BioBITs project – TSCLS Simulator

TSCLS Simulator Demo

Miklós Espák
espakm@gmail.com

Outline

- CLS, SCLS
- TSCLS
- SCLS Simulator
- TSCLS Simulator
  - Development
  - Demonstration
- Further improvements

Calculus of Looping Sequences

- Simple term rewriting system
- Terms:
  - Sequence: $S ::= \varepsilon \mid a \mid S \cdot S$
  - Term: $T ::= S \mid (\overline{S})^L \rfloor T \mid T \mid T$

empty sequence

concatenation

alphabet element

looping sequence

sequence

loop (membrane)^L]content

compartment

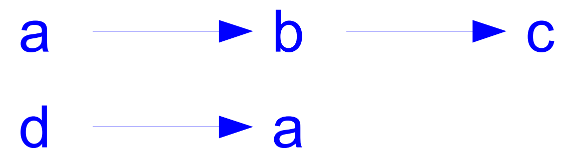
Biological interpretation

Sequences:

$a \cdot b \cdot c$

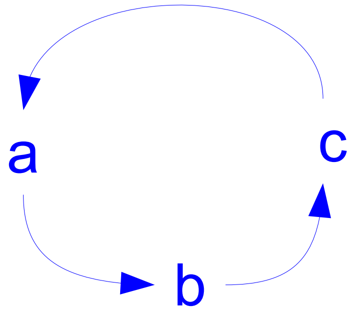
Compartments:

$a \cdot b \cdot c \mid d \cdot a$



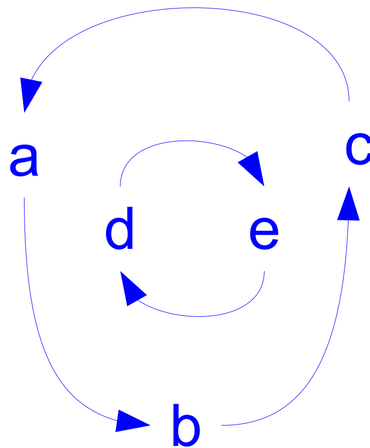
Loops:

$(a \cdot b \cdot c)^L$

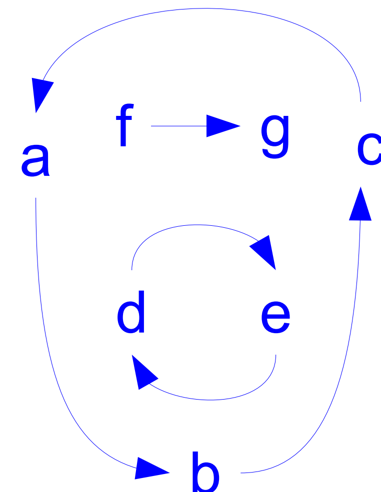


$(a \cdot b \cdot c)^L]\varepsilon$

$(a \cdot b \cdot c)^L](d \cdot e)^L$



$(a \cdot b \cdot c)^L]((d \cdot e)^L \mid f \cdot g)$



Structural congruence of terms

- Equivalence relation on sequences: $\equiv_S$
  - $S_1 \cdot (S_2 \cdot S_3) \equiv_S (S_1 \cdot S_2) \cdot S_3$
  - $S \cdot \varepsilon \equiv_S \varepsilon \cdot S \equiv_S S$
- Equivalence relation on terms: $\equiv_T$
  - $S_1 \equiv_S S_2$ *implies* $S_1 \equiv_T S_2$ *and* $(S_1)^L]T \equiv_T (S_2)^L]T$
  - $T_1 \mid T_2 \equiv_T T_2 \mid T_1$
  - $T_1 \mid (T_2 \mid T_3) \equiv_T (T_1 \mid T_2) \mid T_3$
  - $T \mid \varepsilon \equiv_T T$
  - $(\varepsilon)^L]\varepsilon \equiv_T \varepsilon$
  - $(S_1 \cdot S_2)^L]T \equiv_T (S_2 \cdot S_1)^L]T$

Term rewrite system – variables

- Variables
  - Element variables
    - x, y, z, ...
    - can match exactly one letter
  - Sequence variables
    - $\widetilde{x}$, $\widetilde{y}$, $\widetilde{z}$, ...
    - can match zero or more letter (any sequence)
  - Term variables
    - X, Y, Z, …
    - can match any term

Term rewrite system – patterns

- Patterns:
  - Such *"terms"* that can contain variables
  - Sequence patterns:
    - $SP ::= \varepsilon \mid a \mid SP \cdot SP \mid \tilde{x} \mid x$
  - Term pattern:
    - $P ::= SP \mid (SP)^{\llcorner}]P \mid P \mid P$
- Instantiation (σ):
  - A function that assign an element, a sequence or a term to a variable, according to its type.
  - Can be applied for a pattern, resulting a term.
  - Example:
    - $\sigma(\tilde{x}) = a \cdot b$, $\sigma(X) = a \mid d$, $P = X \mid c \cdot x \mid d$, then $P\sigma = a \mid d \mid c \cdot a \cdot b$

Term rewrite system – patterns

- Matching:
  - A pattern *P* matches to a term *T* when there is an instantiation that produces *T* from *P*.
- Examples (alphabet: {a, b}):
  - $a \cdot x$ matches: a·a, a·b
  - $a \cdot \tilde{x}$ matches: a, a·a, a·b, b·a, b·b, a·a·a, a·a·b, ...
  - $a|X$ matches: a, a|a, b|a, $a|(b)^{L}a$ etc., but *not* a·b

Term rewriting system: rule, context

- Term rewrite rule:
  - $P_1 \rightarrow P_2$
  - P_1 is not empty
  - P_2 must not introduce new variables
- Context:
  - Specifies a certain part of a term (“hole”)
  - Controls, how rules can be applied to a term
  - $C ::= \square \mid C|T \mid T|C \mid (S)^{L}]C$

Applying rules on terms (1)

- Context:
  - $C ::= \square \mid C|T \mid T|C \mid (S)^{L}]C$
- If the left hand side of a rule matches to a certain part of a term within a legal context then it can be replaced to the right hand side of the rule (with the same instantiation)
- Example 1:
  - Rule: a → b | c
  - Term: b | a | (c·b)
  - Matches within context:
    - b | a | (c·b)
  - The “hole” of the context can be replaced to the rhs of the rule:
    - b | b | c | (c·b)

