Generating Term Rewriting Systems with Copster

Nicolas Barré

December 17, 2008

Contents

This document is the reference documentation of *Copster*. For any question or comment you can send an email to Nicolas.Barre@irisa.fr[1].

1 Preamble

Copster is a language designed to easily manipulate *terms* in order to generate *rewriting rules*. In *Copster*, the data structures are only *terms* and *lists*.

In the *rewriting theory*, a *term* is either an *operator* with a null *arity*, either a *variable* intended to substitute any *term*, or an *operator* with a non null *arity* containing other *terms*. The use of *variables* makes sens only in *rewriting rules*. For instance, if we consider the number '1' expressed in peano notation i.e. `succ(zero)`, `zero` is an *operator* of null arity and `succ` is an *operator* of arity 1. Then we can define some *rules* describing the addition between two natural integers :

- `add2(X,zero) => X`
- `add2(X,succ(Y)) => succ(add2(X,Y))`

where `X` and `Y` are *variables* and `=>` defines a rewriting relation between the left and right sides of the *rule*.

Copster allows to create *terms* without writing them down explicitly. For instance if we want to consider the number '100' in Peano notation, we'd like to find a better way than writing `succ(succ(succ(...(zero))))` in a file... With *Copster* we simply do :

```
set x = zero;
for i from 1 to 100
do(
  set x = succ($x);
);
```

Then we can assert that `x` contains the requested value.

Copster was first designed to generate *rewriting systems* from a *Java* byte code program, in order to model its execution on the *Java Virtual Machine* in *rewriting logic*. The generated *rewriting systems* depend on a semantics for the *JVM* and on the program itself. The semantics of the programming language is defined through a set of generic *Copster* rules. Then, starting from those rules and a given byte code program, *Copster* produces a *term rewriting system* encoding the complete execution of the program in rewriting logic. After that, the execution of the program can be simulated in a rewriting tool.

Generally, in the programs designed to manipulate rewriting logic, the *operators* are given a type. For instance, in Maude[2] we should declare `zero`, `succ` and `add2` as follows :

[1]mailto:Nicolas.Barre@irisa.fr
[2]see the [[http://maude.cs.uiuc.edu/][Maude project]]

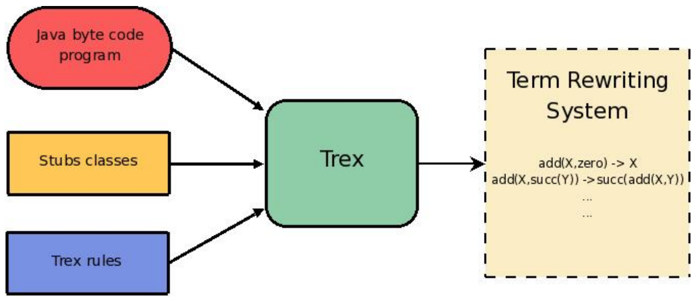

Figure 1: Copster principle

```
sort Natural .
op zero : -> Natural
op succ : Natural -> Natural
op add2  : Natural Natural -> Natural
```

In *Copster*, we can't specify a type for the *operators* we define. Thus every *operator* is a function taking `Term` type arguments and returning a `Term`. In counterpart, we don't need to declare the *operators* in *Copster*, they are inferred.

The operations we can currently do on *terms* with *Copster* are very basic. It's possible to :

- get the *arity* of the root *operator* of a *term*
- get a direct *subterm*
- substitute a direct *subterm* with another
- search for a specific direct *subterm* inside a *term* and get its position if it's found
- get the depth of a *term* according to its root symbol

We define the syntax and see examples of use of those operations in the further sections.

Now let us define the bases of the *Copster* language and see how we can ease the description of a semantics through templates of *term rewriting systems*.

2 Language bases

This section describes the syntax of the *Copster* language, and provides sample codes.

