for arbitrary `Java` classes.

5.3.2 Constructor definition constructs

Once algebraic sorts are declared, Tom provides a mechanism to define sorted signatures for constructors using `%op`, `%oplist` or `%oparray` constructs. When defining a new symbol with the `%op` construct, the user should specify the name of the operator, its codomain, and its domain. The later one is defined by a list of pairs (slot-name, sort).

Let us consider again the class `Person`, and let us suppose that an instance of `Person` has two fields (`name` and `age`), we can define the following operator:

```
%op TomPerson person(name:String, age:int)
```

In this example, the algebraic operator `person` has two slots (`name` and `age`) respectively of sorts `String` and `int`, where `String` and `int` are pre-defined sorts.

In addition to the signature of an operator, several auxiliary functions have to be defined:

- The `is_fsym(t)` { predicate(t) } construct is used to check if a term t is rooted by the considered symbol. The `true` value should correspond to the builtin `true` value of the considered host language (`true` in `Java` or `Caml`, and something different from `0` in `C` for example).

- The `make(t1,...,tn)` construct is parametrized by several variables (i.e. that should correspond to the arity of the symbol). A call to this `make` function should return a term rooted by the considered symbol, where each subterm correspond to the terms given in arguments to the function. When defining a constant (i.e. an operator without argument, `make` can be defined without brace: `make { ... }`).

- The `get_slot(slotName,t)` construct has to be defined for all slots of the signature. The implementation of these constructs should be such that the corresponding subterm is returned.

46

Coming back to our example, checking if an object t is rooted by the symbol `person` can be done by checking that t is an instance of the class `Person`. Building a `person` can be done via the `Java` function `new Person(...)`. Accessing to the slots `name` and `age` could be implemented by an access to the variables of the class `Person`. In practice, the following operator definition should work fine:

```
%op TomPerson person(name:String, age:int) {
  is_fsym(t)  { t instanceof Person }
  make(t1,t2) { new Person(t1,t2) }
  get_slot(name,t) { t.name } // assuming that 'name' is public
  get_slot(age,t)  { t.age  } // assuming that 'age' is public
}
```

When defining a new symbol with the `%oplist` construct, the user has to specify how the symbol is implemented. In addition, the user has to specify how a list can be built and accessed:

- the `make_empty()` construct should return an empty list.

- the `make_insert(e,l)` construct corresponds to a function parametrized by a list variable and a term variable. This function should return a new list `l'` where the element `e` has been inserted at the head of the list `l` (i.e. `equals(get_head(l'),e)` and `equals(get_tail(l'),l)` should be `true`).

- the `get_head(l)` function is parametrized by a list variable and should return the first element of the considered list.

- the `get_tail(l)` function is parametrized by a list variable and should return the tail of the considered list.

- the `is_empty(l)` constructs corresponds to a predicate parametrized by a list variable. This predicate should return `true` if the considered list contains no element.

Similarly, when defining a new symbol with the `%oparray` construct, the user has to specify how the symbol is implemented, how an array can be built, and accessed:

- the `make_empty(n)` construct should return a list of size `n`.

- the `make_append(e,l)` construct corresponds to a function parametrized by a list variable and a term variable.
  Warning: This function should return a list `l'` such that the element `e` is at the `n-th` position.

- the `get_element(l,n)` construct is parametrized by a list variable and an integer. This should correspond to a function that return the `n-th` element of the considered list `l`.

- the `get_size(l)` constructs corresponds to a function that returns the size of the considered list. By convention, an empty list contains `0` element.

The `%oplist` or `%oparray` is complex but not difficult to use. Let us consider the `ArrayList` Java class, and let us define a Tom mapping over this data-structure. The first thing to do consists in defining the sort for the elements and the sort for the list-structure:

```
%typeterm Object {
  implement     { Object       }
  equals(l1,l2) { l1.equals(l2) }
}
%typeterm TomList {
  implement     { ArrayList    }
  equals(l1,l2) { l1.equals(l2) }
}
```

Once defined the sorts, it becomes possible to define the list-operator `TomList conc( Object* )`. This operator has a variadic arity: it takes several `Object` and returns a `TomList`.

```
%oparray TomList conc( Object* ) {
  is_fsym(t)       { t instanceof ArrayList }
  make_empty(n)    { new ArrayList(n)       }
  make_append(e,l) { myAdd(e,(ArrayList)l)  }
  get_element(l,n) { (Object)l.get(n)       }
  get_size(l)      { l.size()               }
}

private static ArrayList myAdd(Object e,ArrayList l) {
  l.add(e);
  return l;
}
```

An auxiliary function `myAdd` is used since the `make_append` construct should return a new list. The `get_element` should return an element whose sort belongs to the domain (`Object`) in this example. Although not needed in this example, in general, a cast (`(Object)l.get(n)`) is needed.

The grammar for the mapping constructs is the following:

Operator	::=	'%op' Type Name
		'(' [SlotName ':' Type (',' SlotName ':' Type)*] ')'
		'{' KeywordIsFsym (KeywordMake | KeywordGetSlot)* '}'
OperatorList	::=	'%oplist' Type Name '(' Type '*' ')'
		'{' KeywordIsFsym (KeywordMakeEmptyList
		| KeywordMakeInsert | KeywordGetHead
		| KeywordGetTail | KeywordIsEmpty)* '}'
OperatorArray	::=	'%oparray' Type Name '(' Type '*' ')'
		'{' KeywordIsFsym (KeywordMakeEmptyArray |
		KeywordMakeAppend | KeywordElement
		| KeywordGetSize)* '}'
KeywordIsFsym	::=	'is_fsym' '(' Name ')' GoalLanguageBlock
KeywordGetSlot	::=	'get_slot' '(' Name ',' Name ')' GoalLanguageBlock
KeywordMake	::=	'make' ['(' Name (',' Name)* ')'] GoalLanguageBlock
KeywordGetHead	::=	'get_head' '(' Name ')' GoalLanguageBlock
KeywordGetTail	::=	'get_tail' '(' Name ')' GoalLanguageBlock
KeywordIsEmpty	::=	'is_empty' '(' Name ')' GoalLanguageBlock
KeywordMakeEmptyList	::=	'make_empty' ['(' ')'] GoalLanguageBlock
KeywordMakeInsert	::=	'make_insert' '(' Name ',' Name ')' GoalLanguageBlock
KeywordGetElement	::=	'get_element' '(' Name ',' Name ')' GoalLanguageBlock
KeywordGetSize	::=	'get_size' '(' Name ')' GoalLanguageBlock
KeywordMakeEmptyArray	::=	'make_empty' '(' Name ')' GoalLanguageBlock
KeywordMakeAppend	::=	'make_append' '(' Name ',' Name ')' GoalLanguageBlock

5.3.3 Predefined sorts and operators

See Section 8.1.

5.4 XML pattern

To deal with XML documents, the XML notation can be used (`<A><B attribute="name"/ ></A>` for example).

When manipulating XML documents, we distinguish two main kinds of operations: retrieving information and transforming a document. Tom provides three different XML notations that ought to simplify the definition of patterns: the "standard" and the "implicit" XML notations are used to define compact (but incomplete) patterns. This notation is well suited to retrieve information. The "explicit"

XML notation is used to precisely describe an XML pattern and all the variables that have to be instantiated. This notation is particularly well suited to perform XML transformation since it allows the programmer to precisely describe how variables have to be instantiated.

To make the XML notation understandable, we have to explain how XML documents are handled by Tom. To each XML document corresponds a DOM (Document Object Model) representation. In Tom, we have defined a mapping from DOM sorts to abstract algebraic sorts: `TNode` and `TNodeList`, which correspond respectively to `Node` and `NodeList`, defined by the `Java` DOM implementation.

Thus, a `Node` object becomes a ternary operator `Element` whose first subterm is the name of the XML node, the second subterm is a list of attributes and the third subterm is a list of subterms (which correspond to XML sub-elements). The second and the third elements are terms of sort `TNodeList` (because they are implemented by `NodeList` objects in DOM).

Thus, when considering the <A></A> XML document, the corresponding algebraic term is `Element("A",[],[])`, where `[]` denotes the empty list. Similarly, <A><B attribute="name"></B></A> is encoded into `Element("A",[],[Element("B",[Attribute("attribute","name")],[])])`.

When defining an XML pattern, the user has to introduce extra list-variables to precisely describe the XML pattern and capture the different contexts. Suppose that we are interested in finding a node <B></B> which is a subterm of a node <A></A> (but not necessary the first subterm). The algebraic pattern should be `Element("A",[_*],[_*,Element("B",[_*],[_*]),_*])`. Using the XML notation, this pattern should be expressed as follows: <A (_*)>(_*,<B (_*)>(_*)</B>,_*)</A>. This notation (called explicit) is precise but error prone. This is why we have introduced the explicit notation, where all context variable can be removed (and () are replaced by []): <A []><B []>[]</B></A>. The last notation (called standard XML notation) allows the user to remove the [] and replace the list-separator (,) by spaces. The previous pattern can be written: <A><B></B></A>.

These three different notations allow the user to choose the level of control he wants to have on the XML pattern matching algorithm.

The formal description of the syntax is the following:

XMLTerm	::=	'<' XMLNameList XMLAttrList '/>'
		'<' XMLNameList XMLAttrList '>' XMLChilds '</' XMLNameList '>'
		'#TEXT' '(' Identifier | String ')'
		'#COMMENT' '(' Identifier | String ')'
		'#PROCESSING-INSTRUCTION'
		'(' (Identifier | String) ',' (Identifier | String) ')'
XMLNameList	::=	XMLName
		'(' XMLName ('|' XMLName)* ')'
XMLAttrList	::=	'[' [XMLAttribute (',' XMLAttribute)*] ']'
		'(' [XMLAttribute (',' XMLAttribute)*] ')'
		(XMLAttribute)*
XMLAttribute	::=	'_*'
		VariableName '*'
		AttributeName '=' [AnnotedName '@'] (Identifier | String)
		[AnnotedName '@'] '_' '=' [AnnotedName '@'] (Identifier | String)
XMLChilds	::=	(Term)*
		'[' Term (',' Term)* ']'

Chapter 6

Gom

Gom is a generator of tree implementations in `Java`, allowing the definition of a preferred canonical form, using *hooks*. The tree implementations Gom generates are characterized by strong typing, immutability (there is no way to manipulate them with side-effects), maximal subterm sharing and the ability to be used with strategies.

6.1 Gom syntax

The basic functionality of Gom is to provide a tree implementation in `Java` corresponding to an algebraic specification.

Gom provides a syntax to concisely define abstract syntax tree, which is inspired from the algebraic type definition of ML languages.