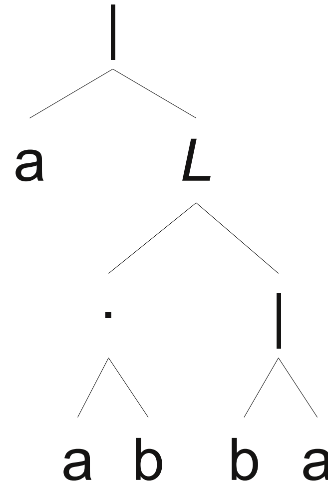
Applying rules on terms (2)

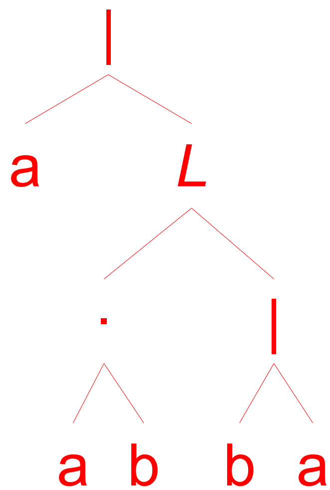
- Context:
  - $C ::= \square \mid C|T \mid T|C \mid (S)^{L}]C$
- If the left hand side of a rule matches to a certain part of a term within a legal context then it can be replaced to the right hand side of the rule (with the same instantiation)
- Example 2:
  - Rule: a → b | c
  - Term: b | (a·b)
  - There are no matches. This is not a valid context. ~~($C ::= S{\cdot}C$)~~.
    - b | (a·b)
  - The rule cannot be applied.

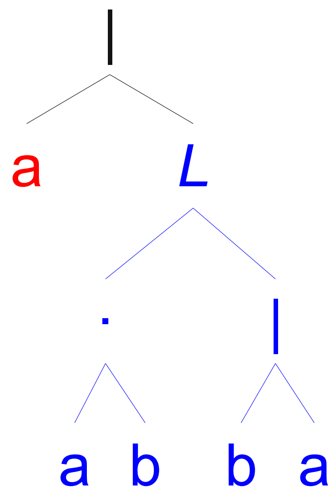
Selecting the matches

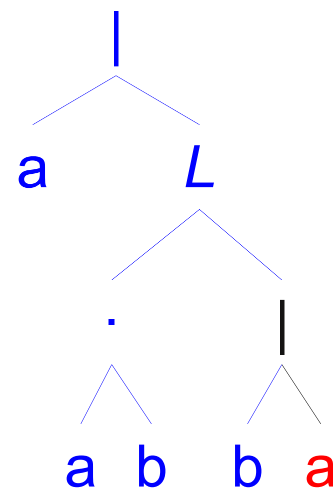
Pattern: a | *X*

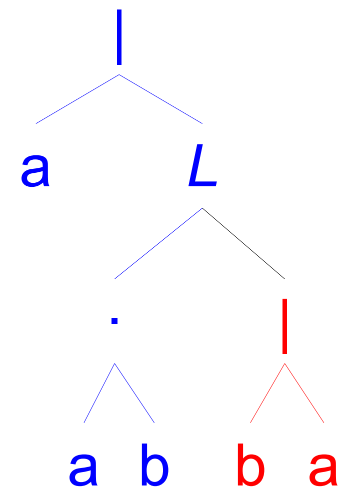
Term: a | $(a \cdot b)^{L}$](b|a)