2.1 Term creation and manipulation

Copster aims to generate *terms rewriting systems*, which means a set of *operators*, *variables* and *rewriting rules*. As we introduced it in the preamble, *operators* and *variables* don't need to be declared. When writing a *term*, the *operators* and *variables* which appear for the first time are registered in the system. Then when encountering an *operator* again, a consistency check is done on the arity. It's possible to store *terms* in *Copster* variables in order to build other *terms* or to apply transformations on them.

#var

2.1.1 Explicit construction

```
set x = zero;
set y = succ($x);
set z = add2(var(x),$x);
```

This simple sample code shows several features. The first instruction sets the *Copster* variable `x` to the *term* `zero`. Then we set `y` to `succ(zero)`. The `$` character is used to refer to a *Copster* variable. Finally, `z` expresses the addition between a *rewriting system variable* and the *Copster* variable `x`. The generated *term* looks like `add2(X,zero)`. In the following, we distinguish between *Copster* variables like the ones above and *rewriting system variables*. In the last instruction we can see that *rewriting system variables* are boxed by the `var` keyword. Otherwise they are understood as *operators*.

2.1.2 Sequences

A sequence is a *list* of *terms*. A *list* is a *term* without a root symbol. `list(zero,zero,succ(zero))` is a *term*, `(zero,zero,succ(zero))` is a *list* of *terms* i.e. a sequence.

There are three predefined ways to build sequences :

- `cseq(zero,3)` builds the constant sequence `(zero,zero,zero)`
- `seq(n,3)` builds the iterative sequence `(n0,n1,n2)`
- `useq(u,3)` builds the iterative unsigned sequence `(u0,u1,u2)` where `u0`, `u1` and `u2` are first defined with arity `-1`. It means that any next instruction that will try to give another *arity* to this *operator* will succeed. We will see an example soon.

2.1.3 Getting a subterm

Getting a *subterm* from a *term* or *list* can be achieved thanks to `getn` primitive.

```
set x = seq(a,3);
set y = getn($x,3);
```

puts `a2` in `y`.

2.1.4 Making substitutions

It's often useful to get a copy of a *term* where one or more *subterms* have been replaced by others. This is what we call substitutions.

```
set x = cseq(zero,3);
set y = subsn($x,1,succ(zero));
```

puts `(zero,zero,zero)` in `x` and `(succ(zero),zero,zero)` in `y`.

#symboldepth

2.1.5 Getting terms length and symboldepth

The length of a *term* is it's number of direct *subterms*. The length of a *list* is it's number of elements. The expression `len((succ(zero),zero))` is evaluated to 2.

Starting from the root symbol `f` of term `t`, `symboldepth` computes the length of the longest path in `t` where each node is labelled by `f`. Invoking `symboldepth` on a *list* makes no sense. `symboldepth` returns at least 1 for any *term*. The `symboldepth` of `cons(a,cons(succ(succ(succ(zero))),cons(c,nil)))` is 3.

2.1.6 Incrementing a term

```
set x = a0;
set y = add($x,3);
```

defines `y = a3`. Obviously the same primitive can be used to make the addition of two integers.

2.1.7 Retrieving the position of a subterm

```
set x = (zero,succ(zero),succ(succ(zero)),succ(zero));
set r = mem(succ(zero),$x);
```

At the end, `r` is 4 because the `mem` primitive browses a *term* until its end. When looking for a *subterm* which is not present, the result of `mem` is the *term* `val_false`.

#concat

2.1.8 Concatenation constructions

There are two concatenation primitives proposed by *Copster* :

- `op(add2,(zero,succ(zero)))` builds the *term* `add2(zero,succ(zero))`
- `concat(a,(d,d2,(zero,succ(zero))))` builds the same *term* and declares `d` and `d2` as *operators* of null *arity*, by inference on the second argument

Indeed, `op` takes as first argument a root identifier and try to concatenate the second argument, whereas the second argument of `concat` is necessarily a *list* whose elements will be concatenated one by one to the first argument.

It's possible to write many inconsistent forms with these constructions, but errors will be thrown when evaluating the *terms*. For instance, `concat(add2,((zero),(zero)))` should generate `add2(zero)(zero)` which has no meaning and raises an error.

