# Representing the Process Algebra CSP in Type Theory

Bashar Igried and Anton Setzer

Swansea University, Swansea,Wales, UK
bashar.igried@yahoo.com , a.g.setzer@swansea.ac.uk

**Abstract**

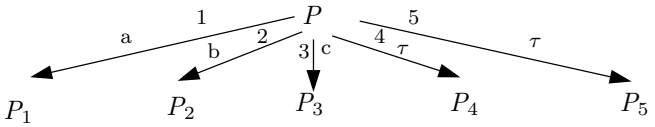
We introduce the library CSP-Agda which represents CSP processes in Agda. CSP-Agda allows to prove in Agda properties of CSP processes. CSP processes are implemented coinductively (or coalgebraically). They are formed like inductive data types from atomic operations, but infinite loops, i.e. non-wellfounded processes, are allowed.

## 1 Introduction

Mathematical induction is a backbone of programming and program verification [3]. Many data structures can be defined as inductive data types, also called algebraic data types. This allows to define programs using them by recursion and to prove properties by induction. In theorem proving elements of inductive types are well-founded, which means that if we consider an element of an inductive data type as a tree, there are no infinite branches. In programming one has to deal as well with potentially non-terminating programs, which corresponds to trees with infinite branches, i.e. non-well-founded data types. They are provided by the dual of inductive types, coinductive data types, also called coalgebras. Coinductive data types can be used as data types, for defining semantics, we can program with them using corecursion or guarded recursion, and reason about them using coinduction. In this paper, we will represent the process algebra CSP in a coinductive form in dependent type theory, and implement it in the Agda. Our approach is inspired by the representation of interactive programs in Agda by Peter Hancock and the second author. Since Agda is an interactive theorem prover, this allows to reason about CSP processes and prove the correctness of programs.

## 2 Representing CSP Processes in Agda

Processes in our approach are similar to interactive programs. Processes are defined using an atomic operation, which defines a new process by determining the transitions it can make together with the processes we obtain when firing these transitions. Processes can loop, therefore they are defined coinductively, using the representation of coalgebras as record types in Agda. The standard operations for forming processes, such as external choice or internal choice, are defined rather than considered as atomic as in process algebras. When defining processes recursively, we require them to be productive, which means for a process we can determine which next transitions it can make and the next processes after firing these transitions. The termination checker of Agda will check whether elements of coalgebraic data types are productive. Our approach is based on the algebraic nature of CSP language and on the operational characterisation of the behavioural semantics.




A CSP process is given by the labelled and silent transitions it can make. Labelled transitions correspond to external choice and silent transitions to internal choice. So we have in case of a process

progressing (1) an index set E of external choices and for each external choice $e$ a Label ($Lab\ e$) and a next process ($PE\ e$); and (2) an index set of internal choices $I$ and for each internal choice $i$ a next process ($PI\ i$). In mathematical notation a process is represented as follows (assuming a set of labels $\mathsf{Label} : \mathsf{Set}$):

$$\begin{aligned}\mathsf{Process} = \{\mathsf{node}(E, Lab, PE, I, PI) \mid\ & E \in \mathsf{Choice}, \quad Lab : E \to \mathsf{Label}, \quad PE : E \to \mathsf{Process}, \\ & I \in \mathsf{Choice}, \quad PI : I \to \mathsf{Process}\ \}\end{aligned}$$

The set of choice sets is given in Agda as a universe. The operations from CSP are given as defined operations. We give here only the simplest operation, the interleaving of two processes. Assuming $P_i = \mathsf{node}(E_i, Lab_i, PE_i, I_i, PI_i)$, we define the interleaving $P_1 \ ||| \ P_2 = \mathsf{node}(E, Lab, PE, I, PI)$ by $E = E_1 + E_2$, $Lab\ (\mathsf{inl}\ e) = Lab_1\ e$, $Lab\ (\mathsf{inr}\ e) = Lab_2\ e$, $PE\ (\mathsf{inl}\ e) = (PE_1\ e) \ ||| \ P_2$, $PE\ (\mathsf{inr}\ e) = P_1 \ ||| \ (PE_2\ e)$, similarly for $I$ and $PI$.

The other operations (external choice, internal choice, parallel operations, hiding, renaming, etc.) are defined in a similar way. Next we define traces model and refinement and show standard laws for processes module trace equivalence, for instance $P \ ||| \ Q$ is equivalent to $Q \ ||| \ P$.

# 3 Related Work

There have been several successful attempts of combining functional programming with CSP. Brown introduced in [1] a library (CHP) in Haskell. Since Haskell lacks explicit support for concurrency, they used a Haskell monad to provide a way to explicitly specify and control sequence and effects. Fontaine [2] gave another successful attempt of implementing the operational semantics of CSP as presented in [6] using the functional programming language Haskell. He presented a new tool for animation and model checking for CSP. Fontaine used a monad in order to model Input/Output, partial functions, state, non-determinism, monadic parser and passing of an environment. Lopez et. al. [4] gave further examples of combining functional programming with process algebras. They used the functional program Eden in order to translate VPSPA specification into Eden programs. Mossakowski et. al. in [5] gave a good example of using coalgebras in order to extend the specification language CASL.

# References


[1] Neil C. C. Brown. Communicating Haskell processes: Composable explicit concurrency using monads. In *The thirty-first Communicating Process Architectures Conference, CPA 2008, organised under the auspices of WoTUG and the Department of Computer Science of the University of York, York, Yorkshire, UK, 7-10 September 2008*, pages 67–83, 2008.

[2] Marc Fontaine. *A Model Checker for CSP-M.* PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2011.

[3] K. Rustan M. Leino and Michal Moskal. Co-induction simply - automatic co-inductive proofs in a program verifier. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 382–398, 2014.

[4] Natalia López, Manuel Núñez, and Fernando Rubio. Stochastic process algebras meet Eden. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 29–48, London, UK, UK, 2002. Springer-Verlag.

[5] Till Mossakowski, Lutz Schröder, Markus Roggenbach, and Horst Reichel. Algebraic-coalgebraic specification in cocasl. *J. Log. Algebr. Program.*, 67(1-2):146–197, 2006.

[6] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.