Branching Out: Existential External Choice in Effpi

Benjamin Robinson
University of Oxford, Oxford, UK
benjamin.robinson@st-hughs.ox.ac.uk

Nobuko Yoshida
University of Oxford, Oxford, UK
nobuko.yoshida@cs.ox.ac.uk

`Effpi` [21, 22] is a framework for writing strongly-typed message-passing programs in Scala, where the compiler enforces the conformance of process implementations to specified protocol types. A compiler plugin is provided to verify properties of protocols, such as deadlock-freedom and liveness, by encoding the behavioural types into a variant of CCS [14].

To address limitations in the expressiveness of the existing toolkit, we extend `Effpi` with external choice by introducing a branching operation. Upon accepting a message via a `branch`, protocols enforce a continuation which depends on the label (type) of the received message. We equip the branching operation with the ability to accept messages over more than one channel. Additionally, we introduce a "catch timeout" operation to allow processes to gracefully handle a lack of incoming messages. The enhanced expressiveness of `Effpi` is demonstrated through a number of examples, including an implementation of the Raft consensus algorithm [16].

1 Introduction & Motivation

Concurrent programs are notoriously difficult to implement correctly. Nuanced bugs often arise from unexpected interactions between numerous components and are hard to reason about. Issues include deadlocks, livelocks, or simple protocol violations. Where possible, detecting and preventing such issues at compile-time is highly desirable. Behavioural type systems, such as session types [9], provide a formalism for encoding a protocol as a type, allowing for static conformance verification.

`Effpi` [21, 22] is a Scala 3 toolkit which aims to address these challenges. The framework provides an embedded DSL for writing message-passing programs, whose implementations are checked against a specified protocol, written as a dependent function type [3]. This allows channels to be tracked at the type-level, which facilitates model checking: protocols (even when defined as a type without a concrete implementation) can be verified for properties such as deadlock-freedom and liveness [22]. A compiler plugin encodes the behavioural types in a variant of CCS [14], and subsequently uses mCRL2 [7].

Despite its many strengths, `Effpi` has some limitations in expressiveness, which we now illustrate by means of two motivating examples. In both, we consider a hypothetical protocol and attempt to encode it as a behavioural type in the existing framework. We hope to represent the protocol faithfully so that the type system can be used to fully verify the correctness of the corresponding implementation.

Example 1.1. Consider a travel agency process which receives an accept/reject message from a client and provides a ticket accordingly, inspired by [23]. The protocol is represented by the `TravelAgency` type and implemented concretely by the `travelAgency` method. A key limitation is that the protocol does not enforce a ticket being returned in exactly the `Accept` case: line 5 uses the union type "`|`" to specify *internal* choice between providing a ticket or not; in reality, this choice should be external (dependent only on the client's message). The concrete process could currently provide tickets to clients who rejected the offer and still technically conform to the type. Ideally, the protocol should force the implementation to send a ticket if and only if the client accepted.

Lorenzo Gheri & Kirstin Peters (Eds.): 17th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software 2026 (PLACES 2026)
EPTCS 444, 2026, pp. 34–44, doi:10.4204/EPTCS.444.4

```
case class Accept(); case class Reject(); type Decision = Accept | Reject

type TravelAgency[C1 <: IChan[Decision], C2 <: OChan[String]] =
  In[C1, Decision, (d: Decision) => // receive the client's decision
    Out[C2, String] | PNil] // send a ticket or stop

def travelAgency(c1: IChan[Decision], c2: OChan[String]):
  TravelAgency[c1.type, c2.type] = {
    recv(c1) { (decision: Decision) =>
      case Accept() => send(c2, "Your ticket")
      case Reject() => nil // sending a ticket would still type-check here!
    }
}
```

Example 1.2. Consider a hypothetical *AuctionHouse* protocol, which accepts *Bid* messages on one channel and *CloseAuction* messages on another (for example, control messages from the auctioneer). Upon receiving a *CloseAuction* message, suppose that the protocol requires no further *Bid* messages be accepted. To implement this in the existing `Effpi` toolkit, we could try to merge the two original channels, so that a single `receive` operation can listen to both bids and control messages. We might attempt to use separate components to forward incoming messages onto the shared channel, as illustrated in Figure 1. However, this subtly adapts the protocol: upon closing the auction, a bid may get trapped in the forwarding process and therefore might never be delivered to the `AuctionHouse` (despite the client seeing the synchronous communication as successful).

Suppose further that we expect the `AuctionHouse` process to respond to *a lack of* incoming bids: for example, to lower the starting price. This cannot be achieved within the existing framework as there is no way to catch the timeout of a `receive` operation.[1] As with the rest of the protocol, this could be implemented as a concrete process, but we cannot represent this behaviour at the type-level.

