Programming with Monadic CSP-Style Processes in Dependent Type Theory

Bashar Igried and Anton Setzer

Swansea University, Swansea,Wales, UK

bashar.igried@yahoo.com , *a.g.setzer@swansea.ac.uk*

TyDe 2016 Type-driven Development, Japan, 18 Sep 2016

Overview

1. Agda
2. Process Algebra CSP
3. CSP-Agda
4. Simulator
5. Conclusion

Agda

- Agda is a theorem prover and dependently typed programming language, which extends intensional Martin-Löf type theory.
- The current version of this language is Agda 2
- Agda has 3 components:
  - Termination checker
  - Coverage checker
  - Type checker
- The termination checker verifies that all programs terminate.
- The type checker which refuses incorrect proofs by detecting unmatched types.
- The coverage checker guarantees that the definition of a function covers all possible cases.

Levels of Types

- There are several levels of types in Agda e.g. Set, Set_1, Set_2, ..., where

$$\text{Set} \overset{\in}{\subset} \text{Set}_1 \overset{\in}{\subset} \text{Set}_2 \overset{\in}{\subset} \text{Set}_3 \overset{\in}{\subset} \cdots$$

- The lowest level for historic reasons called Set.
- Types in Agda are given as:
  - Dependent function types.
  - Inductive types.
  - Coinductive types.
  - Record types (which are in the newer approach used for defining coinductive types).
  - Generalisation of inductive-recursive definitions.

Inductive Data Types

- The 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
   zeroFin : {n : ℕ} → Fin (suc n)
   sucFin  : {n : ℕ} (i : Fin n) → Fin (suc n)
```

- Implicit arguments can be omitted by writing zero instead of zero $\{n\}$.
- Can be made explicit by writing $\{n\}$

Define Functions

Therefore we can define functions by case distinction on these constructors using pattern matching, e.g.

$$
\begin{array}{lcl}
\mathsf{to}\mathbb{N} : \forall \{n\} \to \mathsf{Fin}\ n \to \mathbb{N} & & \\
\mathsf{to}\mathbb{N}\ \ \mathsf{zeroFin} & = & 0 \\
\mathsf{to}\mathbb{N}\ \ (\mathsf{sucFin}\ n) & = & \mathsf{suc}\ (\mathsf{to}\mathbb{N}\ n)
\end{array}
$$

Coinductive Types

There are two approaches of defining coinductive types in Agda.

- The older approach is based on the notion of codata types.
- The newer one is based on coalgebras given by their observations or eliminators

We will follow the newer one, pioneered by Setzer, Abel, Pientka and Thibodeau.

Why Agda?

- Agda supports induction-recursion.
  Induction-Recursion allows to define universes.
- Agda supports definition of coalgebras by elimination rules and defining their elements by combined pattern and copattern matching.
- Using of copattern matching allows to define code which looks close to normal mathematical proofs.

Process Algebra CSP

- "Process algebras" were initiated in 1982 by Bergstra and Klop in order to provide a formal semantics to concurrent systems.
- Process algebra is the study of distributed or parallel systems by algebraic means.
- Three main process algebras theories were developed.
  - Calculus of Communicating Systems (CCS).
    Developed by Robin Milner in 1980.
  - Communicating Sequential Processes (CSP).
    Developed by Tony Hoare in 1978.
  - Algebra of Communicating Processes (ACP).
    Developed by Jan Bergstra and Jan Willem Klop, in 1982.
- Processes will be defined in Agda according to the operational behaviour of the corresponding CSP processes.

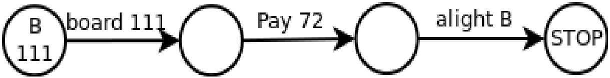
Example Of Processes

CSP Syntax

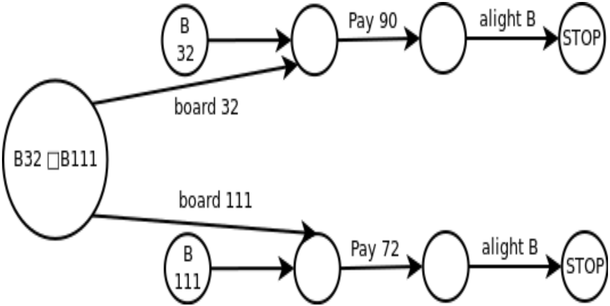
In the following table, we list the syntax of CSP processes:

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	$Q \ ; Q$

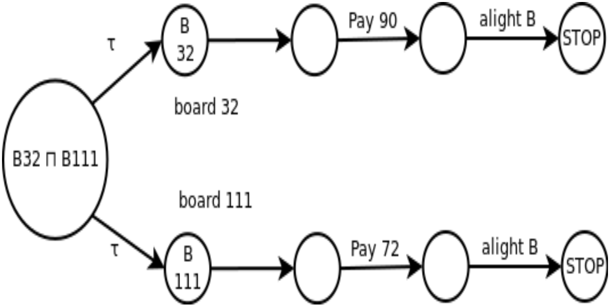
Example Of Processes



Example Of Processes



Example Of Processes



CSP Syntax

In the following table, we list the syntax of 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	$Q \; \|\|\| \; Q$
|	interrupt	$Q \triangle Q$
|	composition	$Q \; ; \; Q$

CSP-Agda

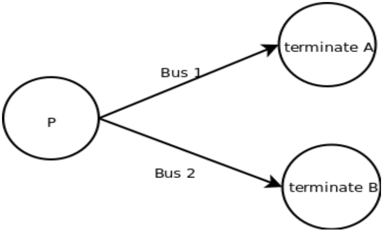
CSP-Agda

- CSP represented coinductively in dependent type theory.
- Processes in CSP can proceed at any time with:
  - Labelled transitions (external choices).
  - Silent transitions (internal choices).
  - $\checkmark$-events (termination).
- Therefore, processes in CSP-Agda have as well this possibility.

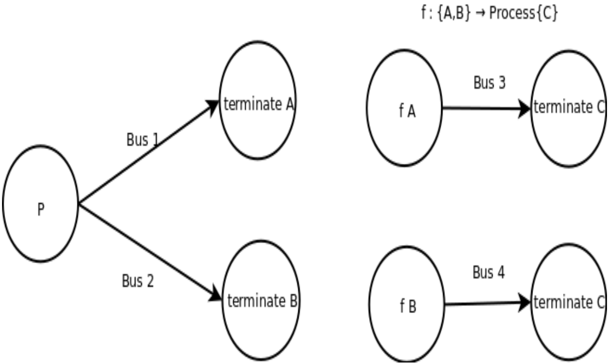
CSP-Agda

- In CSP a terminated process does not return any information except for that it terminated.
- We want to define processes in a monadic way in order to combine them in a modular way.
- If processes terminate additional information to be returned.

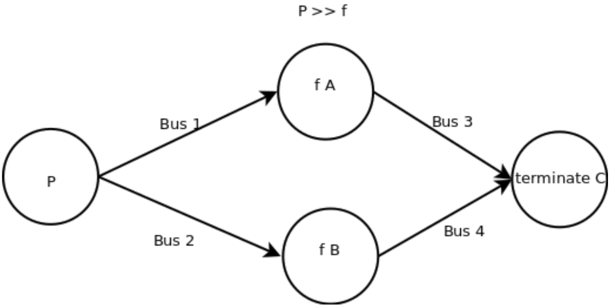
CSP-Agda



CSP-Agda


f : {A,B} → Process{C}
P
Bus 1
terminate A
Bus 2
terminate B
f A
Bus 3
terminate C
f B
Bus 4
terminate C

CSP-Agda



CSP-Agda

