Cashew-Nuts in Haskell

Implementation of a Timed Process Calculus

Simon Foster

University of Sheffield

November 25, 2005

Outline

What is Cashew-Nuts?

- A process calculus based on CCS.
- Extended with the notion of time, facilitated by *multiple abstract clocks*.
- Clocks facilitate *multi-party sychronization* : instantaneous, undirected and simultaneous synchronizations between an unbounded number of processes.
- Clocks can either be deterministic and maximal-progress respecting, or non-deterministic and violating maximal-progress.
- The latest in the timed process calculus family *TPL*, *CSA*, *PMC* and *CaSE*.

Time vs Mobility

- Ongoing discussion of the relative merits of time and mobility.
- Time is better suited to situations when an unspecified number of processes must synchronize simultaneously.
- π is better suited to systems which are not finite state, such as a complex client/server interface and data-types.
- The two primary uses of Cashew-Nuts/CaSE are based around finite state systems (i.e. Digital Signal Processing and workflow orchestration).
- For example the split-join pattern needs a method of resynchronizing all participating sub-workflows.
- π calculus very complicated to implement.
- Expect a mobile timed calculus at some point though.

Cashew-Nuts Syntax

Definition

$$\mathcal{E} ::= \mathbf{0} \mid \nabla \mid \Delta \mid \Delta_\sigma \mid \alpha.\mathcal{E} \mid [\![\mathcal{E}]\!]\sigma(\mathcal{E}) \mid [\![\mathcal{E}]\!]\sigma(\mathcal{E}) \mid \mathcal{E} + \mathcal{E} \mid \mathcal{E}|\mathcal{E} \mid \mathcal{E}[a \mapsto b] \mid \mathcal{E} \setminus a \mid \mathcal{E}/\sigma \mid \mathcal{E}/\!\!/\sigma \mid \mu X.\mathcal{E} \mid X$$

- α and β are action labels, possibly τ (the silent action).
- a and b are action labels, excluding τ.
- σ and ρ are clock labels within the clock universe.

Outline

Synchronization

- As in CCS, if two complementary actions can fire at the same time on different sides of a parallel composition, a synchronization can happen, resulting in a silent action being emitted.

$$a.P \mid \overline{a}.Q \xrightarrow{\tau} P \mid Q$$

Time-outs

$$\lfloor P \rfloor \sigma(Q) \xrightarrow{\sigma^1} Q$$

- Key to the use of time in processes is the timeout operator, similar to that present in *TPL* but with addition of a parameter specifying which clock the timeout relates to.
- Timeout allows a process to choose between 2 different routes depending on whether a specific clock can tick or not.
- All operators except Δ, Δ_σ and τ (except in the case of non-determinism and in contrast to PMC) are patient, that is to say they will allow a clock to tick.
- Thus the process $a.P$ has an implicit self-transition for every clock in the clock-sort.

Time-outs (cont.)

$$\underline{a}_{\sigma}.P \overset{\overset{\sigma}{\nrightarrow}}{\overset{a}{\longrightarrow}} P$$

- Actions can be be made *insistent* (i.e. not allowing ticks) by summing them with a Δ or Δ_σ, i.e. $a.P + \Delta$ or $a.P + \Delta_\sigma$, which we write as $\underline{a}.P$ and $\underline{a}_{\sigma}.P$ respectively.
- Cashew-Nuts has two timeout operators
  - Fragile, which may be interrupted by another clock ticking on the LHS.
  - Stable, which cannot be removed by another clock ticking (instead causing the LHS to advance).

$$\lfloor P \rfloor \sigma(Q) \xrightarrow{\rho^1} P'$$

$$\lceil P \rceil \sigma(Q) \xrightarrow{\rho^1} \lceil P' \rceil \sigma(Q)$$

Maximal Progress

- States that silent actions always take precedence over clock ticks.
- Allows detection of internal activity (or the lack thereof) within a process.
- Inherited from Hennessy's *TPL*.
- In Cashew-Nuts all clock actions (ticks) are annotated with either a 1 or 0 index, indicating whether they are respecting determinism and maximal progress or not. Sometimes known as **red** and **black** ticks.

$$a.P \mid \overline{a}.Q \overset{\sigma^1}{\not\mapsto}$$

$$a.P \mid \overline{a}.Q \overset{\sigma^0}{\longrightarrow} a.P \mid \overline{a}.Q$$

Clock Hiding

- As actions in CCS can be *restricted*, clocks in Cashew-Nuts (and indeed CaSE) can be *hidden*.
- Unlike restriction, clock hiding does not remove possible actions, but instead converts ticks to silent actions (which is somewhat similar to hiding in CSP).
- This forms the idea of hiearchy, since if a hidden clock can tick at the top of a hierarchy of processes, the resulting τs will result in other clocks further down the hiearchy being blocked by maximal progress.