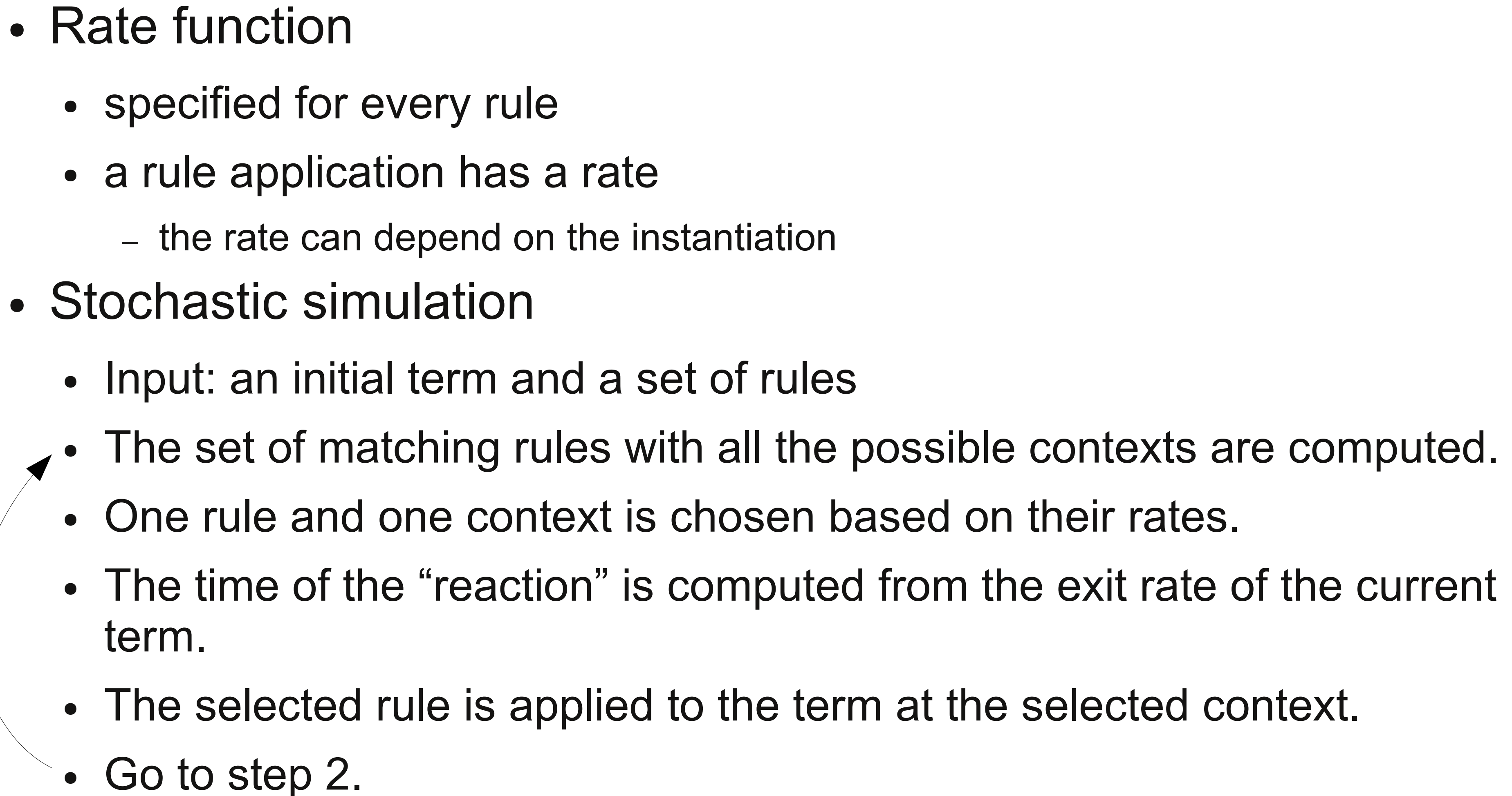




Applying rules on terms

- Example:
  - Rule: a|X → b|X
  - Term: a | (a·b)^{L}](b|a)
  - The pattern (lhs) occurs in the following contexts:
    - a | (a·b)^{L}](b | a) | ε
      - X = (a·b)^{L}](b | a), rhs: b | (a·b)^{L}](b | a) result: b | (a·b)^{L}](b | a) | ε
    - a | (a·b)^{L}](b | a)
      - X = ε, rhs: b|ε ≡ b, result: b | (a·b)^{L}](b | a)
    - a | (a·b)^{L}](b | a)
      - X = b, rhs: b|b, result: a | (a·b)^{L}](b | b)
    - a | (a·b)^{L}](b | a)
      - X = ε, rhs: b|ε ≡ b, result: a | (a·b)^{L}](b)

Stochastic Calculus of Looping Sequencs



- Rate function
  - specified for every rule
  - a rule application has a rate
    - the rate can depend on the instantiation
- Stochastic simulation
  - Input: an initial term and a set of rules
  - The set of matching rules with all the possible contexts are computed.
  - One rule and one context is chosen based on their rates.
  - The time of the “reaction” is computed from the exit rate of the current term.
  - The selected rule is applied to the term at the selected context.
  - Go to step 2.

Outline

- CLS, SCLS
- TSCLS
- SCLS Simulator
- TSCLS Simulator
  - Development
  - Demonstration
- Further improvements

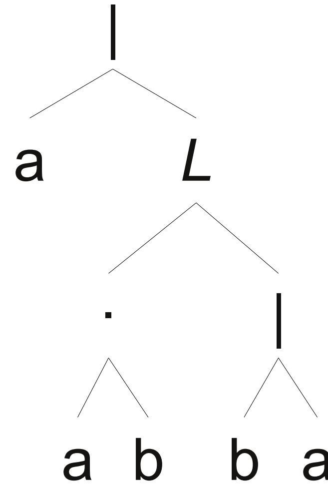
Typed Stochastic Calculus of Looping Sequences

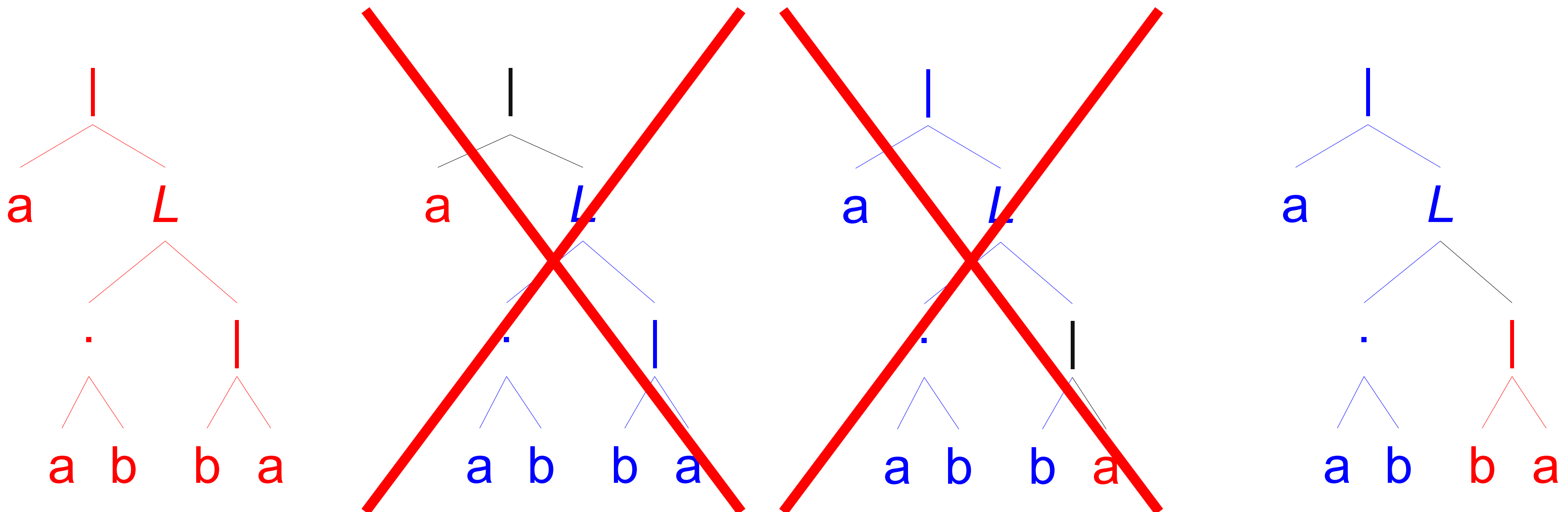
- New context definition:
  - $C ::= \square \mid T \mid (S)^{L}] C$
- The original (CLS) definition was:
  - $C ::= \square \mid C | T \mid T | C \mid (S)^{L}] C)$
- Consequence:
  - Patterns cannot match compartments *partially*

