Dependent protocols for communication

Daniel Gustafsson
IT University of Copenhagen
dagu@itu.dk

Nicolas Pouillard
IT University of Copenhagen
npou@itu.dk

Abstract

Programming correct and precise communicating systems remains a challenging and error-prone venture. Still, communicating systems are not only used in practice everyday but also as a device for game based proofs such as security notions. We present a shallow embedding of protocols within the dependent type theory of Agda, enabling mechanized formal reasoning of processes following precise protocols. Using this shallow embedding we reconstruct the connectives from linear logic, and provide an elegant reasoning tool for communicating systems, all within a purely functional dependently typed programming language.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]

Keywords (Homotopy) Type Theory, Functional Programming Languages, Dependent Types, Mechanized Reasoning, Session Types, Protocols, Linear Logic

1. Introduction

We present a shallow embedding of protocols within the dependent type theory of Agda, enabling mechanized formal reasoning of processes following precise protocols.

A warm-up example: The Diffie-Hellman key exchange protocol is a well established protocol introduced by Whitfield Diffie and Martin Hellman (Diffie and Hellman 1976). It is one of the first constructions of public key cryptography. The core ideas behind this protocol are still in use today. In this setting Alice wants to agree with Bob on a common shared secret. This secret can later be used as a key to secure their communication. They both already agreed on the protocol to follow to reach this goal. The protocol relies on the use of a group structure `G`. The choice of the group is crucial for the security of the protocol but the correctness only relies on its structure. `G` is a multiplicative group of order q with a generator g, meaning that elements of `G` can be written as g^x for some x in $\mathbb{Z}_\mathtt{q}$ (the additive group of integers modulo q).

Alice starts by picking a random value x in $\mathbb{Z}_\mathtt{q}$ and sends g^x to Bob. Bob also picks a random value y in $\mathbb{Z}_\mathtt{q}$ and receives g^x. He responds to Alice by sending g^y. They now both can agree on the common value g^{xy}, Alice can raise the received g^y to the x and Bob can raise the received g^x to the y. The laws of exponentiation ensure that they agree on a common value. In this example let us assume that Alice is then sending this value g^{xy}.

In our system the protocol for Alice can be described as follows:

```
send λ (g^x   : G) →
recv λ (g^y   : G) →
send λ (g^x y : G) →
end
```

We shall give the details of these protocols in section 2 but note that `g^x`, `g^y`, and `g^x y` are unused bound variables. This protocol serves as a type for Alice's behavior. As a type it already rules out errors such as a wrong order of communication or ill-typed messages. However this protocol says nothing about the precise values for these group elements and the relations between the messages. A more precise protocol for Alice is the following:

```
send λ (x     : □ ℤq) →
send λ (g^x   : S⟨ g ^ x ⟩) →
recv λ (y     : □ ℤq) →
recv λ (g^y   : S⟨ g ^ y ⟩) →
send λ (g^x y : S⟨ g ^(x * y) ⟩) →
end
```

This updated version of Alice behavior is more precise: she first has to "send" her secret value `x` but the □ tells that this is a phantom message. These phantom messages are not really sent and thus reveal no dynamic information, however they can be used to precisely describe the remainder of the protocol. She then has to send a real message which given its *singleton* type has to be equal to `g ^ x`. She then receives from Bob the corresponding phantom secret `y` and real value `g^y`. The type of `g^y` ensures that if Bob follows the protocol then `g^y` is equal to `g ^ y`. Finally, Alice can show that she knows the value of `g ^(x * y)`. As previously she sends the secret as her last message.

Bob can be given a similar protocol which is derived from the one for Alice. Indeed for the communication to go well the protocol for Alice and Bob have to match up. As long as they have matching protocols we can show that well typed participants can always communicate and finish the protocol. Two processes communicating using a common protocol can be seen as a combined process which only sends the messages being exchanged, namely the *transcript*. We shall see how these protocols can be used to extract the type of such a transcript.

We shall also consider more advanced protocols. Suppose that Charlie can read the transcript from Alice and Bob. However since Charlie cannot guess the secret by himself he is concurrently communicating with Eve following a different protocol. Eve is assumed to be capable of breaking a game known as IND-CPA. Charlie can then break a supposedly hard problem with the help of Eve. This form of concurrent interaction along multiple protocols is used in cryptography. We elaborate on this example in section 3.4.

TODO, TODO.

ACM TODO. . . $15.00.
http://dx.doi.org/10.1145/TODO

As a separate extension we shall consider protocols synchronizing multiple participants at the same time. For instance Alice, Bob, and Dan wants to agree on a common secret. In section 4, the extension to three participants of Diffie-Hellman key exchange will be shown as an example of multiparty protocols.

