02257 Applied functional programming

Michael R. Hansen
DTU COMPUTE
June 13, 2024

Project 2:
A molecular programming language

The topic of this project is computation on the basis of chemical reactions. It is based on the journal article [1]. You should study this article and solve the tasks given in this description.

In general, you should implement in F#, as much as possible of, the programming language CRN++ described in the article including interpreter, compiler and associated analysis tools.

1 An interpreter for CRN++

In this part you should implement an interpreter for the CRN++ language described in [1], where chemical reactions (non-terminal *RxnS* in the context-free grammar in Listing 1.1) are ignored. For modules, the interpreter should implement the meaning given in the 'Output' column of Table 1.

1. Present a throughout analysis of the description of CRN++ in the article. Particular attention could be given to the grammar in Listing 1.1 and syntactic restrictions either mentioned in the article or implicitly assumed.

   The analysis part must be completed with at least

   - a possibly revised grammar and
   - a list of precisely defined syntactic restrictions (that are not imposed by the grammar).

   The revised grammar and the restrictions together describe the CRN++ programs that are considered *well formed*. The well-formed programs must include all CRN++ programs described in the article.

2. Develop a model for abstract syntax trees for CRN++ programs. This model comprises F# type declarations for the main syntactic categories of CRN++ programs.

3. Define parsers for CRN++ using `FParsec`. You should be able to make abstract syntax trees for the example programs given in the article, e.g. GCD (Fig. 3), Discrete counter (Fig. 6), Factorial (Fig. 7), etc.

4. Implement a type checker for CRN++ programs that can check whether a program is well formed.

A program state (type `State`) records concentrations of species.

A step *stp* is a list of commands that are executed in parallel. In this part you may ignore the parallel execution and consider the interpretation of a step as a function: `State -> State`. You may validate using property-based testing that the evaluation order does not matter for non-conflicting commands.

The main part of a CRN++ program is a list *steps* of steps $stp_1, \ldots, stp_n$. These steps are executed in sequence: first stp_1, then stp_2 and so on. The interpretation of *steps* is also considered a function `State -> State`.

A CRN++ program is executed by executing *steps* indefinitely. That is, the meaning of a CRN++ program is an infinite sequence of states (type `seq<State>`)

$$s_0\ s_1\ s_2 \cdots$$

where state s_0 is given by the initial concentrations of species, state s_i is the state obtained after i iterations of *steps*.

5. Declare an F# type `State` and construct an interpreter for CRN++ programs.
6. Some programs executions may converge to a steady state (e.g. that for GCD) while others will not (e.g. Discrete counter). Construct a visualization component so that figures like Fig. 3(b) and Fig. 6(b) can be shown.
7. Make an assessment of this part. Pay particular attention to the (possibly revised) grammar, the connection between the grammar and the parser and correctness of the interpreter.

You may consider using the programs from Project 1 to visualize abstract syntax trees.

2 Chemical reactions

You should now implement chemical reactions. You may consider such chemical reactions as an abstract chemical machine. In Section 3 we consider how to compile CRN++ programs to this chemical reactions.

8. Make an F# type for reactions. See non-terminal *RxnS* in the grammar and the CRN i examples, $i = 1, 2, 3, \ldots$, in the article.
9. Construct a parser for reactions using `FParsec`.
10. Construct a simulator for chemical reactions on the basis of the ordinary differential equations given on Page 393. On the basis of initial concentrations of species and a time step, this simulator should give an infinite sequence of states.

11. Make visualizations corresponding to those of the Figures 1 and 4, for example.

12. Validate that the chemical reactions for modules in Table 1 behave as expected.

13. Make an assessment of this part.

3 Compiling CRN++ to chemical reactions

The article presents principles behind compiling programs to chemical reaction networks. You should make as much as possible for this compilation. Start with

14. Compile *stp* : `Step`, that is, a list of commands, to chemical reaction networks. (See Table 1).

15. Use property-based testing to validate that interpretation and compilation of steps agree.

Extend your implementation along the lines described in the paper.

References

[1] M. Vasic, D. Soloveichik, and S. Khurshid. CRN++: Molecular programming language. Natural Computing volume 19, pages 391–407 (2020). `https://link.springer.com/article/10.1007/s11047-019-09775-1`

[2] The homepage for `FParsec`: `https://www.quanttec.com/fparsec/`