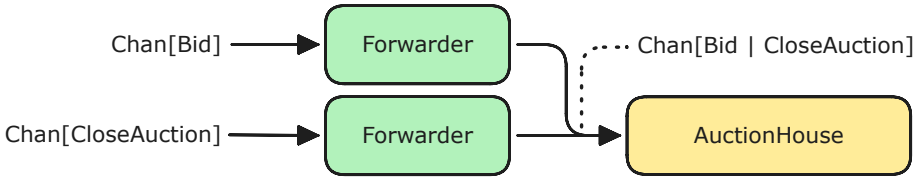

Figure 1: A faulty implementation of the `AuctionHouse` protocol by merging two channels.

With these motivating examples, we have identified three limitations of the existing toolkit: **the lack of external choice, the inability to listen on multiple channels simultaneously,** and **the inability to handle timeouts on inputs.** Whilst match types [5] were a promising initial direction of research, they addressed only one of the above limitations and became verbose for large protocols.[2] Instead, we introduce a custom branching operation, over possibly multiple input channels, and a "catch timeout" operation. We present the design of this extension to the embedded DSL and demonstrate its expressiveness through an implementation of the Raft consensus algorithm [16]. In doing so, we discover further ways in which the framework could be improved. The modified `Effpi` software toolkit and examples are available at:

https://github.com/benrobinson16/effpi

[1] The naive evaluation strategy throws a global runtime exception on timeout, but this cannot be caught in situ. The more sophisticated runtime systems do not handle timeouts at all. Details can be found in the original `Effpi` codebase [21].

[2] Due to the necessity of subtyping `Process`, each match type would need a separate `type` declaration.

2 Formal Calculus & Type System

The `Effpi` toolkit is built on the formal model of the $\lambda^{\pi}_{\leq}$-calculus and its type system [22]. To extend the toolkit with new operations, we present an extended calculus, which we call Extended-$\lambda^{\pi}_{\leq}$. We briefly outline possible modifications to the type system and semantics in the full version of this paper [17]. The focus remains on the practical implementation and applicability of an extension to `Effpi`, with formal soundness proofs of the extended type system left to future work.

The extended syntax is shown in Figure 2, with the new constructs highlighted. We introduce a set of labels $\mathbb{L}$ which are used to tag values. The **branch** operation takes a set of channels to listen on and a set of labelled continuations: upon receiving a labelled message on any of its channels, the corresponding continuation is executed, matching the label received. Meanwhile, the **timeout** operation is modelled on CSP's sliding-choice operator [18]: $\mathbf{timeout}(t,\ t')$ behaves initially as t before becoming t' if no input is received. At the formal level, we refrain from specifying the exact timing of the timeout, instead treating it as a non-deterministic tau action.

$$\mathbb{B} = \{\mathbf{tt},\ \mathbf{ff}\} \qquad \mathbb{C} = \{\mathrm{a},\ \mathrm{b},\ \mathrm{c},\ \ldots\} \qquad \mathbb{X} = \{x,\ y,\ z,\ \ldots\} \qquad \mathbb{L} = \{\ell_1,\ \ell_2,\ \ell_3,\ \ldots\}$$

$$\begin{array}{rcl}
\text{terms } \mathbb{T} \ni t, t', \ldots & ::= & \mathbb{X} \mid \mathbb{V} \mid \neg t \mid \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid \mathbf{let}\ x = t\ \mathbf{in}\ t' \mid t\ t' \mid \mathbf{chan}() \mid \mathbb{P} \\
\text{values } \mathbb{V} \ni u, v, \ldots & ::= & \mathbb{B} \mid \mathbb{C} \mid \lambda x.t \mid () \mid \mathbf{err} \mid \ell(t) \\
\text{processes } \mathbb{P} \ni p, q, \ldots & ::= & \mathbf{end} \mid \mathbf{send}(t,\ t',\ t'') \mid \mathbf{recv}(t,\ t') \mid t \parallel t' \mid \mathbf{branch}(\{t_j\}_{j\in J},\ \{\ell_k \mapsto t_k\}_{k\in K}) \\
 & & \mathbf{timeout}(t,\ t') \qquad \text{where } J, K \text{ are finite, non-empty index sets}
\end{array}$$

Figure 2: Syntax of the Extended-$\lambda^{\pi}_{\leq}$ calculus.