The difference between the first argument of `op` and `concat` is that `op` can also take a *Copster* variable instead of an explicit root identifier. In that case, the *Copster* variable must refer to an unsigned *operator* to avoid errors :

```
set s = useq(u,3);
set x = getn($u,1);
set y = getn($u,2);
set r1 = op($x,(zero));
set r2 = op($y,(zero,zero));
```

works, and sets `r1` to `u0(zero)` and `r2` to `u1(zero,zero)`.

Whereas

```
set x = u0;
set r = op($x,(zero));
```

doesn't work because `u0` is declared with arity 0 by the first instruction and `op` tries to give it arity 1.

2.1.9 Merging two lists

The `append` primitive builds the concatenation of two *lists*. For instance, `append((zero,zero),(succ(zero)))` is the *list* `(zero,zero,succ(zero))`.

2.2 Rules generation

Copster is an interpreter of *copster* source code which holds basic data structures during code processing :

- a list of declared *operators* with their *arity*
- a list of declared *variables*
- a stack of defined *Copster* variables associated to their value (this stack structure allows to handle the scoping)
- a list of *rewriting rules*

We'll discuss later about the scoping of *Copster* variables, however it's important to notice that *operators*, *variables* and *rewriting rules* can only been added and never deleted from *Copster* held data structures.

There is a single instruction to generate a *rewriting rule* :

- `genrule(add2(var(x),zero),var(x))` generates the *rule* `add2(X,zero) => X`.

The `genrule` primitive takes as arguments any expressions that can be evaluated to a *term*.

2.3 Imperative constructions

Until now we've had an overview of *Copster* showing instructions separated by semi-colons in a very imperative way. Indeed, *Copster* is mainly imperative and furthermore doesn't allow recursive constructions. Fortunately *Copster* handles iterations on variables or sequences and comparisons between *terms*.

2.3.1 Iterations

Here is an example of a `for` loop :

```
set x = cseq(zero,3);
set y = zero;
for i from 1 to len($x)
do(
  set y = succ($y);
  set x = subsn($x,$i,$y);
);
```

At the end of the `for` loop, `x` has the value `(succ(zero),succ(succ(zero)),succ(succ(succ(zero))))`

It's also possible to iterate on sequences :

```
set x = ((add2(var(x),zero),var(x)),(add2(var(x),succ(var(y))),succ(add2(var(x),var(y))
for r in $x
do(
  genrule(getn($r,1),getn($r,2));
);
```

2.3.2 Conditional statements

In *Copster*, everything is considered true except the `val_false` *term*.

```
set x = (zero,succ(zero),succ(zero));
for i from 1 to len($x)
do(
  if (getn($x,$i) = zero) then
  (
    genrule(pos($x,$i),val_true);
  ) else
  (
    genrule(pos($x,$i),val_false);
    );
  );
```

generates the rules :

- `pos((zero,succ(zero),succ(zero)),1) => val_true`
- `pos((zero,succ(zero),succ(zero)),2) => val_false`
- `pos((zero,succ(zero),succ(zero)),3) => val_false`

The `else` clause is optional and can be replaced by a semi-colon.

Inside a conditionally statement, the following comparison operators are allowed : `=`, `<`, `>`, `<=` and `>=`.

Furthermore, in conditional statements, logical operators like `&&` and `||` are allowed and the expressions are evaluated according to their writing order. For instance, the expression `val_true && val_false || val_true` is false, and the expression `val_true && (val_false || val_true)` is not allowed by the language syntax. Such expressions can be boxed by a `not` primitive.

#or However, if necessary, such conditions can be encoded before an `if` statement as follows :

```
set x = and(val_true,or(val_false,val_true));
if ($x) then (
  ...
);
```

2.3.3 Terms modifications

We've seen in former sections that we could create new *terms* from an existing one by getting a direct *subterm* or making substitutions. There also exists a primitive `setn` to substitute a *subterm* in place.

```
set x = (a,b,c,d);
setn (x,1) = e;
```

At the end, `x = (e,b,c,d)`. The last instruction is equivalent to `set x = subsn($x,1,e);` but is more concise.