$$\lfloor P \rfloor \sigma(Q)/\sigma \xrightarrow{\tau} Q$$

Non-deterministic time

- Hennessy's TPL stated that if a clock tick σ from a process P resulted in two processes Q and R, $Q \equiv R$
- CSA and CaSE also had this restriction, but Cashew-Nuts removes it.
- Instead ticks are characterised by a 1 or 0 subscript, which define whether a tick is deterministic and maximal-progress respecting or not (respectively).
- Thus where CaSE included a single clock hiding operator, Cashew-Nuts has 2
  - Single hiding, P/σ which converts only deterministic (i.e. subscript 1) ticks.
  - Double hiding, $P/\!/\sigma$, which converts both types of ticks (thus resulting in 2 τs).

Non-deterministic time (cont.)

- Similarly, whereas in CaSE there existed just one of each type of timeout operator, in Cashew-Nuts there are 2
  - Non-deterministic time-out, i.e. $\llbracket P \rrbracket \sigma(Q)$ and $\lceil\!\lceil P \rceil\!\rceil \sigma(Q)$, in which a σ can occur on either side, a tick on the LHS being indicated by a σ_0.
  - Deterministic time-out, which is derived from the above, simply blocking σ ticks on the LHS.

Definition

$$\lfloor P \rfloor \sigma(Q) \stackrel{\text{def}}{=} \llbracket P + \Delta_\sigma \rrbracket \sigma(Q)$$

$$\lceil P \rceil \sigma(Q) \stackrel{\text{def}}{=} \lceil\!\lceil P + \Delta_\sigma \rceil\!\rceil \sigma(Q)$$

Non-deterministic time (cont.)

- Non-deterministic ticks cause deterministic ticks to be demoted, so that if $P \xrightarrow{\sigma^1} P'$ and $Q \xrightarrow{\sigma^0} Q'$ then $Q \mid P \xrightarrow{\sigma^0} Q' \mid P'$
- Thus Cashew-Nuts also includes a "weak" version of Δ, which only prevents deterministic ticks: the ∇ process. This can only perform σ^0 ticks and thus all surrounding ticks are demoted in its presence.

Implementation in Haskell

- Haskell is a purely functional programming language.
- It has been used to implement the Cashew-Nuts process calculus since, as shall be demonstrated, it naturally represents the syntax.
- Large derived syntax can be represented elegantly via functions.
- Original version developed by Barry Norton for CaSE, and then enhanced by Andrew Hughes and Simon Foster for the CASheW-s (now Cashew) Darwin Project. My work has been to add bisimulation semantics and make it realistic for implementation of the Cashew workflow semantics.

Outline

Syntax in Haskell

- We use parametric polymorphism to introduce the action, clock and process label sets, as well as for providing process bindings.

Definition

data `Process a c p h v m` = ···

- a, c, and p are types representing action, clock and process labels respectively.
- h is the type of *Action Headers* (defined later).
- v is the type of (message) value that will be passed between processes.
- m is the type of Monad the process's bindings will be executed in.

On Action Headers

- Action Headers provide a generic method of attaching meta-data to actions.
- The primary intention for implementing them is to allow type data to be associated with input and output actions such that type-checking can be performed just by looking at the resulting silent action.
- We'll look into this in more detail in the typing section.

Syntax constructors

- We first define the simplest parts of the syntax:

Definition

data `Process a c p h v m` $=$

$NIL \mid TLD \mid TL\ c$	$\mid -0, \Delta, \Delta_\sigma$
$SUM\ (Process\ a\ c\ p\ h\ v\ m)\ (Process\ a\ c\ p\ h\ v\ m)$	$\mid -P + Q$
$PAR\ (Process\ a\ c\ p\ h\ v\ m)\ (Process\ a\ c\ p\ h\ v\ m)$	$\mid -P \mid Q$
$RES\ (Process\ a\ c\ p\ h\ v\ m)\ a$	$\mid -P \setminus a$
$HID\ (Process\ a\ c\ p\ h\ v\ m)\ c$	$\mid -P/\sigma$
$REN\ (Process\ a\ c\ p\ h\ v\ m)\ (a, a)$	$\mid -P[a \rightarrow b]$

Recursion

- Recursion is facilitated by point-free μ expressions (e.g. $\mu X.a.\overline{b}.X$).
- The expression with the variable in it is expressed as a function which inserts the recurring process in place (e.g. $\lambda X.a.\overline{b}.X$).
- The variable constructor allows serialization and deserialization of such constructions, by allowing replacement of recursive parts.

