Programming with Monadic CSP-Style Processes in Dependent Type Theory

Bashar Igried   Anton Setzer

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

Abstract

We introduce a library called CSP-Agda for representing processes in the dependently typed theorem prover and interactive programming language Agda. We will enhance processes by a monad structure. The monad structure facilitates combining processes in a modular way, and allows to define recursion as a direct operation on processes. Processes are defined coinductively as non-well-founded trees. The nodes of the tree are formed by a an atomic one step relation, which determines for a process the external, internal choices, and termination events it can choose, and whether the process has terminated. The data type of processes is inspired by Setzer and Hancock's notion of interactive programs in dependent type theory. The operators of CSP will be defined rather than atomic operations, and compute new elements of the data type of processes from existing ones.

The approach will make use of advanced type theoretic features: the use of inductive-recursively defined universes; the definition of coinductive types by their observations, which has similarities to the notion of an object in object-oriented programming; the use of sized types for coinductive types, which allow coinductive definitions in a modular way; the handling of finitary information (names of processes) in a coinductive settings; the use of named types for automatic inference of arguments similar to its use in template Meta-programming in C++; and the use of interactive programs in dependent type theory.

We introduce a simulator as an interactive program in Agda. The simulator allows to observe the evolving of processes following external or internal choices. Our aim is to use this in order to simulate railway interlocking system and write programs in Agda which directly use CSP processes.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.2.4 [*Software/Program Verification*]: Formal methods; D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.3 [*Language Constructs and Features*]: Abstract data types; Input/output; Patterns; D.4.1 [*Process Management (concurrency)*]; F.1.2 [*Modes of Comptuation*]: Interactive and reactive computation; F.3.1 [*Specifying and Verifying and Reasoning about Programs Mechanical verification*]; F.3.2 [*Semantics of Programming Languages*]: Process models; CSP; F.3.3 [*Studies of Program constructs*]: Functional constructs; Type structure; Agda; Dependent Type Theory; Dependently Typed Programming

General Terms Languages, Theory, Verification

Keywords Agda, CSP, Dependent Type Theory, Monadic Programming, Process Algebras, Interactive Program, Monad, IO-Monad, Coalgebras, Coinductive Data Types, Sized Types, Induction-Recursion, Universes.

TyDe'16, September 18, 2016, Nara, Japan
© 2016 ACM. 978-1-4503-4435-7/16/09...$15.00
http://dx.doi.org/10.1145/2976022.2976032

1. Introduction

Process algebras are one of the most important concepts for describing concurrent behaviours of programs. In functional programming a lot of work has been invested in developing concepts for defining interactive, usually sequential programs. The main approach is Moggi's IO monad [41]. Hancock and Setzer [32–34] have developed a version of the IO monad in dependent type theory, which for the sake of brevity we call in this paper HS-monad. The HS-monad has been used together with other ideas for formalising IO in Idris [14, 15]. The HS-monad covers currently only sequential programs. In this article we explore the representation of processes in dependent type theory as a step towards concurrent interactive programs. We will present as well an executable interactive program in Agda, which simulates processes. Our vision is to use this approach for writing concurrent programs in Agda, similarly to as it is done in the Java library JCSP [50]. The main example we are investigating are processes in the context of the European Rail Traffic Management System ERTMS [28], for which the first author has carried out some initial modelling in CSP. Our vision is that prototypes can be executed in Agda directly. Other examples one can envisage is to develop programs for networking in Agda.

The basis of functional programming are inductive data types and function types. In order to represent interactive programs, which are potentially non-terminating, in pure functional programming, special constructs are needed. The most commonly used construct is Moggi's IO monad [41]. The idea is that an element of $(\mathsf{IO}\ A)$ is an interactive program, which may or may not terminate, and if it terminates returns an element of type A. We can use the monadic bind to compose a $p : \mathsf{IO}\ A$ with a function $f : A \to \mathsf{IO}\ B$ to form an element of $(\mathsf{IO}\ \ B)$. The program is executed by first running p. If p terminates with result a, one continues running $(f\ \ a)$. Using nicely defined syntactic constructs, one can write sequences of operations in a way which looks similar to sequences of assignments in imperative style programming languages.

The HS-monad reduces the IO monad to coinductively defined types. An element of $(\mathsf{IO}\ \ A)$ is either a terminated program, or it is node of a non-well-founded tree having as label a command to be executed, and as branching degree the set of responses the real world gives in response to this command. In CSP-Agda we will model processes in a similar way. A CSP-Agda process can either terminate, returning a result. Or it can be a tree branching over external and internal choices, where for each such choice a continuing process is given. So instead of forming processes by using high level operators, as it is usually done in process algebras, our processes are given by these atomic one step operations. The high level operators are defined operations on these processes. This introduces a new concept to process, namely that of a monadic processes which when terminating returns a value. This facilitates the combination of processes in a modular way. Since processes are defined coinductively, we can introduce processes directly corecursively without having to use the recursion combinator.

Abel, Pientka, Thibodeau and Setzer have [5, 49] developed the notion of coinductive types as being defined by their elimination rules or observations. This notion has now been implemented in Agda. This has strong similarity to the notion of classes and objects in object oriented programming. Classes are essentially defined by their methods, and therefore given by their observations. Setzer, Abel and Adelsberger have used this approach in order to develop the notion of objects in dependent type theory [6, 48]. In CSP-Agda we will make extensive use of this approach. Using a record type, we access directly for non-terminating processes the choice sets and corresponding subprocesses. It turns out that this makes programming with processes much easier, since it avoids the use of auxiliary functions.

We will make extensive use of sized types [3]. The main reason is that in its puristic form, primitive corecursion or guarded recursion doesn't allow to apply any functions to the corecursion hypothesis. With sized types size preserving functions can be applied and therefore coinductive programs be written in a modular way. The index sets of processes will be given by universes, which are defined inductive-recursively [26].

We will make use of named types, as they are used in C++ template meta programming [8], to facilitate type inference. One example is the product type $A \times B$. In the Agda standard library, it translates into $\Sigma\ [x \in A]\ B$. If we have an element $(a\ ,\ b)$ of this type, we cannot infer B from it, since it we know only the instance $B\ [x := a]$. If we define $A \times B$ directly, we can infer the type. We have two types for representing finite sets directly, $(\mathsf{Fin}\ n)$ and $(\mathsf{NamedElements}\ s)$. If we evaluate the latter to $(\mathsf{Fin}\ n)$, type inference cannot differentiate between the two. Defining $(\mathsf{NamedElements}\ s)$ as a separate inductive type solves this problem.

The **structure of this paper** is as follows: In Sect. 2 we review the process algebra CSP. In Sect. 3 we give a brief introduction into the type theoretic language of Agda. In Sect. 4 we model CSP processes in Agda, and define the most important CSP operators in Agda. In Sec. 5 we introduce a simulator for CSP-Agda. In Sect. 6 we investigate related work, followed by a conclusion in Sect. 7.

Literal Agda code. All display style Agda code in this paper has been generated from type checked literal Agda files. We have hidden in the paper bureaucratic code, such as import of libraries, and restricted ourselves to the most important definitions. The full code of the library can be found at [36].

2. CSP

"Process algebras" were initiated in 1982 by Bergstra and Klop [10] in order to provide a formal semantics to concurrent systems. A "process" is a representation of the behaviour of a concurrent system. "Algebra" means that the system is dealt with in an algebraic and axiomatic way [9]. Process algebras allow to study distributed or parallel systems in an algebraic way. Most process algebras have basic operators to construct finite processes, synchronisation and parallel constructs to express concurrency, and a notion of recursion to obtain infinite behaviour. The main process algebras approaches are Calculus of Communicating Systems (*CCS*) [40], Communicating Sequential Processes (*CSP*) [16] and Algebra of Communicating Processes (*ACP*) [11]. The process algebras CSP [35, 45, 46] was developed by Hoare in 1978 [35].

Processes in CSP form a labelled transition system, where the one step transitions is written as

$$P \xrightarrow{\mu} Q \qquad \text{where } P, Q \text{ are processes and } \mu \text{ is an action,}$$

which means that process P can evolve to process Q by event μ. The event μ can be a label, the silent transition τ, or the termination event $\checkmark$. For example, the execution of the process $a \rightarrow b \rightarrow \mathsf{STOP}$ can be described by the LTS:

$$(\ a \rightarrow b \rightarrow \mathsf{STOP}) \xrightarrow{a} (b \rightarrow \mathsf{STOP}) \xrightarrow{b} \mathsf{STOP}$$

The operational semantics of CSP defines processes as states, and defines the transition rules between the states using firing rule. In Sect. 4 we will introduce firing rules for CSP operators (taken from [46]), and model them in Agda.