Contributions:

- We present an exploration of the consequences of using (only) type theory quantifiers (Σ-types for sending, Π-types for receiving) as the types for communication.
- What is the result of communication? Our answer is that when a sender meets a receiver they can communicate and then appear as a sender. A sender being a Σ-type, the result of a communication is a transcript of the exchanged messages.
- We express processes communicating on multiple channels as a single channel communication. As a result in section 3 we obtain a shallow embedding of the additive and multiplicative fragment from linear logic.
- Put together in section 3.4 we express: games from security notions; simulators between these games; proofs about these simulators.
- In section 4 we scale to multiparty protocols (still using sequential protocols). Our solution is twofold: first we describe the whole protocol at once and provide projections for any subsets of the participants; second we can still keep a form of dependent protocols through the use of broadcast and phantom messages (shape-polymorphism, `□ A`).
- The full AGDA development is available online (Gustafsson and Pouillard 2014).

Notations: Throughout the paper, our definitions are presented in AGDA (Norell 2007) notation. With $\star$ we denote the type of types. The function space is written `A → B`, while the dependent function space is written `(x : A) → B x`, `∀ (x : A) → B x`, or `Π A B`. An implicit parameter, can be introduced via `∀{x : A} → B x`, and can be omitted at a call site if its value can be uniquely inferred from the context. There are shortcuts for introducing multiple arguments at once or for omitting a type annotation, as in `∀{A} {i j : A} x → e`. We will use mixfix declarations, such as `_⊎_`, where underscores denote where arguments go. AGDA is strict about whitespace, for instance $g^{x\,y}$ or $p{\leq}q$ are single identifiers because they contain no space.

Core types: As a tool AGDA comes with no predefined concepts other than types and functions: everything else must therefore be defined. In particular, there is no specific sort for propositions: everything is in $\star$. We denote the empty type as $\mathbb{O}$ and it is used to represent falsity. The type $\mathbb{1}$, has one value namely 0_1, and it is used to represent trivial truth. The type $\mathbb{2}$ has two values (0_2 and 1_2), and it is used both to denote a single bit of information and as a Boolean value where 0_2 denotes false and 1_2 denotes true. The type `_≡_` is the type of propositional equality, also called the identity type. AGDA reserves the usual equality symbol `=` for definitions; we apply this convention to our mathematical statements as well.

A note on Σ-types and Equivalences: In type theory `Σ A B` is used to denote a dependent sum (sometimes called a dependent pair). Here `A` is a type and `B` is a dependent type over `A` (hence `B` has type `A → ★`). These pairs can be built using the `_,_` constructor which has the type `(x : A) → B x → Σ A B`. Moreover, pairs come with two projection functions `fst : Σ A B → A` and `snd : (p : Σ A B) → B (fst p)`. The type `A ≃ B` is used to denote equivalences between types `A` and `B`: it means there are two functions, one going from `A → B` and the other from `B → A` such that composing them either way yields the identity function. The type `_≃_` is an equivalence relation for types.

Shape-Polymorphism and Singleton Types: In this paper we will use an experimental modality available in AGDA. This modality is introduced by `..(x : A) → B` in AGDA but we wrap it as a data type called `□ : ★ → ★`. It is similar but different from the proof irrelevance modality. Terms can safely be erased before run-time, however they still matter for type checking. This means that it is not possible to pattern match on a value of type `□ A`, since at run-time there might be nothing to pattern match against. This type, is useful for situations where a term is needed in order to type-check another expression, but is not used in the program. Throughout the paper we will silently convert to the type `□` to reduce clutter. We use this in conjunction with a singleton type `S⟨_⟩ : □ A → ★`, which has one constructor, namely `S[_] : (x : A) → S⟨ x ⟩`. There is only one value of the type `S⟨ v ⟩` namely `v` but with the singleton type this value now becomes available for inspection.

2. Protocols as dependent types

2.1 Protocols, Processes, and Communication

To assign types to a communication we first focus on typing each *participant* individually. We define *protocols* as a type, namely $\texttt{Proto}_0$ (the subscript, $_0$, can be disregarded for the moment). This type describes the type of messages to be sent and received by one participant. When there is nothing left to send or receive the protocol ends. The protocols we describe are solely made of send and receive actions. However after a communication (send or receive), the protocol can use the communicated message to determine how the protocol unfolds. This notion of *dependency* makes direct use of the type formers for dependent types, namely Π-types and Σ-types. For now, protocols are either `end` or an action $\texttt{send}'$ or $\texttt{recv}'$ together with the type of the message and the remaining protocol. We shall see the generalized definition of $\texttt{Proto}_0$ soon but first here are two protocols:

```
P₀ P₁ : Proto₀
P₀ = recv′ ℕ (recv′ ℕ (send′ ℕ end))
P₁ = send′ ℕ (send′ ℕ (recv′ ℕ end))
```