The syntax of types is extended similarly, as shown in Figure 3. We add support for unions over index sets, which we assume to be finite and non-empty. The indexed union can be viewed as a generalisation and replacement of the previous binary type $T \vee U$, and is useful when representing the possible input values to a branching operation (by taking the union over the continuation labels and arguments). The type $\langle \ell : T \rangle$ represents a labelled value, and the types $\mathbf{b}[\ldots,\ldots]$ and $\mathbf{timed}[\ldots,\ldots]$ represent the branching and timeout operations respectively.

$$\mathrm{bool} \mid () \mid \top \mid \bot \mid \Pi(\underline{x} : U)T \mid \mu \underline{x}.T \mid \underline{x} \mid \bigvee_{j\in J} T_j \mid \langle \ell : T \rangle \mid \mathrm{c}^{\mathrm{io}}[T] \mid \mathrm{c}^{\mathrm{i}}[T] \mid \mathrm{c}^{\mathrm{o}}[T]$$

$$\mathbf{proc} \mid \mathbf{nil} \mid \mathbf{o}[S,T,U] \mid \mathbf{i}[S,T] \mid \mathbf{p}[T,U] \mid \mathbf{b}[\{S_j\}_{j\in J}, \{\ell_k \mapsto U_k\}_{k\in K}] \mid \mathbf{timed}[T,U]$$

Figure 3: Syntax of types for Extended-$\lambda^{\pi}_{\leq}$.

Extended typing rules and semantics must build on the existing rules from [22]. As part of these rules, we expect to enforce conditions on the new constructs: for the timeout operation, we require that the first term is either a receive or a branch (the operations that can time out) and the second term is a lambda abstraction. For branching, we require:

- **Label distinctness:** Each continuation must have a distinct label.
- **Continuation coverage:** Every possible incoming label has a continuation.
- **Payload compatibility:** The payload type of each incoming message must be compatible with the expected argument type of the corresponding continuation.

Example 2.1. Consider again the travel agency protocol from Example 1.1. In the extended type system, we represent the protocol more accurately by using the branching type: the expected continuations after receiving accept/reject labels are given explicitly. We define $T_{\text{Decision}} = \langle \ell_{\text{Accept}} : () \rangle \vee \langle \ell_{\text{Reject}} : () \rangle$ to be the union of two labelled unit types, representing the client's decision.

$$T_{\text{Agency}} = \Pi(\underline{c1} : \mathrm{c}^{\mathrm{i}}[T_{\text{Decision}}])\ \Pi(\underline{c2} : \mathrm{c}^{\mathrm{o}}[\mathit{string}])\ \mathbf{b}\left[\{\underline{c1}\}, \begin{cases} \ell_{\text{Accept}} \mapsto \Pi()\ \mathbf{o}[\underline{c2}, \mathit{string}, \Pi()\ \mathbf{nil}] \\ \ell_{\text{Reject}} \mapsto \Pi()\ \mathbf{nil} \end{cases}\right]$$

A corresponding process is given below. Using extended typing rules, we verify that the process correctly implements the protocol; more precisely, we derive $\Gamma \vdash \mathit{myAgency} : T_{\text{Agency}}$. No process which returns a ticket after a ℓ_{Reject} label can be typed with T_{Agency}, so we have improved implementation safety.

$$\mathit{myAgency} = \lambda c1.\ \lambda c2.\ \mathbf{branch}\left(\{c1\}, \begin{cases} \ell_{\text{Accept}} \mapsto \lambda\_.\ \mathbf{send}(c2,\ \text{“ticket”},\ \lambda\_.\ \mathbf{end}) \\ \ell_{\text{Reject}} \mapsto \lambda\_.\ \mathbf{end} \end{cases}\right)$$

3 Extending the Effpi Toolkit

3.1 Existing Codebase

`Effpi` is implemented as an embedded DSL. At its core are classes representing the possible actions of a process, such as sending and receiving. Functions are provided to instantiate these classes in a manner that appears natural: as though the operation is truly happening at the point of invocation. Instead, the produced hierarchy of classes and continuations is interpreted at runtime. The framework additionally includes a compiler plugin, which encodes the behavioural types for model checking. The plugin is not modified in this paper, and it is left for future work to extend it for the new operations.