```
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
```

CSP-Agda

```
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
```

CSP-Agda

- Process∞ bundles processes as one coinductive type with one main one eliminator.
- So we have in case of a process progressing:
  (1) an index set E of external choices and for each external choice *e* the Label (Lab *e*) and the next process (PE *e*);
  (2) an index set of internal choices I and for each internal choice *i* the next process (PI *i*); and
  (3) an index set of termination choices T corresponding to ✓-events and for each termination choice *t* the return value PT $t : A$.
- In CSP termination is an event
  – for compatibility reasons we allow in CSP-Agda termination events as well.

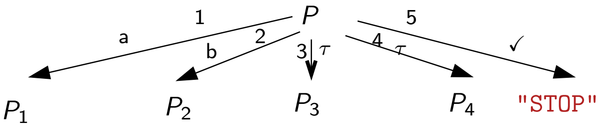
Example

$$P = \text{node}\ (\text{process+}\ E\ Lab\ PE\ I\ PI\ T\ PT\ \texttt{"P"})$$
$$: \text{Process String} \quad \text{where}$$

$E =$ code for $\{1, 2\}$ $\qquad I =$ code for $\{3, 4\}$

$T =$ code for $\{5\}$

$Lab\ 1$	$=$	a	$Lab\ 2$	$=$	b	$PE\ 1$	$=$	P_1
$PE\ 2$	$=$	P_2	$PI\ 3$	$=$	P_3	$PI\ 4$	$=$	P_4
$PT\ 5$	$=$	"STOP"						



Choices Set

- Choice sets are modelled by a universe.
- Universes go back to Martin-Löf in order to formulate the notion of a type consisting of types.
- Universes are defined in Agda by an inductive-recursive definition.

Choice Sets

We give here the code expressing that Choice is closed under fin, ⊎ and subset'.