2.4 Copster variables scoping

In *Copster*, the scoping is lexical, except for variables contained in function definitions that we'll see next. The instruction `let ... in` allows to build a new context which is destroyed at the end.

```
set x = 1; (* defines x = 1 in the toplevel *)
let x = 3 and y = 2 in (
  genrule(numpred(x),y);
);
```

generates the rule `numpred(3) => 2`. At the end `x = 1` and `y` doesn't exist any more.

The instruction `set x` defines a new *Copster* variable `x` in the toplevel, only if `x` is not already defined in any enclosing context. Otherwise, a bottom-up lookup is processed through the context stack and the first occurrence found of `x` is modified.

2.5 Functions

We don't distinguish between functions and procedures in *Copster* meaning that every expression of the language must be an instruction and vice versa. That's why we need to define a result expression for instructions which don't return a value. Like in *Lisp* dialects, it is the empty *list* `()`. A function returns the expression associated to its last instruction.

```
defun f(a) =
(
  let b = add($a,1) in
  (
    $b;
  );
);
```

returns `a + 1`.

As we mentioned it in the previous subsection, the scoping is dynamic for functions. Indeed the syntax of a function is checked during it's definition but its content is interpreted only during the function call. Calling a function is done the same way than referring to variables except you have to provide the list of parameters.

```
defun f(a) =
(
  let b = add($a,$c) in
  (
    $b;
  );
);

set c = 5;
set r = $f(2);
```

This sample works even if `c` isn't defined before the function definition, and `r = 7` at the end.

3 Environment imports

An environment is a set of *operators*, *variables*, *Copster* variables and *rewriting rules*. Thus importing an environment means merging such a set with the current environment. An environment can come from an other *Copster* source file (e.g. modules), or can be natively defined in *Copster* in order to serve any purpose. There currently exists only one kind of native import in *Copster*, meeting our first needs on *Java* byte code programs.

3.1 Modules

Copster allows modular programming by splitting code into multiple files. a module may include another, otherwise module environments are totally isolated.

```
(* file definitions.rex *)
set x = a0;
set y = (succ(zero),zero);

(* file main.rex *)
load ./definitions.rex
genrule($x,$y);
```

Relative paths are allowed.

Loading a module is merging its environment with the current one, it's to say adding the *Copster* variables, *operators*, *variables* and *rules* which are not already defined in the current environment. If one of the imported elements is inconsistent according to the current environment, an appropriate error is raised.

3.2 The Java byte code Environment

Importing an environment built from a *Java* byte code program is done by invoking the primitive `import java_bytecode`. The program name and location are not written in *Copster* source file but are given as parameters to *Copster* command line as shown in 3.2.5 section.

The aim is to express a *Java Virtual Machine* semantics in *Copster*. That's why we have to import an environment containing everything we could need in order to write this semantics. For instance, we have to know classes names, methods names, fields names and a lot of information associated to them.

3.2.1 Naming conventions

First, we need to define a naming convention in order to avoid collisions between program symbols, coming from a Java byte code program, and the *operators* defined by the user in a *Copster* source file. By convention, all *operators* coming from a *Java* program are bracketed with <>. For instance, the class `java.lang.Object` will create three *operators*, <`java`>, <`lang`> and <`Object`>. Thus by convention if you don't refer to a program symbol and you want to avoid collisions you mustn't name your *operators* with that kind of brackets.

Copster is able to export the generated *rewriting systems* in Timbuk and Maude formats. However those formats don't allow the use of characters < and >, that's why *Copster* relies on a renaming process to generate valid output. The renaming algorithm is very simple, it consists in suppressing the brackets and if a collision is found, in incrementing a counter concatenated to the `operator` symbol until collisions are resolved.

3.2.2 Quick semantic overview

The *Java* imported data structures depends on the semantics we chose to describe classes, methods, fields and their attributes. So let us present it quickly.