3.2 DSL Design

Branching. Like the existing constructs in `Effpi`, we introduce the branching operation by extending `Process`. We use tuples to represent the set of channels and continuations, as this allows for variable length whilst maintaining the specific type information of each element. Using a list would cast the channels and continuations to a common supertype, losing the vital type information required to enforce correct implementations. The specific type information is also required for model checking later on.

```
1 case class Branch[A, Chans <: Tuple, Matches <: Tuple]
2   (channels: Chans, matches: Matches, timeout: Duration)
3   (using val valid: ValidBranch[A, Chans, Matches],
4          val wrapper: WrapMatches[A, Matches]) extends Process
```

Implementing this construct has a few challenges; notably, we must enforce the validity of the operation (as previously defined) and also maintain the type information of each of the continuation arguments. To enforce the operation's validity:

- **Label distinctness:** In the formal model, we require each continuation label to be distinct. In Scala, we use types (e.g. case classes) as labels, but checking for overlapping types is non-trivial. One approach is to require sealed traits and use `Mirror.SumOf[A]` to analyse distinctness. However, the loss of flexibility when using overlapping label sets throughout a larger program is undesirable. Instead, we re-interpret the implementation as imposing an ordering on the continuations (which is natural for a `Tuple`). At runtime, the first matching continuation is selected.

- **Continuation coverage:** Every incoming message (every instance of `A`) must have a corresponding continuation: or more precisely, the union of continuation argument types must cover `A`. This is checked via a `using` clause [6], which requires an implicit instance of `ValidBranch` to be witness to the validity of the branching operation. This instance is provided by a `given` definition, listed below, allowing it be automatically derived at compile-time so the DSL code remains clear [4].
- **Payload compatibility:** The provided channels must support only subtypes of `A`. Again, this is checked via the `ValidBranch` implicit instance.

```
given valid[A, Chans <: Tuple, Matches <: Tuple](using
  // The channels are all input channels accepting some subtype of A
  ev1: Tuple.Union[Chans] <:< IChan[A],
  // The match cases are all functions to some process
  ev2: Tuple.Union[Matches] <:< Function1[?, Process],
  // The match cases cover exactly all possible inputs of A
  ev3: Tuple.Union[Tuple.Map[Matches, ArgumentOf]] =:= A,
): ValidBranch[A, Chans, Matches] with {}
```

In comparison to match expressions, this implementation provides more control to the framework authors. We develop a syntax that still feels natural to the programmer but maintains the necessary type information, especially for model checking in the future. Whilst every incoming message is guaranteed to have a corresponding continuation, we unfortunately lose the ability to detect duplicated labels compared to the formal model.

As well as preserving type information for compile-time checking of implementations, it is also necessary to preserve the types of the continuation arguments at *runtime*, so that the evaluator can select the correct continuation for each incoming message. We achieve this with another implicit parameter `WrapMatches`, which converts the tuple of continuations into a tuple of so-called `MatchCase` instances. Each instance gets tagged with a `TypeTest` for its argument type which we may use during evaluation.

With the types in place, the DSL can provide a `branch` function for writing branching operations, similar to the existing `send` and `receive` functions. This simply instantiates the `Branch` class.

Example 3.1. Using the provided `branch` DSL extension, we may reimplement the travel agency from Example 1.1 with enforced continuations dependent on the received `Decision` message.

```
type TravelAgency[C1 <: IChan[Decision], C2 <: OChan[String]] =
  Branch1[Decision, C1, (
    (a: Accept) => Out[C2, String], // Provide a ticket
    (r: Reject) => PNil // Do nothing
  )]

def travelAgency(c1: IChan[Decision], c2: OChan[String]):
  TravelAgency[c1.type, c2.type] = {
    branch1(c1, (
      (a: Accept) => send(c2, "Your ticket"),
      (r: Reject) => nil // Sending a ticket here would be a type error
    ))
}
```

Timeouts. To implement the timeout operator, we extend the DSL in a similar way: by subtyping `Process` (below). Note the restriction to `TimeoutableProcess`, which ensures that the first process provided is an `In` or `Branch` operation. (The `Out` operation cannot time out in the current toolkit.)

```
case class CatchTimeout[P <: TimeoutableProcess, Q <: Process]
  (p: () => P, onTimeout: () => Q) extends Process
```

As before, we provide a DSL function to instantiate the class. One notable property is that the `CatchTimeout` does not impose a timeout duration itself. Instead, the duration is specified by the wrapped `In`/`Branch` process.

Example 3.2. We may now represent the `AuctionHouse` protocol as originally described in Example 1.2, which was not previously possible. Using `CatchTimeout` allows us to respond to a lack of incoming bids, and the multi-channel branching allows us to correctly handle the auction closing.

```
type AuctionHouse[C1 <: IChan[Bid], C2 <: OChan[CloseAuction]] =
  CatchTimeout[
    Branch[Bid | CloseAuction, (C1, C2), (
      (b: Bid) => ..., // Handle the bid
      (c: CloseAuction) => PNil // Stop the auction
    )],
    ... // Handle timeout, e.g. reduce starting price
  ]
```

3.3 Evaluating the New Operations

`Effpi` provides three evaluation strategies: a naive strategy where each process runs on its own thread; and two more sophisticated strategies where processes are scheduled cooperatively to run on a fixed number of executor threads. We briefly describe the modifications to each.