Selecting the matches

Pattern: a | *X*

Term: a | (a·b)^{L}](b|a)





Typed Stochastic Calculus of Looping Sequences

- Alphabet elements are typed.
- Let $\Gamma(a) = t_a$
  - If *a* occurs in itself, its type is t_a.
  - If *a* occurs in a (looping) sequence, its type is $\tilde{t}_a$.
- The type of a sequence or looping sequence:
  - the multiset of the type of its elements
- The type of a loop:
  - the type its *membrane* (the content does not matter!)
- The type of a compartment:
  - the union of the type of its children

Type of terms: example

- $\Gamma(a) = t_1$, $\Gamma(b) = t_2$, $\Gamma(c) = t_1$
  - $type(\ a \mid a \mid c\ ) = \{\ t_1, t_1, t_1\ \}$
  - $type(\ a \cdot b \cdot b \cdot a \cdot c\ ) = \{\ \tilde{t}_1, \tilde{t}_2, \tilde{t}_2, \tilde{t}_1, \tilde{t}_1\ \}$
  - $type(\ (a \cdot b \cdot b \cdot a \cdot c)^L\ ) = \{\ \tilde{t}_1, \tilde{t}_2, \tilde{t}_2, \tilde{t}_1, \tilde{t}_1\ \}$
  - $type(\ a \mid (b)^L](a|b|c)\ ) = \{\ t_1, \tilde{t}_2\ \}$

Rules

- Rule definition
  - A list of basic and sequence types are specified for each variable occurring in the rule
  - If a match found, for each variable the number of elements of the types relevant for the variable is counted
  - A rate function is defined that gets these numbers as its argument
  - The stochastic simulation is based on this rate

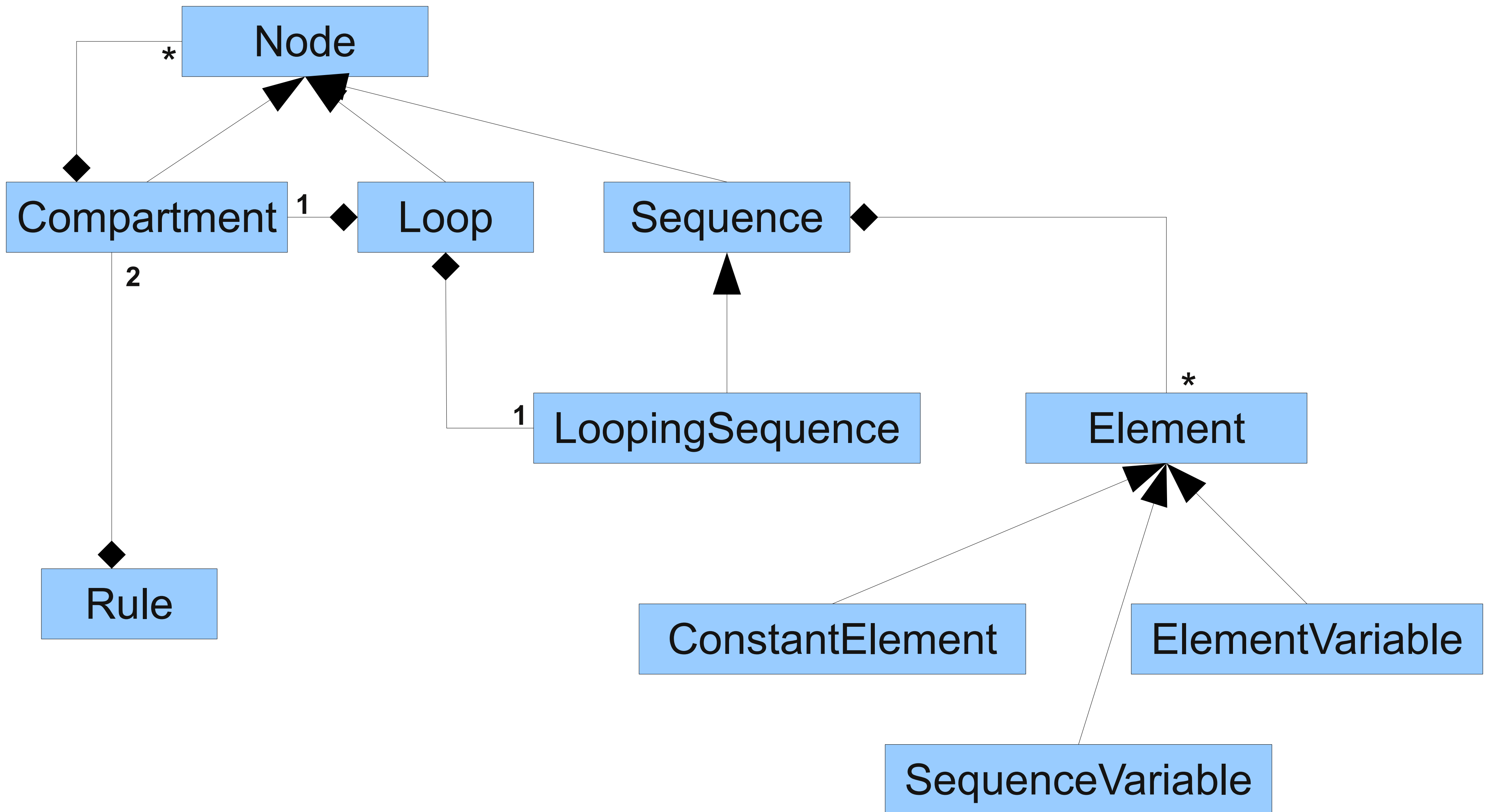
Outline

- CLS, SCLS
- TSCLS
- SCLS Simulator
- TSCLS Simulator
  - Development
  - Demonstration
- Further improvements

SCLS Simulator

- Guido Scatena, M.Sc. thesis, University of Pisa
- Technical details of the project
  - About 10 000 lines of code
  - Mostly written in F#
  - The parser:
    - Coco/R grammar
    - Semantic actions are coded in C#
  - Rate functions
    - C# source code generated and compiled
    - Invoked through reflection at run-time