Definition

$$\begin{array}{ll} MU\ p\ (Process\ a\ c\ p\ h\ v\ m \rightarrow Process\ a\ c\ p\ h\ v\ m) & |- \mu X \\ VAR\ p & - X \end{array}$$

Timeouts

Definition

$$\lfloor P \rfloor \sigma(Q)$$

$$\lceil P \rceil \sigma(Q)$$

Definition

$$STO\ (Process\ a\ c\ p\ h\ v\ m)\ c\ (Process\ a\ c\ p\ h\ v\ m)|$$
$$TO\ (Process\ a\ c\ p\ h\ v\ m)\ c\ (Process\ a\ c\ p\ h\ v\ m)\ |$$

Binding Actions to Process Behaviour

- Remaining syntax more complicated - we need to take note of monadic bindings.
- Ultimately, the execution of a process should result in an operation of type `Monad m => m ()`, i.e. a pure side-effect.
- Whenever a silent action is emitted it should be possible to construct a pure side-effect which can be bound to the existing state.
- There are three places in Cashew-Nuts where this binding can occur:
  - Action Synchronization;
  - Explicit Silent Action ($\tau.P$);
  - During Clock Hiding (which can either be deterministic or non-deterministic).

Binding to τ

$$(a.\tau.0 \mid \overline{a}.0) \setminus a$$

$$\downarrow \tau$$

$$(\tau.0 \mid 0) \setminus a$$

$$\downarrow \tau$$

$$(0 \mid 0) \setminus a$$

- Each τ transition should perform a monadic interaction with the environment.

Action Synchronization

- An action synchronization occurs when two complementary actions can occur at the same instant.
- In essence this can be thought of as some sort of message being passed from one party to the other.
- This is represented in the syntax like so:

Definition

$$\begin{aligned}&\textbf{data}\ \texttt{Process a c p h v m} = \\ &\quad \mathit{OBS}\ (\mathit{Action}\ a\ h\ v\ m)\ (\mathit{Process}\ a\ c\ p\ h\ v\ m) \mid \cdots \\ &\textbf{data}\ \texttt{Action a h v m} = \\ &\quad \mathit{Input}\ a\ h\ (v \rightarrow m\ ()) \mid \\ &\quad \mathit{Output}\ a\ h\ (m\ v)\end{aligned}$$

- The resulting action is produced by binding the input action to the output action.

Explicit Silent Action

- On the surface, bindings seem trivial; simply attaching an operation of type `m ()` to the syntax will suffice.
- However, silent actions can also be used to facilitate external choice (e.g. $\tau.P + \tau.Q$).
- Although represented in the abstract syntax as non-deterministic choice, the choice phase needs to be more explicit in the implementation:

Definition

data `Process a c p h v m` $=$
$\quad TAU\ (m\ Int)\ [Process\ a\ c\ p\ h\ v\ m] \mid \cdots$

- i.e. an explicit silent action is a Monad producing an index, and a list of processes to pick from.
- If no choice is required, the operation should just return 0 and the list contain exactly 1 element.

Clock Hiding

- Hidden clocks (i.e. P/σ or $P/\!\!/\sigma$) give rise to a silent action each time the clock ticks within the operator's scope.
- The binding is fairly trivial, it is simply a case of attaching an operation of type `m ()` to the hiding operator.
- Double hiding though is given *two* actions, as it allows a τ produced either by a σ^0 or σ^1 and it's wise to make a distinction.

Definition

```
data Process a c p h v m =
  HID  (Process a c p h v m) c (m ()) |
  UHID (Process a c p h v m) c (m ()) (m ()) | ···
```

Example Process

- Here is an example process, and how it might be encoded (without bindings or meta-data) using the Haskell syntax.

Definition

$$((a.0 + b.0) \mid \overline{a}.0) \setminus a$$

$$((OBS\ (Input\ "a"\ \ nullHd\ \ nullIn)\ \ NIL\ `SUM`$$
$$\quad OBS\ (Input\ "b"\ \ nullHd\ \ nullIn)\ \ NIL)$$
$$`PAR`\ \ OBS\ (Output\ "a"\ \ nullHd\ \ nullOut)\ \ NIL)\ `RES`\ \ "a"$$

- This process may have type $Process\ String\ String\ String\ ()\ ()\ IO$.

Outline

Step Function

- We give semantics to the Cashew-Nuts syntax by the `step` function.

Definition

$$\begin{aligned} step :: \; & (Ord\ a, Ord\ c, Monad\ m) \Rightarrow \\ & [c] \rightarrow (Process\ a\ c\ p\ h\ v\ m) \rightarrow [Transition\ a\ c\ p\ h\ v\ m] \end{aligned}$$

- Takes a clock sort and process and returns all the possible transitions which can result.
- The clock sort is needed because of patience; every patient process must have a self-transition for each clock in the universe which isn't explicitly blocked.
- A transition is a label and the set of resulting processes.