Naive strategy. This strategy uses a recursive blocking function `eval` to interpret the operation tree. We support branching by polling on each of the channels (in a shuffled order, for fairness) until a message is available. Supporting timeouts involves setting a timeout continuation for the next recursive call of `eval`, which propagates to the evaluation of the wrapped `In` or `Branch` operation. Receiving directly from a channel can specify a timeout duration, and we catch this exception to trigger the timeout continuation.

Optimised runtime systems. These systems share a common executor-based architecture. Readers are referred to the linked code repository for the full details of the modified implementation. At a high level, the modifications include generalising channel queues to support enqueueing both `In` and `Branch` operations as waiting to receive messages. We introduce a timer thread to handle timeouts and schedule their continuations for execution.

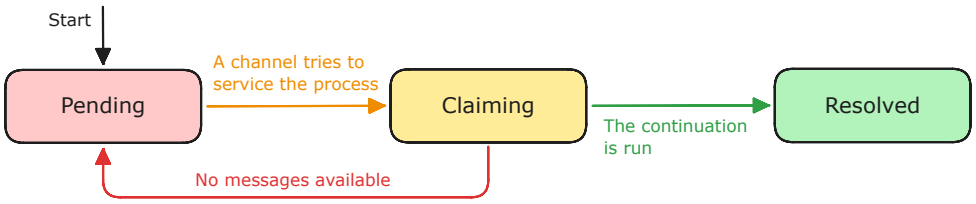

Figure 4: The state of a `WaitingProcess`, tracked by an atomic variable.

Careful consideration is given to the interaction of multiple channels attempting to resolve the same `Branch` operation, and to races arising between timeouts and incoming messages. It is important to ensure that only one continuation (whether timeout or message-based) is triggered for each operation, regardless of the number of channels involved. We introduce a 3-state atomic variable to track the state of a waiting process: `Pending`, `Claiming`, and `Resolved`. The intermediate `Claiming` state is used to ensure only one channel (or timeout) can attempt to resolve the operation at a time. When a timeout fires but is unable to claim the waiting process, it is rescheduled to try again after a short delay, until it can claim the process or the process is resolved by an incoming message.

4 The Raft Election Algorithm

4.1 Overview

Raft [16] is a consensus algorithm for managing a replicated state machine across a distributed system. It provides guarantees (such as election safety and leader completeness), even in the presence of node failures. Raft is widely praised for its understandability, especially when compared to the Paxos algorithm [12], although the two algorithms share a similar approach to consensus [11].

A key component of the Raft algorithm is the leadership election, which we explore in detail. Nodes exist in one of three states (*follower*, *candidate*, or *leader*) and transition between states as internal timeouts expire or as *remote procedure calls* (RPCs) are exchanged. These have two forms: *RequestVote* (sent by candidates to gather votes) and *AppendEntries* (sent by leaders to replicate the log).

Each RPC carries a *term number* which serves as a logical clock and is incremented with each election. The key property of *election safety* ensures that there is at most one leader per term.

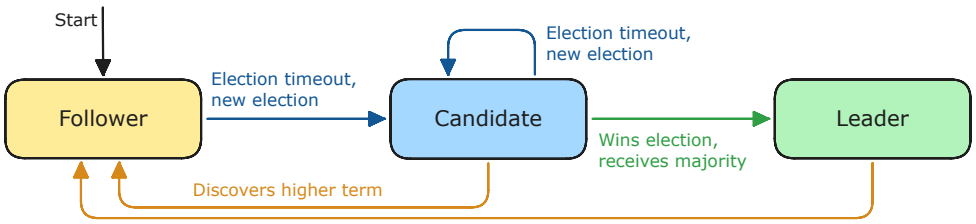

Figure 5: State machine representing Raft node states, adapted from [16].

We may summarise the election algorithm with respect to the role of each node state. Additional details, such as the conditions attached to granting votes, have been omitted for brevity (see [16]).

- **Follower:** Listens for incoming RPCs, replying appropriately. If no *AppendEntries* are received for the election timeout period, transition to candidate and stand in a new election.
- **Candidate:** Starts a new election, incrementing the term number and broadcasting *RequestVote* RPCs to the other nodes. If a majority of nodes grant their vote, transition to leader. Otherwise, if the election timeout occurs without a majority, start a new election and stand for election again.
- **Leader:** Sends periodic *AppendEntries* RPCs to the other nodes, acting as heartbeats. In the full Raft protocol, the leader is additionally responsible for writing new entries to the replicated log.
  - In both the *leader* and *candidate* states, if an RPC is received with a higher term number, the node transitions back to being a *follower*.