Each Gom file consists in the definition of one or more modules. In each module we define sorts, and operators for the sorts. Additionally, it allows to define *hooks* modifying the default behavior of an operator.

GomGrammar	::=	Module
Module	::=	'module' ModuleName [Imports] Grammar
Imports	::=	'imports' (ModuleName)*
Grammar	::=	'abstract syntax' (TypeDefinition |HookDefinition)*
TypeDefinition	::=	SortName '=' ['|'] OperatorDef ('|' OperatorDef)*
OperatorDef	::=	OperatorName '('[SlotDef(','SlotDef)*]')'
	|	OperatorName '(' SortName'*' ')'
SlotDef	::=	SlotName ':' SortName
HookDefinition	::=	OperatorName ':' HookOperation '{' TomCode '}'
HookOperation	::=	('make' |'make_insert') '(' (Identifier)* ')'
ModuleName	::=	Identifier
SortName	::=	Identifier
OperatorName	::=	Identifier
SlotName	::=	Identifier

6.1.1 Builtin sorts and operators

Gom supports integers and String builtins. Those builtins define respectively the sorts `int` and `String`, that can be used as sorts for slot definitions. It is not possible to define a new operator whose domain is one of the builtin sorts, since those sorts are defined in another module.

Note: to be able to use a builtin sort, it is necessary to declare the import of the corresponding module (`int` or `String`) in the Imports section.

6.1.2 Example of signature

The syntax of Gom is quite simple and can be used to define many-sorted abstract-datatypes. The `module` section defines the name of the signature. The `imports` section defines the name of the imported

51

signatures. The `abstract syntax` part defines the operators with their signature. For each argument, a sort and a name (called slot-name) has to be given.

```
module Expressions
imports String int

abstract syntax
Bool = True()
     | False()
     | Eq(lhs:Expr, rhs:Expr)
Expr = Id(stringValue:String)
     | Nat(intValue:int)
     | Add(lhs:Expr, rhs:Expr)
     | Mul(lhs:Expr, rhs:Expr)
```

The definition of a signature in Gom has several restrictions:

- there is no overloading: two operators cannot have the same name

- for a given operator, all slot-names must be different

- if two slots have the same slot-name, they must belong to the same sort. In the previous example, `Eq`, `Add`, and `Mul` can have a slot called `lhs` of sort `Expr`. But, `Id` and `Nat` cannot have a same slot named `value`, since their sort are not identical (the slots are respectively of sorts `String` and `int` for `Id` and `Nat`).

6.2   Generated API

For each Module, the Gom compiler generates an API specific of this module, possibly using the API of other modules (declared in the Imports section). This API is located in a Java package named using the ModuleName lowercased, and contains the tree implementation itself, with some additional utilities:

- an abstract class named ModuleNameAbstractType is generated. This class is the generic type for all nodes whose type is declared in this module. It declares generic functions for all tree nodes: a `toATerm()` method returning a `aterm.ATerm` representation of the tree; a `symbolName()` method returning a `String` representation for the function symbol of the tree root; the `toString()` method, which returns a string representation. It declares also all `accept(Visitor v)` to implement the visitor combinators pattern used by the strategy language.

- the Visitor, BasicStrategy and Forward classes for a module named Module are respectively named `ModuleNameVisitor`, `ModuleNameBasicStrategy`, `ModuleNameForward`. Those classes implement the visitor combinator design pattern for the module, by providing a typed Visitor interface (one `visit` method per sort), a Forward visitor providing a simple forwarding visitor respecting types, and a BasicStrategy, which is a visitable visitor built on top of the Forward class.

- in a subpackage `types`, Gom generates one class for each sort defined in the module, whose name corresponds to the sort name. Each sort class extends AbstractType for the module, and declares methods `boolean isOperatorName()` for each operator in the module (`false` by default). It declares getters (methods named `getSlotName()`) for each slot used in any operator of the sort (throwing an exception by default). A method `SortName fromTerm(aterm.ATerm trm)` is generated, allowing to use ATerm as an exchange format.

- for each sort, generate classes for the operators in a package `types.SortName`. All classes for the operator are named following the operator name, and extend the class generated for the corresponding sort. Thus they provide getters for the slots of the operator, and the `isOperatorName()` method is overriden to return `true`. Those classes implement the `jjtraveler.Visitable` interface. It is worth noting that builtin fields are not accessible from the `jjtraveler.Visitable`, and thus will not be visited by strategies. The operator class also implements a static `make` method, building a new instance of the operator. This `make` method is the only way to obtain a new instance.

52

- for each module, one file `ModuleName.tom` providing a Tom mapping for the sorts and operators defined or imported by the module.

See section 6.2.4 for an example.

6.2.1 Hooks

Hooks are used to alter operations of the operators. 'make' and 'make_insert' hooks are altering the creation operations for respectively algebraic and variadic operators. 'make_insert' is simply a derivative case of 'make' for the `Cons` of variadic operators (see 6.2.3).

The hook operation type is followed by a list of arguments name between '()'. The creation operation takes those arguments in order to build a new instance of the operator. Thus, the arguments number has to match the slot number of the operator definition, and types are inferred from this definition.

Then the body of the hook definition is constructed of `Java` of Tom code. The Tom code is compiled using the mapping definition for the current module, and thus allows to build and match terms from the current module. This code can also use the `realMake` function, which consists in the "inner" default allocation function. This function takes the same number and types of arguments as the hook. In any case, if the hooks code does not perform a `return` itself, this `realMake` function is called with the hooks arguments at the end of the hook execution.

6.2.2 Hook example

Using the `expression` example introduced in 6.1.2, we can add *hooks* to implement the computation of `Add` and `Mul` when both arguments are known integers (i.e. when they are `Nat(x)`)

```
module Expressions
imports String int

abstract syntax
Bool = True()
     | False()
     | Eq(lhs:Expr, rhs:Expr)
Expr = Id(stringValue:String)
     | Nat(intValue:int)
     | Add(lhs:Expr, rhs:Expr)
     | Mul(lhs:Expr, rhs:Expr)
Add:make(l,r) {
  %match(Expr l, Expr r) {
    Nat(lvalue), Nat(rvalue) -> {
      return 'Nat(lvalue + rvalue);
    }
  }
}
Mul:make(l,r) {
  %match(Expr l, Expr r) {
    Nat(lvalue), Nat(rvalue) -> {
      return 'Nat(lvalue * rvalue);
    }
  }
}
```

Using this definition, it is impossible to have an expression containing unevaluated expressions where a value can be calculated. Thus, a procedure doing constant propagations for `Id` whose value is known could simply replace the `Id` by the corresponding `Nat`, and rely on this mechanism to evaluate the expression. Note that the arguments of the `make` hook are themselves elements built on this signature, and thus the hooks have been applied for them. In the case of hooks encoding a rewrite system, this corresponds to using an innermost strategy.

6.2.3   Variadic operators

Variadic operators are operators that have no fixed arity. They can have any number of arguments, and are usually used to represent lists. Variadic operators are declared using the syntax:

$$\text{OperatorDef} \quad ::= \quad \text{OperatorName '(' SortName'*' ')'}$$

The generated code contains two operator classes: one name `EmptyOperator` which is used to represent the empty list of arity 0, and the other named `ConsOperator` having two fields: one with the codomain sort, and one with the domain sort of the variadic operator, respectively named `HeadOperator` and `TailOperator`, leading to getter functions `getHeadOperator` and `getTailOperator`. This allows to define lists as the composition of many `Cons` and one `Empty` objects.

As the interface is different than the one for standard algebraic operators, the `make_insert` hook operation is used to modify the creation operation for variadic operators.

6.2.4   Generated API example

We show elements of the generated API for a very simple example, featuring variadic operator. It defines natural numbers as `Zero()` and `Suc(n)`, and lists of natural numbers.

```
module Mod
abstract syntax
Nat = Zero()
    | Suc(pred:Nat)
    | Plus(lhs:Nat,rhs:Nat)
    | List(Nat*)
```

Using the command `gom Mod.gom`, the list of generated files is:

```
mod/Mod.tom                 (the Tom mapping)
mod/ModAbstractType.java
mod/ModBasicStrategy.java
mod/ModForward.java
mod/ModVisitor.java
mod/types/Nat.java          (abstract class for the "Nat" sort)
mod/types/nat/ConsList.java   \
mod/types/nat/EmptyList.java  / Implementation for the operator "List"
mod/types/nat/Plus.java     (Implementation for "Plus")
mod/types/nat/Suc.java      (Implementation for "Suc")
mod/types/nat/Zero.java     (Implementation for "Zero")
```

The `ModAbstractType` class declares generic methods shared by all operators in the `Mod` module:

```
public aterm.ATerm toATerm()
public String symbolName()
public String toString()
public ModAbstractType accept(mod.ModVisitor v) throws jjtraveler.VisitFailure
```

The classes `ModVisitor`, `ModForward`, and `ModBasicStrategy` implement the visitor combinator design pattern for the data structure of the `Mod` module. The `mod/types/Nat.java` class provides an abstract class for all operators in the `Nat` sort, implementing the `ModAbstractType` and contains the following methods. First, the methods for checking the root operator, returning `false` by default:

```
public boolean isConsList()
public boolean isEmptyList()
public boolean isPlus()
public boolean isSuc()
public boolean isZero()
```

Then getter methods, throwing an `UnsupportedOperationException` by default, as the slot may not be present in all operators. This is convenient since at the user level, we usually manipulate objects of sort `Nat`, without casting them to more specific types.

```
public mod.types.Nat getpred()
public mod.types.Nat getlhs()
public mod.types.Nat getHeadList()
public mod.types.Nat getrhs()
public mod.types.Nat getTailList()
```

The `fromTerm` static method allows Gom data structure to be interoperable with `ATerm`
```
public static mod.types.Nat fromTerm(aterm.ATerm trm)
```
The operator implementations redefine all or some getters for the operator to return its subterms. It also provides a static `make` method to build a new tree rooted by this operator, and implements the `jjtraveler.Visitable` interface. For instance, in the case of the `Plus` operator, the interface is:

```
public static Plus make(mod.types.Nat lhs, mod.types.Nat rhs)
public int getChildCount()
public jjtraveler.Visitable getChildAt(int index)
public jjtraveler.Visitable setChildAt(int index, jjtraveler.Visitable v)
```

completed with the methods from the `Nat` class and the `ModAbstractType`. For the `ConsList` class, we obtain:

```
/* the constructor */
public static ConsList make(mod.types.Nat _HeadList, mod.types.Nat _TailList) { ... }
public String symbolName() { ... }
/* From the "Nat" class */
public boolean isConsList() { ... }
public mod.types.Nat getHeadList() { ... }
public mod.types.Nat getTailList() { ... }
/* From the "ModAbstractType" class */
public aterm.ATerm toATerm() { ... }
public static mod.types.Nat fromTerm(aterm.ATerm trm) { ... }
/* The jjtraveler.Visitable interface */
public int getChildCount() { ... }
public jjtraveler.Visitable getChildAt(int index) { ... }
public jjtraveler.Visitable setChildAt(int index, jjtraveler.Visitable v) { ... }
/* The MuVisitable interface */
public jjtraveler.Visitable setChilds(jjtraveler.Visitable[] childs) { ... }
```

6.3   Combining Gom with Tom

A first solution to combine Tom with Gom is to use Gom as a standalone tool, using the command line tool or the ant task.

