Type-safe Bidirectional Channels in Idris 2

Guillaume Allais[1]

University of Strathclyde, Glasgow, Scotland, United Kingdom
guillaume.allais@strath.ac.uk

Introduction

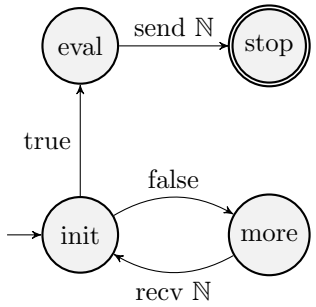
Session types are a channel typing discipline ensuring that the communication patterns of concurrent programs abide by a shared protocol. A meta-theoretical analysis of the typing discipline can then ensure that the communicating processes will have good properties e.g. deadlock-freedom. We show how to use a linear dependently typed programming language to define a direct and type-safe embedding of expressive binary session types.

Expressive Session Types

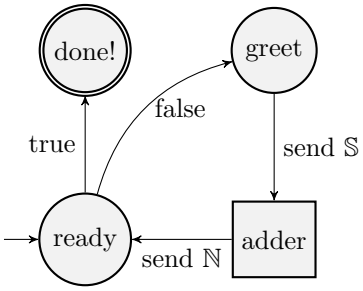
Our session types are closed under nested smallest ($\mu x.\ \cdot$) and largest ($\nu x.\ \cdot$) fixpoints, typed sends ($!A.\ \cdot$) and receives ($?A.\ \cdot$), the empty protocol (`end`) and binary branch offerings ($\cdot \,\&\, \cdot$) and selections ($\cdot \oplus \cdot$).

$$\begin{aligned} p,\ q,\ \ldots ::= {} & x \mid \mu x.\ p \mid \nu x.\ p \\ & \mid\ !A.\ p \mid ?A.\ p \mid \texttt{end} \\ & \mid p \,\&\, q \mid p \oplus q \end{aligned}$$

The session $\nu i.\ ((!\mathbb{N}.\ \texttt{end}) \,\&\, (?\mathbb{N}.\ i))$ types a server for the 'adder' protocol represented as a finite state machine on the right hand side. The server accepts an arbitrarily long sequence of natural numbers (sent by a client repeatedly selecting the *more* $= (?\mathbb{N}.\ x)$ branch) before sending back their sum as a single number (when the *eval* $= (!\mathbb{N}.\ \texttt{end})$ branch is selected) and terminating. The encoding of the server's type uses a largest fixpoint because the client is the one responsible for ensuring the process terminates by eventually selecting the *eval* branch.



We can use nested fixpoint to describe more complex protocols. For instance, the session described by $\nu r.\ (\texttt{end} \,\&\, (!\mathbb{S}.\ \nu i.\ ((!\mathbb{N}.\ r) \,\&\, (?\mathbb{N}.\ i))))$ informally corresponds to the finite state machine shown on the right hand side. It types a server repeatedly offering to handle the adder protocol described above after greeting the user with a string. We replaced adder's `end` with r to let the protocol loop back to the 'ready' state once the adder is done (in the state machine, that corresponds to merging 'stop' and 'ready' hence the bottom 'send ℕ'-labeled arrow from the 'adder' sub-machine back to 'ready').



Type-safe Implementation

Pairs of Unityped Channels We represent a bidirectional channel parametrised by a session type as a pair of unityped unidirectional channels. The key idea behind this safe implementation is to collect all the received (respectively sent) types used in a protocol and to manufacture a big sum type in which we can inject all of the received (respectively sent) values. Because

the received types of a protocol are equal to the sent types of its dual (and vice-versa), we can prove that the directed channels of two communicating parties are indeed type-aligned. The definition of this sum type is a type safe realisation of Kiselyov and Sabry's open union type [5] using the encoding of scope-safe De Bruijn indices [4] introduced by Brady in Idris 2's core [3]. We include its definition below: given a list `ts` of values of type `a` and a decoding function `elt` turning such values into types, (`UnionT elt ts`) defines a union type. Its one constructor named `Element` takes a natural number `k` functioning as a tag, a proof of type (`AtIndex t ts k`) that the value `t` is present at index `k` in `ts` and injects a value of type (`elt t`) into the union. By picking `elt` to be the identity function, we recover the usual notion of tagged union of types.

```
data UnionT : (elt : a -> Type) -> (ts : List a) -> Type where
  Element : (k : Nat) -> (0 _ : AtIndex t ts k) -> elt t -> UnionT elt ts
```