4.2 Implementation in Effpi

With the help of the newly introduced operations, we may begin to express the Raft election protocol in `Effpi` at the type-level. For clarity, we approach this task in several stages.

Timer process. The protocol uses timeouts to trigger state transitions: for example, followers become candidates if they do not receive heartbeats within a randomly chosen election timeout period. To model this behaviour, we define a `timer` process which is responsible for producing timeout events and handling reset instructions from the main node process. We model the timer as a separate process to allow for more fine-grained control over when to reset the timer and listen for timeout events. Many of Raft's receiving/branching operations should only reset the timer on some of their continuations, for example.

```
type Timer = Rec[RecX,
  In[timerReset.type, TimerReset, TimerReset => Rec[RecY,
    CatchTimeout[
      // If we receive a timer reset before timeout, we loop to RecY.
      In[timerReset.type, TimerReset, TimerReset => Loop[RecY]],
      // Once we time out, we loop to RecX: no pending timeout!
      Out[timeoutChan.type, TimerExpired] >>: Loop[RecX]
    ]]]]
```

By using two recursive variables, `RecX` and `RecY`, we model the different behaviours of the timer process depending on whether it has received a reset instruction since the last timeout event. After receiving the first `TimerReset`, we use the new `CatchTimeout` operator: we accept reset messages, but produce a `TimerExpired` message if no reset message is received in time.

RPC reply behaviours. Before defining the processes for each node state, we define common behaviours shared between them. Replying to a *RequestVote* RPC involves deciding to grant or deny the vote and performing the corresponding actions. We encapsulate this in `GrantVoteBehaviour` and its `Deny` counterpart, and call their union `VoteReplyBehaviour`. The type parameter `C` represents the channel to reply to, ensuring the correct channel is used. After granting a vote, the node is always expected to become a follower, whereas after denying a vote, it should return to its current state. We capture this via another type parameter `V`, representing the recursion to loop back to.

```
type GrantVoteBehaviour[C <: OChan[VoteResponse]] =
  Out[C, GrantVote] >>: Out[timerResetChan.type, TimerReset]
    >>: Loop[RecFollower]

type RefuseVoteBehaviour[C <: OChan[VoteResponse], V[A] <: RecVar[A]] =
  Out[C, RefuseVote] >>: Loop[V]
```

Follower state. We now define the full Raft election protocol state by state, beginning with followers. At its core, we define the process as repeatedly receiving messages over the inbox and timeout channels, replying to the RPCs appropriately. Dependent function types [3] allow us to extract the type of the reply channel from the incoming message. Passing the reply channel type as a parameter to the reply behaviours ensures the correct channel is used for replies. Upon receiving a timeout, we transition to become a candidate. The `branch` operation is instrumental in enforcing the correct continuations are run and enables us to listen on both an external inbox channel and the internal timeout channel.

```
type Follower = Rec[RecFollower,
  Branch[RpcMessage | TimerExpired, (inbox.type, timeoutChan.type), (
    (rv: RequestVote[_]) => VoteReplyBehaviour[rv.reply.type, RecFollower],
    (ae: AppendEntries[_]) => AEReplyBehaviour[ae.reply.type, RecFollower],
    TimerExpired => Candidate
  )]]
```

From this type-level description, we may implement the process straightforwardly, simply using the DSL and filling in details such as the logic for deciding between granting and denying votes. The definition of the protocol as a type constrains the process definition, reducing possible implementation errors. Readers are referred to the open source code repository for the full program.

Candidate state. In addition to replying to incoming RPCs like followers do, candidates must also initiate an election, broadcasting *RequestVote* messages to the other nodes. We define the candidate in two parts: first, a `CandidateElection` type which handles the lifetime of a single election; and second,

the full `Candidate` type which starts a new election. Splitting the definition in this way allows us to capture the fresh reply channel as type parameter `C`. This forces the broadcasted *RequestVote* messages to specify the same channel as the one we listen on for incoming vote responses.

```
type CandidateElection[C <: Chan[RequestVote]] = Par[
  Broadcast[RequestVote[C]], // Broadcast vote requests, with replyChan = C
  Rec[RecCandidate,
    Branch[RPC | TimerExpired | VoteResponse, (C, /* channels */), (
      ... // Handle the incoming RPCs and vote responses
    )]
  ]]

type Candidate = Rec[RecElection, Out[timerReset.type, TimerReset]
  >>: CandidateElection[Chan[VoteResponse]]]
```