In that case, the module name of the Gom specification and the `package` option determine where the files are generated. To make things correct, it is sufficient to import the generated `Java` classes, as well as the generated Tom file. In the case of a Gom module called `Module`, all files are generated in a directory named `module` and the Tom program should do the following:

```
import module.*;
import module.types.*;

class MyClass {
  ...
  %include { module/Module.tom }
...
}
```

A second possibility to combine Tom with Gom is to use the `%gom` construct offered by Tom. In that case, the Gom module can be directly included into the Tom file, using the `%gom` instruction:

```
package myPackage;

import myPackage.myclass.expressions.*;
import myPackage.myclass.expressions.types.*;

class MyClass {
  %gom{
    module Expressions

    abstract syntax
    Bool = True()
         | False()
      ...
    Expr = Mul(lhs:Expr, rhs:Expr)
  }
  ...
}
```

Note that the Java code is generated in a package that corresponds to the current package, followed by the class-name and the module-name. This allows to define the same module in several classes, and avoid name clashes.

Chapter 7

Strategies

Strategies provides a way to control how a transformation rule is applied. In Tom, users can define *elementary strategies* (corresponding to a set of rewriting rules) and combine them using the *MuTraveler library*. An *elementary strategy* corresponds to a list of `MatchStatement` (one associated to each sort) and is defined using the construction `%strategy`. The *MuTraveler library*, inspired by ELAN, Stratego, and JJTraveler, allows to easily define various kinds of traversal strategies.

Note: strategies are only supported associated with `Java`.

7.1   Elementary strategy

7.1.1   Elementary strategies from MuTraveler

The two elementary strategies are the identity and the failure. There are defined in *MuTraveler library*. These two strategies have no effect on the term. The identity strategy written `'Identity()` returns the term unchanged. The failure strategy written `'Fail()` throws a `jjtraveler.VisitFailure` exception.

7.1.2   Elementary strategies defined by users

Users can define elementary strategies by using the `%strategy` construction. Here is the `%strategy` grammar:

StrategyConstruct	::=	'`%strategy`' StrategyName '(' [StrategyArguments] ')'
		'extends' '`'`' ExtendsTerm '{' StrategyVisitList '}'
StrategyArguments	::=	SubjectName ':' AlgebraicType (',' SubjectName ':' AlgebraicType)*
StrategyVisitList	::=	(StrategyVisit)*
StrategyVisit	::=	'`visit`' AlgebraicType '{' (PatternAction)* '}'

This strategy has a name, followed with mandatory parenthesis. Inside these parenthesis, we have optional parameters.

The strategy has an `extendsTerm` which defines the behavior of the default strategy that will be applied. In most cases, two default behaviours are used:

- the failure: `'Fail()`

- the identity: `'Identity()`

The body of the strategy is a list of visited sorts. Each `StrategyVisit` contains a list of `PatternAction` that will be applied to the corresponding sort. In other words, a `StrategyVisit` is translated into a `MatchStatement`.

For strategies, all `AlgebraicType` must have a `TomForwardType`. For a given strategy, all visited sorts must have the same `TomForwardType`. This `TomForwardType` represents a `Java` class. This class must implement the interface `VisitableVisitor`. By default this mechanism is done automatically by Gom 6.2. If you are defining a new sort using `%typeterm`, you have to define `visitor_fwd` 5.3.1:

```
%typeterm Term {
  implement { strategy.term.types.Term }
  visitor_fwd { strategy.term.termVisitableFwd }
  equals(t1,t2) { t1.equals(t2) }
}
```

For instance, here is an elementary strategy:

```
%strategy RewriteSystem() extends 'Identity() {
  visit Term {
      a()  -> { return 'b(); }
      b()  -> { return 'c(); }
  }
}
```

Then you can instantiate it:

```
VisitableVisitor rule = 'RewriteSystem();
```

Trick
As strategy parameters must be algebraic types, if for instance you need to pass Java variables, you have to use mappings to do so:

```
%typeterm HashSet {
  implement { java.util.HashSet }
}
```

For instance, here is an elementary strategy:

```
%strategy RewriteSystem(hs:HashSet) extends 'Identity() {
  visit Term {
    a() -> { hs.add('a()); }
    b() -> { return 'c(); }
  }
}
```

Then you can instantiate it:

```
HashSet hashset = new HashSet();
VisitableVisitor rule ='RewriteSystem(hashset);
```

Strategies defined by '**%strategy**' are local to the file defining them. If you want to export your strategy to another package, you have to create a public function exporting an instance of the strategy built with '''.

Note: you cannot use **%strategy** inside a function.

7.2   Basic strategy operators

The following operators are the key-component that can be used to define more complex strategies. In this framework, the application of a strategy to a term can fail. In Java, the *failure* is implemented by an exception (jjtraveler.VisitFailure).

$$
\begin{array}{lcl}
(\mathsf{Identity})[t] & \Rightarrow & t \\
(\mathsf{Fail})[t] & \Rightarrow & \textit{failure} \\
(\mathsf{Sequence(s1,s2))}[t] & \Rightarrow & \textit{failure} \text{ if } (\mathsf{s1})[t] \text{ fails} \\
& & (\mathsf{s2})[\mathsf{t'}] \text{ if } (\mathsf{s1})[t] \Rightarrow \mathsf{t'} \\
(\mathsf{Choice(s1,s2))}[t] & \Rightarrow & \mathsf{t'} \text{ if } (\mathsf{s1})[t] \Rightarrow \mathsf{t'} \\
& & (\mathsf{s2})[t] \text{ if } (\mathsf{s1})[t] \text{ fails} \\
(\mathsf{All(s)})[f(t1,...,tn)] & \Rightarrow & f(t1',...,tn') \text{ if } (\mathsf{s})[t1] \Rightarrow t1', ..., (\mathsf{s})[tn] \Rightarrow tn' \\
& & \textit{failure} \text{ if there exists i such that } (\mathsf{s})[ti] \text{ fails} \\
(\mathsf{All(s)})[cst] & \Rightarrow & c \\
(\mathsf{One(s)})[f(t1,...,tn)] & \Rightarrow & f(t1,...,ti',...,tn) \text{ if } (\mathsf{s})[ti] \Rightarrow ti' \\
& & \textit{failure} (\mathsf{s})[t1] \text{ fails, ..., } (\mathsf{s})[tn] \text{ fails} \\
(\mathsf{One(s)})[cst] & \Rightarrow & \textit{failure}
\end{array}
$$

For example, we define myFirstStrat = All(s) where s is defined as the elementary strategy:

```
%strategy RewriteSystem() extends 'Fail() {
  visit Term {
    a() -> { return 'b(); }
  }
}
```

If we apply this strategy to different subjects, we obtain:

$$
\begin{array}{lcl}
(\mathsf{myFirstStrat})[f(a(),a())] & \Rightarrow & f(b(),b()) \\
(\mathsf{myFirstStrat})[f(a(),b())] & \Rightarrow & \textit{failure} \\
(\mathsf{myFirstStrat})[b()] & \Rightarrow & b()
\end{array}
$$

Sometimes, it is interesting to get the postiion where the strategy is applied. Positions in a term are represented as sequences of integers. The class `MuTraveler` provides a static method `getPosition(VisitableVisitor s)` to collect the current position in the term visited by `s`. To use this method or the following strategies which depend on it, you must call the static method `init` on your strategy.

```
try {
  VisitableVisitor s = MuTraveler.init('OnceBottomUp(rule));
  s.visit(subject));
} catch(VisitFailure e) {
  System.out.prinltn("Failure at position" + MuTraveler.getPosition(s));
}
```

The library gives several basic strategies using the position:

$$
\begin{array}{lcl}
(\mathsf{Omega(i,s)})[f(t1,...,tn)] & \Rightarrow & f(t1,...,ti',...,tn) \text{ if } (\mathsf{s})[ti] \Rightarrow ti' \\
& & \textit{failure} \text{ if } (\mathsf{s})[ti] \text{ fails} \\
(\mathsf{MuTraveler.getReplace(i,t)})[f(t1,...,tn)] & \Rightarrow & f(t1,...,t,...,tn) \\
(\mathsf{MuTraveler.getSubterm(i,t)})[f(t1,...,tn)] & \Rightarrow & ti
\end{array}
$$

Note: the static method MuTraveler.getReplace(i,t) corresponds to the strategy Omega(i,s) where s is a strategy reduced to one rule $x \rightarrow t$.

Note: the static method MuTraveler.getPosition(i,t) returns a strategy that is not type preserving.

7.3   Strategy library

In order to define recursive strategies, we introduce the `mu` abstractor. This allows to give a name to the current strategy, which can be referenced later.

$$
\begin{array}{lcl}
\mathsf{Try(s)} & = & \mathrm{Choice(s,Identity)} \\
\mathsf{Repeat(s)} & = & \mathrm{mu\ x.Choice(Sequence(s,x),Identity())} \\
\mathsf{OnceBottomUp(s)} & = & \mathrm{mu\ x.Choice(One(x),s))} \\
\mathsf{BottomUp(s)} & = & \mathrm{mu\ x.Sequence(All(x),s))} \\
\mathsf{TopDown(s)} & = & \mathrm{mu\ x.Sequence(s,All(x)))} \\
\mathsf{Innermost(s)} & = & \mathrm{mu\ x.Sequence(All(x),Try(Sequence(s,x))))}
\end{array}
$$

The Try strategy never fails: it tries to apply the strategy s. If it succeeds, the result is returned. Otherwise,the Identity strategy is applied, and the subject is not modified.

The Repeat strategy applies the strategy s as many times as possible, until a failure occurs. The last unfailing result is returned.

The strategy OnceBottomUp tries to apply the strategy s once, starting from the leftmost-innermost leaves. BottomUp looks like OnceBottomUp but is not similar: s is applied to all nodes, starting from the leaves. Note that the application of s should not fail, otherwise the whole strategy also fails.