In the following table, we list the constructs for forming CSP processes. Here Q represent CSP processes:

Q ::=	STOP	STOP
|	SKIP	SKIP
|	prefix	$a \rightarrow Q$
|	external choice	$Q \Box Q$
|	internal choice	$Q \sqcap Q$
|	hiding	$Q \setminus a$
|	renaming	$Q[R]$
|	parallel	$Q\ {}_X\|\|_Y\ Q$
|	interleaving	$Q \mid\mid\mid Q$
|	interrupt	$Q \triangle Q$
|	composition	$Q \fatsemi Q$

There are as well indexed versions of $\Box$, $\sqcap$, $\|$, $|||$. They are indexed over finite sets, and therefore can be reduced to the binary case. We will treat explicitly in this article only the binary case.

In this article we follow the version of CSP used in [45, 46]. All rules are taken from [46]. In the rules we follow the convention of this book that a ranges over $\mathrm{Label} \cup \{\checkmark\}$ and μ over $\mathrm{Label} \cup \{\checkmark, \tau\}$.

3. Agda

Agda [7, 13] is a theorem prover and dependently typed programming language, which extends intensional Martin-Löf type theory [39]. It is closely related to the theorem prover Coq [20, 21, 44]. Predicates are given as types, the elements of which are proofs of that property. Agda has a termination and coverage checker. This makes Agda a total language, so each Agda program terminates. Without the termination and coverage checker, Agda would be inconsistent. The current version of this language is Agda 2 which has been designed and implemented by Ulf Norell in his PhD in 2007 [43]. The user interface of Agda is Emacs. This interface has been useful for interactively writing and verifying proofs [12]. Programs can be developed incrementally, since we can leave parts of the program unfinished, and programmers can get useful information from Agda on how to fill in the missing parts by type checking. Agda has a type checker which refuses incorrect proofs by detecting unmatched types. The type checker in Agda shows the goals and the environment information related to proof. The coverage checker guarantees that the definition of a function covers all possible cases, and the termination checker verifies that all programs terminate.

There are several levels of types in Agda, the lowest is for historic reasons called Set. Types in Agda are given as dependent function types, inductive types and coinductive types. In addition there exist record types (which are in the newer approach used for defining coinductive types) and a generalisation of inductive-recursive definitions. Inductive data type are dependent versions of algebraic data types as they occur in functional programming. Inductive data types are given as sets A together with constructors which are strictly positive in A. For instance the collection of finite sets is given as

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} (i : Fin n) → Fin (suc n)
```

Here $\{n : \mathbb{N}\}$ is an implicit argument. Implicit arguments are omitted, provided they can be uniquely determined by the type checker. We can make a hidden argument explicit by writing for instance zero $\{n\}$.

The above definition introduces a new type Fin : $\mathbb{N} \to$ Set where (Fin n) is a type with n elements. The elements of (Fin n) are those constructed from applying these constructors. Therefore we can define functions by case distinction on these constructors using pattern matching, e.g.

```
toℕ : ∀ {n} → Fin n → ℕ
toℕ zero      = 0
toℕ (suc n)   = suc (toℕ n)
```

Nested patterns are allowed. The coverage checker checks completeness and the termination checker checks that the recursive calls follow a schema of extended primitive recursion.

There are two approaches of defining coinductive types in Agda. The older approach is based on the notion of codata types. We will follow the newer one, pioneered by the second author, Abel, Pientka and Thibodeau [5, 49], which is based on coalgebras given by their observations or eliminators. Consider

```
record Stream (i : Size) : Set where
  coinductive
  constructor cons'
  field
    head : ℕ
    tail : {j : Size< i} → Stream j

open Stream
```

If we first ignore the arguments Size, Size< we see that the type Stream is given as a record type in Agda. It is defined coinductively by its observations head, tail. We have an automatically generated constructor cons', which is only useful for records not involving sizes. For records involving Sizes, one can use the self-defined constructor cons:

```
cons : ∀ {i} → ℕ → Stream i → Stream (↑ i)
head (cons n s) = n
tail (cons n s) = s
```

Here we used the notation $\forall\ \{a\} \to \cdots$, which stands for $\{a : A\} \to \cdots$, where A can be inferred by Agda.

Elements of Stream are defined by copattern matching, i.e. by determining the result of applying head, tail to them. Without sizes, only recursive calls to the function being defined are possible, with no restrictions on the arguments they are applied to. No functions can be applied to the recursive calls. This restriction is called the principle of guarded recursion [22] or primitive corecursion. An example is the tail component in the pointwise addition of two streams:

```
_+s_ : ∀ {i} → Stream i → Stream i → Stream i
head (s +s s') = head s + head s'
tail (s +s s') = tail s +s tail s'
```

Here we use Agda's mechanism for mixfix operators, where the arguments of a mixfix operator are denoted by underscore (_). s +s s' stands for (_+s_ s s').

_+s_ makes a recursive call to tail s +s tail s'. Note that s, s' are arguments of _+s_, so we can apply tail to them freely. Without the guarded recursion restriction, one could define non productive definitions, e.g. define tail $(f\,x)$ = tail $(f\,x)$.

However this restriction makes it difficult to define streams in a modular way, therefore sized types [2, 3] are used. Sizes are essentially ordinals (without infinite branching one can think of them as natural numbers), however there is an additional infinite size ∞. We can explicitly only access the size ∞, the successor operation on sizes $\uparrow$ and smaller sizes using Size<. The idea is that for ordinal sizes $i \neq \infty$, a stream s : Stream i allows up to i times of applications of tail, whereas an s : Stream ∞ allows arbitrary many applications of tail. When defining an element f : (i : Size) $\to A\ i \to$ Stream i by corecursion, tail $(s\ (f\ i\ a))\ \{j\}$ must be an element of size $\geq j$ which can refer to a recursive call $(f\ j\ a')$, and we can apply functions to it as long as the resulting size is $\geq j$. Since we don't have access to any size $< j$ (j could be the smallest size), we are not able to eliminate on the recursive call itself. This means that we can apply size preserving and size increasing functions to the recursive call, which guarantees that streams are productive. We have ∞ : Size< ∞, so a recursive definition of elements of Stream ∞ can refer to itself. One could say that with sized types we define two functions: One using ordinal sizes, which is used to calculate the correct usage of sizes. The other one is where sizes are replaced by ∞.

An example of applying the size preserving function _+s_ to the recursion hypothesis is the stream of Fibonacci numbers as defined e.g. in [4]:

```
fib : ∀ {i} → Stream i
head fib              = 1
head (tail fib)       = 1
tail (tail fib)       = fib +s tail fib
```

4. The Library CSP-Agda

In process algebras, if a process terminates, it does not return any information except for that it terminated. [1] We want to define processes in a monadic way in order to combine them in a modular way. Therefore, if processes terminate, they should return some additional information, namely the result returned by the process.

In functional programming, a monad is given by a functor M together with morphisms $\gg\!=$: M $A \to (A \to$ M $B) \to$ M B and return : $A \to$ M A such that the following laws hold:

$$\begin{array}{lcl}\mathsf{return}\ a \gg\!= f & = & f\ a \\ p \gg\!= \mathsf{return} & = & p \\ (p \gg\!= f) \gg\!= g & = & p \gg\!= (\lambda x. f\ x \gg\!= g)\end{array}$$

[1] See below for a discussion on terminated processes vs terminating events as they occur in CSP.

The type of interactive programs can be considered as a monad in the following way:

- For a given set A, $(\mathsf{M}\ A)$ is the set of interactive programs which may or may not terminate, and if they terminate, they will return a result $a : A$.
- Assume P is program in $(\mathsf{M}\ A)$, and Q is a function which for $a : A$ returns a program in $(\mathsf{M}\ B)$. then $P \gg= Q$ is the program which executes as follows: First P is executed. If P terminates with result a then we continue executing $(Q\ a)$. The result of the whole process is the result of $(Q\ a)$ (if it terminates).
- The program $(\mathtt{return}\ a)$ will terminate without any interaction with result a.

Processes in our approach are similar to interactive programs. They are defined using an atomic operation, corresponding to the next transitions they can make. Since processes can loop for ever, they are defined coinductively. The standard CSP-operators are in our approach defined rather than atomic as in process algebras. Since processes are given coinductively, we can introduce processes by primitive corecursion (also called guarded recursion). The principle of primitive corecursion, which is enforced by Agda's termination checker, will guarantee processes to be productive, which means for a process we can determine whether it terminates or not, and, in case it terminates, the result returned, and, in case it doesn't terminate, which next transitions it can make, and the next processes after firing these transitions.

Terminated processes vs termination events. In CSP termination is handled by events. A process can terminate, which is modelled by an event with reserved label ✓. If $P \xrightarrow{\checkmark} P'$, then P' is a deadlocked process, which is in all standard semantic models of CSP equal to the process STOP. A first step towards a monadic version of processes is that we add a return value to ✓-transitions. This is the result returned when the process terminates, which can be used for choosing a continuation e.g. in monadic bind. Since P' is equal to STOP we can omit it and just write $P \xrightarrow{\checkmark,a}$ for P having a termination event with return value a.

Adapted to the monadic setting, we have the CSP process $(\mathsf{SKIP}\ \ a)$ (for a return value a) which has as only transition $\mathsf{SKIP}\ \ a \xrightarrow{\checkmark,a}$. We want to have as well a terminated process $(\mathsf{terminate}\ \ a)$ with result a, which is our name for the monadic $(\mathtt{return}\ a)$. $(\mathsf{terminate}\ a)$ is very similar to $(\mathsf{SKIP}\ a)$, except that $(\mathsf{terminate}\ \ a)$ has terminated, whereas $(\mathsf{SKIP}\ \ a)$ will terminate. If we lift $(\fatsemi)$ to a monadic $\gg=$ we get $\mathsf{SKIP}\ \ a \gg= Q \xrightarrow{\tau} Q\ a$, whereas we want definitionally $\mathsf{terminate}\ a \gg= Q = Q\ a$ without a τ-transition. Semantically this doesn't make a difference, since in the various semantics of CSP we have $\tau \longrightarrow P = P$. When using it, it makes a difference, since when composing processes we don't want a τ-transition in between.

Because of the equation $\tau \longrightarrow P = P$, we could use $(\mathsf{SKIP}\ a)$ for $(\mathsf{terminate}\ a)$ and optimise the rules to guarantee $\mathsf{SKIP}\ \ a \gg= Q\ \ =\ \ Q\ \ a$. However, this makes the code very complex. It seems to be a better approach to have a separate process $(\mathsf{terminate}\ \ a)$. That process will be in CSP semantics equal to $(\mathsf{SKIP}\ a)$. When defining CSP operators applied to arguments, we define it for the argument $(\mathsf{terminate}\ \ a)$ in the same way as for the argument $(\mathsf{SKIP}\ a)$. However, if the result is a process which has only one τ-transition to a process P, we return instead, when the argument is $(\mathsf{terminate}\ \ a)$, directly process P without the τ-transition. So we have two kinds of terminated processes in CSP-Agda. One is the result of following a termination event ✓, and one is the new terminated process $(\mathsf{terminate}\ a)$.

4.1 Representing CSP Processes in Agda

In a monadic version, a process $P\ \ :\ \ \mathsf{Process}\ A$ is either a terminating process $(\mathsf{terminate}\ a)$, which has return value $a\ \ :\ \ A$, or it is process $(\mathsf{node}\ P)$ which progresses. Here $P\ \ :\ \ \mathsf{Process+}\ A$, where $(\mathsf{Process+}\ A)$ is the type of progressing processes. A progressing process can proceed at any time with labelled transitions (external choices), silent transitions (internal choices), or ✓-events (termination). After a ✓-event, the process becomes deadlocked, so there is no need to determine the process after that event. However, as discussed before we will add a return value $a\ \ :\ \ A$ to ✓-events.

Elements of $(\mathsf{Process+}\ A)$ are therefore determined by

(1) an index set E of external choices and for each external choice e the Label $(\mathsf{Lab}\ e)$ and the next process $(\mathsf{PE}\ e)$;

(2) an index set of internal choices I and for each internal choice i the next process $(\mathsf{PI}\ i)$; and

(3) an index set of termination choices T corresponding to ✓-events and for each termination choice t the return value $\mathsf{PT}\ t\ \ :\ \ A$.

One might consider reducing the number of components by unifying the choice sets and adding τ and ✓ to the set of labels. However, the operators of CSP handle external, internal, and termination transitions quite differently. If we encoded them as one choice set, we would for each operator have to select the choices corresponding to these categories, form the new choices and recombine them. Keeping them as separate entities makes programming much easier.

We define $(\mathsf{Process+}\ A)$ as a record. Definition of elements of it by copattern matching is very convenient, since it avoids the need to define the components of $(\mathsf{Process+}\ \ A)$ as auxiliary functions as one would have to do when using data.

Processes need to be defined coinductively instead of inductively – otherwise processes would always after finitely many transitions eventually terminate. Processes will therefore be defined by primitive corecursion or guarded recursion. The left hand side of a primitive corecursion scheme needs to have an observation applied to the element of the coinductive type to be defined. We could do this using $(\mathsf{Process+}\ A)$. However, this would mean that for defining a process by primitive corecursion, we need to define all the 7 components. It seems to be natural to define processes by primitive corecursion where we want to equate the result of applying an eliminator to a process directly with a process formed from other processes, without having to define all 7 components. Because of this we introduce a third type of processes, $(\mathsf{Process}\infty\ A)$ which has a observation forcep returning an element of $(\mathsf{Process}\ A)$.

We will develop a simulator for processes, which displays the evolving of processes following external and internal choices. The simulator needs to display processes as strings. Since processes are infinite objects, we cannot directly compute such finite strings. The solution is to add a new field $\mathsf{Str+}$ to $(\mathsf{Process+}\ A)$ which determines the string. That string needs to be user-defined. We need to add as well a field $\mathsf{Str}\infty$ to $(\mathsf{Process}\infty\ \ A)$. The reason is that we can only use $\mathsf{Str+}$ to obtain a string from an element of $(\mathsf{Process}\infty\ \ A)$, if we have a smaller size available, which in general is not the case. This is no artificial restriction imposed by sizes: Without this field it is in general not possible to compute a string. For instance, we could define elements of $(\mathsf{Process}\infty\ \ A)$ corecursively without assigning to it a string directly. Then any string computed would need to be infinite.

We model the sets of external, internal, and termination choices as elements of an inductive-recursively defined universe Choice. Elements c of Choice are codes for finite sets, and $(\mathsf{ChoiceSet}\ c)$ is the set it denotes. In addition we define a string $(\mathsf{choice2Str}\ c)$ representing c, and a function $\mathsf{choice2Enum}$ which computes from

c a list of all choices. This will be used to print a list of choices in the simulator for CSP processes.

We require as well that the set of return values are elements of Choice. This allows us to print the result returned when a process terminates. However, for the return types it is not needed that they are finite sets. We plan to introduce in future a separate universe for return values, where we only require that a string can be computed for each element, but drop the requirement to compute an enumeration of its elements. Then we could have the set of natural numbers as a return value, which could be useful for defining processes by recursion over the natural numbers.

The resulting code for processes in Agda is as follows:

```
mutual
  record Process∞ (i : Size) (c : Choice) : Set where
    coinductive
    field
      forcep : {j : Size< i} → Process j c
      Str∞  : String

  data Process (i : Size)  (c : Choice) : Set where
    terminate  : ChoiceSet  c    → Process i c
    node       : Process+   i c  → Process i c

  record Process+ (i : Size) (c : Choice) : Set where
    constructor process+
    coinductive
    field
      E    :  Choice
      Lab  :  ChoiceSet  E  →  Label
      PE   :  ChoiceSet  E  →  Process∞  i  c
      I    :  Choice
      PI   :  ChoiceSet  I  →  Process∞  i  c
      T    :  Choice
      PT   :  ChoiceSet  T  →  ChoiceSet c
      Str+  :  String
```

An example of a process is as follows:

```
P =    node (process+ E Lab PE I PI T PT "P")
       : Process String  where
E =    code for {1,2}            I =  code for {3,4}
T =    code for {5}
Lab 1   =   a         Lab 2   =   b        PE 1   =   P₁
PE 2    =   P₂        PI 3    =   P₃       PI 4   =   P₄
PT 5    =   "STOP"
```

$$\begin{tikzcd}
 & & P \arrow[lld, "1", "a"'] \arrow[ld, "2", "b"'] \arrow[d, "3"', "\tau"] \arrow[rd, "4", "\tau"'] \arrow[rrd, "5", "\checkmark"'] & & \\
P_1 & P_2 & P_3 & P_4 & \texttt{"STOP"}
\end{tikzcd}$$

The universe of choices is given by a set Choice of codes for choice sets, and a function ChoiceSet, which maps a code to the choice set it denotes. Universes were introduced by Martin-Löf (e.g. [39]) in order to formulate the notion of a type consisting of types. Universes are defined in Agda by an inductive-recursive definition [23–26]: we define inductively the set of codes in the universe while recursively defining the decoding function.

We give here the code expressing that Choice is closed under fin, ⊎', ×', subset', Σ' and namedElements. Closure under other operations can easily be added as needed. The type (NamedElements l) is essentially (Fin (length l)). The function choice2Str will for elements of this set print the nth element of l, giving them more meaningful names. We don't equate (NamedElements l) with (Fin (length l)). This facilitates type inference. subset $A\ f$ is the set of $a : A$ such that $(f\ a)$ is true.

We could have defined Choice simply as the collection of finite sets (Fin n). However, then the indices of choice sets would loose connection with the actual types constructed. For instance in case of external choice $P\ \square\ Q$, in our setting a choice (inj₁ x) refers to P, and a choice (inj₂ x) refers to Q.

```
data NamedElements (s : List String) : Set where
  ne : Fin (length s) → NamedElements s

mutual
data Choice : Set where
  fin      : ℕ → Choice
  _⊎'_   : Choice → Choice → Choice
  _×'_   : Choice → Choice → Choice
  namedElements : List String → Choice
  subset'  :  (E  :  Choice)  →  (ChoiceSet  E  →  Bool)
                →  Choice
  Σ'          :  (E  :  Choice)  →  (ChoiceSet  E  →  Choice)
                →  Choice

ChoiceSet : Choice → Set
ChoiceSet  (fin n)          =   Fin n
ChoiceSet  (s ⊎' t)        =   ChoiceSet  s   ⊎   ChoiceSet  t
ChoiceSet  (E ×' F)        =   ChoiceSet  E   ×   ChoiceSet  F
ChoiceSet  (namedElements s)  =  NamedElements s
ChoiceSet  (subset' E f)        =  subset (ChoiceSet E) f
ChoiceSet  (Σ' A B)   =   Σ[ x ∈ ChoiceSet A ]
                           ChoiceSet (B x)

choice2Str : {c : Choice} → ChoiceSet c  → String
choice2Str {fin n} m = showℕ (toℕ m)
 ...

choice2Enum : (c : Choice) → List (ChoiceSet c)
choice2Enum (fin n) = fin2Option0 n
 ...
```

(Lab P) can return the same value for different elements of (Lab P), therefore a process can have several transitions with the same label. This is in accordance with CSP. One could instead demand that for each label there is at most one transition possible, and replace processes having several transitions with the same label by one which has one transition followed by silent transitions to the different choices.

4.2 Sequential Composition

In CSP the semi-colon operator ($\fatsemi$) is used for sequencing two processes, where, if the first process terminates, control is passed to a second one. The rules for sequential composition in CSP are as follows:

$$\frac{P \xrightarrow{\checkmark} \bar{P}}{P \fatsemi Q \xrightarrow{\tau} Q} \qquad \frac{P \xrightarrow{\mu} \bar{P}}{P \fatsemi Q \xrightarrow{\mu} \bar{P} \fatsemi Q}\ [\mu \neq \checkmark]$$

In CSP-Agda we have monadic composition $P \gg= Q$, where Q depends on the return value of P. We obtain therefore in monadic form SKIP $a \gg= Q \xrightarrow{\tau} Q\ a$ as only transition. (terminate a) should behave as (SKIP a), however we omit unnecessary τ-transitions. Therefore we define terminate $a \gg= Q = Q\ a$. If P has a $\checkmark, a$-event, we cannot define $P \gg= Q = Q\ a$, since P could have other external or internal choices. Therefore, a τ-transition before continuing with $(Q\ a)$ will be added, as it happens in original CSP. In the following code choice2Str2Str converts a function ChoiceSet $c \to$ String into a meaningful string, making a case distinction on the argument. In $P' := (P \gg= + Q)$, external choices of P become external choices of P' using a recursive call, similarly for internal choices. For termination events of P with return value PT P c $=$ a, we get additional internal choice transitions $P' \xrightarrow{\tau} Q\ a$.

```
_≫=Str_ : {c₀ : Choice} → String
   → (ChoiceSet c₀ → String)          → String
s ≫=Str f = s  ++s ";" ++s choice2Str2Str f
```

```
mutual
   _≫=∞_ : {i : Size} → {c₀ c₁ : Choice}
           → Process∞ i c₀
           → (ChoiceSet c₀ → Process∞ i c₁)
           → Process∞ i c₁
   forcep (P ≫=∞ Q)  =  forcep P  ≫= Q
   Str∞  (P ≫=∞ Q)  =  Str∞ P ≫=Str (Str∞ ∘ Q)

   _≫=_ : {i : Size} → {c₀ c₁ : Choice}
           → Process i c₀
           → (ChoiceSet c₀ → Process∞ (↑ i) c₁)
           → Process i c₁
   node       P  ≫=  Q  =  node    (P ≫=+ Q)
   terminate x  ≫=  Q  =  forcep  (Q x)

   _≫=+_ : {i : Size} → {c₀ c₁ : Choice}
           → Process+ i c₀
           → (ChoiceSet c₀ → Process∞ i c₁)
           → Process+ i c₁
   E    (P ≫=+ Q)           = E    P
   Lab  (P ≫=+ Q)           = Lab  P
   PE   (P ≫=+ Q) c         = PE  P  c     ≫=∞  Q
   I    (P ≫=+ Q)           = I P ⊎' T P
   PI   (P ≫=+ Q) (inj₁ c)  = PI   P  c     ≫=∞  Q
   PI   (P ≫=+ Q) (inj₂ c)  = Q (PT P c)
   T    (P ≫=+ Q)           = ∅'
   PT   (P ≫=+ Q) ()
   Str+ (P ≫=+ Q)           = Str+ P ≫=Str (Str∞ ∘ Q)
```

The above code introduces a pattern of defining operators on Process by defining simultaneously operators on the three categories of processes Process∞, Process, and Process+, where the qualifier ∞, p, + attached to the operator refer to the 3 categories of processes, respectively. We often omit p, and omitted it in case of _≫=_. We have as well a string forming operation indicated by Str, and sometimes a result type forming operation indicated by Res. For some binary operators we need versions where the arguments are from different categories of processes, in which case we add two qualifiers to the operators. We will in the following only present the main cases of the operators. Especially, we will omit the functions involving Process∞, which in most cases follow the same pattern as the definition of _≫=∞_ above, and the string forming operation, which is easy to define. The full code can be found at [36].

4.3 The Recursion Operator

We can define recursion in a similar way to _≫=_. The operation takes an s : String, f : ChoiceSet $c_0 \to$ Process+ i $(c_0 \uplus' c_1)$ and an a : ChoiceSetc_0 and returns a process (rec s f a) which operates as follows: We start with process $(f\ a)$ and follow its external and internal choices. If it terminates with result (inj₂ x), the recursion terminates with result x. If it terminates with result (inj₁ a'), we recursively start again, with a replaced by a'.

However, in case $(f\ \ x)$ terminates immediately, this procedure (unless we put a τ-transition after each loop iteration) will result potentially in a black hole recursion. To avoid this we require f x : Process+, which is the type of processes which have not terminated. We have an argument s which is the name of the resulting process, since an automatically generated name would in most cases be unreadable.

The Agda code is as follows ((renameP *name* P) renames the Str+ component of process P to *name*):

```
mutual
   rec : {i : Size} → {c₀ c₁ : Choice}
           → (s : String)
           → (ChoiceSet  c₀ → Process+ (↑ i) (c₀ ⊎' c₁))
           → ChoiceSet   c₀
           → Process∞    i c₁
   forcep (rec s f a)  =   renameP s
                           (f a ≫=+p recaux s f)
   Str∞   (rec s f a)  =   s

   recaux  : {i : Size} →  {c₀ c₁ : Choice}
           → (s : String)
           → (ChoiceSet c₀ → Process+ (↑ i) (c₀ ⊎' c₁))
           → (ChoiceSet c₀ ⊎ ChoiceSet c₁)
           → Process∞ i c₁
   recaux s f (inj₁ x)  =  rec     s f x
   recaux s f (inj₂ x)  =  delay  (terminate x )
```

4.4 STOP, SKIP, Terminate, DIV

The STOP process in CSP is the deadlocked process, which refuses all communication. It has no transition rule. It can be modelled as a process which has empty external, internal and termination choice sets ∅'. The components Lab, PE, PI, PT have as domain the empty set, and can be given by the function efq (for ex falsum quodlibet) which is defined by the empty case distinction, as denoted by (). The name of STOP is "STOP".