Note that the proof of type (`AtIndex t ts k`) is marked as erased through the use of the `0` quantity which means this datatype compiles down to a pair of a GMP-style `Integer` (the optimised runtime representation of the tag of type `Nat`) and a value of the appropriate type. This gives us an encoding that has constant time injections and partial projections (via an equality check on the `Nat` tag) in and out of the big sum type.

Dealing with Changing Sessions The session type, by design, changes after each communication but the big sum types used when communicating across the unityped channels need to stay the same. Our solution is to record an offset remembering where we currently are in the protocol and thus allowing us to keep injecting values into the initial shared sum types. Our offsets are proven correct using a gadget that can be understood as a cut down version of McBride's one hole contexts [8]. Instead of recording the full path followed by our programs in the finite state machine describing the protocol, we merely record a de-looped version.

Types of the Primitives We include code snippets showing the types of the primitives dealing with communications. Note that Idris 2's funny arrow (`-@`) denotes a linear function space, and consequently all of these primitives are linear in the channel they take as an input (ensuring it cannot be reused after the protocol has been stepped) and systematically return, wrapped in a linear IO monad called `IO1`, a channel indexed by the updated protocol. First, `send` and `recv`, the primitives dealing with sending arbitrary values back and forth. `send` takes an argument of type `ty` while `recv` returns one as the `Res`ult of the communication.

```
send : Channel (Send ty s) e -@ (ty -> IO1 (Channel s e))
recv : Channel (Recv ty s) e -@ IO1 (Res ty (\ _ => Channel s e))
```

Second, `offer` and `select`, the primitives dealing with branching. Both involve a choice in the form of a `Bool` named `b` that is used to compute the next channel state. `offer` gets it from the other thread as a `Res`ult of some communication while `select` requires the caller to pass it and will forward the decision to the other party.

```
offer : Channel (Offer s1 s2) e -@
  IO1 (Res Bool (\ b => ifThenElse b (Channel s1 e) (Channel s2 e)))
select : Channel (Select s1 s2) e -@
  ((b : Bool) -> IO1 (ifThenElse b (Channel s1 e) (Channel s2 e)))
```

Third, `unroll` and `roll`, the primitives dealing with the iso-recursive nature of our (co)inductive protocols. The types of these primitives reveal the role of `Channel`'s second parameter: it is a stack of open session types corresponding to the bodies of the nested fixpoints encountered during execution. `unroll` opens a fixpoint and thus pushes a new entry onto the stack `e` while `roll` does the opposite.

```
unroll : Channel (Fix nm s) e -@ IO1 (Channel s (s :: e))
roll   : Channel s (s :: e) -@ IO1 (Channel (Fix nm s) e)
```

Last but not least, `rec` steps a channel that has reached a recursive call to a fixpoint by looking up the open session and environment associated to the specified `pos`ition in the stack and reinstating them.

```
rec : {pos : _} ->  Channel (Rec pos) e -@
  IO1 (Channel (Fix nm (SessionAt pos e)) (EnvAt pos e))
```