The strategy Innermost tries to apply s as many times as possible, starting from the leaves. This construct is useful to compute normal forms.

For example, we define myFirstInnerMost = InnerMost(s) where s is defined as the elementary strategy:

```
%strategy RewriteSystem() extends 'Fail() {
  visit Term {
    a()        -> { return 'b(); }
    b()        -> { return 'c(); }
    g(c(),c()) -> { return 'c(); }
  }
}
```

The application of this strategy to different subject terms gives:

$$
\begin{array}{lcl}
(\mathsf{myFirstInnerMost})[g(a(),b())] & \Rightarrow & c() \\
(\mathsf{myFirstInnerMost})[f(g(g(a,b),g(a,a)))] & \Rightarrow & f(c(),c()) \\
(\mathsf{myFirstInnerMost})[g(d(),d())] & \Rightarrow & g(d(),d())
\end{array}
$$

We can notice that Innermost strategy never fails. If we try myFirstBottomUp = BottomUp(s) with the same subjects, we obtain always *failure* because if s fails on a node, the whole strategy fails.

7.4 Strategies with identity considered as failure

In order to get more efficient strategies (in particular when performing leftmost-innermost normalization), we consider variants where the notion of *failure* corresponds to the *identity*. This means that when a term cannot can be transformed by a strategy (into a different term), this is considered as a *failure*.

$$
\begin{array}{lcl}
(\mathsf{SequenceId}(s1,s2))[t] & \Rightarrow & (s2)[t'] \text{ if } (s1)[t] \Rightarrow t' \text{ with } t!=t' \\
& & t \text{ otherwise} \\
(\mathsf{ChoiceId}(s1,s2))[t] & \Rightarrow & t' \text{ if } (s1)[t] \Rightarrow t' \text{ with } t!=t' \\
& & (s2)[t] \text{ otherwise} \\
(\mathsf{OneId}(s))[f(t1,...,tn)] & \Rightarrow & f(t1,...,ti',...,tn) \text{ if } (s)[ti] \Rightarrow ti' \text{ with } ti!=ti' \\
& & f(t1,...,tn) \text{ otherwise} \\
(\mathsf{OneId}(s))[cst] & \Rightarrow & cst \\
\\
\mathsf{TryId}(s) & = & s \\
\mathsf{RepeatId}(s) & = & mu\ x.SequenceId(s,x)) \\
\mathsf{OnceBottomUpId}(s) & = & mu\ x.ChoiceId(OneId(x),s)) \\
\mathsf{OnceTopDownId}(s) & = & mu\ x.ChoiceId(s,OneId(x))) \\
\mathsf{InnermostId}(s) & = & mu\ x.Sequence(All(x),SequenceId(s,x))) \\
\mathsf{OutermostId}(s) & = & mu\ x.Sequence(SequenceId(s,x),All(x)))
\end{array}
$$

We can define a strategy trying to apply a simple rewrite system to the root of a term, replacing a() by b(), b() by c(), and g(c(),c()) by a(), and otherwise returning the identity:

```
import jjtraveler.reflective.VisitableVisitor;
import tom.library.strategy.mutraveler.*;
import jjtraveler.Visitable;
import jjtraveler.VisitFailure;
```

We also need to import the corresponding mapping:

```
%include { mutraveler.tom }
```

Then we define an elementary strategy:

```
%strategy RewriteSystem() extends 'Fail() {
  visit Term {
    a()        -> { return 'b(); }
    b()        -> { return 'c(); }
    g(c(),c()) -> { return 'c(); }
  }
}
```

Then, it becomes quite easy to define various strategies on top of this elementary strategy:

```
Term subject = 'f(g(g(a,b),g(a,a)));
VisitableVisitor rule = 'RewriteSystem();
try {
  System.out.println("subject       = " + subject);
  System.out.println("onceBottomUp  = " +
      MuTraveler.init('OnceBottomUp(rule)).visit(subject));
  System.out.println("innermost    = " +
      MuTraveler.init('Choice(BottomUp(rule),Innermost(rule))).visit(subject));
} catch (VisitFailure e) {
  System.out.println("reduction failed on: " + subject);
}
```

7.5   Implementing elementary strategies in Java

The simplest way to use strategies is to use them with data-structures generated by Gom, as this data structure implementation provides the interfaces and classes needed to use '%strategy'.

However, it is not impossible (nor difficult) to use '%strategy' with another term implementation. The requirements for using '%strategy' on the data structure are:

- the classes implementing the terms (declared in the 'implements' section of a '%typeterm') have to implement the jjtraveler.Visitable interface

- A 'visitor_fwd' or BasicStrategy class has to be provided for the data structure, implementing the jjtraveler.reflective.VisitableVisitor interface, and providing dispatch of the visit method on type preserving visit_<type> methods.

- All '%typeterm' for the data structure shall have the same 'visitor_fwd' class

We detail here on a simple example of hand written data structure how to adapt the data structure to be able to use '%strategy'. Given a Java class Person we can define an algebraic mapping for this class:

```
%typeterm TomPerson {
  implement { Person }
  equals(t1,t2) { t1.equals(t2) }
  visitor_fwd { PersonVisitor }
}
```

For this example, we consider a data-structure those Gom equivalent could be

```
Term = A()
     | B()
     | G(arg:Slot)
     | F(arg1:Term, arg2:Term)
Slot = Name(name:String)
```

7.5.1 Simple implementation of the data structure

We first present a very straightforward implementation of this data structure. It will serve as a basis and will be extended to provide support for strategies.

We first define abstract classes for the two sorts of this definition, `Term` and `Slot`, and classes for those operators extending those sort classes.

```
public abstract class Term { }
public abstract class Slot { }

/* A and B are similar up to a renaming */
public class A extends Term {
  public String toString() {
    return "A()";
  }
  public boolean equals(Object o) {
    if(o instanceof A) {
      return true;
    }
    return false;
  }
}


/* G is similar to F, but has only one child */
public class F extends Term {
  public Term a;
  public Term b;
  public F(Term arg0, Term arg1) {
    a = arg0;
    b = arg1;
  }
  public String toString() {
    return "F("+a.toString()+", "+b.toString()+")";
  }
  public boolean equals(Object o) {
    if(o instanceof F) {
      F f = (F) o;
      return a.equals(f.a) && b.equals(f.b);
    }
    return false;
  }
}

public class Name extends Slot {
  public String name;
  public Name(String s) {
    this.name = s;
  }
  public String toString() {
    return "Name("+name+")";
  }
  public boolean equals(Object o) {
    if(o instanceof Name) {
      Name n = (Name) o;
      return name.equals(n.name);
    }
    return false;
```

```
  }
}
```

We only took care in this implementation to get a correct behavior for the `equals` method. Then, the Tom mapping is simply

```
%include { string.tom }
%typeterm Term {
  implement { Term }
  equals(t1,t2) {t1.equals(t2)}
}
%typeterm Slot {
  implement { Slot }
  equals(t1,t2) {t1.equals(t2)}
}
%op Term A() {
  is_fsym(t) { (t!= null) && (t instanceof A) }
  make() { new A() }
}
%op Term F(arg1:Term, arg2:Term) {
  is_fsym(t) { (t!= null) && (t instanceof F) }
  get_slot(arg1,t) { ((F) t).a }
  get_slot(arg2,t) { ((F) t).b }
  make(t0, t1) { new F(t0, t1) }
}
...
%op Slot Name(name:String) {
  is_fsym(t) { (t!= null) && (t instanceof Name) }
  get_slot(name,t) { ((Name) t).name }
  make(t0) { new Name(t0) }
}
```

7.5.2   Extending to support strategies

We first need to make sure all classes representing operators do implement the `jjtraveler.Visitable` interface. We also need those classes to provide a `accept` method taking as argument the class that will serve as `visitor_fwd`. We decide here to call this class `BasicStrategy`, and its code and behavior will be detailed later.

```
public abstract class Term implements jjtraveler.Visitable {
  public Term accept(BasicStrategy v) throws jjtraveler.VisitFailure {
    return v.visit_Term(this);
  }
}

public abstract class Slot implements jjtraveler.Visitable {
  public Slot accept(BasicStrategy v) throws jjtraveler.VisitFailure {
    return v.visit_Slot(this);
  }
}
```

Once the abstract classes for sorts declare they implements `jjtraveler.Visitable`, we have to adapt the classes for the operators to actually implement it. This interface is composed of three methods:

- `public int getChildCount()`
  This method returns the number of direct subterms of the term. It returns 0 for constants.

- `public jjtraveler.Visitable getChildAt(int index)`
  This method returns the subterm at the `index` position. The first subterm is at position 0.

- `public jjtraveler.Visitable setChildAt(int index, jjtraveler.Visitable v)`
  This method replaces the subterm at position `index` by the argument `v`

As all subterms have to implement the `jjtraveler.Visitable` interface, we do not provide access to subterms of *builtin* sorts like `String` od `int`, but discard them. Thus those children will not be traversed by the strategies.

For constants like the class `A`, we get:

```
public class A extends Term {
  public int getChildCount() { return 0; }
  public jjtraveler.Visitable getChildAt(int index) {
    throw new IndexOutOfBoundsException();
  }
  public jjtraveler.Visitable setChildAt(int index, jjtraveler.Visitable v) {
    throw new IndexOutOfBoundsException();
  }
  public String toString() {
    return "A()";
  }
  public boolean equals(Object o) {
    if(o instanceof A) {
      return true;
    }
    return false;
  }
}
```

Then for `F`, which has two subterms, we implement setters and getters for children:

```
public class F extends Term {
  public Term a;
  public Term b;
  public F(Term arg0, Term arg1) {
    a = arg0;
    b= arg1;
  }
  public int getChildCount() { return 2; }
  public jjtraveler.Visitable getChildAt(int index) {
    if(index == 0) {
      return a;
    } else if(index == 1) {
      return b;
    }
    throw new IndexOutOfBoundsException();
  }
  public jjtraveler.Visitable setChildAt(int index, jjtraveler.Visitable v) {
    if(index == 0) {
      return new F((Term)v,b);
    } else if(index == 1) {
      return new F(a,(Term)v);
    }
    throw new IndexOutOfBoundsException();
  }
  public String toString() {
    return "F("+a.toString()+", "+b.toString()+")";
  }
  public boolean equals(Object o) {
    if(o instanceof F) {
```

```
      F f = (F) o;
      return a.equals(f.a) && b.equals(f.b);
    }
    return false;
  }
}
```