```
mutual
data Choice : Set where
   fin     : ℕ → Choice
   _⊎'_    : Choice → Choice → Choice
   subset' : (E : Choice) → (ChoiceSet E → Bool)
      → Choice

ChoiceSet : Choice → Set
ChoiceSet (fin n)        =     Fin n
ChoiceSet (s ⊎' t)       =     ChoiceSet s ⊎ ChoiceSet t
ChoiceSet (subset' E f)  = subset (ChoiceSet E) f
```

Interleaving operator

- In this process, the components P and Q execute completely independently of each other.
- Each event is performed by exactly one process.
- The operational semantics rules are straightforward:

$$\frac{P \xrightarrow{\checkmark} P' \qquad Q \xrightarrow{\checkmark} Q'}{P \mid\mid\mid Q \xrightarrow{\checkmark} P' \mid\mid\mid Q'} \qquad \frac{P \xrightarrow{\mu} P'}{P \mid\mid\mid Q \xrightarrow{\mu} P' \mid\mid\mid Q} \; \mu \neq \checkmark$$

$$\frac{Q \xrightarrow{\mu} Q'}{P \mid\mid\mid Q \xrightarrow{\mu} P \mid\mid\mid Q'} \; \mu \neq \checkmark$$

Interleaving operator

We represent interleaving operator in CSP-Agda as follows:

```
_|||_ : {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
```

Interleaving operator

$$
\begin{array}{l}
\_|||{+\!\!+}\_ : \{i : \mathsf{Size}\} \to \{c_0\ c_1 : \mathsf{Choice}\} \\
\quad \to \mathsf{Process+}\ i\ c_0 \to \mathsf{Process+}\ i\ c_1 \\
\quad \to \mathsf{Process+}\ i\ (c_0 \times' c_1)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{E}\ (P\ |||{+\!\!+}\ Q) & = & \mathsf{E}\ P \uplus' \mathsf{E}\ Q \\
\mathsf{Lab}\ (P\ |||{+\!\!+}\ Q)\ (\mathsf{inj}_1\ c) & = & \mathsf{Lab}\ P\ c \\
\mathsf{Lab}\ (P\ |||{+\!\!+}\ Q)\ (\mathsf{inj}_2\ c) & = & \mathsf{Lab}\ Q\ c \\
\mathsf{PE}\ (P\ |||{+\!\!+}\ Q)\ (\mathsf{inj}_1\ c) & = & \mathsf{PE}\ P\ c\ |||{\infty}{+}\ Q \\
\mathsf{PE}\ (P\ |||{+\!\!+}\ Q)\ (\mathsf{inj}_2\ c) & = & P\ |||{+}{\infty}\ \mathsf{PE}\ Q\ c \\
\mathsf{I}\ (P\ |||{+\!\!+}\ Q) & = & \mathsf{I}\ P \uplus' \mathsf{I}\ Q \\
\mathsf{PI}\ (P\ |||{+\!\!+}\ Q)\ (\mathsf{inj}_1\ c) & = & \mathsf{PI}\ P\ c\ |||{\infty}{+}\ Q \\
\mathsf{PI}\ (P\ |||{+\!\!+}\ Q)\ (\mathsf{inj}_2\ c) & = & P\ |||{+}{\infty}\ \mathsf{PI}\ Q\ c \\
\mathsf{T}\ (P\ |||{+\!\!+}\ Q) & = & \mathsf{T}\ P \times' \mathsf{T}\ Q \\
\mathsf{PT}\ (P\ |||{+\!\!+}\ Q)\ (c\ ,,\ c_1) & = & \mathsf{PT}\ P\ c\ ,,\ \mathsf{PT}\ Q\ c_1 \\
\mathsf{Str+}\ (P\ |||{+\!\!+}\ Q) & = & \mathsf{Str+}\ P\ |||\mathsf{Str}\ \mathsf{Str+}\ Q
\end{array}
$$

Interleaving operator

- 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.

Interleaving operator

- A termination event can happen only if both processes have a termination event.
- If both processes terminate with results a and b, then the interleaving combination terminates with result $(a ,, b)$.
- 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.

A Simulator of CSP-Agda

Simulator is programmed in Agda using compiled version of Agda.

- Simulator requires String
- 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
- Choice set need to be displayed, so we use a universes of choices with a ToString function

A Simulator of CSP-Agda

The simulator does the following:

- It will display to the user
  - The selected process.
  - The set of termination choices with their return value.
  - 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.
- ✓-events are only displayed but one cannot follow them, because afterwards the system would stop.

A Simulator of CSP-Agda

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
```

Future Work

- We would like to model complex systems in Agda.
- Model examples of processes occurring in the European Train Management System (ERTMS) in Agda.
- Show correctness.

Conclusion

- A formalisation of CSP in Agda has been developed using coalgebra types and copattern matching.
- The other operations (external choice, internal choice, parallel operations, hiding, renaming, etc.) are defined in a similar way.
- Developed a simulator of CSP processes in Agda.
- Define approach using Sized types (Which allow us to apply function to CO-IH).

The End