Consider the following class in the package `java/lang` :

```
class String extends Object{
    public char charAt(int i){...};
    public int length(){...};
    public String substring(int){...};
  ...
}
```

The class `java.lang.String` is represented by the *term* `ConsName(<java>, ConsName(<lang>, ConsName(<String>, NilName)))`.

The methods are represented by the *terms*

- `Method(<charAt>, ConsType(TInt, NilType), TChar)`
- `Method(<length>, NilType, TInt)`
- `Method(<substring>, ConsType(TInt, NilType), OType(ConsName(<java>, ConsName(<lang>, ConsName(<String>, NilName)))))`

The `ConsName` *operator* is used to build a class name qualified with its package name, the *operator* `ConsType` represents a list of types.

Moreover we also define a `Field` constructor whose arguments are the class name, the field name and the field type. For instance, `Field(ConsName(<A>,NilName),<x>,TInt)` represents a field `x` of type `int` in a class `A`.

There exist other *operators* to represent basic types : `TShort`, `TBool`, `TDouble`, `TFloat`, `Tlong`, `TByte`, `void`.

The classes, methods and fields attributes are represented by the *operators* `AccDefault`, `AccPublic`, `AccPrivate`, `AccAbstract`, `AccNative`, `AccStatic` and `AccSynchronized`.

3.2.3 Imported data structures

The imported data structures are *terms lists* contained in *Copster* variables. We don't need to import *rewriting rules* because these structures provide all the information required to build the *rules* we want to express.

Here are the imported *Copster* variables :

- `max_locals` contains the maximum size of local variables arrays
- `max_pc` contains the maximum number of instructions among all the methods defined in the considered program
- `insts` is a *list* containing every *Java* byte code instruction at every program point in every method defined in every existing class : `((ConsName(<A>,NilName),Method(<foo> ...)`
- `classes` is a *list* gathering all classes names present in the given *Java* program : `(ConsName(<A>,NilName),ConsName(<B>,NilName), ...)`
- `methods` is a *list* containing all the methods defined in the given program
- `methods_per_classes` is a *list* with the same length as `classes` containing *lists* of methods defined in the corresponding classes : `((Method(<foo>,NilType,void), ...),( ... ), ...)`
- `fields_per_classes` is a *list* with the same length as `classes` containing *lists* of fields defined in the corresponding classes or their superclasses : `((Field(ConsName(<A>,NilName),<x>,TInt), ...),( ... ), ...)`
- `fields` is `fields_per_classes` flatten
- `init_fields_per_classes` contains the default values taken by the fields when they are initialized : `zero` for numbers, `nilchar` for characters and `val_null` for objects
- `subclasses_per_classes`
- `superclasses_per_classes`
- `classes_flags` : `((AccPublic),(AccPublic,AccAbstract, ...), ...)`
- `fields_flags_per_classes`
- `fields_flags` is `fields_flags_per_classes` flatten
- `methods_flags_per_classes`

Moreover, we import variables such as `nb_insts`, `nb_classes`, `nb_fields`, `nb_methods`, `nb_fields_per_classes` and `nb_methods_per_classes` even if they can be retrieved by using the primitive `len` on the appropriate variables.

3.2.4 The stubs classes

The classes belonging to the *Java* API can be declared in a special file with `.jstub` extension. This file contains classes, fields and methods signatures using a syntax very close to *Java*. For instance here is the `stubclasses.jstub` file that we use to generate a *term rewriting system* from a *Java* program :

```
/** stubclasses.jstub file **/

public class java.lang.Object{
    void <init>{};
}

public class java.io.IOException extends java.lang.Object{
}

public class java.io.InputStream extends java.lang.Object{
    public int read{};
}

public class java.io.PrintStream extends java.lang.Object{
    public void println{int};
    public void println{char};
    public void println{java.lang.String};
}

public class java.lang.System extends java.lang.Object{
    public static java.io.InputStream in;
    public static java.io.PrintStream out;
}

public class java.lang.String extends java.lang.Object{
    public native char charAt{int};
    public native java.lang.String concat{java.lang.String};
    public native int length{};
    public native java.lang.String substring{int};
}

public class java.lang.StringBuilder extends java.lang.Object{
}

public class java.lang.Thread extends java.lang.Object{
    public void <init>{};
    public void start{};
    public void join{};
}

public class java.lang.InterruptedException extends java.lang.Object{
}
```

Some classes are empty when they are present in the byte code but we don't provide any implementation yet (e.g the exception classes).