The case of `Name` is special, since it has only one child, which is of sort `String`, and thus do not implement jjtraveler.Visitable

```
public class Name extends Slot {
  public String name;
  public Name(String s) {
    this.name = s;
  }
  /* Do not visit the "String" child, as we can't make it Visitable */
  public int getChildCount() { return 0; }
  public jjtraveler.Visitable getChildAt(int index) {
    throw new IndexOutOfBoundsException();
  }
  public jjtraveler.Visitable setChildAt(int index, jjtraveler.Visitable v) {
    throw new IndexOutOfBoundsException();
  }
  public String toString() {
    return "Name("+name+")";
  }
  public boolean equals(Object o) {
    if(o instanceof Name) {
      Name n = (Name) o;
      return name.equals(n.name);
    }
    return false;
  }
}
```

Then, the `BasicStrategy` has to be implemented. This class implements an interface from the JJTraveler library: `jjtraveler.reflective.VisitableVisitor`. It also has an arguments which itself is a strategy, and its `visit` method either call the `accept` method of both `Term` and `Slot` sorts (which in turn will call the type preserving `visit` methods `BasicStrategy` declares), or delegate the `visit` call to its argument.

```
public class BasicStrategy implements jjtraveler.reflective.VisitableVisitor {
  protected jjtraveler.Visitor any;

  public BasicStrategy(jjtraveler.Visitor v) {
    this.any = v;
  }

  public jjtraveler.Visitable visit(jjtraveler.Visitable v)
                               throws jjtraveler.VisitFailure {
    if (v instanceof Term) {
      return ((Term) v).accept(this);
    } else if (v instanceof Slot) {
      return ((Slot) v).accept(this);
    } else {
      return any.visit(v);
    }
```

```
  }

  public Term visit_Term(Term arg) throws jjtraveler.VisitFailure {
    return (Term) any.visit(arg);
  }

  public Slot visit_Slot(Slot arg) throws jjtraveler.VisitFailure {
    return (Slot) any.visit(arg);
  }

  /* jjtraveler.Visitable interface */
  public int getChildCount() { return 1; }
  public jjtraveler.Visitable getChildAt(int index) {
    if(index == 0) {
      return (jjtraveler.Visitable) any;
    }
    throw new IndexOutOfBoundsException();
  }
  public jjtraveler.Visitable setChildAt(int index, jjtraveler.Visitable v) {
    if(index == 0) {
      any = ((jjtraveler.Visitor) v);
      return this;
    }
    throw new IndexOutOfBoundsException();
  }
}
```

Once this is set up, we still have to modify the Tom mapping to add a `visitor_fwd`, pointing to the `BasicStrategy` class.

```
%typeterm Term {
  implement { Term }
  visitor_fwd { BasicStrategy }
  equals(t1,t2) { t1.equals(t2) }
}
%typeterm Slot {
  implement { Slot }
  visitor_fwd { BasicStrategy }
  equals(t1,t2) { t1.equals(t2) }
}
```

Those changes make possible the use of '`%strategy`' with any term implementation. While the core Tom functionalities can work without any modification of the user data structure, the use of '`%strategy`' requires some changes to the data structure to be used. However, those changes are relatively easy to implement, and the `jjtraveler.Visitable` interface is reasonable for tree-like structures.

Chapter 8

Runtime Library

8.1 Predefined mappings

8.1.1 Builtin sorts

The system comes with several predefined signature-mappings for C, Java and Caml. Among them, let us mention:

- `int.tom` (written 1, 2, 3, *etc.*)

- `long.tom` (written 1l or 1L, *etc.*)

- `double.tom` (using Java grammar for double)

- `boolean.tom` (true or false)

- `char.tom` (written 'a', 'b', *etc.*)

- `string.tom` (written "a", "ab", *etc.*)

These mappings define, for each builtin sort of the host language, an algebraic sort that can be used in a `%match` or a signature definition construct. Thus, builtin values, such as `f(5)`, `g('a')` or `h("foo")`, can be used in patterns.

The `string.tom` mapping is interesting because it provides an associative operator (`concString`) which allows the programmer to consider a string as a list of characters. Thus, the string `"foo"` can be seen as the algebraic object `concString('f','o','o')`. By using this mapping, it becomes possible to perform pattern matching against the content of a string. The pattern `concString('f',X*)` will match any string which begins with the character `'f'`. By using the unnamed-symbol capability, this pattern can be written: `('f',X*)` instead of `concString('f',X*)`.

To match any string which begins with the substring `"fo"`, the corresponding pattern should be `('f','o',X*)`. To simplify the definitions of such patterns, Tom supports an exception which allows the programmer to write `('fo',X*)` instead. Internally, the ill-formed character `'fo'` is expanded into the list `('f','o')`. Thus, the annotated notation `y@'fo'` is expanded into `(y@'f',y@'o')`, and thus may be dangerous to use in this case.

In addition to these mappings, several other predefined mappings come with the system:

- `aterm.tom` and `atermlist.tom` provide a mapping to the C and Java version of the ATerm Library

- `list.tom` provides a mapping to the builtin notion of List in Caml

- `dom.tom` provides a mapping to the Java version of the DOM Library

67

8.1.2 MuTraveler

To support strategies, Tom provides a predefined mapping to allow the description of strategies.

- `mutraveler.tom`: maps the basic strategies classes, and provides a `mu` operator to express the recursion as well a `MuVar(v:String)` operator to allow the definition of complex strategies as described in Section 7.3.

8.2 XML

To support the transformation of XML documents, Tom provides a specific syntax for defining patterns, as well as several predefined mappings:

- `dom.tom`: maps XML notation to a `Java` implementation of the DOM library

- `dom_1.5`.tom: maps XML notation to a `Java` (version 1.5) implementation of the DOM library

- `TNode.tom`: maps XML notation to an ATerm based representation, generated by ApiGen

See Section 5.4 for a detailed description of the Xml facilities offered by Tom.

Part III

Tools

Chapter 9

Installation

This chapter describes how to install and use the Tom system.

9.1   Requirements

Tom is a Java application written with JDK 1.4. Tom is platform independent and runs on various systems. It has been used successfully on many platforms, including Linux (Mandrake, Debian), FreeBSD, NetBSD, MacOS X, and Windows XP.

The only requirement to run Tom is to have a recent Java Runtime Environment installed (version 1.4.x or newer) and/or the Eclipse platform (3.0 or newer). In addition, Ant can be useful to compile the system and the examples:

- `Java` is available for download at java.sun.com.

- Eclipse is available for download at Eclipse

- Ant is available for download at ant.apache.org

9.2   Installing Tom

The binary distribution of Tom consists of the following directory layout:

```
tom
 +--- bin  // contains launcher scripts
 +--- doc  // contains documentation
 +--- lib  // contains Tom jars and necessary dependencies
 +--- share
      +--- contrib
      +--- tom    // contains predefined mapping
      |     +--- adt
      |     +--- c
      |     +--- caml
      |     +--- java
      +--- man
```

Only the `bin/` and `lib/` directories are required to run Tom. To install Tom, choose a directory and copy the distribution files there. This directory will be known as `TOM_HOME`.

Before you can run Tom there is some additional set up you will need to do:

- Add the bin directory to your path.

- Set the `TOM_HOME` environment variable to the directory where you installed Tom.

- JDK 1.4.2 or newer is needed. Optionally, set the `JAVA_HOME` environment variable. This should be set to the directory where your JDK is installed.

9.2.1 Windows

We assume that Tom is installed in `c:\tom\`. The following steps are needed for setting up the environment:

- set environment variable `TOM_HOME=c:\tom\`

- add `%TOM_HOME%\bin` to your PATH variable

For a detailed description of setting environment variables in Windows, please refer to Windows documentation. For Windows XP, some information can be found here:
http://support.microsoft.com/default.aspx?scid=kb;en-us;310519&sd=tech

9.2.2 Windows with Cygwin

We assume that Tom is installed in `c:\tom\`. Please issue the following for setting up Tom:

```
export TOM_HOME=/cygdrive/c/tom
export PATH=${PATH}:${TOM_HOME}/bin
```

We also suggest you to define a `TOM_LIB` variable and update your `CLASSPATH` accordingly:

```
TOM_LIB=`echo ${TOM_HOME}/lib/*.jar | tr ' ' ':'`
export CLASSPATH=${TOM_LIB}:${CLASSPATH}
```

Note: we recommend to add current directory . in your `CLASSPATH`.

9.2.3 Unix

Assuming Tom is installed in `/usr/local/tom`. The following sets up the environment:

```
(bash)
  export TOM_HOME=/usr/local/tom
  export PATH=${PATH}:${TOM_HOME}/bin

(csh)
  setenv TOM_HOME /usr/local/tom
  set path=( $path ${TOM_HOME}/bin )
```

We also suggest you to define a `TOM_LIB` variable and update your `CLASSPATH` accordingly:

```
(bash)
  TOM_LIB=`echo ${TOM_HOME}/lib/*.jar | tr ' ' ':'`
  export CLASSPATH=${TOM_LIB}:${CLASSPATH}

(csh)
  set TOM_LIB=`echo ${TOM_HOME}/lib/*.jar | tr ' ' ':'`
  setenv CLASSPATH ${TOM_LIB}:${CLASSPATH}
```

Note: we recommend to add current directory . in your `CLASSPATH`.

9.2.4 Eclipse plugin

The Tom eclipse plugin is available directly from the Eclipse platform via the update link:
http://tom.loria.fr/eclipse-update
From the Help menu, choose Help Content and look at the Tom Section. You will find there everything to use Tom in a very simple way.

9.3 Getting some examples

Many examples have been written to test or illustrate some particular behaviors. This collection of examples is available at the http://tom.loria.fr/download.php page.

The Tom and Gom compilers are written in Tom itself. The compiler is certainly the most complex program written in Tom. Most of the constructs provided by Tom are used and tested in the project itself.

9.4 Compiling Tom from sources

9.4.1 Getting the sources

The Tom and Gom sources are available through anonymous **cvs(1)**. To get them, use the commands:

```
cvs -d :pserver:anonymous@scm.gforge.inria.fr:/cvsroot/tom login
cvs -d :pserver:anonymous@scm.gforge.inria.fr:/cvsroot/tom checkout -P jtom
```

Developer access is possible via `ssh`. See Gforge for detailed instructions.

The `cvs` checkout will get a new repertory in your current working directory, called `jtom`. We will call this directory `${TOM_BASE}`.

Note: older releases of Tom are available from cvs using tags of the form `tom-2_2`.

9.4.2 Prepare for the build

First of all, please read `${TOM_BASE}/INSTALL`. Even if the current guide should be enough, this file may contain useful information.