Example Server Omitting the somewhat horrifying types, we include the implementation of the server repeatedly offering to greet the user and execute the adder protocol that we sketched earlier. We use Idris 2's named argument syntax to explicitly pass the name `nm` to the `unroll` and `rec` calls in order to clarify what step of the protocol is being reached.

```
server id ch = do
  ch <- unroll {nm = Nu "ready"} ch
  b # ch <- offer ch
  if b then close ch else do
    let msg = "Hello n°\{show id}!"
    ch <- send ch msg
    ch <- adder 0 ch
    server (1 + id) ch
```

```
adder acc ch = do
  ch <- unroll {nm = Nu "adder"} ch
  b # ch <- offer ch
  if b then do
      ch <- send ch acc
      rec {nm = Nu "ready"} ch
    else do
      (n # ch) <- recv ch
      ch <- rec {nm = Nu "adder"} ch
      adder (acc + n) ch
```

On the left-hand side we are acting as a `server` parametrised by a client `id` number. We first `unroll` the largest fixpoint corresponding to the 'ready' state repeatedly offering the adder protocol, and wait for the client to choose one of the branches we `offer`. If they are done we `close` the channel and terminate, otherwise we `send` a greeting, call the `adder` routine, and eventually continue acting as the server with an incremented id number.

On the right-hand side we are acting as the `adder` parametrised by an accumulator `acc`. We `unroll` the largest fixpoint corresponding to the 'adder' protocol and `offer` the client a choice between getting the running total or sending us more numbers. If they are done, we `send` back the value in the accumulator and, having reached a `Rec` variable in the protocol, use `rec` to jump back to the 'ready' state. Otherwise we `recv` an additional number, add it to the accumulator, use `rec` to jump back to the 'adder' state and continue acting as the `adder`.

Limitations and Future Work

Encoding Uniqueness via Linear Types Unlike Brady's prior work in Idris 1 [2], we are using Idris 2 which does not currently have uniqueness types. We are forced to use the linearity granted to us by Quantitative Type Theory (QTT) [9, 1] to emulate uniquess via a library-wide

invariant. This adds a degree of noise and trust to the implementation. We would ideally want a host system combining both linearity and uniqueness to benefit from enhanced expressivity and efficiency as described by Marshall, Vollmer, and Orchard [7].

Small Runtime Overhead In the current version of our library, we compute a handful of key offset values by induction over the protocol. Correspondingly the protocol is not marked as erased anymore and, if compilation does not specialise the relevant combinators agressively, then we may very well be evaluating these relatively small computations at runtime. As far as we understand, directly adding typed staging to Idris 2 through the use of a two-level type theory à la Kovács [6] would not be enough to solve our issue: for typing purposes, the protocol does need to persist through staging. But it should be possible to move it from QTT's unrestricted to its erased modality.

Ergonomics We found writing the types of inner loops of programs with non-trivial communication patterns to be tedious as they expose quite a lot of the powerful but noisy encoding of syntaxes with binding we use. Figuring out a more user-friendly surface syntax for our protocols is left as future work.

Totality Checking Our lightweight embedding does distinguish smallest and largest fixpoints but Idris 2's productivity checker does not currently take advantage of the knowledge that the user is making steady progress in a potentially infinitely repeating protocol ($\nu x.\ P$) to see that a program is total. Experiments suggest we may be able to introduce a lightweight encoding of a Nakano-style modality [10] for guarded programming to let the user justify that their co-recursive calls are all justified by regular progress.

References

[1] Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018.

[2] Edwin C. Brady. Type-driven development of concurrent communicating systems. *Comput. Sci.*, 18(3), 2017.

[3] Edwin C. Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[4] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[5] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In Chung-chieh Shan, editor, *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, pages 59–70. ACM, 2013.

[6] András Kovács. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.*, 6(ICFP):540–569, 2022.

[7] Danielle Marshall, Michael Vollmer, and Dominic Orchard. Linearity and uniqueness: An entente cordiale. In Ilya Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 346–375. Springer, 2022.

[8] Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 287–295. ACM, 2008.

[9] Conor McBride. I got plenty o' nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016.

[10] Hiroshi Nakano. A modality for recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*, pages 255–266. IEEE Computer Society, 2000.