The implementation of the classes declared in the `stubclasses.jstub` file is done in a *Copster* source file.

3.2.5 A sample code

We present here a full example of the imported environment when considering a very simple program.

Consider the following program :

```
class A{
  int x;
  void foo(){this.bar();}
  void bar(){x=1;}
}

class B extends A{
  void bar(){x=2;}
}

class M{
  public static void main(String[] argv){
    A o1 = new A();
    A o2 = new B();
  }
}
```

The imported variables are :

- `insts = ((Class(ConsName(<M>,NilName)),Method(<<init>>,NilType,void),pp0,load(local0`
  `...)`
- `nb_insts = 29`
- `max_locals = 3`
- `max_pc = 8`
- `nb_classes = 12`
- `nb_fields = 4`
- `nb_methods = 14`
- `nb_fields_per_classes = (0,1,1,0,0,0,0,2,0,0,0,0)`
- `nb_methods_per_classes = (2,2,3,1,0,1,3,0,4,0,3,0)`

- classes = (Class(ConsName(<M>,NilName)), Class(ConsName(<B>,NilName)),
  Class(ConsName(<A>,NilName)), Class(ConsName(<java>,ConsName(<lang>,ConsName(<Objec
  Class(ConsName(<java>,ConsName(<io>,ConsName(<IOException>,NilName)))),
  Class(ConsName(<java>,ConsName(<io>,ConsName(<InputStream>,NilName)))),
  Class(ConsName(<java>,ConsName(<io>,ConsName(<PrintStream>,NilName)))),
  Class(ConsName(<java>,ConsName(<lang>,ConsName(<System>,NilName)))),
  Class(ConsName(<java>,ConsName(<lang>,ConsName(<String>,NilName)))),
  Class(ConsName(<java>,ConsName(<lang>,ConsName(<StringBuilder>,NilName)))),
  Class(ConsName(<java>,ConsName(<lang>,ConsName(<Thread>,NilName)))),
  Class(ConsName(<java>,ConsName(<lang>,ConsName(<InterruptedException>,NilName)))))

- fields = (Field(Class(ConsName(<A>,NilName)),<x>,TInt), Field(Class(ConsName(<A>,Nil
  ...)

- methods = (Method(<<init>>,NilType,void), Method(<main>,ConsType(...),void),
  Method(<bar>,NilType,void), Method(<foo>,NilType,void), ...)

- fields_per_classes = ((), (Field(Class(ConsName(<A>,NilName)),<x>,TInt)),
  (Field(Class(ConsName(<A>,NilName)),<x>,TInt)), ...)

- methods_per_classes = ((Method(<<init>>,NilType,void), Method(<main>,ConsType(...),vo
  (Method(<<init>>,NilType,void), Method(<bar>,NilType,void)),
  (Method(<<init>>,NilType,void), Method(<bar>,NilType,void),
  Method(<foo>,NilType,void)), ...)

- init_fields_per_classes = ((), (zero), (zero), (), (), (), (),
  (val_null, val_null), (), (), (), ())

- subclasses_per_classes = ((1), (2), (3,2), (4,12,11,10,9,8,7,6,5,3,2,1),
  (5), (6), (7), (8), (9), (10), (11), (12))

- superclasses_per_classes = ((1,4), (2,3,4), (3,4), (4), (5,4),
  (6,4), (7,4), (8,4), (9,4), (10,4), (11,4), (12,4))

- classes_flags = ((AccDefault), (AccDefault), (AccDefault), (AccPublic),
  (AccPublic), (AccPublic), (AccPublic), (AccPublic), (AccPublic),
  (AccPublic), (AccPublic), (AccPublic))