The Tom build process uses apache ant to automate the all parts of the process. This build process is specified by the `${TOM_BASE}/build.xml` file. You will need to install `apache ant` and `java 1.4` in order to compile Tom.

The build process requires `apache ant 1.6` or more, and a `Java` environment compatible with `Java 1.4`.

Optionally, to customize the build environment, you will have to copy the template configuration file `${TOM_BASE}/local.properties_template` to `${TOM_BASE}/local.properties`, and edit the latter it to reflect your setup. The most common is to set `build.compiler=jikes` to use the `jikes` Java compiler, and `build.compiler.emacs=true` to get errors in a format the `emacs` editor can handle.

9.4.3 Build the Tom distribution

To compile the stable distribution of Tom, you have to use the `${TOM_BASE}/build.sh` script. To use it, first make sure to **cd(1)** to `${TOM_BASE}`. Then you can build the stable distribution.

```
$ ./build.sh stable
```

This creates the `stable` distribution in the directory `${TOM_BASE}/stable/dist`.

To build and install the source distribution, you have to do the following:

```
$ ./build.sh stable
$ ./build.sh src.dist
```

This creates the `src` distribution in the directory `${TOM_BASE}/src/dist`.

To list all the available targets for the build:

```
$ ./build.sh -projecthelp
```

Note: for Windows, just use "build.bat" instead of "build.sh"

9.4.4 Setup

To setup Tom after a source build, you can simply follow the configuration instructions for the binary distribution found in Section 9, using either `${TOM_BASE}/stable/dist` or `${TOM_BASE}/src/dist` for value for the `TOM_HOME` environment variable.

Chapter 10

Using Tom

10.1 The Tom compiler

The Tom compiler is a non-intrusive pattern matching compiler. It takes a Tom program, combination of a host programming language (Java, C or Caml) extended by Tom constructs. Each Tom program can define its own data structures and manipulate them in order to write pattern-matching based algorithms.

10.2 Development process

As mentioned earlier, Tom generates host language code. The command `tom` invokes the compiler on a Tom source file and generates a new source file (with `.java`, `.tom.c`, or `.tom.ml` extension, depending on the compilation command-line). The generated code has to be compiled/interpreted by a tool which can handle the corresponding host language: `javac` for Java, `cc` for C and `ocamlc` for Caml.

A typical development cycle is the following:

- edit a Tom program (by convention, a Tom source code has the `.t` extension)

- run the Tom compiler on this file

- compile the generated file (`javac file.java`, or `cc file.tom.c`,...)

- execute the program (`java file`, or `a.out`,...)

As `cc` and `javac`, `tom` can compile several input files. By default, the generated files are saved in the same directory as the corresponding input files. When compiling a Java file (i.e. a Tom file where Java is the host language), Tom is smart enough to parse the package definition and generate the pure-Java file in the same package architecture. Similarly to `javac`, `tom` supports a `-d` option which allows the user to indicate where the files have to be generated. To be compatible with `cc` and classical `Makefile`, Tom also supports a `-o` option which can be used to specify the name of the generated file.

10.3 Command line tool

tom [options] filename[.t]

The command takes only one file argument, the name of the input file to be compiled. By convention the input file name is expected to be of the form *filename.t*, whatever the used host language is.

Options:

`-X <file>`	Define an alternate XML configuration file
	Allow to define the plugins instantiated by the Tom platform
`--cCode | -c`	Generate C code (default is Java)
	The output file has the extension `.tom.c`
`--camlCode`	Generate Caml code (default is Java)

`--compile`	Activate the compiler (by default)
`--destdir | -d`	Specify where to place generated files
	By default, Tom generates files close to the input file.
	However (like `javac`), this option allows to specify a
	directory where generated files have to be put. When compiling a
	`Java` file, Tom is smart enough to parse the package definition and generate
	the pure-`Java` file in the same package architecture.
`--eclipse`	Activate Eclipse mode
`--expand`	Activate the expander (by default)
`--help | -h`	Show the help
	Give a brief description of each available options
`--import <path> | -I`	Path to included files
	Even if inclusion can be performed using relative path, this option specifies
	list of path where Tom look for when an inclusion has to be done by %include construct
`--intermediate | -i`	Generate intermediate files
	The compiler manipulates Abstract Syntax Trees.
	This option dumps the AST after each phase
	(parsing, checking, expansion, compilation)
	This option is useful to analyze the compilation process
`--jCode | -j`	Generate Java code (by default)
`--lazyType | -l`	Use universal type
	This option makes Tom using a less restrictive backend.
	In `Java`, the universal sort `Object` is used more often.
	This reduces static typing but allows to manipulate inherited data-structures
`--noDeclaration | -D`	Do not generate code for declarations
	Avoid multiple declaration of symbols when structures are inherited or when
	multiple inclusion of common data structures are performed
`--noOutput`	Do not generate code
	No output file is generated. It allows to see warnings and errors without
	generating the result
`--noSyntaxCheck`	Do not perform syntax checking
`--noTypeCheck`	Do not perform type checking
`--optimize | -O`	Optimize generated code
	Add an optimization phase to the compilation process.
	This removes unused variables and performs some inlining.
`--optimize2 | -O2`	Further optimize generated code (does not imply `-O`)
`--output | -o`	Set output file name
	By default, Tom infers the name of the generated file by replacing
	the initial file extension by `.java`, `.tom.c` or `.tom.ml`, depending on the chosen target language.
	This option allows to explicitly give the name of the generated file.

--parse	Activate the parser (by default)
--pretty | -p	Generate readable code with indentation
	By default, the generated code is synchronized with the source code.
	This simplifies error reporting but makes the code more difficult to read.
	This option beautifies the generated code.
--protected	Generate protected functions
	In Java, this option forces the generation of protected functions, instead of private ones,
	for symbol manipulation functions
--noStatic	Generate non static functions
	In Java, this option forces the generation of non static methods for symbol
	manipulation functions
--verbose | -v	Set verbose mode on
	Give duration information about each compilation passes
--version | -V	Print the version of Tom
--wall	Print all warnings
	Useful in debugging phase

10.4 Ant task

Tom provides an ant task for running the Tom compiler within the apache ant build system.

The Tom ant task is very close in use to the `javac` ant task. Since this task is not part of the official ant tasks, you have first to declare this task in your buildfile, in order to use it.

This is done by the following code:

```
<taskdef name="tom" classname="tom.engine.tools.ant.TomTask">
    <classpath refid="tom.classpath"/>
</taskdef>
```

where `tom.classpath` is a path reference containing all the jar's in Tom's distribution.

This task is used to produce Java code from Tom programs. A typical use of the tom ant task is:

```
<tom config="${stable.configfile}"
        srcdir="${src.src}"
        destdir="${src.gen}"
        options="-I ${src.mapping}">
    <include name="**/*.t"/>
</tom>
```

Here, we want to compile all Tom source files in {src.src}, having the generated code in {src.gen}, and we configure the compiler to use the {stable.configfile} config file, and pass the options we want, just as we do for Tom in command line.

Another use of this task is:

```
<tom config="${stable.configfile}"
    srcdir="${src.src}"
        outputfile="${src.gen}/tom/engine/parser/TomLanguage.g"
        options="-I ${src.mapping} -I ${src.gen}/tom/engine/adt">
    <include name="**/TomLanguage.g.t"/>
</tom>
```

Here we compile only one Tom file, and specify directly the output file name we want to use.

The main usecases for the Tom ant task can be found in Tom's own buildfile.

The Tom ant task takes the following arguments:

Attribute	Description	Required
srcdir	Location of the Tom files.	Yes, unless nested `<src>` elements are present
destdir	Location of the java files.	No
outputfile	Destination file to compile the source. This require to have only one source file.	No
config	Location of the Tom configuration file.	Yes
logfile	Which log file to use.	No
optimize	Indicates whether source should be compiled with optimization; defaults to `off`	No
stamp	Indicates whether the generated code should use stamps or not, to simulate concrete ADT. Defaults to `false`	No
failonerror	Indicates whether the build will continue even if there are compilation errors; defaults to `true`	No
options	Custom options to pass to the Tom compiler	No
verbose	Asks the compiler for verbose output; defaults to `no`	No
pretty	Asks the compiler to generate more human readable code; defaults to `no`	No
nowarn	Asks the compiler not to report all warnings; defaults to `no`	No
classpath	The classpath to use, given as a reference to a path defined elsewhere. This variable is **not** used currently.	No

Chapter 11

Using Gom

11.1 Command line tool

People interested in using Gom as a standalone tool can use the Gom command line interface.

gom [options] filename [... filename]

Options:

`-X <file>`	Defines an alternate XML configuration file
`--destdir <dir> | -d`	Specify where to place generated files
`--verbose | -v`	Display compilation information
`--debug | -vv`	Display debugging information
`--intermediate | -i`	Leave generated ADT file for all compilation phases
`--import <path> | -I`	Path for searching included modules
`--package <pack> | -p`	Specify a prefix package name (optional)
`--wall | -W`	Print warnings
`--version | -V`	Print version
`--help | -h`	Show this help

11.2 Ant task

Gom can also be used within ant, and has its own ant tasks.

To use the Gom ant task, you have to first declare it the same way as the Tom ant task.

```
<taskdef name="gom"
         classname="tom.gom.tools.ant.GomTask"
         classpathref="tom.classpath"/>
```

Once the task is defined, you can use them to compile a Gom file.

The Gom task need the user to specify the configuration file to use with the `config` attribute. The default configuration file for Gom is included in the Tom distribution as `Gom.xml`.

```
<gom config="${stable.gomconfigfile}"
     package="tom.gom.adt"
     destdir="${src.gen}">
  <include name="tom/gom/adt/*.gom"/>
</gom>
```

The Gom task takes the following arguments:

Attribute	Description	Required
config	Location of the Gom configuration file.	Yes
options	Custom options to pass to the Gom compiler.	No
package	Package for the generated **Java** files.	No
destdir	Location of the generated files.	No

The set of `.gom` files to compile is specified with a nested `include` element.

Chapter 12

Configuring your environment

12.1   Editor

12.1.1   Vim

The Tom distribution contains some filetype plugins for Tom and Gom support in Vim. They are located in `${TOM_HOME}/share/contrib/vim`.