Transition Type

Definition

```
data Transition a c p h v m =
  Tran (TransitionLabel a c p h v m) [Process a c p h v m]
data TransitionLabel a c p h v m =
  Tick (c, Bool) | Observable (Action a h v m) |
                   Silent (SilentType h) (m Int)
data SilentType h = ClockTick | UClockTick |
                    External  | Synchronize h h
```

- Multiple possible processes exist because of external choice (a single *TAU* in the syntax mutates to several processes).
- Silent action information is maintained mostly for action headers (for comparison between 2 synchronizing actions).

Step Semantics

- We use list comprehensions to draw out transitions from the component processes of a construct and use these to build the semantics. (c.f. principle of compositionality)

$$
\begin{aligned}
&step\ c\ p@(OBS\ l\ p') = [Tran\ (Observable\ l)\ [p']] +\!\!+ \\
&\qquad\qquad [Tran\ (Tick(x, True))\ [p] \mid x \leftarrow c] \\
&step\ c\ (SUM\ p\ q) = \\
&\quad \textbf{let}\ p' = step\ c\ p;\ q' = step\ c\ q\ \textbf{in} \\
&\quad [p'' \mid p''@(Tran(Silent\_\ \_)\_) \leftarrow (p' +\!\!+ q')] \quad +\!\!+ \\
&\quad [p'' \mid p''@(Tran(Observable\_)\_) \leftarrow (p' +\!\!+ q')] \quad +\!\!+ \\
&\quad [Tran\ (Tick\ (cn1, ct1\ \&\&\ ct2))\ [SUM\ p''\ q''] \mid \\
&\qquad\quad Tran\ (Tick\ (cn1, ct1))\ [p''] \leftarrow p', \\
&\qquad\quad Tran\ (Tick\ (cn2, ct2))\ [q''] \leftarrow q', \\
&\qquad\quad cn1 == cn2]
\end{aligned}
$$

Outline

Compiling a Labelled Transition System

- There are 2 LTSs which can be produced from processes in general:
  - Strong transition systems, which shows both silent and non-silent transitions.
  - Weak transitions systems, which shows only non-silent transitions, taking extra steps made by silent actions into account (e.g. $a.\tau.P$ would have a transition directly from a to P)
- The former is used for strong bisimulation checking, and the latter for weak bisimulation checking.
- Other modelling-checking uses possible, but I don't know what they are yet(!)

The LTS data-type

Definition

$$
\begin{aligned}
&\textbf{data}\ \mathit{LTS}\ l\ t\ s = \\
&\quad \mathit{LTS}\ \{\ \mathit{mutations} &&:: t \longrightarrow [s] \\
&\qquad ,\ \mathit{transitionLabel} &&:: t \longrightarrow l \\
&\qquad ,\ \mathit{isSilentAction} &&:: l \longrightarrow \mathit{Bool} \\
&\qquad ,\ \mathit{prepareState} &&:: s \longrightarrow s \\
&\qquad ,\ \mathit{statesIdentical} &&:: s \longrightarrow s \longrightarrow \mathit{Bool} \\
&\qquad ,\ \mathit{step} &&:: s \longrightarrow [t] \\
&\qquad \}
\end{aligned}
$$

- A set of rules for how a particular triple defines an LTS with τ.
- For Cashew-Nuts the type of labels is *Process*

The LTS data-type (cont.)

- Any data-type which can have step semantics defined in this way can also have a pair of graphs created for it.
- This enables bisimulation checking and also graph sketching via the Functional Graph Library's (fgl) graphviz interface.
- LTS for Cashew-Nuts looks like this (with states as process):

$$
\begin{aligned}
nutsLTS\ cs = LTS\ \{\ & mutations &&= aprioriMutations \\
,\ & transitionLabel &&= getTransitionLabel \\
,\ & isSilentAction &&= isSilent \\
,\ & prepareState &&= expandMUs \\
,\ & statesIdentical &&= (==) \\
,\ & Data.LTS.step &&= step\ cs \\
\}
\end{aligned}
$$