- fields_flags_per_classes = ((), ((Field(Class(ConsName(<A>,NilName)),<x>,TInt),
  (AccDefault))), ((Field(Class(ConsName(<A>,NilName)),<x>,TInt),
  (AccDefault))), (), (), (), (), ((Field(Class(ConsName(<java>,ConsName(<lang>,ConsNam
  <in>, OType(Class(ConsName(<java>,ConsName(<io>,ConsName(<InputStream>,NilName)))))
  (AccStatic,AccPublic)), (Field(Class(ConsName(<java>,ConsName(<lang>,ConsName(<Syste
  <out>, OType(Class(ConsName(<java>,ConsName(<io>,ConsName(<PrintStream>,NilName))))
  (AccStatic,AccPublic))),(),(),(),())

- fields_flags = ((Field(Class(ConsName(<A>,NilName)),<x>,TInt),
  (AccDefault)), (Field(Class(ConsName(<A>,NilName)),<x>,TInt),
  (AccDefault)), ...)

- `methods_flags_per_classes = (((Method(<<init>>,NilType,void), (AccDefault)), (Method(<main>,ConsType(...),void), (AccStatic,AccPublic))), ... )`

4 Copster command line

This section presents the `copster` command usage. If you type `copster -help` in a terminal, you can see :

```
~/> copster -help
Usage : copster -aterms filename [-rules file] [-sysname expr] [-maude file]
[-timbuk file] [-screen-width size] [-javaclass file] [-stubs file]
[-no-recompile]
  -rules specifies the rules file (default : rules.rex)
  -aterms specifies the aterms result file
  -maude specifies the maude result file
  -timbuk specifies the timbuk result file
  -sysname specifies the name of the system (default : S)
  -classpath specifies the classpath where to search for .class files
        Must be placed before -javaclass option
  -javaclass specifies the .class file to import
        Must not contain the .class extension
  -stubs specifies the file where are defined stubs
  -no-recompile If the aterms file is already generated,
                it's not useful to parse the java class file again
  -screen-width specifies the max size of lines in result files
  -debug prints information useful for debugging
  -help  Display this list of options
  --help  Display this list of options
```

Example of usage :

```
./build/copster -rules ./rules/monothread/rules.rex -stubs ./rules/stubclasses.jstub
-classpath ./tests/ -javaclass Ex1 -aterms result.aterms -maude result.maude
```

4.1 Provided outputs

The main output is the Aterms format and is specified as follows :

```
#### Example.aterms ###
specification(
   operatorList([
      op(stack,2),op(name,2),op(succ,1),op(zero,0),...]),
   varList([
      x,y,f,...]),
   trsList([system("S",[
      rule(initialJavaState(var(x)),
           IO(state(frame(name(Method(main,...),...),...),...),...)),
      ... ]), ...])
   )
```

The *term rewriting system* is given a name (`S` by default), but this is not used yet.

Specifying a `-aterms` argument to `copster` is compulsory because the others output formats are built from the Aterms format. There are two others output formats, Maude and Timbuk, where files are given after `-maude` and `-timbuk` options, respectively.

It possible to generate Maude and Timbuk outputs later, independently of any byte code program. For this you need to specify the previously generated Aterms file and use the `-no-recompile` option.

Example : `./copster -aterms result.aterms -maude res_maude.maude -timbuk res_timbuk.timbuk --no-recompile`

4.2 Error Management

Copster handles syntax and execution errors. In both cases, a stack trace is printed, giving the filenames and the instructions lines numbers which lead to the error.

Moreover, there is a very useful `-debug` option which prints the content of *operators list*, *variables list* and *Copster* variables *list* when the program stops its execution, normally or abnormally.

5 Index of reserved expression keywords

The expression keywords are reserved. It means that you can't use them directly in order to create *terms*. For instance, if you want to generate the *rule* `add(X,zero) => X` you must write `genrule(op(add,(var(x),zero)),var(x))`. Otherwise `add` is understood as the addition primitive.

6 Index of instruction keywords