To install them, you should put the content of the `contrib/vim` directory in `${HOME}/.vim/`. Then in order for these plugins to be automatically used whenever it is necessary, you can put in your `${HOME}/.vimrc`:

```
" indentation depends on the filetype
filetype indent on
filetype plugin on

autocmd BufNewFile,BufRead *.t   setfiletype tom
autocmd BufNewFile,BufRead *.tom setfiletype tom
autocmd BufNewFile,BufRead *.gom setfiletype gom
```

For Tom developers, the preferred setup for indenting Tom and `Java` code is as follows:

```
autocmd FileType ant    set expandtab
autocmd FileType ant    set sw=2

" automatically indent tom and java code
autocmd FileType tom    set cindent autoindent
autocmd FileType java   set cindent autoindent
autocmd FileType tom    set encoding=utf-8
autocmd FileType tom    set fileencoding=utf-8
autocmd FileType java   set fileencoding=utf-8

" how to indent: in java and tom, 2 spaces, no tabs
autocmd FileType java,tom,tex   set expandtab
autocmd FileType java,tom,tex   set sw=2
autocmd FileType java,tom,tex   set tabstop=2
autocmd FileType java,tom,tex   set nosmarttab
```

Compiling Tom under Vim

To compile Tom programs (like Tom) under Vim, and get a correct error reporting, you will have to set the `${CLASSPATH}` environment value to the path of `junit.jar`, and use the `makeprg` variable to have `:make` call ant through a script maintaining the link between Tom and the generated `Java` files.

```
let $CLASSPATH = "\$\{HOME\}/workspace/jtom/src/lib/junit.jar"
set makeprg=ant\ -find\ build.xml\ -emacs\ $*\\\|\
           \ awk\ -f\ \$\{HOME\}/workspace/jtom/utils/ant-tom.awk
```

12.2   Shell

12.2.1   Zsh

Zsh completion functions for Tom and Gom are available in the `share/contrib/zsh` directory. To install them, you only have to add those files to zsh's `fpath` variable.

For example, assuming you already set up the `TOM_HOME` environment variable to the Tom installation directory, you can add to your `${HOME}/.zshrc`:

```
fpath=(${TOM_HOME} $fpath)
```

12.3   Build Tom projects using Ant

To build complex projects using Tom, it is useful to use Ant as build system, to manage generation of data structure using Gom and the compilation of Tom and `Java` code.

To ease the use of Ant, the file `${TOM_HOME}/lib/tom-common.xml` is provided in the Tom distribution. This file provide initialization for Tom and Gom Ant tasks. To load `tom-common.xml`, you just have to put in your Ant script:

```
<property environment="env"/>
<property name="tom.home" value="${env.TOM_HOME}"/>
<import file="${tom.home}/lib/tom-common.xml"/>
```

Then, each target depending on the `"tom.init"` task will allow the use of the Gom and Tom ant tasks, as well as `tom.preset` and `javac.preset`, providing usual values for the attributes of those tasks. Also, `tom-common.xml` provides several properties, as `tomconfigfile` and `gomconfigfile`, providing the location of the configuration files for Tom and Gom, and `tom.classpath`, containing all classes related to the Tom installation.

Chapter 13

Migration Guide

This chapter documents important changes in the Tom language that impact running code. Thus, it exposes how Tom code for an older version should be changed to be able to compile and work under a newer version.

13.1   Migration from 2.2 to 2.3

Mapping definition and term notation

To avoid ambiguities, since version 2.3, constants should be written using explicit parentheses: `a()`, `nil()`, for example. To be coherent, the parentheses have also to be used in the mapping definition:

```
%op T a() {
  ...
}
```

Static functions

Since version 2.3, Tom generate static functions in `Java`, instead of object methods. This allows to use Tom constructs in the `main()` function, this improves the efficiency of the generated code, and this makes safer the use of strategies.

To be compatible with the previous version, the `--noStatic` flag can be used.

From Vas to Gom

In the future, Gom will replace VAS. To make the migration easier, the Gom compiler accepts the VAS syntax. However, there are some minor changes:

- in Gom the importation of builtin modules (`int`, `String`,...) should be explicit,

- when using the new ML-like syntax, the defined sorts should no longer be declared,

- factories are no longer needed by the generated code. As a consequence, `Java` imports and constructors are simpler,

- the methods to get and set values of subterms are named after the slot name: for a slot named `slot` of type `SlotType`, the access and modification are `getslot()` and `setslot(SlotName value)`. The `String getName()` function is now `String symbolName()`, and there is no more `int getArity()`.

- The methods `boolean isOperator()` now use the exact name of the operator, without capitalizing it.

- when defining a list-operator, Gom automatically generates Java access functions. `getHead`, `getTail`, and `isEmpty` are replaced respectively by `getHeadconc`, `getTailconc`, and `isEmptyconc` for a list-operator named `conc`.

- objects generated by Gom do no longer implement the ATerm interface. Therefore, some functionalities have disappear. In particular, `genericCollect` and `genericTraversal` can no longer be used. The strategy language has to be used instead.

- to implement strategies on a module `Module`, the class `ModuleBasicStrategy` replaces the `ModuleVisitableFwd` of VAS.

13.2 Migration from 2.1 to 2.2

Mapping definition

Since version 2.2, the mapping definition formalism has been simplified. Therefore, we are no longer able to be fully compatible with the old syntax. Some small and simple modifications have to be performed:

- in `%op`, the slot-names are no longer optional. For each argument of a constructor, a name has to be given. This name can then be used within the "bracket" notation.

  `%op T f(T,T)` has to be replaced by `%op T f(arg1:T,arg2:T)`, where `arg1` and `arg2` are slot-names.

- for each slot-name, a `get_slot(name,t)` construct has to be specified. This defines, given an object `t`, how to access to the slot-name named `name`.

  The definition of `get_slot` can be defined by using the previously defined construct `get_subterm` (the first slot corresponds to the $0 - th$ subterm, second slot to the $1 - st$ subterm, *etc.*). Thus, for the previous example, we can have:

```
%op T f(arg1:T, arg2:T) {
  get_slot(arg1,t) { get_subterm(t,0) }
  get_slot(arg2,t) { get_subterm(t,1) }
}
```

- for each operator (`%op`, `%oplist`, and `%oparray`), the `is_fsym(t)` predicate has to be defined. This function should return `true` when `t` is rooted by the considered symbol, and `false` otherwise. In practice, the constructs can be deduced by combining `get_fun_sym`, `cmp_fun_sym`, and `fsym`:

```
%op T f(arg1:T, arg2:T) {
  is_fsym(t) { cmp_fun_sym(get_fun_sym(t), body of the old fsym construct) }
  get_slot(arg1,t) { get_subterm(t,0) }
  get_slot(arg2,t) { get_subterm(t,1) }
}
```

- in `%oplist`, get_head, `get_tail`, and `is_empty` have to be defined. In practice, these definitions have to be copied from `%typelist` to `%oplist`

- in `%oparray`, get_element, and `get_size` have to be defined. In practice, these definitions have to be copied from `%typearray` to `%oparray`

- `%typelist`, and `%typearray` do no longer exist: `%typeterm` has to be used to define any kind of sort. This is not a problem since access functions are now defined inside operator definitions (`%op`, `%oplist`, or `%oparray`)

- get_fun_sym, cmp_fun_sym, and get_subterm do no longer exist in `%typeterm`

- `fsym` do no longer exists in `%op`, `%oplist`, and `%oparray`

As an example, let us consider the following "old" mapping definition:

```
%typeterm TomTerm {
  implement          { ATermAppl        }
  cmp_fun_sym(t1,t2) { t1 == t2         }
  get_fun_sym(t)     { t.getName()      }
  get_subterm(t, n)  { t.getArgument(n) }
  equals(t1, t2)     { t1 == t2         }
}

%typelist TomList {
  implement          { ATermList                    }
  get_fun_sym(t)     { ((t instanceof ATermList)?"conc":null) }
  cmp_fun_sym(t1,t2) { t1 == t2                      }
  equals(l1,l2)      { l1 == l2                      }
  get_head(l)        { (ATermAppl)l.getFirst()       }
  get_tail(l)        { l.getNext()                   }
  is_empty(l)        { l.isEmpty()                   }
}

%op TomTerm a {
  fsym   { "a" }
  make() { factory.makeAppl(factory.makeAFun("a", 0, false)) }
}

%op TomTerm f(TomTerm) {
  fsym     { "f" }
  make(t1) { factory.makeAppl(factory.makeAFun("f", 1, false),t1) }
}

%oplist TomList conc( TomTerm* ) {
  fsym            { "conc"           }
  make_empty()    { factory.makeList() }
  make_insert(e,l) { l.insert(e)       }
}
```

The two first type definitions have to be replaced by:

```
%typeterm TomTerm {
  implement      { ATermAppl }
  equals(t1, t2) { t1 == t2  }
}

%typeterm TomList {
  implement      { ATermList }
  equals(l1,l2) { l1 == l2  }
}
```

The operator definition can be replaced by the following code:

```
%op TomTerm a {
  is_fsym(t) { t.getName() == "a" }
  make()     { factory.makeAppl(factory.makeAFun("a", 0, false)) }
}
```

```
%op TomTerm f(arg:TomTerm) {
  is_fsym(t)      { t.getName() == "f" }
  get_slot(arg,t) { (TomTerm) t.getArgument(t,0) }
  make(t1)        { factory.makeAppl(factory.makeAFun("f", 1, false),t1) }
}

%oplist TomList conc( TomTerm* ) {
  is_fsym(t)        { t instanceof ATermList }
  get_head(l)       { (ATermAppl)l.getFirst() }
  get_tail(l)       { l.getNext() }
  is_empty(l)       { l.isEmpty() }
  make_empty()      { factory.makeList() }
  make_insert(e,l)  { l.insert(e) }
}
```

Disjunction of patterns

Since version 2.2, the disjunction of patterns construct has become deprecated. It can still be used, but we recommend to use the disjunction of symbols instead. This last construct allows to share patterns which have similar subterms but different head-symbols. Therefore, the construct `g(g(a)) | g(h(a))` can be replaced by `g( (g | h)(a) )`. Since version 2.2, this construct has been extended to support disjunction of symbols which do not have identical signature: only involved slots have to be compatible.

Considering `%op T f(arg1:T, arg2:T)` and `%op T g(arg1:T)`, it is now possible to write `g( (g | f)[arg1=a] )`.

Variable

To prepare a future extension where variables have to be distinguished from constructors of arity 0, we have turned into error all ambiguities between variables and constructors. As a consequence, a variable cannot any longer have the same name as a constant.

Vas

Since version 2.2, APIGEN and VAS have been updated:

- name of list-operators defined in VAS are now significant and can be used in `match` constructs.

- generated Tom mappings are now stored in the same directory as the generated factory. In practice, imports of the form `%include{ file.tom }` have to be replace by `%include{ package/file.tom }`.

Strategy library

The `TravelerFactory` class has been removed and replaced by a Tom mapping: `mutraveler.tom`. Therefore, backquote notation has to be used: `travelerFactory.Repeat(travelerFactory.OnceBottomUp(rule))` is replaced by `'Repeat(OnceBottomUp(rule))`.