```
efq : {A : Set} → Fin 0 →  A
efq ()

STOP+ : {i : Size} → (c : Choice) →  Process+ i c
STOP+ c =  process+ ∅' efq efq ∅' efq ∅' efq "STOP"
```

The CSP process SKIP terminates immediately. Its only transition is

$$\text{SKIP} \xrightarrow{\checkmark} \text{STOP}$$

In CSP we have that SKIP $\fatsemi P \xrightarrow{\tau} P$ instead of SKIP $\fatsemi P = P$. Therefore SKIP is not the process (terminate a) but a process which has no external or internal choices and only one ✓ choice with a given return value. Let ⊤' $=$ fin 1 be the one element choice set. SKIP is defined as follows:

```
SKIP+ : {i : Size} → {c : Choice} → (a : ChoiceSet c)
          → Process+ i c
SKIP+ a
  = process+ ∅' efq efq ∅' efq ⊤' (λ _ → a)
    ("SKIP(" ++s choice2Str a ++s ")")
```

We have as well the terminating process given by

```
terminate: {i : Size} → (c : Choice) → (a : ChoiceSet c)
           → Process+ i c
```

Direct divergence in the sense of black hole recursion does not occur in CSP-Agda, since productivity is guaranteed by Agda's termination checker. Note that in case of recursion, productivity is guaranteed by referring to the type of not-terminated processes Process+. However one can easily define a process which has infinitely many τ transitions to itself:

```
mutual
  DIV∞  : {i : Size} → {c : Choice} → Process∞ i c
  forcep  DIV∞  = node (process+ ∅' efq efq ⊤'
     (λ _ → DIV∞) ∅' efq "DIV")
  Str∞   DIV∞  =  "DIV"
```

4.5 Prefix

The prefix operator $a \rightarrow P$ has only one transition

$$(a \rightarrow P) \xrightarrow{a} P$$

So it is the process with one external choice with label a and continuation P, and empty internal and ✓-choices:

```
_⟶+_ : {i : Size} → {c : Choice} → Label
   → Process∞ i c → Process+ i c
l ⟶+ P = process+ ⊤' (λ _ → l) (λ _ → P) ∅' efq ∅' efq
     (l ⟶Str Str∞ P )
```

4.6 Internal Choice

The internal choice operator has the following transitions:

$$P \sqcap Q \xrightarrow{\tau} P \qquad\qquad P \sqcap Q \xrightarrow{\tau} Q$$

It is modelled in CSP-Agda by having as internal choice set bool and otherwise empty choices:

```
bool : Choice
bool = fin 2

if_then_else : {A : Set} → ChoiceSet bool → A → A → A
if zero then a else b        = a
if (suc zero) then a else b  =  b
if (suc (suc ())) then a else b

_⊓+_ : {i : Size} → {c : Choice} → Process∞ i c
   → Process∞ i c → Process+ i c
P ⊓+ Q =
   process+ ∅' efq efq bool (λ b → if b then P else Q) ∅' efq
     (Str∞ P ⊓Str Str∞ Q)
```

4.7 External Choice

External choice allows the environment to make the choice between the behaviour of the processes. For instance, the process $(a \rightarrow P \,\square\, b \rightarrow Q)$ can engage in either of the events a or b. If the first event chosen was a, the posterior behaviour is described by P, and if it was b, the process will behave as Q. The inference rules for external choice are as follows (having an inference rule with two conclusions is an abbreviation for two inference rule, one deriving the first and one deriving the second conclusion):

$$\frac{P \xrightarrow{a} \bar{P}}{\begin{array}{c} P \,\square\, Q \xrightarrow{a} \bar{P} \\ Q \,\square\, P \xrightarrow{a} \bar{P} \end{array}} \qquad\qquad \frac{P \xrightarrow{\tau} \bar{P}}{\begin{array}{c} P \,\square\, Q \xrightarrow{\tau} \bar{P} \,\square\, Q \\ Q \,\square\, P \xrightarrow{\tau} Q \,\square\, \bar{P} \end{array}}$$

Assume processes P : Process i c_0 and Q : Process i c_1 and consider $P \,\square\, Q$. If P or Q terminates, then $P \,\square\, Q$ can terminate with the return value of that process. In case both processes are of the form terminate we need to be consistent with the behaviour we would have if both processes were SKIP: in that case the process could have two ✓-events corresponding to each of the two return value. So we get again return values in c_0 or c_1. The result returned is therefore always in c_0 or c_1, i.e. an element of the disjoint union $(c_0 \uplus c_1)$ of c_0 and c_1. In case P and Q have not terminated the defining equations are obvious from the rules. The only problem is that we have to map the return values of the processes (PE P c) to the return value of $P \,\square\, Q$. We do this by using the function fmap defined below.

If both processes terminate, as said before we obtain a process which can terminate with each of two given return values. So we obtain (2-✓ a b) which is the process which can make ✓ transitions for return values (inj₁ a) and (inj₂ a). We would prefer to return (terminate (a ,, b)), but being consistent with that (terminate a) should be semantically equal to (SKIP a) requires this choice. In case of (terminate a □ P) we get a more complex behaviour: (1) the combined process can terminate with result a; (2) it can follow an internal choice of P, after which the possibility having a transition as in (1) remains; (3) we can have a termination event of P, in which case the result returned is that of P; (4) we can have an external choice of P, in which case information about termination of the first process is lost. What we get is that the combined process behaves as P, but the return value needs to be mapped to the return value of the combined value. In addition, we need to add using addTimed✓ a timed tick event, which provides the possibility of having a transition $\xrightarrow{✓,a}$, as long as the process hasn't performed an external choice operation. We obtain the following code:

```
mutual
   _□_ : {c₀ c₁ : Choice} → {i : Size} → Process i c₀
        → Process i c₁ → Process i (c₀ ⊎' c₁)
   node P        □ Q               =  P □+p Q
   P             □ node Q          =  P □p+ Q
   terminate a   □ terminate b     =  2-✓ a b

   _□+p_ : {c₀ c₁ : Choice} → {i : Size}
     → Process+ i c₀ → Process i c₁
     → Process i (c₀ ⊎' c₁)
   P □+p  terminate b  = addTimed✓ (inj₂ b)
     (node (fmap+ inj₁ P) )
   P □+p  node Q        =  node (P □+ Q)

   _□+_ : {c₀ c₁ : Choice} → {i : Size}
     → Process+ i c₀ → Process+ i c₁
     → Process+ i (c₀ ⊎' c₁)
   E    (P □+ Q)            = E P ⊎' E Q
   Lab  (P □+ Q) (inj₁ x)  = Lab P x
   Lab  (P □+ Q) (inj₂ x)  = Lab Q x
   PE   (P □+ Q) (inj₁ x)  = fmap∞ inj₁ (PE P x)
   PE   (P □+ Q) (inj₂ x)  = fmap∞ inj₂ (PE Q x)
   I    (P □+ Q)            = I P ⊎'  I Q
   PI   (P □+ Q) (inj₁ c)  = PI P c □∞+ Q
   PI   (P □+ Q) (inj₂ c)  = P  □+∞ PI Q c
   T    (P □+ Q)            = T P ⊎' T Q
   PT   (P □+ Q) (inj₁ c)  = inj₁ (PT P c)
PT   (P □+ Q) (inj₂ c)  = inj₂ (PT Q c)
Str+ (P □+ Q)           = Str+ P □Str Str+ Q
```

We used here the function which adds the possibility of terminating with result a, which is only available, as long as the process hasn't performed an external choice