Term representation



Some remarks

- Terms are stored in a concise and normalized way:
  - Compartments are stored as hashtables
    - node * repetitions
  - Looping sequences are stored in their “normal form”
    - their “least” form (elements are comparable)
- But:
  - There are compartments with a single child
    - The content of loops is a compartment, not node
    - The left and right hand side of loops is compartment

Bottom-up tree pattern matching

- Original idea published by Hoffmann and O'Donnel
- A longer preparation phase to allow faster pattern recognition
- Preparation phase
  - All the subtrees of the patterns are collected
  - The set of possible transitions between the subtrees are precomputed and stored in a table
- Recognition phase
  - Attributes are assigned to the leaves of the subject tree
  - From the leaves to the root, a set of attributes is computed based on:
    - the attributes of their children
    - the precomputed transition table
  - A rule can be applied if its lhs is among the attributes of the root

Non-determ. finite state automata

- The leaves the tree are sequences
- Their attributes are computed by NFAs
  - An NFA is generated for every (looping) sequence occurring in the left hand sides of the rules (preprocessing phase)
  - At the recognition phase all the automata are started simultaneously for the input

Some drawbacks of the project

- Not well documented
- The code is not clean
  - Not OO style
    - Many “instance of” checks instead of late binding
    - Direct field accesses (even for writing)
    - Static methods accepting the instance of the same class
- Too many transformations on data
  - Frequent use of temporary lists, arrays for performing a certain computation

Outline

- CLS, SCLS
- TSCLS
- SCLS Simulator
- TSCLS Simulator
  - Development
  - Demonstration
- Further improvements

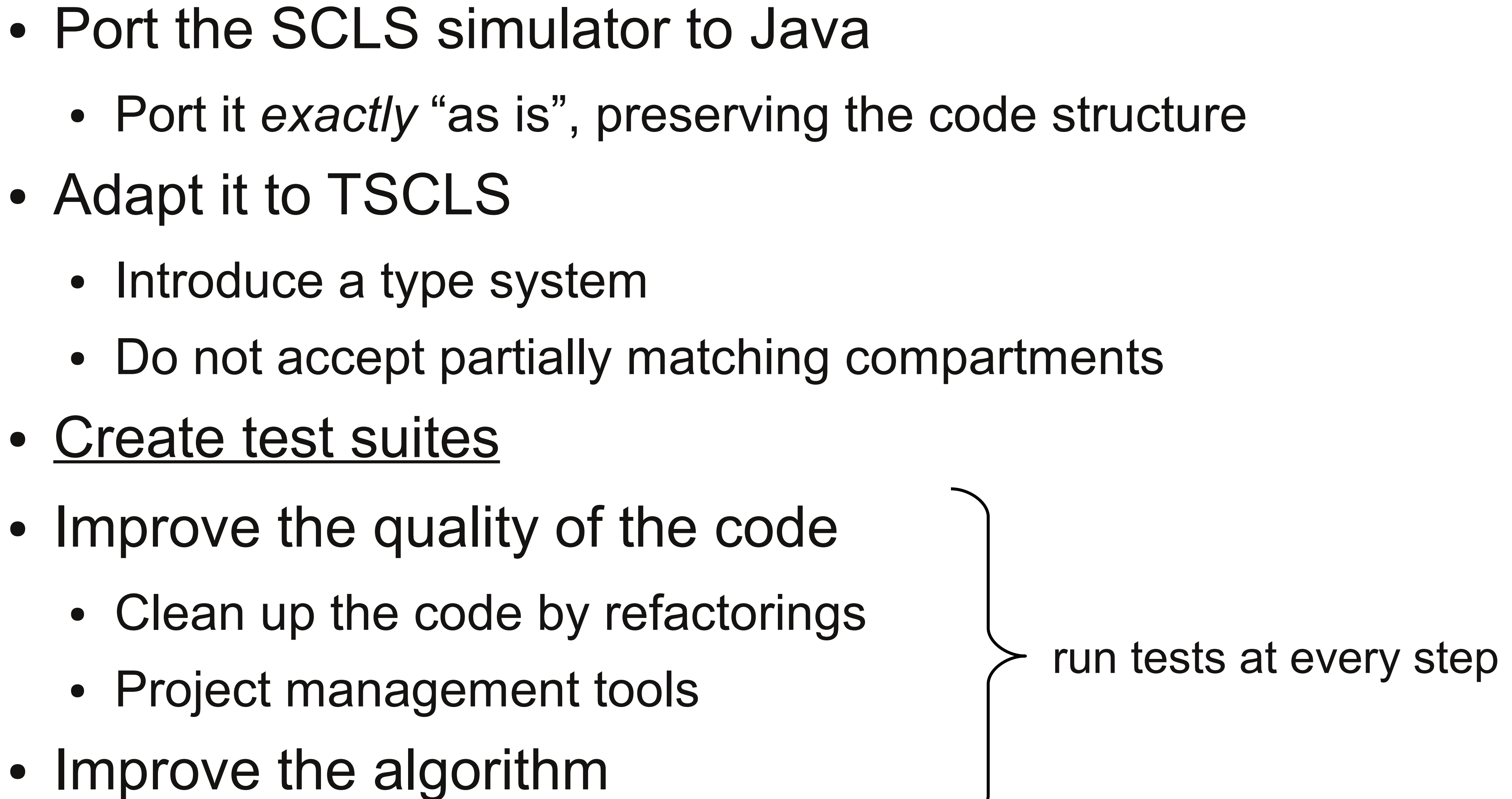
TSCLS Simulator

- Goals
  - *Correct* implementation of TSCLS
  - Should be *more efficient* then the SCLS simulator
  - Java platform