13.3   Migration from 2.0 to 2.1

Tom library

Since version 2.1, the runtime library has been reorganized. Java importations have to be updated according to the following hierarchy:

```
tom
 |--- library
 |        |--- adt              (datatype definitions)
 |        |--- mapping          (predefined mapping)
 |            |--- adt          (generated mapping)
 |        |--- plugin           (platform tools)
 |        |--- set              (to handle sets)
 |        |--- strategy
 |            |--- concurrent   (to support parallel strategy)
 |        |--- traversal        (generic traversal)
 |        |--- xml              (xml tools)
 |
 |--- platform                  (the Tom Plugin Platform, known as Tom server)
          |--- adt
```

Command line options

- `--protected` is a new option which makes Tom generates protected access functions. By default, access functions are now private instead of public,

- `--noWarning` has been replaced by its dual `--wall` (for printing warnings).

13.4   Migration from 1.5 to 2.0

New backquote usage

Since version 2.0, Tom integrates an optimizer which removes unused variable assignments and performs inlining optimizations. In order to make these optimizations correct, the Tom compiler needs to know whether a Tom variable (instantiated by pattern matching) is used or not.

In previous versions of Tom, it was possible to use or re-assign a Tom variable without any particular requirement. Since version 2.0, it is recommended (and needed if you want to use the optimizer) to use the backquote mechanism to retrieve the value of a Tom variable. The syntax of a backquote construct has been modified in the following way:

- `‘Name` (like `‘x`) is now correct. This allows access to the value of a Tom variable. This construct also allows access to the value of list-variable, but the `‘Name‘*`' construct is to be preferred,

- `‘Name‘*`' (like `‘X*`) is now correct. This allows access to the value of a Tom list-variable,

- `‘Name( ... )` is correct, as in previous versions. This construct builds a term in memory,

- `‘( ... )` is now correct and can be used to simulate the previous syntax. Since version 2.0, it is no longer possible to start a backquote construct by using something different from a `Name` or a `‘(`. Thus, `‘1+x` is no longer valid: `‘(1+x)` or `1+‘x` has to be used instead,

- `‘xml( ... )` has to be used to create an XML term.

Command line options

- `--destdir` (short version: `-d`) option: as in `javac`, this option specify where to place generated files,

- `--output` (short version: `-o`) option: as in `gcc`, this option specify the name of the *unique* generated file. This option is not compatible with `--destdir`,

- `-o` is no longer the short version of `--noOutput`.

Predefined mapping

Since version 2.0, Tom does no longer support predefined builtin sorts, via `%typeint`, `%typestring` or `%typedouble`. Tom comes now with a library of predefined mappings. Among them, `int.tom`, `string.tom` and `double.tom`. In order to use these mappings, the `%include` construct should be used (i.e. `%include{ int.tom }` to import the predefined `int` mapping).

Part IV

Cookbook

Chapter 14

Strategies

14.1 Retrieving information from trees

Problem:
We want to retrieve all subtrees that are instances of one type.
Solution:
Strategies can browse trees in several ways.
Example:
We seek all chairs in the house.

Here is the `%gom` datatype definition:

```
module House
imports String

abstract syntax

House = concRoom(Room*)
Room = room(name: String, furniture: Furniture)
Furniture = concPieceOfFurniture(PieceOfFurniture*)
PieceOfFurniture = bed()
                 | chair()
                 | armchair()
                 | fridge()
```

Here the corresponding import:

```
import tutorial.cookbook.house.*;
import tutorial.cookbook.house.types.*;

import tom.library.strategy.mutraveler.MuTraveler;
import tom.library.strategy.mutraveler.Identity;
import jjtraveler.reflective.VisitableVisitor;
import jjtraveler.Visitable;
import jjtraveler.VisitFailure;
```

Don't forget to include the `mutraveler` library:

```
%include { mutraveler.tom }
```

We build a *House*:

```
House myHouse =
  `concRoom(room("kitchen",concPieceOfFurniture(chair(),chair(),fridge())),
           room("bedroom",concPieceOfFurniture(bed(),chair())));
```

We create a elementary strategy that seeks all chairs in there. This strategy extends 'Identity(), because we want to go through even if a failure happens. In other cases, we'd rather use 'Fail() to stop if we encounter a failure.

```
%strategy SeekChairs() extends 'Identity() {
  visit PieceOfFurniture {
    chair() -> {
      System.out.println("1 Chair !");
    }
    armchair() -> {
      System.out.println("1 Armchair !");
    }
  }
}
```

Here we visit the tree using a BottomUp strategy:

```
try {
  MuTraveler.init('BottomUp(seekChairs)).visit(myHouse);
} catch (VisitFailure e) {
  System.out.println("failed on = " + myHouse);
}
```

14.2 Modifying information on trees

Problem:
We want to modify information on trees.
Solution:
We use a strategy that modifies the inspected node.
Example:
We want to replace chairs by armchairs.

To do so, we use a strategy that replaces chairs by armchairs. We use the same datatype definition as in the previous example.

```
%strategy ReplaceChairs() extends 'Identity() {
  visit PieceOfFurniture {
    chair() -> { return 'armchair(); }
  }
}
```

Here we browse the tree with a BottomUp strategy:

```
try {
  myHouse = (House)MuTraveler.init('BottomUp(replaceChairs)).visit(myHouse);
} catch (VisitFailure e) {
  System.out.println("failed on = " + myHouse);
}
```

Chapter 15

Using Gom generated structures

15.1   Extending Gom generated structures

Problem:
I want to use Gom to generate a data structures, but I also need to use my own hand made containers
for this structure, and I do want to use strategies to traverse the resulting trees.
Solution:
You have to extend the `BasicStrategy` provided by Gom to add support for your own containers. Then,
you will be able to build strategies traversing both Gom and your containers by combining visitors.
Example:
We define in Gom how to describe points and lines:

```
module Plan
imports int
abstract syntax
Point = Point(abs:int, ord:int)
Comp = Line(p1:Point,p2:point)
     | Triangle(p1:Point,p2:Point,p3:Point)
```

And we want to add a `Carre` class, composed of two `Line`, but without using Gom. This class must
implement the `jjtraveler.Visitable` interface to allow the use of strategies:

```
import plan.types.*;
public class Carre implements jjtraveler.Visitable {
  public Line l1;
  public Line l2;
  Carre(Line l1, Line l2) {
    this.l1 = l1;
    this.l2 = l2;
  }
  /* implements the Visitable interface */
  public int getChildCount() { return 2; }
  public jjtraveler.Visitable getChildAt(int index) {
    switch(index) {
      case 0: return l1;
      case 1: return l2;
      default: throw new IndexOutOfBoundsException();
    }
  }
  public jjtraveler.Visitable setChildAt(int index, jjtraveler.Visitable v) {
    switch(index) {
      case 0: l1=(Line)v; return this;
```

```
      case 1: l2=(Line)v; return this;
      default: throw new IndexOutOfBoundsException();
    }
  }
}
```

The last step to be able to define strategies over both `Carre` and the Gom structure is to define a new `BasicStrategy`, extending the one generated by Gom.

```
public class CarreBasicStrategy extends plan.PlanBasicStrategy {
  public CarreBasicStrategy(jjtraveler.reflective.VisitableVisitor v) {
    super(v);
  }
  public jjtraveler.Visitable visit(jjtraveler.Visitable v)
                                     throws jjtraveler.VisitFailure {
    if (v instanceof Carre) {
      return this.visit_Carre((Carre)v);
    } else {
      return super.visit(v);
    }
  }
  public Carre visit_Carre(Carre arg) throws jjtraveler.VisitFailure {
    return (Carre) any.visit(arg);
  }
}
```

It is then possible to define a strategy over the composed structure by extending this `CarreStrategy`. See the `TestCarre.t` file in the `gom` example from the Tom distribution for a running example.

Chapter 16

Encoding rewrite systems

16.1 Transition systems

Problem:
Encode a simple transition system and compute normal forms.
Solution:
Use the *hook* mechanism provided by Gom.

16.2 Computing all reachable terms

Problem:
Given a rewrite system and a term, compute the immediate successors.
Solution:
This cannot be done using *hooks*, use a strategy instead.

Suppose we have the following Gom definition:

```
%gom {
  module Term
  abstract syntax
  Term = a()
       | b()
       | s(pred:Term)
       | f(arg1:Term)
}
```

Given a rewriting system s, we want to construct all the terms generated by s. In Tom, the solution consists in defining the rewriting system by a basic strategy `%strategy` and apply it to each subterm using the *MuTraveler* operator `BottomUp`. All possible terms reachable in one step are collected in a collection.

```
public void collectOneStep(final Collection collection, Term subject) {
 try {
  MuTraveler.init('BottomUp(OneStep(subject,collection))).visit(subject);
  System.out.println(collection);
 } catch (VisitFailure e) {
   System.out.println("Failed to get successors " +subject);
 }
}

%strategy OneStep(subject:Term,c:Collection) extends 'Identity() {
  visit Term {
    f(x)    -> { c.add(MuTraveler.getReplace(this,'f(s(x))).visit(subject)); }
```

```
    f(s(x)) -> { c.add(MuTraveler.getReplace(this,'f(f(x))).visit(subject)); }
    s(s(x)) -> { c.add(MuTraveler.getReplace(this,'f(x)).visit(subject)); }
  }
}
```

Notice that we need to use the `MuTraveler` static method `getReplace` that replaces at the current position (given due to the parameter `this` corresponding to the current strategy) with the term given in the second parameter (for example `f(s(x))` in the first rule). For more explanations, see 7.2.

Moreover, to use a `Java Collection` as an algebraic sort, we need to add the following code:

```
%typeterm Collection {
   implement { Collection }
}
```

From now, we have just a collection of reachable terms in one step. If we want to know if a term is reachable in an undetermined number of steps, we define the function `reach(Term start,Term end)`:

```
public boolean reach(Term start,Term end) {
    Collection result = new HashSet();
    Collection predResult = new HashSet();
    Collection c1 = new HashSet();
    c1.add(start);
    while (result.isEmpty() || predResult.containsAll(result)) {
      Collection c2 = new HashSet();
      Iterator it = c1.iterator();
      while(it.hasNext()) {
        collectOneStep(c2,(Term)it.next());
      }
      c2.removeAll(result);
      c1 = c2;
      predResult.addAll(result);
      result.addAll(c2);
      if(result.contains(end)) {
        return true;
      }
    }
    return false;
  }
```

This function can loop if the system s is not terminating or if no term is reachable from `start`.