```
addTimed✓+ : ∀ {i} → {c : Choice}
  → (a : ChoiceSet c)
  → Process+ i c → Process+ i c
E   (addTimed✓+ a P)   = E P
Lab (addTimed✓+ a P)   = Lab P
PE  (addTimed✓+ a P) s = PE P s
I   (addTimed✓+ a P)   = I P
PI  (addTimed✓+ a P) s =
  addTimed✓∞ a (PI P s)
T   (addTimed✓+ a P)   = T'  ⊎'  T P
PT  (addTimed✓+ a P) (inj₁ _) = a
PT  (addTimed✓+ a P) (inj₂ c) = PT P c
Str+ (addTimed✓+  a P)  =
  addTimed✓Str a      (Str+ P)
```

The process having two tick events for two values is defined as follows:

```
2-✓+ : ∀ {i} → {c₀ c₁ : Choice} → (a : ChoiceSet c₀)
  → (a' : ChoiceSet c₁) → Process+ i (c₀ ⊎' c₁)
2-✓+ a a' = process+ ∅' efq efq ∅' efq bool
    (λ b → if b then (inj₁ a) else (inj₂ a'))
    (2-✓Str a a')
```

The function fmap mapping (Process i c_0) to (Process i c_1) by applying a function (f : ChoiceSet c_0 → ChoiceSet c_1) to the return values can be defined using monadic composition:

```
fmap : {c₀ c₁ : Choice} → {i : Size}
  → (f : ChoiceSet c₀ → ChoiceSet c₁)
  → Process i c₀ → Process i c₁
fmap f P = P ≫= (delay ∘ terminate ∘ f)
```

4.8 Renaming

The renaming operator takes a process and renames the external choice labels by applying a function to them. It is modelled in CSP-Agda as follows:

```
Rename+ : {i : Size} → {c : Choice}
  →  (f : Label → Label)
  →  Process+ i c → Process+ i c
E   (Rename+ f P)   = (E P)
Lab (Rename+ f P) c = f (Lab P c)
PE  (Rename+ f P) c = Rename∞ f (PE P c)
I   (Rename+ f P)   = I P
PI  (Rename+ f P) c = Rename∞ f (PI P c)
T   (Rename+ f P)   = T P
PT  (Rename+ f P) c = PT P c
Str+ (Rename+ f P)  = RenameStr f (Str+ P)
```

4.9 Parallel Operator, Interleaving, and Interrupt

The parallel and interleaving operators enforce two processes to work together and interact through synchronous events. The Agda-code for the parallel operator is quite long (see the CSP-library [36] for its code), and therefore we present in this short paper only the interleaving operator. In the library one can find as well the interrupt operator which we omit because of lack of space in this article. The interleaving operator executes the external and internal choices of its arguments P and Q completely independently of each other. The CSP rules are as follows:

$$\frac{P \xrightarrow{\checkmark} \bar{P} \qquad Q \xrightarrow{\checkmark} \bar{Q}}{P \mid\mid\mid Q \xrightarrow{\checkmark} \bar{P} \mid\mid\mid \bar{Q}} \qquad \frac{P \xrightarrow{\mu} \bar{P}}{\begin{array}{c} P \mid\mid\mid Q \xrightarrow{\mu} \bar{P} \mid\mid\mid Q \\[1ex] Q \mid\mid\mid P \xrightarrow{\mu} Q \mid\mid\mid \bar{P} \end{array}}\ \mu \neq \checkmark$$

The definition in CSP-Agda is as follows:

```
mutual
 _|||_ : {i : Size} → {c₀ c₁ : Choice} → Process i c₀
      → Process i c₁ → Process i (c₀ ×' c₁)
 node P  ||| node Q  =  node  (P |||++ Q)
 terminate a |||  Q    =  fmap  (λ b → (a ,, b)) Q
 P |||  terminate b    =  fmap  (λ a → (a ,, b)) P

 _|||++_ : {i : Size} → {c₀ c₁ : Choice}
    → Process+ i c₀ → Process+ i c₁
    → Process+ i (c₀ ×' c₁)
 E    (P |||++ Q)            =  E P ⊎' E Q
 Lab  (P |||++ Q) (inj₁ c)   =  Lab P c
 Lab  (P |||++ Q) (inj₂ c)   =  Lab Q c
 PE   (P |||++ Q) (inj₁ c)   =  PE P c |||∞+  Q
 PE   (P |||++ Q) (inj₂ c)   =  P |||+∞ PE Q c
 I    (P |||++ Q)            =  I P ⊎' I Q
 PI   (P |||++ Q) (inj₁ c)   =  PI P c |||∞+  Q
 PI   (P |||++ Q) (inj₂ c)   =  P |||+∞ PI Q c
 T    (P |||++ Q)            =  T P ×' T Q
 PT   (P |||++ Q) (c ,, c₁)  =  PT P c ,, PT Q c₁
 Str+ (P |||++ Q)            =  Str+ P |||Str Str+ Q
```

When processes P and Q haven't terminated, then $P \mid\mid\mid Q$ will not terminate. The external choices are the external choices of P and Q. The labels are the labels from the processes P and Q, and we continue recursively with the interleaving combination. The internal choices are defined similarly. A termination event can happen only if both processes have a termination event.

If one process terminates but the other not, the rules of CSP express that one continues as the other other process, until it has terminated. We can therefore equate, if P has terminated, $P \mid\mid\mid Q$ with Q. However, we record the result obtained by P, and therefore apply fmap to Q in order to add the result of P to the result of Q when it terminates. If both processes terminate with results a and b, then the interleaving combination terminates with result $(a \,,\, b)$.

4.10 Hiding

Hiding allows to hide some external transitions and replace them by silent ones in order to hide them from other processes. The behaviour of the hiding operator is shown by the following firing rules:

$$\frac{P \xrightarrow{a} \bar{P}}{P \setminus A \xrightarrow{\tau} \bar{P} \setminus A}\ [a \in A] \qquad \frac{P \xrightarrow{\mu} \bar{P}}{P \setminus A \xrightarrow{\mu} \bar{P} \setminus A}\ [\mu \notin A)]$$

In our approach we model this operator as follows (the parameter *hide* determines whether a label is hidden or not):

```
Hide+ : {i : Size} → {c : Choice}
  → (hide : Label → Bool) → Process+ i c
  → Process+ i c
E   (Hide+ f P)   = subset' (E P) (¬ b ∘ f ∘ (Lab P))
Lab (Hide+ f P)   c = Lab  P (projSubset c)
PE  (Hide+ f P)   c = Hide∞ f (PE P (projSubset c))
```

```
I    (Hide+ f P)  = I P ⊎' subset' (E P) (f ∘ Lab P)
PI   (Hide+ f P)  (inj₁ c) = Hide∞ f (PI P c)
PI   (Hide+ f P)  (inj₂ c) =
   Hide∞ f (PE P (projSubset c))
T    (Hide+ f P)  = T   P
PT   (Hide+ f P)  = PT  P
Str+ (Hide+ f P)  = HideStr f (Str+ P)
```

Here ¬ b is Boolean negation. In our approach the external choice E P is the subset of the external choices for which is Lab P is not hidden, and the internal choice I P is the union of the internal choice and the subset of the external choice for which Lab P is hidden. Informally:

$$
\begin{array}{lcll}
PE' & = & \{x \in PE & | \quad \text{Label}(x) \text{ is not hidden } \} \\
PI' & = & \{x \in PE & | \quad \text{Label}(x) \text{ is hidden } \} + PI
\end{array}
$$

5. A Simulator of CSP-Agda

We have written a simulator in Agda. It turned out to be more complicated than expected, since we needed to convert processes, which are infinite entities, into strings, which are finitary. The solution was to add string components to Process and Process∞. The need to add it to Process∞ was unexpected, since Process+ already seemed to have this information – however, one can only access it only if one has a smaller size available. We needed as well to add a conversion of choice sets to labels and restrict result sets to choice sets to make them printable.

The simulator does the following: It will display to the user the selected process, the set of termination choices with their return value (we don't allow the user to follow them, because it will always deadlock), and allows the user to choose an external or internal choice as a string input. If the input is correct, then the program continues with the process which is obtained by following that transition, otherwise an error message is returned and the program asks again for a choice. The simulator is implemented using a cut down version of the IO library of ooAgda [6], which makes use of the HS-monad. The IO library defines a version IOConsole of the IO monad with console commands (putStrLn s) for writing a string to console with a return type Unit, and getLine for getting user input with return type String.

The simulator displays the process as a string. Then it computes and displays the set of ✓-events and their results, and of external and internal choices together with their labels. We use the function choice2Enum to compute the list of choices and choice2Str to create a string representing a choice. Then the simulator asks for a user input, a string. The input is then compared with the choices available, yielding a Maybe applied to the list of external and internal choices. If the input was correct, the program continues with the next process, otherwise the user is asked to enter another choice. ✓-events are only displayed but one cannot follow them, because afterwards the system would stop.