Outline

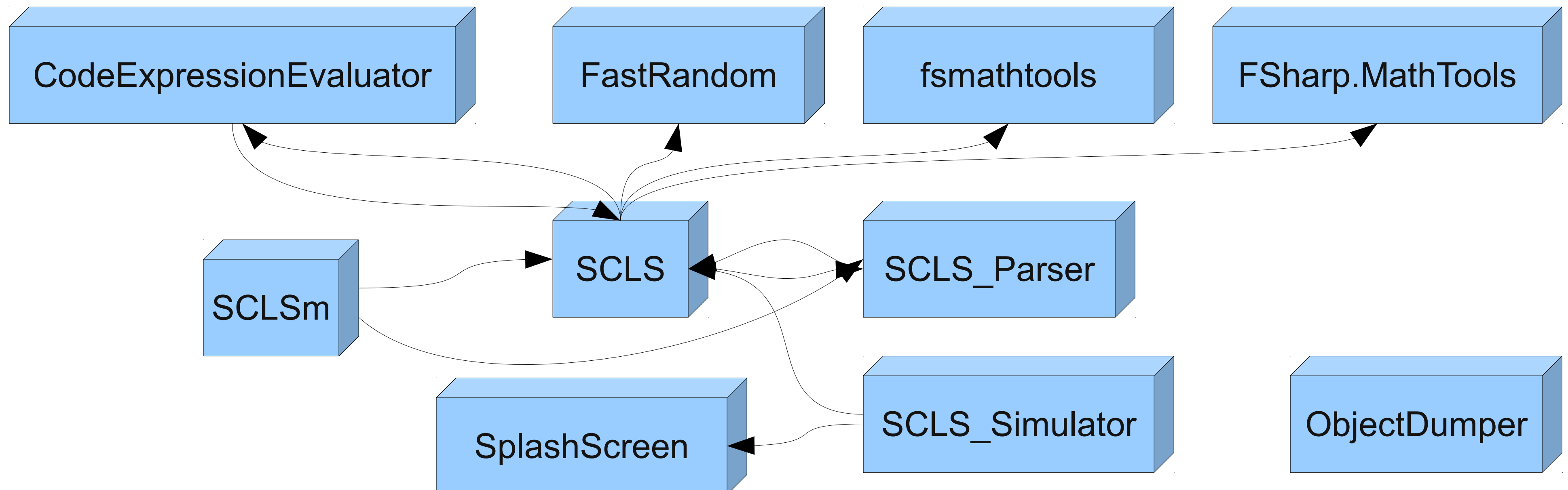
- CLS, SCLS
- TSCLS
- SCLS Simulator
- TSCLS Simulator
  - Development
  - Demonstration
- Further improvements

TSCLS development



Porting the SCLS simulator to Java

- Setting up a development environment for SCLS simulator
  - Windows 7, Visual Studio 2010, ...
- Exploring the dependencies between the subprojects:



Differences between F# and Java

- Frequent use of the “option” type
  - .NET specific, frequently used in F#
  - In Java: any reference is can be set to null
  - Each use of the option types has to be eliminated
- “Out” formal arguments
  - Do not exist in Java
  - Can be converted to
    - a return value (if there is none)
    - a “one element” array

Differences between F# and Java

- Frequent use of tuples
  - Two generic class introduced:
    - Pair<T1, T2>
    - Triple<T1, T2, T3>
  - For tuples having more elements a specific class has been introduced.
  - Tuples do not have name (neither their fields)
    - It is hard to recognize their purpose
    - Frequent use of tuples is a bad programming style
    - Use classes instead!

Misusing tuples in F#

A two dimensional array of triples containing a triple, a list and a pair.

```
let crossIntantiations(pi: ((Node ref*int64*int64) * bindings list *
                            (int64*int64) ) array array)
 :((Node ref*int64*int64) * bindings list *(int64*int64)) array array
```

The return value is of the same type.

Differences between F# and Java

- Type inference vs. static type checking
  - In F# variables need not to be declared (they are inferred automatically)
  - Keeps the code concise and safe
  - The code is much harder to read: the role of the variables remains hidden
- Nested tuples and type inference
  - *It's a real challenge to figure out what's happening!*

Porting the parser

- Coco/R grammar
  - Coco/R can generate Java code as well
  - The grammar is written very badly
    - the name of the rules does not reflect their purpose well
      - rule “terms” accepts a compartment
      - rule “term” accepts a term (sequence, term variable or loop)
      - rule “sequences” accepts *one* sequence
      - rule “sequence” accepts an element
    - rules use many arguments
      - the responsibility of rules is not clean
  - Semantic actions are written in C#

Extending the grammar

- Create tests accepting the current inputs
- Extending the grammar:
  - Constant elements has to be declared
  - The interested types of the variables has to be specified for the rules
  - A rate function has to be specified for the rules

Example input file

```
types

a: ta;
b: tb;
c: tc;
d, e: te;

rules

r1 :=
    a|$t:X -> b|$t:X,
    <<ta, tc>>,
    function (n) {
        var k1 = 0.1;
        var k2 = 0.5;
         return ((n[0] + 1) * k1) / (n[1] == 0 ? 1 : n[1] * k2);
    }
    ;

term
a|a|c
```

Several constants can have the same type.

If there are more variables, several tuples are here.

Processing the rate function

- A rate function is a valid *JavaScript* function
- Java Scripting API
  - JSR 223, introduced in Java 6
  - JavaScript Engine
  - Contains many other engines, too: Python, Ruby, ...
- How it is used:
  - The engine is created once
  - The rate functions are <u>compiled</u> once
  - The rate functions are invoked when calculating the rate

Project management

- Introducing Maven 2
  - Supports project management through the whole project life cycle
  - Automatic dependency management
  - Subprojects (called modules)
  - Build the project, build the tests, run the tests, build the project web site, deploy all the artifacts, etc. with a single command
  - Supports creating reports from the project
  - Supports creating releases

Maven 2

- There are two subprojects now:
  - SCLS
    - The project ported in its original state
    - Only clean-ups and bug fixes
    - The “occ” function is not implemented
  - TSCLS
    - Contains the current state of the code

Logging

- System.out.println is regarded as bad
- Introducing the SLF4J framework
  - Simply to use logging framework
  - Used in numerous projects (e.g. Apache projects)

Coco/R → ANTLR

- The grammars have been reimplemented in ANTLR
  - The code needed a heavy clean-up
  - ANTLR is one of the most successful Java projects
  - Its syntax is cleaner than of Coco/R
  - More fault tolerant
  - Supported by Maven 2

Test grammar

- Special input file format for writing tests
  - *Declaration* of constant elements
  - Set of *rules*
  - A set of *test cases*, whose elements contain:
    - An initial term
    - A *match set*, whose elements contain:
      - the name of the applicable rule
      - the term produced by applying the rule
      - the rate of applying the rule

Example test file

```
types
a: ta;
b: tb;

rules
r1 := a|$t:X -> b|$t:X,
      <<ta, ~ta>>,
      function (int[] n) {
        return n[0] * 2 + n[1] * 3;
      };

r2 := a -> b,
      <>,
      function (n) {
        return 1.0;
      };

tests

a | loop(a)[a|a] | loop(a)[a|a] ->
   <r1, b | loop(a)[a|a] | loop(a)[a|a], 6>,
   <r1, a | loop(a)[a|b] | loop(a)[a|a], 2>;
```

Note that different prime numbers are used so that we get different rate for different inputs.