Leader state. Finally, a leader process may be defined, following the same design principles. A key difference is that the leader must broadcast *AppendEntries* RPCs periodically. This is achieved by making use of the timer process and using two recursive variables, much like the candidate state. The full implementation is again deferred to the code repository.

```
type Leader = Rec[RecLeaderHeartbeat, // First recursive variable
  Out[timerReset.type, TimerReset]
    >>: Par[
      Broadcast[AppendEntries[Chan[AckAppendEntries]]],
      Rec[RecLeader, // Second recursive variable
        Branch[RPC | TimerExpired, (inbox.type, timeoutChan.type), (
          (ae: AppendEntries[_]) => ..., // As before
          (rv: RequestVote[_]) => ..., // As before
          TimerExpired => Loop[RecLeaderHeartbeat]
        )]
      ]
    ]]
```

Discussion. By extending the `Effpi` toolkit, we have been able to better express the Raft election algorithm at the type-level, capturing the required behaviours more faithfully. The implementation of each process is type-checked against the defined protocol type, making it easier to identify errors now that branching and timeout operations are enforced by the type system. However, there are still a number of limitations, such as the lack of model checking support for programs using the new operations. Creating a fresh channel (as in `CandidateElection`) could also be better supported. Note we restrict Raft to a fixed number of nodes: the toolkit could natively support any number of nodes by introducing a true broadcasting operation.

5 Conclusion & Related Work

In this work, we presented an extension to the `Effpi` toolkit, by means of the Extended-$\lambda^{\pi}_{\leq}$ calculus and concrete Scala implementation. The new branching and timeout operations improve the expressiveness of `Effpi`, which we demonstrated through two smaller examples and an implementation of the Raft election algorithm. The enhanced expressiveness enables more faithful representations of protocols as types, allowing us to encode more precise protocol specifications and thereby better ensure process implementations adhere to their protocols.

Related work. Effpi was introduced by Scalas *et al.* [21, 22] and has since been used for code generation from global Scribble types [1]. Few other extensions have been proposed to the toolkit to date. Frameworks such as Akka Typed [13] and `lchannels` [20] are alternative toolkits for writing typed message-passing programs in Scala, the latter using session types as the underlying type system.

External choice is a common operation in concurrency theories. Multiparty session types [24] include branching and selection as core constructs; unlike our approach, there are no separate send/receive operations, only branch/select. This shows how the branching operation could be used as a replacement for the existing receive operation, rather than as an extension. The π-calculus implements choice through a "+" operator, which offers input on possibly multiple channels [15]. Communicating Sequential Processes (CSP) also includes external choice [8].

Timeouts have been studied in a number of works. Notably, CSP includes a sliding choice operator [18], which inspired our own operation. Timed extensions to the π-calculus have been proposed [19] as well as extensions to multiparty session types [10], both of which handle time constraints more generally.

Future work. Future work includes proving the formal properties of the extended type system and implementing model checking support for the new constructs in the compiler plugin. This will enable the verification of deadlock-freedom, among other properties, of programs that use branching or timeouts. We aim to verify the Raft implementation presented here and explore the use of the model checking to prove properties such as election safety. Raft has previously been modelled in mCRL2 [2], which may provide a useful comparison for this verification.

Acknowledgements. The authors would like to thank Dylan McDermott and the PLACES reviewers for their helpful comments. The first author is partially supported by a travel grant from St Hugh's College, University of Oxford. The second author is partially supported by EPSRC grants EP/T006544/2, EP/T014709/2, EP/Y005244/1, EP/V000462/1, EP/X015955/1, EP/Z0005801/1; Horizon EU TaRDIS 101093006 (UKRI No. 10066667); and ARIA.

References

[1] Adam D. Barwell, Ping Hou, Nobuko Yoshida & Fangyi Zhou (2023): *Designing Asynchronous Multiparty Protocols with Crash-Stop Failures*. In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 263, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 1:1–1:30, doi:10.4230/LIPIcs.ECOOP.2023.1.

[2] Parth Bora, Pham Duc Minh & Tim A.C. Willemse (2024): *Modelling the Raft Distributed Consensus Protocol in mCRL2*. *Electronic Proceedings in Theoretical Computer Science* 399, pp. 7–20, doi:10.4204/eptcs.399.4.

[3] Scala Developers (2021): *Scala Documentation: Dependent Function Types*. Available at `https://www.scala-lang.org/api/3.3.7/docs/docs/reference/new-types/dependent-function-types.html`.

[4] Scala Developers (2021): *Scala Documentation: Givens*. Available at `https://www.scala-lang.org/api/3.3.7/docs/docs/reference/contextual/givens.html`.

[5] Scala Developers (2021): *Scala Documentation: Match Types*. Available at `https://www.scala-lang.org/api/3.3.7/docs/docs/reference/new-types/match-types.html`.

[6] Scala Developers (2021): *Scala Documentation: Using Clauses*. Available at `https://www.scala-lang.org/api/3.3.7/docs/docs/reference/contextual/using-clauses.html`.

[7] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko & Muck van Weerdenburg (2007): *The Formal Specification Language mCRL2*. In: *Methods for Modelling Software Systems (MMOSS)*, *Dagstuhl*

Seminar Proceedings (DagSemProc) 6351, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 1–34, doi:10.4230/DagSemProc.06351.12.

[8] C. A. R. Hoare (1978): *Communicating Sequential Processes*. *Commun. ACM* 21(8), pp. 666–677, doi:10.1145/359576.359585.

[9] Kohei Honda (1993): *Types for Dyadic Interaction*. In Eike Best, editor: *CONCUR'93*, *Lecture Notes in Computer Science* 715, Springer, Berlin, Heidelberg, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[10] Ping Hou, Nicolas Lagaillardie & Nobuko Yoshida (2024): *Fearless Asynchronous Communications with Timed Multiparty Session Protocols*. In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 313, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 19:1–19:30, doi:10.4230/LIPIcs.ECOOP.2024.19.

[11] Heidi Howard & Richard Mortier (2020): *Paxos vs Raft: Have we reached consensus on distributed consensus?* *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, doi:10.17863/CAM.51885.

[12] Leslie Lamport (1998): *The Part-Time Parliament*. *ACM Transactions on Computer Systems* 16(2), pp. 133–169, doi:10.1145/279227.279229.

[13] Inc Lightbend (2019): *Akka Typed Documentation*. Available at https://doc.akka.io/libraries/akka-core/current/typed/.

[14] Robin Milner (1989): *Communication and Concurrency*. Prentice-Hall international series in computer science, Prentice-Hall, New York.

[15] Robin Milner (1999): *Communicating and Mobile Systems: The Pi-calculus*. Cambridge University Press, USA.

[16] Diego Ongaro & John Ousterhout (2014): *In Search of an Understandable Consensus Algorithm*. In: *USENIX ATC'14*, USENIX Association, USA, pp. 305–320. Available at https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[17] Benjamin Robinson & Nobuko Yoshida (2026): *Branching Out: Existential External Choice in Effpi (Full Version)*. Available at https://benrobinson.dev/papers/places26-full.pdf.

[18] Andrew W. Roscoe (2000): *The Theory and Practice of Concurrency*. Prentice Hall series in computer science, Prentice Hall, London.

[19] Neda Saeedloei & Gopal Gupta (2013): *Timed Pi-calculus*. In Martín Abadi & Alberto Lluch Lafuente, editors: *Trustworthy Global Computing*, Springer International Publishing, Cham, pp. 119–135, doi:10.1007/978-3-319-05119-2_8.

[20] Alceste Scalas & Nobuko Yoshida (2016): *Lightweight Session Programming in Scala*. In Shriram Krishnamurthi & Benjamin S. Lerner, editors: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 56, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 21:1–21:28, doi:10.4230/LIPIcs.ECOOP.2016.21.

[21] Alceste Scalas, Nobuko Yoshida & Elias Benussi (2019): *Effpi: Verified Message-Passing Programs in Dotty*. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, Scala '19, Association for Computing Machinery, New York, NY, USA, pp. 27–31, doi:10.1145/3337932.3338812.

[22] Alceste Scalas, Nobuko Yoshida & Elias Benussi (2019): *Verifying Message-Passing Programs with Dependent Behavioural Types*. In: *PLDI'19*, ACM, Phoenix AZ USA, pp. 502–516, doi:10.1145/3314221.3322484.

[23] Nobuko Yoshida & Lorenzo Gheri (2020): *A Very Gentle Introduction to Multiparty Session Types*. In: *Distributed Computing and Internet Technology: 16th International Conference, ICDCIT 2020, Proceedings*, Springer-Verlag, Berlin, Heidelberg, pp. 73–93, doi:10.1007/978-3-030-36987-3_5.

[24] Nobuko Yoshida & Ping Hou (2024): *Less is More Revisited*. In Ana Cavalcanti & James Baxter, editors: *The Practice of Formal Methods: Essays in Honour of Cliff Jones, Part II*, Springer Nature Switzerland, Cham, pp. 268–291, doi:10.1007/978-3-031-66673-5_14.