LTS Facilities

Definition

$$
\begin{array}{lll}
traces & :: & LTS\ l\ t\ s \rightarrow s \rightarrow [[l]] \\
ltsToGraphs & :: & DynGraph\ gr \Rightarrow LTS\ l\ t\ s \rightarrow s \rightarrow \\
 & & (LtsGraph\ gr\ l\ s, LtsGraph\ gr\ l\ s) \\
bisim & :: & (Eq\ b, Graph\ gr) \Rightarrow\ gr\ a\ b \rightarrow gr\ a\ b \rightarrow \\
 & & (Node, Node) \rightarrow \\
 & & StateT\ (Set\ (Node, Node))\ Maybe\ () \\
sbisim & :: & Eq\ l \Rightarrow (LTS\ l\ t\ s, s) \rightarrow (LTS\ l\ t\ s, s) \rightarrow \\
 & & Maybe[(s, s)] \\
wbisim & :: & Eq\ l \Rightarrow (LTS\ l\ t\ s, s) \rightarrow (LTS\ l\ t\ s, s) \rightarrow \\
 & & Maybe[(s, s)] \\
allReachable & :: & Graph\ gr \Rightarrow gr\ a\ b \rightarrow Bool
\end{array}
$$

Executing Processes

Definition

$$
\begin{aligned}
&\mathit{run} :: \mathit{Monad}\ m \Rightarrow \\
&\quad \mathit{Maybe}\ \mathit{Int} \\
&\quad \rightarrow ([\mathit{Transition}\ a\ c\ h\ v\ m] \rightarrow m\ (\mathit{Transition}\ a\ c\ h\ v\ m)) \\
&\quad \rightarrow \mathit{Process}\ a\ c\ h\ v\ m \\
&\quad \rightarrow m\ ()
\end{aligned}
$$

- This takes a process and uses the step semantics to transform it into a monadic interaction of type `Monad m => m ()` for the given number of runs.
- As a process may be non-deterministic, we also need a picker function which decides which transition to run next.

Motivation

- When we build a simple wire process (for linking two agent's data-flow), it is possible that some sort of state interaction is taking place, though not explicit.

Definition

$$Wire = \mu X.a.\overline{b}.X$$

- In these circumstances the message that action a inputs should be available for the action $\overline{b}$ to output.
- This can be dealt with in two ways:
  - By changing the calculus to a value passing calculus.
  - By writing variables into a global state.
- We will now consider both and show that only the latter is suitable.

Value Passing

- Option 1 : Use the value-passing calculus (i.e. Pi-Calculus sans channel passing)

Definition

$$Wire = \mu X.a(v).\overline{b(v)}.X$$

- So we store the state in the process itself, when *Wire* synchronizes it also unifies v with the value.
- Immediate problem with this is that we immediately restrict variable scope to the process level; useless if we want consumer/producer processes parallel composed for data-flow.
- This can be solved by causing unification to occur at the restriction boundary for the associated channel.

Value Passing (cont.)

- This is pretty odd though, since value passing is supposed to behave like λ, but this doesn't.
- Implementing a run-time unifier is likely to make process execution very slow.
- However the biggest problem is explicit silent actions; as they don't say what they consume or produce we can't know what to bind.
- On top of that we can't actually unify anyway, since to know what value we are going to unify with we need to run the silent actions, but we can't run them until we get to the top-level process. i.e. Operational Semantics depends on bindings which is not a good thing.
- Thus we abandon this approach for the time-being.

Global State Variables

- Option 2 : Store messages in a global state for later retrieval
- Recall that processes are compiled to monadic interactions, if we add mutable state to the monad we are using we can allow bindings to store variables.
- Immediate advantage that it fits into the existing model and doesn't need a language extension.
- Plus this is arguably the correct way of viewing processes, as if they are simulating real-world agents (e.g. web-services) they will most likely share some sort of state.
- We will use this method because it doesn't require a change to the calculus.

Outline

Scoping Global State

- To avoid clashes between two processes sharing the same variable names, we need some form of scoping.
- This is based on restriction boundaries, i.e. on a binding level we are treating restriction like π's *new* operator.
- Thus each label in the action sort has 2 associated variables, one associated with inputs and one with outputs.
- Then each restriction boundary is identified by some unique value which is used to ensure that each variable name is unique in its scope.
- For example, we might identify each variable with a pair $(a, \mathit{Integer})$, where a is the action variable type and the integer identifies the restriction boundary.
- We will use rules for our bindings which will enable us to establish how the τ actions interact with other actions.

Variables Uses

- Input action variables (e.g. a_1) are written to by input actions only and read from by explicit silent actions and output actions.
- Output action variables (e.g. $\overline{b_3}$) are written to by explicit silent actions, and read by output actions.

$$a.b.\tau.\overline{c}.0$$

- This process would use
  - a_n and b_n (where n is set by the scope) to store values input by a and b, from which the τ action and $\overline{c}$ can read.
  - $\overline{c_n}$ which the τ action could write to and the $\overline{c}$ action could read from.
- Difficult to enforce these rules, but they must be observed if variable clash is to be avoided.

Example

- Consider the following process:

$$(\overline{a}.0 \mid a.\overline{b}.0 \mid (\overline{a}.0 \mid a.b.\tau.\overline{c}.0) \setminus \{a\}) \setminus \{a, b\}$$

- There are two distinct a action scopes, one b and one c, which makes a total of 8 variables (although not all are necessarily used). Their initial values are shown below

a_1	$\overline{a_1}$	a_2	$\overline{a_2}$	b_1	$\overline{b_1}$	c_1	$\overline{c_1}$
	3		6				

Example

- Consider the following process:

$$(\overline{a}.0 \mid a.\overline{b}.0 \mid (\overline{a}.0 \mid a.b.\tau.\overline{c}.0) \setminus \{a^2\}) \setminus \{a^1, b^1\}$$

- There are two distinct a action scopes, one b and one c, which makes a total of 8 variables (although not all are necessarily used). Their initial values are shown below

a_1	$\overline{a_1}$	a_2	$\overline{a_2}$	b_1	$\overline{b_1}$	c_1	$\overline{c_1}$
	3		6				

Example run-through

$$(\overline{a}.0 \mid a.\overline{b}.0 \mid (\overline{a}.0 \mid a.b.\tau.\overline{c}.0) \setminus \{a\}) \setminus \{a, b\}$$

$$\downarrow$$

$$(0 \mid \overline{b}.0 \mid (\overline{a}.0 \mid a.b.\tau.\overline{c}.0) \setminus \{a\}) \setminus \{a, b\}$$

a_1	$\overline{a_1}$	a_2	$\overline{a_2}$	b_1	$\overline{b_1}$	c_1	$\overline{c_1}$
3	3		6				

$$\downarrow$$

$$(0 \mid \overline{b}.0 \mid (0 \mid b.\tau.\overline{c}.0) \setminus \{a\}) \setminus \{a, b\}$$

a_1	$\overline{a_1}$	a_2	$\overline{a_2}$	b_1	$\overline{b_1}$	c_1	$\overline{c_1}$
3	3	6	6				

Example run-through (cont.)

$$(0 \mid \overline{b}.0 \mid (0 \mid b.\tau.\overline{c}.0) \setminus \{a\}) \setminus \{a, b\}$$

$$\downarrow$$

$$(0 \mid 0 \mid (0 \mid \tau.\overline{c}.0) \setminus \{a\}) \setminus \{a, b\}$$

a_1	$\overline{a_1}$	a_2	$\overline{a_2}$	b_1	$\overline{b_1}$	c_1	$\overline{c_1}$
3	3	6	6	3			

$$\downarrow$$

$$(0 \mid 0 \mid (0 \mid \overline{c}.0) \setminus \{a\}) \setminus \{a, b\}$$

a_1	$\overline{a_1}$	a_2	$\overline{a_2}$	b_1	$\overline{b_1}$	c_1	$\overline{c_1}$
3	3	6	6	3			9

- Simple example, but illustrates how variable clash is avoided.

Outline

The Scoping class

- This is nice, but how to implement it.
- Really what we want is scoping to be transparent. That is when we access variables from the environment by name, it should automatically adjust that name to take scope into account.
- Thus we've created the scoping class.

Definition

class `Scoping m a` **where**

$$\begin{aligned} &\mathit{scope} &&:: a \to \mathit{Integer} \to m\ () \\ &\mathit{unscope} &&:: a \to m\ () \end{aligned}$$

The Scoping class (cont.)

- The idea of the scoping class is that the step semantics need, at the point of a restriction boundary, "scope up" the variable label in the monad.
- Every action which passes through such a restriction must have its binding correctly scoped. The *Integer* identifies the scope boundary.
- Instances of this class for monads which cannot scope variables will simply perform no action.
- Where scoping does exist, at the point of restriction for an action a, and assuming the restriction boundary has identity n, the following change would be made to the binding:

$$scope\ a\ n \gg op \gg unscope\ a$$

A Simple Scoping Monad

- To illustrate how scoping would work, here is the simplest form of monad for scoping.

Definition

type `MonadScope k v` $=$
$State\ (Map\ ((k, Bool), Integer)\ v, [(k, Integer)])$

- A state monad which stores a map from variable key/restriction boundary key pairs to values, allowing different values for variables in different boundaries.
- It also stores a list of scope keys that have been stacked for each channel. When a variable is accessed it will use the top most name to get the scope key. May be important as synchronizing parties have their side-effects combined.