The protocol $\texttt{P}_0$ (respectively $\texttt{P}_1$) is a type for processes receiving (respectively sending) two natural numbers, then sending (respectively receiving) a natural number, and finally ending. Two processes $\texttt{t}_0$ and $\texttt{t}_1$ obeying protocols $\texttt{P}_0$ and $\texttt{P}_1$ respectively can be made to communicate with each other. For example $\texttt{t}_0$ could be sending the sum or the product of two received numbers. The communication will not get stuck since the two protocols $\texttt{P}_0$ and $\texttt{P}_1$ are opposite or *dual* protocols. In short, to achieve two-party communication for any protocol, we put together any two processes obeying a protocol and its dual.

What are the possible pure behaviors of a participant? For $\texttt{P}_0$ a participant is going to receive two natural numbers and send back a natural number. Can this participant be described by a mapping from the type $(\mathbb{N} \times \mathbb{N})$ to $\mathbb{N}$? An equivalent type is obtained by decoding the action $\texttt{recv}'\ \texttt{A}\ \texttt{B}$ to `A → B`, $\texttt{send}'\ \texttt{A}\ \texttt{B}$ to `A × B`, and `end` to `End` (a custom unit type equivalent to $\mathbb{1}$ whose constructor is `end` as well). This decoding is called $[\![\_]\!]$ and it maps protocols to AGDA types. For instance $[\![\ \texttt{P}_0\ ]\!] \equiv \mathbb{N} \to (\mathbb{N} \to (\mathbb{N} \times \texttt{End}))$ and $[\![\ \texttt{P}_1\ ]\!] \equiv \mathbb{N} \times (\mathbb{N} \times (\mathbb{N} \to \texttt{End}))$. A process of type $[\![\ \texttt{P}_0\ ]\!]$ can be defined as: $\texttt{t}_0 = \lambda\ \texttt{m}\ \texttt{n} \to (\texttt{m + n , end})$. From a type theory standpoint, $[\![\_]\!]$ is defined using a so-called "large elimination", which means that it is a function that computes a type. In short, a receiver is a function of the received value, and a sender is a pair whose first component is the value to send.

Why restrain ourselves to pure behaviors? First, termination is critical to get the total correctness results we are looking for. Second, while pure, these processes can still carry state through the steps of the protocol. Third, this is enough flexibility to model our games and adversaries. Finally, most effects can be handled through the use of communication to special processes which actually perform these effects.

Serialization: We want to stress that we are not addressing *how* the communication should take place in a distributed setting. For instance we do not explain how the messages should be encoded and serialized. Moreover we assume a well-typed setting where the processes would all be well-typed under consistent assumptions. We will use the fact that one can "send"/"receive" functions and types. This is because we focus on a notion of local communication that we are going to describe within our language. Our goal is to reason about these protocols and processes. Later on we shall describe what kind of reasoning we have about processes but we will mainly speak about the messages which have been exchanged by two processes following a protocol `P`.

Channels: We have no explicit channels. Channels would enable a much richer topology for communication between processes. Channels were not necessary to model the protocols and processes used in the games arising from the game based security notion. Moreover typing the use of these channels usually requires a form of linearity. Indeed, every time a process sends or receives on a channel, the type of this channel has to make one step through the protocol. In section 3 we describe concurrent protocols; a process can then follow two protocols concurrently and this is akin to having access to two channels. When composing these "2-channels" processes this can be thought of as putting these processes in parallel connecting each other by one shared channel.

Concurrency: The protocols we propose are completely sequential and therefore involve no concurrency, which means we have neither race conditions nor deadlocks. In section 3 we introduce a way to combine protocols to emulate concurrency.

The type of dependent protocols: We aim at a richer notion of protocols while keeping the simplicity of send and receive. Similar to how dependent type theory generalizes the notion of functions and pairs, i.e. Π-types and Σ-types, we will generalize the protocols to dependent protocols. Without further ado here is the definition for protocols and their decoding as types:

```
data Proto₀ : ★₁ where
  end  : Proto₀
  send : {M : ★₀}(P : M → Proto₀ ) → Proto₀
  recv : {M : ★₀}(P : M → Proto₀ ) → Proto₀

⟦_⟧ : Proto₀ → ★₀
⟦ end       ⟧ = End
⟦ send {M} P ⟧ = Σ M (λ m → ⟦ P m ⟧)
⟦ recv {M} P ⟧ = Π M (λ m → ⟦ P m ