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

JAIST, Japan, 6 Sep 2016
Overview

1. Agda
2. Why Agda?
3. Process Algebra
4. CSP
5. CSP-Agda
6. Choice Sets
7. Simulator
8. Future Work
9. 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 which has been designed and implemented by Ulf Norell in his PhD in 2007.
- Agda has a termination and coverage checker. This makes Agda a total language, so each Agda program terminates.
- The termination checker verifies that all programs terminate.
- Without the termination and coverage checker, Agda would be inconsistent.
- 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.
The user interface of Agda is Emacs.

This interface has been useful for interactively writing and verifying proofs.

Programs can be developed incrementally, since we can leave parts of the program unfinished.
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.
- coinductive types.
- record types (which are in the newer approach used for defining coinductive types).
- generalisation of inductive-recursive definitions.
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

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

- Here $\{n : ℕ\}$ 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 \( \text{Fin} : \mathbb{N} \rightarrow \text{Set} \) where \((\text{Fin } n)\) is a type with \( n \) elements.

The elements of \((\text{Fin } n)\) are those constructed from applying these constructors.
Therefore we can define functions by case distinction on these constructors using pattern matching, e.g.

\[
\text{toN} : \forall \{n\} \to \text{Fin } n \to \mathbb{N}
\]

\[
\text{toN } \text{zero} = 0
\]

\[
\text{toN } (\text{suc } n) = \text{suc } (\text{toN } n)
\]
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?
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.
Overview Of Process Algebras
“Process algebra” was initiated in 1982 by Bergstra and Klop [1], in order to provide a formal semantics to concurrent systems.

Baeten et. al. 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 considered as a formal specification language, developed in order to describe concurrent systems. By identifying their behaviour through their communications.

CSP is a notation for studying processes which interact with each other and their environment.

In CSP we can describe a process by the way it can communicate with its environment.

A system contains one or more processes, which interact with each other through their interfaces.
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>Q ::= STOP</td>
<td>STOP</td>
</tr>
<tr>
<td></td>
<td>SKIP</td>
</tr>
<tr>
<td></td>
<td>prefix $a \rightarrow Q$</td>
</tr>
<tr>
<td></td>
<td>external choice $Q ⊕ Q$</td>
</tr>
<tr>
<td></td>
<td>internal choice $Q \cap Q$</td>
</tr>
<tr>
<td></td>
<td>hiding $Q \setminus a$</td>
</tr>
<tr>
<td></td>
<td>renaming $Q[R]$</td>
</tr>
<tr>
<td></td>
<td>parallel $Q X∥ Y Q$</td>
</tr>
<tr>
<td></td>
<td>interleaving $Q</td>
</tr>
<tr>
<td></td>
<td>interrupt $Q \triangle Q$</td>
</tr>
<tr>
<td></td>
<td>composition $Q ; Q$</td>
</tr>
</tbody>
</table>
Example Of Processes
Example Of Processes

B32 □B111

- Board 32
- Board 111

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

1. B 111
2. Pay 72
3. Alight B
4. STOP
Example Of Processes

1. Start at B32 and B111
2. From B32, follow the path to B32
3. From B32, follow the path to Pay 90
4. From Pay 90, follow the path to alight B, then STOP
5. From B111, follow the path to Pay 72
6. From Pay 72, follow the path to alight B, then STOP

Note: 
- Board 32
- Board 111
CSP-Agda
CSP-Agda

- We will represent the process algebra CSP in a coinductive form in dependent type theory.
- Implement it in Agda.
- can proceed at any time with labelled transitions (external choices), silent transitions (internal choices), or ✓-events (termination).
- Therefore, processes in CSP-Agda have as well this possibility.
In process algebras, if a process terminates, it 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.

Therefore, if processes terminate, they should return some additional information, namely the result returned by the process.
In Agda the corresponding code is as follows:

```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
```
In Agda the corresponding code is as follows:

```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
```
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 $\text{PT } t : A$.  

As an example the following Agda code describes the process pictured below:

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

\[
P : \text{Process String} \quad \text{where}
\]

\[
E = \text{code for } \{1, 2\} \quad I = \text{code for } \{3, 4\}
\]

\[
T = \text{code for } \{5\}
\]

\[
Lab 1 = a \quad Lab 2 = b \quad PE 1 = P_1
\]

\[
PE 2 = P_2 \quad PI 3 = P_3 \quad PI 4 = P_4
\]

\[
PT 5 = "\text{STOP}"\]
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.
We give here the code expressing that Choice is closed under \( \text{fin} \), \( \uplus \) and \( \text{subset}' \).

\[
\text{mutual}
\]

\[
data \text{Choice} : \text{Set} \text{ where}
\]

\[
\begin{align*}
\text{fin} & : \mathbb{N} \to \text{Choice} \\
\uplus' & : \text{Choice} \to \text{Choice} \to \text{Choice} \\
\text{subset'} & : (E : \text{Choice}) \to (\text{ChoiceSet} \ E \to \text{Bool}) \\
& \to \text{Choice}
\end{align*}
\]

\[
\text{ChoiceSet} : \text{Choice} \to \text{Set}
\]

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

\[
\begin{align*}
P \xrightarrow{\checkmark} \bar{P} & \quad Q \xrightarrow{\checkmark} \bar{Q} \\
\hline
P \parallel Q \xrightarrow{\checkmark} \bar{P} \parallel \bar{Q} \\
\end{align*}
\]

\[
\begin{align*}
P \xrightarrow{\mu} \bar{P} & \quad \mu \neq \checkmark \\
P \parallel Q \xrightarrow{\mu} \bar{P} \parallel Q \\
Q \parallel P \xrightarrow{\mu} Q \parallel \bar{P} \\
\end{align*}
\]
We represent interleaving operator in CSP-Agda as follows

\[-\|\|+\| \colon \{i : \text{Size}\} \rightarrow \{c_0, c_1 : \text{Choice}\}\]

\rightarrow \text{Process}+ i c_0 \rightarrow \text{Process}+ i c_1

\rightarrow \text{Process}+ i (c_0 \times' c_1)

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

```plaintext
((b → (a → STOP)) □ ((((c → STOP) □ (a → STOP)) □ SKIP(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
```

Choose Event
Looking to the future, 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.
- A simulator of CSP processes in Agda has been developed.
Conclusion

- Define approach using Sized types.
- For complex examples (e.g. recursion) sized types are used to allow application of functions to the co-IH.
The End