A snapshot of the monad for the above example

- Recall the state of the following process before the τ action was executed.

$$(0 \mid 0 \mid (0 \mid \tau.\overline{c}.0) \setminus \{a\}) \setminus \{a, b\}$$

a_1	$\overline{a_1}$	a_2	$\overline{a_2}$	b_1	$\overline{b_1}$	c_1	$\overline{c_1}$
3	3	6	6	3			

- Using the Scoping Monad, this would be represented like this:

$$(\{((\text{"}a\text{"}, \mathit{True}), 2) \Rightarrow 6, ((\text{"}b\text{"}, \mathit{True}), 1) \Rightarrow 3, \cdots\},$$
$$[(\text{"}a\text{"}, 2), (\text{"}b\text{"}, 1), (\text{"}a\text{"}, 1)])$$

Generating Restriction IDs

- Just one thing remains, we need to generate a unique ID for each restriction boundary in a process.
- We do this with the *genScopeKeys* function:

Definition

$$genScopeKeys :: Process\ a\ c\ p\ h\ v\ m \rightarrow Process\ a\ c\ p\ h\ v\ m$$

- This preprocessor function adds scope keys to each restriction boundary, so that when transitions pass through the boundary, it is possible to scope them up.

Outline

Typing action variables

- So, now we've got a model for how state in terms of variable binding should be viewed, we need to look at how these things are going to be typed.
- Type-checking is based on the notion that functions bound to τ actions (and sometimes inputs/outputs although we prefer to avoid these) provide information which can be used to established what type of information channels should be passing around.
- For example, if $\overline{a}$ has an action bound of type $m\ Int$ and a an action of type $String \rightarrow m\ ()$ the two can synchronize operationally, but the resulting silent action would not be executable.

Objectives

- Original idea was that type-checking would be done at runtime; synchronizations whose associated bindings did not have matching types could not occur.
- However, this doesn't solve the problem of two τs with differing views of what type a variable should be - the system can still crash.
- In any case, rejecting synchronizations can only be a bad thing, most likely leading to a stall unless we build in some recovery mechanism.
- Instead our aim is to do type-checking before run-time with the help of a complete LTS.
- Also has the advantage of full static typing which Haskell is best at.

Typing Limitations

- Recall that the *Process* data-type is parameterised over a single message type v, i.e. the type of value which is exchanged by synchronizations.

Definition

data `Process a c p h v m` $=$
$OBS\ (Action\ a\ h\ v\ m)\ (Process\ a\ c\ p\ h\ v\ m) \mid \cdots$

- This means that any exchanges within a single processes must work on the same data-type, by virtue of Haskell's type-system.

Typing Limitations (cont.)

- In order to actually allow bound operations in the syntax to be heterogeneous dependant types would be required.
- Haskell's support for dependant types is at best very poor, and thus the type of message values are (in this context) limited to `Dynamic`.
- `Dynamic` is a data-type which can store any value so long as its type can be established via the `Typeable` class (more on this in a bit).
- Turns out to be pretty elegant, and regular heterogenous functions can be converted to fully dynamic interactions without too much difficulty, so long as they're typeable.
- But we need to recast functions from several input to several outputs as interactions with state.

The *Typeable* class

- The *Typeable* class allows a type-representation to be assigned to types in Haskell.
- In this way it is possible to perform simple type-casting from one typeable value to another if their types are the same.

Definition

class `Typeable a` **where**
  $typeOf :: a \rightarrow TypeRep$
$cast :: (Typeable\ a, Typeable\ b) \Rightarrow a \rightarrow Maybe\ a$

The *Dynamic* data-type

- A data-type which can contain any value.
- Based on the typeable class, we can cast the abstract the data inside to a suitable value.

Definition

$$\begin{aligned} &\textbf{data}\ \texttt{Dynamic} \quad = \quad \cdots \\ &\mathit{toDyn} \qquad :: \mathit{Typeable}\ a \Rightarrow a \rightarrow \mathit{Dynamic} \\ &\mathit{fromDynamic} :: \mathit{Typeable}\ a \Rightarrow \mathit{Dynamic} \rightarrow \mathit{Maybe}\ a \end{aligned}$$

Outline

Step 1 : A suitable monad

- We first need to design a class of monads which can get and set variables into the state carried by the monad.
- The simplest form of variable monad is the state monad encapsulating a map from variable names to values, i.e. *State* (*Map k Dynamic*)
- However, as we want to be generic we will define our own class called *MonadVar*

Definition

class `Monad m` $\Rightarrow$ `MonadVar k m` | `m` $\rightarrow$ `k` **where**
$\quad$ *getVar* :: *Typeable v* $\Rightarrow$ *k* $\rightarrow$ *m v*
$\quad$ *setVar* :: *Typeable v* $\Rightarrow$ *k* $\rightarrow$ *v* $\rightarrow$ *m* ()

Step 2 : Converting Functions to interactions

- Each explicit silent action can glean the data it needs to run from state variables, and write its outputs to other state variables.
- Ultimately we want to be able to take heterogenous functions with a number of inputs and an output, mapping these to monadic interactions on the state alone.
- Such interactions also need to have two sets of variable names associated with them, one for inputs and one for outputs.
- Along with a state interaction, a list associating names with types is produced: this is key to static type-checking.

Converting Functions to interactions (cont.)

- The *tauM* function converts a function with a single input and output to an interaction and a list of variable label/type pairs (the boolean indicates whether it is an input or output variable).

Definition

$$\begin{aligned} tauM \;::\; & (Typeable\ a, Typeable\ b, MonadVar\ m\ l) \Rightarrow \\ & l \rightarrow l \rightarrow (a \rightarrow m\ b) \rightarrow \\ & (m\ (), [((l, Bool), TypeRep)]) \end{aligned}$$

- It takes a variable label (l) for the input, one for the output and the actual function.

Converting Functions to interactions (cont.)

- Similarly we could define the *tauM2* function:

Definition

$$\begin{aligned} tauM2 \;::\; & (Typeable\ a, Typeable\ b, Typeable\ c, MonadVar\ m\ l) \Rightarrow \\ & (l, l) \rightarrow l \rightarrow (a \rightarrow c \rightarrow m\ b) \rightarrow \\ & (m\ (), [((l, Bool), TypeRep)]) \end{aligned}$$

- We could create a class *Function f* which would deal with a large number of functions, but from experience with all the undecidability which is required, it's not worth the effort.

Example

- So if we had a simple function
  $add :: Monad\ m \Rightarrow Int \rightarrow Int \rightarrow m\ Int$, this would be converted to an interaction like so:

Definition

$$tauM2\ (\text{"}a\text{"}, \text{"}b\text{"})\ \text{"}c\text{"}\ add$$
$$\Longrightarrow$$
$$(\ \ll operation \gg,$$
$$[((\text{"}a\text{"}, True), Int), ((\text{"}b\text{"}, True), Int), ((\text{"}c\text{"}, False), Int)])$$

- With the operation getting the 2 inputs from variables a and b, and writing the result to variable c.

Step 3 : A Nasty Kludge

- So now we can convert a function to a binding this needs to be integrated into the *TAU* syntax via a function, e.g.

$$TAU :: (m\ Int) \rightarrow [Process \cdots] \rightarrow Process \cdots$$
$$tauF :: (m\ (), [((a, Bool), TypeRep)]) \rightarrow Process\ a\ c\ p\ h\ v\ m \rightarrow Process\ a\ c\ p\ h\ v\ m$$

- However, we've now got a bit of a problem. Although we've can insert the monadic interaction generated straight into the *TAU* syntax, we've got nowhere to put the variable allocations.
- Only option is to add an extra *Dynamic* element into *TAU*, so that we can store the list of types.

Type-Check

- There are 2 steps to actually performing type-checking:
  - Checking for consistency between τ actions.
  - LTS-based checking of synchronizations.
- The former can be achieved by decending through the process tree, making a note of all the variables each τ manipulates and the scopes in which they work.
- This would be achieved with the following function.

Definition

$$getScopedVariableTypes :: Process\ a\ c\ p\ h\ v\ m \rightarrow [(((a, Bool), Integer), TypeRep)]$$

LTS Type Check

- If the τ bindings consistently type the variables, the types can be extended to input and output actions.
- Finally we use the action headers; for typing the following type is used:

Definition

$$\textbf{data}\ \texttt{ActionType} \quad = \quad \texttt{TopType} \mid \texttt{HaskellType TypeRep}$$

- Just using *TypeRep* is insufficient; channels which do not convey data cannot be assigned a type. Instead they are assigned *TopType* which matches, and is matched with any type.

LTS Type Check (cont.)

- Recall that the Transition label Type allowed the comparison of the headers for the two halves of synchronizations.

Definition

$$\textbf{data}\ \texttt{SilentType h} = \mathit{ClockTick} \mid \mathit{UClockTick} \mid \mathit{External} \mid \mathit{Synchronize}\ h\ h$$

- We can now take advantage of this in the LTS we produce, it is simply a case of creating a graph and checking that each synchronization transition compatible types.
- Thus the type-checking is complete and the process can be safely executed.

ConCalc

- ConCalc is an experimental tool for forming and testing Cashew-Nuts processes.
- It tries to have a similar sort of interface to GHCi.
- Key features are bisimulation checking and graph sketching.
- Demo (if it happens to be working at this point ;)).

Future Work

- Now we've got type and bisimulation checking, the next step is to look at model checking.
- Also we'd like to create a process executor which uses Concurrent Haskell as the GHC team is nearing support for SMP on it. Currently execution support is purely mono-processor.

Conclusion

- We've presented a type-safe implementation of the process calculus Cashew-Nuts in Haskell.
- We believe that once the type-checker is completed it should be workflow-ready.
- ConCalc gives a simple tool which can be used for testing processes.