Note that r2 does not match to the term.

Refactoring, clean-up

- Lines of code
  - The original F# project: 5271 lines
  - The first Java port (“as is”): 4412 lines
    - Java is more verbose in general
  - After some basic refactorings, clean-ups: 3964 lines
    - Removing unused code
    - Introducing constructors
    - Restructuring the code
    - ...

Outline

- CLS, SCLS
- TSCLS
- SCLS Simulator
- TSCLS Simulator
  - Development
  - Demonstration
- Further improvements

Usage

- Command line application
  - The graphical user interface has not been ported.
  - Arguments:
    - input file
    - time limit
    - output file
    - time sampling rate
  - Produces a comma separated output
    - can be processed by MS Excel / OpenOffice.org Calc
    - columns: time, elements...
    - rows: the time and the concentrations at the given time

Example test file

```
types
NH3: T_NH3;
NH4+: T_NH4+;

rules

r1 := $t:X | NH3 -> NH4+ | $t:X,
      <<T_NH3+>>,
      function (int[] n) {
        var k1 = 0.000018;
        return (n[0] + 1) * k1;
      };

r2 := $t:X | NH4+ -> NH3 | $t:X,
      <<T_NH4+>>,
      function (int[] n) {
        var k2 = 0.000000000562;
        return (n[0] + 1) * k2;
      };

term
NH3 * 1382388 | NH4+ * 1382388
```

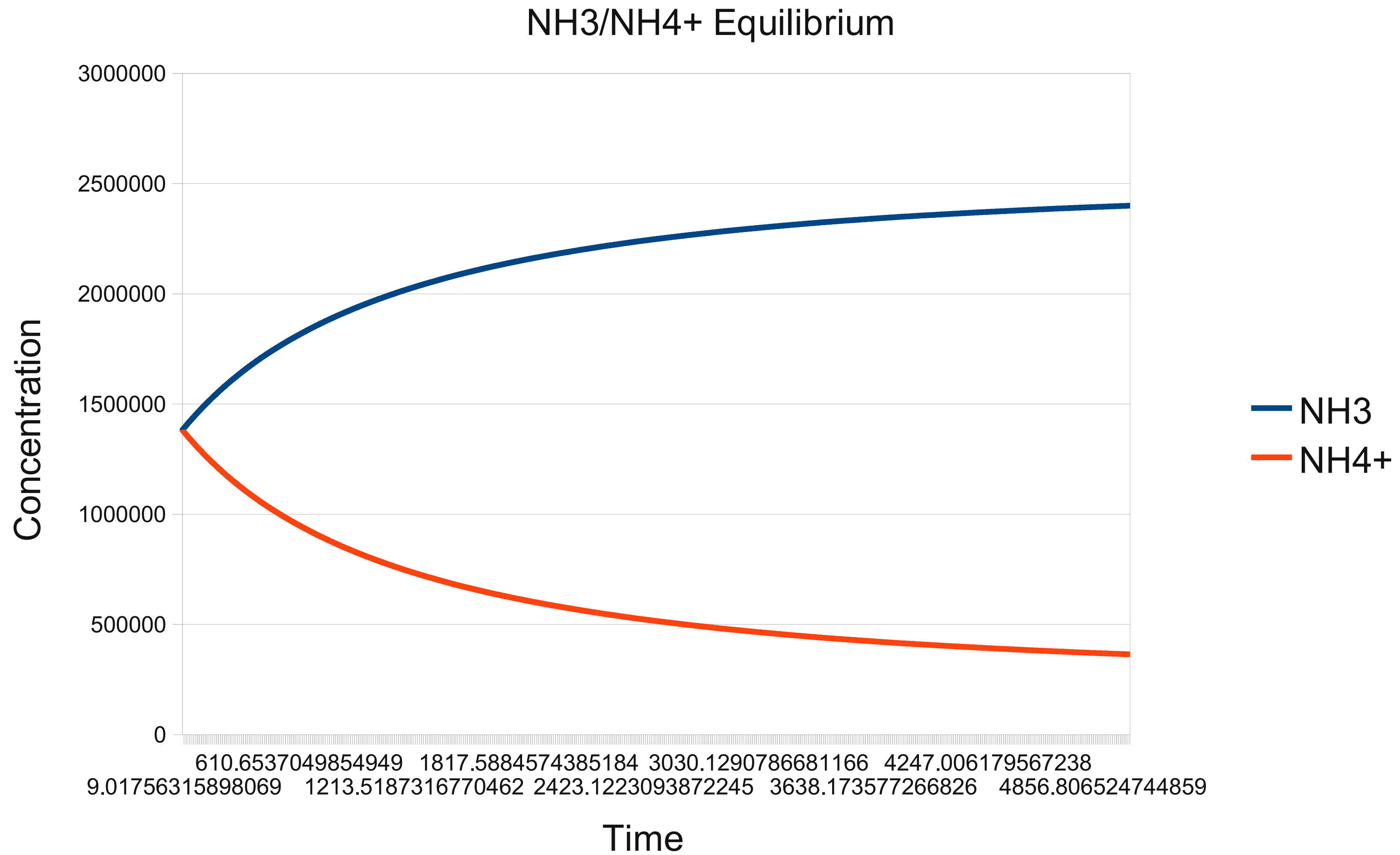
A variable has been introduced to allow counting of types.