```
mutual
  simulator : ∀ {i} → {c₀ : Choice}
     → Process ∞ c₀ → IOConsole i Unit
  forceIO (simulator P) =
     do' (putStrLn (Str P))                     λ _ →
     do  (putStrLn ("Termination-Events:"
        ++s showTicks P))                       λ _ →
     do  (putStrLn
        ("Events" ++s showLabels₁ P))           λ _ →
     do  (putStrLn ("Choose Event"))            λ _ →
     do  getLine                                λ s →
     simulator₁ P (lookupChoice
                  (processToE P)
                  (processToI P) s)

  simulator₁ : ∀ {i} → {c₀ : Choice}
     → (P : Process ∞ c₀)
     → Maybe ((ChoiceSet (processToE P))
        ⊎ (ChoiceSet (processToI P)))
     → IOConsole i Unit
  forceIO (simulator₁ P nothing) =
     do' (putStrLn
        "please enter a choice amongst") λ _ →
     do (putStrLn (showLabels₁ P))        λ _ →
     simulator P
  simulator₁ P (just c₁) =
     simulator (processToSubprocess P c₁)

main  : NativeIO Unit
main = translateIOConsole (simulator myProcess)
```

An example run of the simulator is as follows:

```
((b →  (a →  STOP)) □  (((c →  STOP) ⊓ (a →  STOP)) □  SKIP(STOP)))
Termination-Events: (inr (inr 0)):(inr (inr STOP))
Events: e-(inl 0):b i-(inr (inl 0)):τ i-(inr (inl 1)):τ
Choose Event
i-(inr (inl 0))
((b →  (a →  STOP)) □  ((c →  STOP) □  SKIP(STOP)))
Termination-Events: (inr (inr 0)):(inr (inr STOP))
Events: e-(inl 0):b e-(inr (inl 0)):c
Choose Event
e-(inl 0)
(fmap inl (a →  STOP))
Termination-Events:
Events: e-0:a
Choose Event
```

Internal choice and termination events are labelled by `"i-"` and `"t-"`, respectively. Since it is difficult to type in on the terminal inj₁, inj₂, we use the traditional names inl and inr instead. We have in the first step one ✓-event (inr (inr 0)), one external choice (inl 0), and two internal choices (inr (inl 0) and (inr (inl 1). In this run the user chose one internal choice and then one external choice. We used a more complex version of fmap, which displays a more readable string.

6. Related Work

There have been several successful approaches of combining functional programming with the CSP process Algebra. Brown [17] introduced a library (Communicating Haskell Process library, CHP) in Haskell. Since Haskell lacks explicit support for concurrency, he used a Haskell monad to provide a way to explicitly specify and control sequence and effects. Brown et al. [18] present a new technique in order to generate CSP models of Haskell implementations using the CHP library. This approach is characterised by the need for a detailed semantics of the Haskell language. In order to check the model generated by this approach against deadlock (using the FDR and ProB tools) they used it as well to perform refinement checks. López et al. [38] 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. Eden extends Haskell, and is considered as a concurrent functional language. Similarly, Fontaine [29] gave another successful attempt of implementing the operational semantics of CSP [45] 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. Sellink [47] gave a successful attempt to represent process algebras in type theory. Sellink used μCRL, a language for reasoning about the Algebra of Communicating Processes, in order to implement it in the type theoretic proof assistant Coq [21, 44].

Cleaveland et al. [19] gave an earlier attempt to implement process algebras in type theory. They implemented the Calculus for Communicating Systems [40] in the proof assistant Nuprl [1], which is similar to Agda, but based on extensional Martin-Löf type theory. Elliott [27] proposed an approach of representing concurrent programs in the dependently typed programming language Idris, and compiling them into the functional programming language Erlang using the Actor Model of Erlang. This allows to produce verified concurrent programs. Conceptually Erlang is similar to the Occam programming language. CSP was highly influential in the design of Occam.

Goncharov et al. in [30] defined a framework for concurrent processes where atomic steps have side effects. Goncharov et al. used the monadic principle in order to encapsulate the effects. Processes in that approach are modelled as infinite resumptions using a final coalgebra. The main result of this paper is a corecursion scheme over the base language and a new semantics for operators on processes such as parallel composition. They extended the framework to cover safety properties. Mossakowski et al. [42] gave a good example of using coalgebras in order to extend the specification language CASL. Goncharov et al. [31] also developed a semantic framework that combines monads, operations and recursive definitions. Their metalanguage for effectful recursion definitions was inspired by Moggi's computational metalanguage. They integrated the coalgebraic and monad aspects of the computations into a single framework. Using the notion of a complete Elgot monad, the authors developed a metalanguage. The work closest to our research is Goncharov et al. [31], who have also formalised corecursive definitions of process algebraic operations on processes with side effects using a new metalanguage.

7. Conclusion

The aims of this research is to give the type theoretic interactive theorem prover Agda the ability to model and in a next step verify concurrent programs by representing the process algebra CSP in monadic form. The set of processes forms a monad (Process A), which depends on a set A. Using this we can define a dependent composition (monadic bind) and a dependent loop construct rec for processes.

In our approach we define processes coinductively. The termination checker of Agda guarantees productivity of processes. This allows to define processes recursively without having to reduce them to the recursion combinator. The processes in our approach are formed from an atomic one step iteration. The operators of CSP are defined operations, which combine processes defined from atomic operations.

Future work. We have already defined trace semantics and bisimilarity for a previous version of CSP-Agda and verified selected laws of CSP with respect to those semantics. In that version Process+ was defined by using data rather than record, which caused problems since lots of auxiliary functions had to be defined. First experiments with adapting it to our new approach show that these definitions become much cleaner, and we are working on converting all of them and include the other CSP-semantics. A particular focus will be on showing that the laws regarding the interplay between different operators hold in our setting. One should note that the laws only hold subject to renaming of the results by a bijection. For instance in case of commutativity of _|||_ one needs to switch from result choice set $(c_0 \times' c_1)$ to $(c_1 \times' c_0)$.

As a case study, we are planning to implement CSP-processes, which the first author has developed for modelling elements of the European Rail Traffic Management System ERTMS [28], in CSP-Agda and verify their properties. Here we can build on Kanso's PhD thesis [37] in which he verified real world railway interlocking systems in Agda. Verifying larger examples might require to upgrade the integration of SAT solvers into Agda2, which has been developed by Kanso [37], to the current version of Agda. We plan as well to integrate the CSP model checker FDR2 into Agda. One future project is to write prototypes of programs, e.g. of some elements of the ERTMS, in Agda and make them directly executable in Agda. This uses the unique feature of Agda of being both a theorem prover and a dependently typed programming language, and that in Agda there is no distinction between proofs and programs, between data types and propositions.

Acknowledgments

This research was supported by the CORCON FP7 Marie Curie International Research Project, PIRSES-GA-2013-612638; COMPUTAL FP7 Marie Curie International Research Project, PIRSES-GA-2011-294962; and by CA COST Action CA15123 European research network on types for programming and verification (EUTYPES).

References

[1] E. Aaron. A user-level introduction to the Nuprl proof development system. Technical Report (CIS) 822, University of Pennsylvania, Department of Computer Science, 2001. URL `http://repository.upenn.edu/cis_reports/822`.

