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.
There are several levels of types in Agda e.g. Set, Set₁, Set₂, ..., where

\[ \text{Set} \subseteq \text{Set₁} \subseteq \text{Set₂} \subseteq \text{Set₃} \subseteq ... \]

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.
In this section, we discuss inductive data types. 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

\[
data \text{Fin} : \mathbb{N} \to \text{Set} \quad \text{where}
\]

\[
\text{zeroFin} : \{ n : \mathbb{N} \} \to \text{Fin} (\text{suc} \ n)
\]

\[
\text{sucFin} : \{ n : \mathbb{N} \} (i : \text{Fin} n) \to \text{Fin} (\text{suc} \ n)
\]

Implicit arguments can be omitted by writing \text{zero} instead of \{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.

\[
\begin{align*}
to\mathbb{N} &: \forall \{n\} \rightarrow Fin \ n \rightarrow \mathbb{N} \\
to\mathbb{N} \ zeroFin &= 0 \\
to\mathbb{N} \ (sucFin \ n) &= suc \ (to\mathbb{N} \ n)
\end{align*}
\]
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 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

B 32 → board 32 → Pay 90 → alight B → STOP
CSP Syntax

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Q ::= STOP</strong></td>
<td>STOP</td>
</tr>
<tr>
<td></td>
<td>SKIP</td>
</tr>
<tr>
<td></td>
<td>prefix</td>
</tr>
<tr>
<td></td>
<td>external choice</td>
</tr>
<tr>
<td></td>
<td>internal choice</td>
</tr>
<tr>
<td></td>
<td>hiding</td>
</tr>
<tr>
<td></td>
<td>renaming</td>
</tr>
<tr>
<td></td>
<td>parallel</td>
</tr>
<tr>
<td></td>
<td>interleaving</td>
</tr>
<tr>
<td></td>
<td>interrupt</td>
</tr>
<tr>
<td></td>
<td>composition</td>
</tr>
</tbody>
</table>
Example Of Processes
Example Of Processes

- Board 32
- Pay 90
- Alight B
- Stop

- Board 111
- Pay 72
- Alight B
- Stop
Example Of Processes

1. Board 32
2. Pay 90
3. Alight B
4. STOP

1. Board 111
2. Pay 72
3. Alight B
4. STOP

The diagram illustrates processes involving boarding and paying for transportation.
In the following table, we list the syntax of CSP processes:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
<td>STOP</td>
</tr>
<tr>
<td>SKIP</td>
<td>SKIP</td>
</tr>
<tr>
<td>prefix</td>
<td>$a \rightarrow Q$</td>
</tr>
<tr>
<td>external choice</td>
<td>$Q \square Q$</td>
</tr>
<tr>
<td>internal choice</td>
<td>$Q \sqcap Q$</td>
</tr>
<tr>
<td>hiding</td>
<td>$Q \setminus a$</td>
</tr>
<tr>
<td>renaming</td>
<td>$Q[R]$</td>
</tr>
<tr>
<td>parallel</td>
<td>$Q \parallel Y Q$</td>
</tr>
<tr>
<td>interleaving</td>
<td>$Q</td>
</tr>
<tr>
<td>interrupt</td>
<td>$Q \triangle Q$</td>
</tr>
<tr>
<td>composition</td>
<td>$Q ; Q$</td>
</tr>
</tbody>
</table>
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).
- ✓-events (termination).

Therefore, processes in CSP-Agda have as well this possibility.
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

Diagram:

- **P**
  - **Bus 1** to **terminate A**
  - **Bus 2** to **terminate B**
CSP-Agda

```
Bus 1
P

Bus 2

terminate A

terminate B

Bus 3

f A

Bus 4

f B

f : \{A,B\} \rightarrow \text{Process}\{C\}

terminate C

terminate C
```
CSP-Agda

P >> f

P

f A

Bus 1

f B

Bus 2

terminate C

Bus 3

Bus 4
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
CSP-Agda

- **Process**\(^\infty\) 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 \( \text{Lab} e \) and the next process \( \text{PE} e \);
  2. an index set of internal choices \( I \) and for each internal choice \( i \) the next process \( \text{PI} i \); and
  3. an index set of termination choices \( T \) corresponding to \( \checkmark \)-events and for each termination choice \( t \) the return value \( \text{PT} t : A \).

- In CSP termination is an event – for compatibility reasons we allow in CSP-Agda termination events as well.
Example

\[ P = \text{node (process+ } E \text{ Lab } PE \ I \ PI \ T \ PT \ "P" \text{)} \]

: Process String where

\[ E = \text{code for } \{1, 2\} \quad I = \text{code for } \{3, 4\} \]
\[ T = \text{code for } \{5\} \]
\[ \text{Lab 1 } = a \quad \text{Lab 2 } = b \quad \text{PE 1 } = P_1 \]
\[ \text{PE 2 } = P_2 \quad \text{PI 3 } = P_3 \quad \text{PI 4 } = P_4 \]
\[ PT 5 = "\text{STOP}" \]
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.
We give here the code expressing that Choice is closed under \( \text{fin} \), \( \uplus \) and \( \text{subset'} \).

\[
\begin{align*}
\text{mutual} \\
data \text{ Choice} & : \text{ Set where} \\
\quad \text{fin} & : \mathbb{N} \rightarrow \text{ Choice} \\
\quad \uplus' & : \text{ Choice} \rightarrow \text{ Choice} \rightarrow \text{ Choice} \\
\quad \text{subset'} & : (E : \text{ Choice}) \rightarrow (\text{ ChoiceSet } E \rightarrow \text{ Bool}) \rightarrow \text{ Choice} \\
\end{align*}
\]

\[
\begin{align*}
\text{ChoiceSet} & : \text{ Choice} \rightarrow \text{ Set} \\
\text{ChoiceSet} \ (\text{fin} \ n) & = \text{ Fin } n \\
\text{ChoiceSet} \ (s \uplus' \ t) & = \text{ ChoiceSet } s \uplus \text{ ChoiceSet } t \\
\text{ChoiceSet} \ (\text{subset'} \ E \ f) & = \text{ subset } (\text{ ChoiceSet } E) \ f
\end{align*}
\]
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:

\[ P \xrightarrow{\checkmark} P' \quad Q \xrightarrow{\checkmark} Q' \]

\[ P \parallel Q \xrightarrow{\checkmark} P' \parallel Q' \]

\[ P \xrightarrow{\mu} P' \quad Q \xrightarrow{\mu} Q' \quad \mu \neq \checkmark \]

\[ P \parallel Q \xrightarrow{\mu} P \parallel Q' \]

\[ Q \xrightarrow{\mu} Q' \quad \mu \neq \checkmark \]

\[ P \parallel Q \xrightarrow{\mu} P \parallel Q' \]
Interleaving operator

We represent interleaving operator in CSP-Agda as follows:

\[ \_ \parallel \parallel \_ : \{ i : \text{Size} \} \to \{ c_0, c_1 : \text{Choice} \} \to \text{Process} i \ c_0 \to \text{Process} i \ c_1 \to \text{Process} i \ (c_0 \times' c_1) \]

\[ \text{node } P \parallel \parallel \text{node } Q = \text{node } (P \parallel \parallel + + Q) \]

\[ \text{terminate } a \parallel \parallel Q = \text{fmap } (\lambda \ b \to (a ,, b)) \ Q \]

\[ P \parallel \parallel \text{terminate } b = \text{fmap } (\lambda \ a \to (a ,, b)) \ P \]
Interleaving operator

\[ _{|||+\_} : \{ i : \text{Size} \} \to \{ c_0, c_1 : \text{Choice} \} \]

\[ \to \text{Process} + i \ c_0 \to \text{Process} + i \ c_1 \]

\[ \to \text{Process} + i \ (c_0 \times \ c_1) \]

\[
\begin{align*}
E \ (P \ |||++ \ Q) &= E \ P \uplus' \ E \ Q \\
Lab \ (P \ |||++ \ Q) (\text{inj}_1 \ c) &= \text{Lab} \ P \ c \\
Lab \ (P \ |||++ \ Q) (\text{inj}_2 \ c) &= \text{Lab} \ Q \ c \\
PE \ (P \ |||++ \ Q) (\text{inj}_1 \ c) &= PE \ P \ c \ |||\infty+ \ Q \\
PE \ (P \ |||++ \ Q) (\text{inj}_2 \ c) &= P \ |||+\infty \ PE \ Q \ c \\
I \ (P \ |||++ \ Q) &= I \ P \uplus' \ I \ Q \\
PI \ (P \ |||++ \ Q) (\text{inj}_1 \ c) &= PI \ P \ c \ |||\infty+ \ Q \\
PI \ (P \ |||++ \ Q) (\text{inj}_2 \ c) &= P \ |||+\infty \ PI \ Q \ c \\
T \ (P \ |||++ \ Q) &= T \ P \times' \ T \ Q \\
PT \ (P \ |||++ \ Q) (c,, \ c_1) &= PT \ P \ c,, \ PT \ Q \ c_1 \\
\text{Str}+ \ (P \ |||++ \ Q) &= \text{Str}+ \ P \ |||\text{Str} \ \text{Str}+ \ Q
\end{align*}
\]
Interleaving operator

- When processes $P$ and $Q$ haven’t terminated, then $P ||| Q$ 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, $P || Q$ with $Q$.
  - However, we record the result obtained by $P$, and therefore apply $\text{fmap}$ 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
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.
An example run of the simulator is as follows:

```plaintext
((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.
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