The generated .csv file

time	NH3	NH4+
0.0	1382388	1382388
1.000080033097823	1383439	1381337
2.002979844634081	1384445	1380331
3.003715839751098	1385522	1379254
4.006293765024959	1386565	1378211
5.00702023440904	1387631	1377145
6.009312875654464	1388711	1376065
7.011813002366265	1389806	1374970
8.014032384019455	1390903	1373873
9.01756315898069	1391932	1372844
10.0182696099603	1392947	1371829
11.0205918693573	1393923	1370853
12.0212280382097	1394973	1369803
13.0249626870125	1395998	1368778
14.0273208276178	1397004	1367772
15.0288086074625	1398052	1366724
16.0321004292053	1399096	1365680
17.0348566195187	1400184	1364592
18.0351642663636	1401169	1363607
19.0381191978863	1402182	1362594
20.0413086026870	1403258	1361518
21.0447110908186	1404247	1360529
22.04653160277958	1405282	1359494
23.0474933443397	1406245	1358531
24.04888925865987	1407220	1357556

NH_3/NH_4^+ Equilibrium

Time limit: 5000, Sampling rate: 1



The same reaction using SCLS

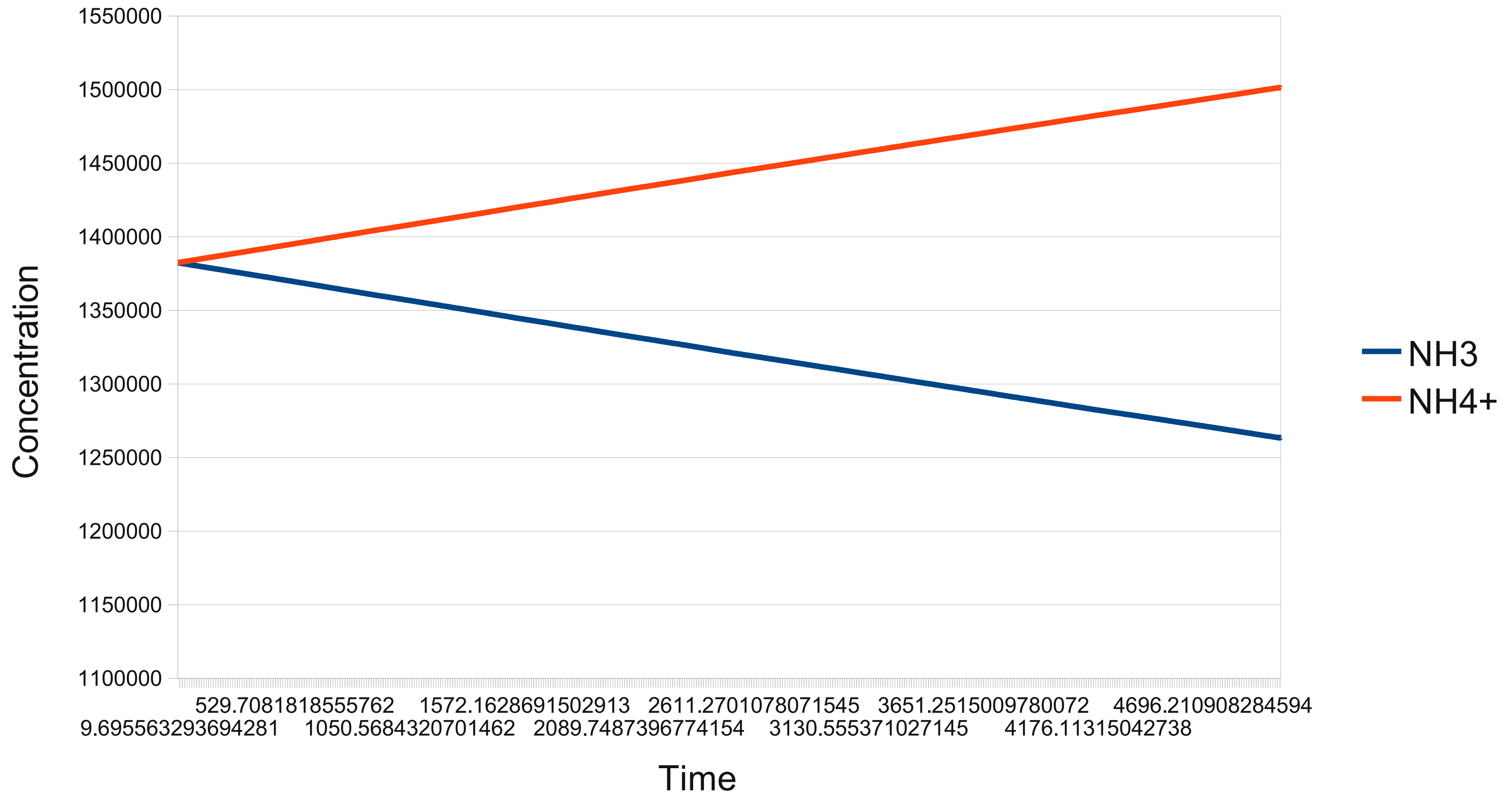
```
rules

r1: (NH3, NH4+, 0.000018)
r2: (NH4+, NH3, 0.000000000562)

term
NH3 * 1382388 | NH4+ * 1382388
```

NH_3/NH_4^+ Equilibrium

Time limit: 5000, Sampling rate: 1



Outline

- CLS, SCLS
- TSCLS
- SCLS Simulator
- TSCLS Simulator
  - Development
  - Demonstration
- Further improvements

Further improvements

- Speed
  - The implementation is still slow. :-(
  - The type of the terms is recomputed every time
    - It should be stored in the attributes of the nodes!
  - The structural dependencies of the subterms is not used
    - Dependency graph should be created
  - The structural dependencies of (looping) sequences is not used. (Independent NFAs run parallelly)
    - Multiple pattern matching algorithm should be used.

Further improvements

- Exploring the bottlenecks of the implementation
  - JProfiler
- Improving code quality
  - Eclipse Metrics
  - CheckStyle
  - PMD
  - FindBugs

Thank you for the attention!