[2] A. Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types.* PhD thesis, Ludwig-Maximilians-Universität München, 2006. URL `http://www2.tcs.ifi.lmu.de/~abel/publications.html`.

[3] A. Abel. Compositional coinduction with sized types. In I. Hasuo, editor, *Coalgebraic Methods in Computer Science*, pages 5–10. Springer, 2016. ISBN 978-3-319-40370-0. doi: 10.1007/978-3-319-40370-0_2.

[4] A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In G. Morrisett and T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13*, pages 185–196. ACM, 2013.

[5] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In R. Giacobazzi and R. Cousot, editors, *Proceedings of POPL'13*, pages 27–38. ACM, 2013. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429075. URL `http://doi.acm.org/10.1145/2429069.2429075`.

[6] A. Abel, S. Adelsberger, and A. Setzer. Interactive Programming in Agda – Objects and Graphical User Interfaces. To appear in Jour. Functional Programming, 2016. URL `http://www.cs.swan.ac.uk/~csetzer/articles/ooAgda.pdf`.

[7] Agda Community. The Agda Wiki. 2015. URL `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

[8] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley, 2001.

[9] J. Baeten, D. A. van Beek, and J. Rooda. Process algebra. *Handbook of Dynamic System Modeling*, pages 19–1, 2007. URL `http://mate.tue.nl/mate/pdfs/8509.pdf`.

[10] J. A. Bergstra and J. W. Klop. Fixed point semantics in process algebras. CWI technical report, Stichting Mathematisch Centrum. Informatica-IW 206/82, 1982. URL `http://oai.cwi.nl/oai/asset/6750/6750A.pdf`.

[11] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and control*, 60(1):109–137, 1984.

[12] A. Bove and P. Dybjer. Language engineering and rigorous software development. In A. Bove, L. S. Barbosa, A. Pardo, and J. S. Pinto, editors, *Language Engineering and Rigorous Software Devel-*

opment, pages 57–99. Springer, 2009. ISBN 978-3-642-03152-6. doi: 10.1007/978-3-642-03153-3_2.

[13] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda — a functional language with dependent types. In *Proceedings of TPHOLs '09*, pages 73–78. Springer, 2009. ISBN 978-3-642-03358-2. doi: 10.1007/978-3-642-03359-9_6.

[14] E. Brady. Idris, a language with dependent types – Extended abstract. 2008. URL `http://www.cs.st-and.ac.uk/~eb/drafts/ifl08.pdf`.

[15] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013. ISSN 1469-7653. doi: 10.1017/S095679681300018X.

[16] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, June 1984. ISSN 0004-5411. doi: 10.1145/828.833.

[17] N. C. C. Brown. Communicating Haskell processes: Composable explicit concurrency using monads. In *The thirty-first Communicating Process Architectures Conference, CPA 2008*, pages 67–83, 2008. doi: 10.3233/978-1-58603-907-3-67.

[18] N. C. C. Brown. Automatically generating CSP models for communicating Haskell processes. *ECEASST*, 23, 2009. URL `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/325`.

[19] R. Cleaveland and P. Panangaden. Type theory and concurrency. *International Journal of Parallel Programming*, 17(2):153–206, 1988.

[20] Coq Community. The Coq Proof Assistant. 2015. URL `https://coq.inria.fr/`.

[21] Coq Development Team. The Coq proof assistant. Reference manual. https://coq.inria.fr/distrib/current/refman/, 2015.

[22] T. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, volume 806, pages 62–78. LNCS, 1994. doi: 10.1007/3-540-58085-9_72.

[23] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical frameworks*, pages 280 – 306. Cambridge University Press, 1991.

[24] P. Dybjer. Universes and a general notion of simultaneous inductive-recursive definition in type theory. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Proceedings of the 1992 workshop on types for proofs and programs, Båstad*, June 1992. URL `http://www.lfcs.inf.ed.ac.uk/research/types-bra/proc/proc92.ps.gz`.

[25] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525 – 549, June 2000.

[26] P. Dybjer and A. Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 124:1 – 47, 2003.

[27] A. S. Elliott. A concurrency system for IDRIS & ERLANG. Bachelors Dissertation, University of St Andrews, 2015. URL `http://lenary.co.uk/publications/dissertation/`.

[28] ERTMS. The European Rail Traffic Mangement System. 2013. URL `http://www.ertms.net/`.

[29] M. Fontaine. *A Model Checker for CSP-M*. PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2011. URL `http://docserv.uni-duesseldorf.de/servlets/DerivateServlet/Derivate-21671/dissertation_marc_fontaine.pdf`.

[30] S. Goncharov and L. Schröder. A coinductive calculus for asynchronous side-effecting processes. *Inf. Comput.*, 231:204–232, Oct. 2013. ISSN 0890-5401. doi: 10.1016/j.ic.2013.08.012.

[31] S. Goncharov, L. Schröder, and C. Rauch. (Co-)algebraic foundations for effect handling and iteration. *CoRR*, abs/1405.0854, 2014.

[32] P. Hancock and A. Setzer. The IO monad in dependent type theory. In *Electronic proceedings of the workshop on dependent types in programming, Göteborg, 27-28 March 1999*, 1999. URL `http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/dtp99.html`.

[33] P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic*, LNCS, Vol. 1862, pages 317 – 331, 2000. doi: 10.1007/3-540-44622-2_21.

[34] P. Hancock and A. Setzer. Specifying interactions with dependent types. In *Workshop on subtyping and dependent types in programming, Portugal,7 July 2000*, 2000. URL `http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html`. Electronic proceedings.

[35] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978. ISSN 0001-0782. doi: 10.1145/359576.359585.

[36] B. Igried and A. Setzer. CSP-Agda. Agda library, 2016. URL `http://www.cs.swan.ac.uk/~csetzer/software/agda2/cspagda/`.

[37] K. Kanso. *Agda as a Platform for the Development of Verified Railway Interlocking Systems*. PhD thesis, Dept. of Computer Science, Swansea University, Swansea, UK, August 2012. Available from http://www.swan.ac.uk/csetzer/articlesFromOthers /index.html and http://cs.swan.ac.uk/~cskarim/files/.

[38] N. López, M. Núñez, and F. Rubio. Stochastic process algebras meet Eden. In *Proceedings of IFM '02*, pages 29–48. Springer, 2002. ISBN 3-540-43703-7. URL `http://dl.acm.org/citation.cfm?id=647983.743555`.

[39] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, Naples, 1984. ISBN 88-7088-105-9.

[40] R. Milner. *A Calculus of Communicating Systems*. Springer, 1982. ISBN 0387102353.

[41] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. ISSN 0890-5401. doi: 10.1016/0890-5401(91)90052-4.

[42] T. Mossakowski, L. Schröder, M. Roggenbach, and H. Reichel. Algebraic-coalgebraic specification in CoCasl. *J. Log. Algebr. Program.*, 67(1-2):146–197, 2006. doi: 10.1016/j.jlap.2005.09.006.

[43] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

[44] C. Paulin-Mohring. Introduction to the coq proof-assistant for practical software verification. In B. Meyer and M. Nordio, editors, *Tools for Practical Software Verification: LASER*, pages 45–95. Springer, 2012. ISBN 978-3-642-35746-6. doi: 10.1007/978-3-642-35746-6_3.

[45] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN 0136744095.

[46] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley, 1st edition, 1999. ISBN 0471623733.

[47] M. P. A. Sellink. Verifying process algebra proofs in type theory. In *Proceedings of the International Workshop on Semantics of Specification Languages (SoSL)*, pages 315–339. Springer, 1994. ISBN 3-540-19854-7. URL `http://dl.acm.org/citation.cfm?id=645878.672054`.

[48] A. Setzer. Object-oriented programming in dependent type theory. In *Conference Proceedings of TFP 2006*, 2006. Available from http://www.cs.nott.ac.uk/~nhn/TFP2006/TFP2006-Programme.html and http://www.cs.swan.ac.uk/~csetzer/index.html.

[49] A. Setzer, A. Abel, B. Pientka, and D. Thibodeau. Unnesting of copatterns. In G. Dowek, editor, *Rewriting and Typed Lambda Calculi*, volume 8560, pages 31–45. LNCS, 2014. ISBN 978-3-319-08917-1. doi: 10.1007/978-3-319-08918-8_3.

[50] P. H. Welch, N. Brown, J. Moores, K. Chalmers, and B. H. Sputh. Integrating and extending JCSP. In *CPA*, volume 65, pages 349–370, 2007.