Act II

Smol-Zooid: multiparty with shallower embedding
Goals

1. Certifying individual processes of a distributed system
2. Extracting runnable code
3. Avoiding complex formalisations of binders, whenever possible
Overview

\[ \begin{align*}
G & \xrightarrow{\mathcal{R}} G^c & \text{global trace} \\
L & \xrightarrow{\mathcal{R}} L^c & \text{local trace} \\
\text{proc} & \xrightarrow{\mathcal{L}_T} \text{process trace}
\end{align*} \]

\[ \begin{align*}
\exists & \Rightarrow \text{LTS} \\
| & \Rightarrow \text{LTS} \\
\Downarrow & \Rightarrow \text{erase}
\end{align*} \]

\[ \begin{align*}
\text{extraction} & \rightarrow \text{OCaml code} \\
\text{DSL layer} & \rightarrow \text{Zoooid}
\end{align*} \]
Smol Zooid

- We combine **shallow/deep embeddings** of binders
  - We use DeBruijn indices for the deeply embedded binders
- SZooid constructs are **well-typed by construction**
- We leverage **Coq code extraction** mechanism
- For simplicity, SZooid does not cover **choices**
Core Processes

In: http://github.com/emtst/gentleAdventure

```
Inductive proc :=
| Inact  | Rec (e : proc) | Jump (X : nat)
| Send (p : participant) {T : type}
    (x : interp_type T) (k : proc)
| Recv (p : participant) {T : type}
    (k : interp_type T -> proc)
| ReadIO {T : type} (k : interp_type T -> proc)
| WriteIO {T : type} (x : interp_type T) (k : proc).
```
Payload Types

We need to define a type for payload types:
- We need a decidable equality on payload types
- We need a decidable equality on payload values

```
Inductive type := Nat | Bool | ...
Definition interp_type : type -> Type := ...
```
Semantics: events

The semantics is an LTS:
- the labels are the **communication events**
- it is parameterised by a **payload interpretation function**
- traces are obtained as the greatest fixpoint of the LTS step

\[
\text{Inductive } \text{action} := \text{a\_send} \mid \text{a\_recv}.
\]

\[
\text{Record } \text{event interp\_payload} :=
\begin{align*}
\{ & \text{action\_type} : \text{action;} \\
& \text{subj} : \text{participant;} \\
& \text{party} : \text{participant;} \\
& \text{payload\_type} : \text{type;} \\
& \text{payload} : \text{interp\_payload payload\_type}
\end{align*}
\]
p_unroll : proc -> proc

p_unroll exposes the first communication action in a process:
• “runs” any I/O action
• unfolds recursion

Definition p_unroll : proc -> proc := ...
Semantics: step

The step of the LTS is defined as a function:

```latex
Definition step' e E :=
  match e with
  | Send p T x k =>
    if (action_type E == a_send) && (party E == p) &&
      (eq_payload (payload E) x)
    then Some k else None
  | Recv p T k => ... | _ => None
end.

Definition step e := step' (p_unroll e).
```
Local Types

We introduce a typing discipline that associates processes with local types, that characterise their communication behaviour:

Inductive lty :=
  | l_end
  | l_jump (X : nat)
  | l_rec (k : lty)
  | l_send (p : participant) (T : type) (l : lty)
Type System

Inductive of_lty : proc -> lty -> Prop :=
| lt_Send p T k L x :
  of_lty k L -> of_lty (@Send p T x k) (l_send p T L)
| lt_ReadIO T k L :
  (forall x, of_lty (k x) L) -> of_lty (@ReadIO T k) L
| ...
.
It would be tedious to type up both a local type and a process. Users would need to provide a proof that processes are well-typed.

We define **SZooid** (Smol Zooid), to write well-typed processes by construction, avoiding repetition.
Definition SZooid L := \{ p \mid \text{of\_lty } p \text{ L}\}.

Definition z_Send p T x L (k : SZooid L) : SZooid (l_send p T L) := \text{exist } _ _ (lt_Send p x (proj2_sig k)).

...
Inferring Local Types

SZooid constructs fully determine their types from their inputs, so we can ask Coq to infer local types associated with SZooid terms:

```
Definition AZooid := { L & SZooid L }.
```
Subject Reduction

Theorem preservation (e : proc) (L : lty)
(H : of_lty e L) (E : rt_event) :
\[\forall e',\\]
\[\text{step } e \ E = \text{Some } e' \to\\]
\[\exists L', \ lstep L (ev_{\text{erase}} E) = \text{Some } L' \land\\]
\[\text{of_lty } e' \ L'.\]
Extraction

- We convert proc to function calls in an ambient monad
- We extract the monadic code to OCaml
- The ambient monad needs to be implemented in OCaml
- Processes are extracted using Higher-Order modules, so it is straightforward to change the underlying transport

**Remark:** SZooid does not provide an implementation of the ambient monad, but Zooid does, using TCP/IP sockets: https://github.com/emtst/zooid-cmpst

Fixpoint extract_proc (d : nat) (p : proc) : MP.t unit :=
    match p with
    | Send p T x k =>
        MP.bind (MP.send T p x)
        (fun=> extract_proc d k)
    ... End ProcExtraction.
Example Extraction

Module ALICE (MP : ProcessMonad) : PROCESS(MP).
Module PE := ProcExtraction(MP).
Definition proc :=
    Eval compute in PE.extract_proc 0 alice.
End ALICE.

Extraction ALICE.
Summary

- We have seen how to encode a small calculus of Multiparty Processes, with a basic type system.
- Next: how do we relate traces of individual processes to a larger system?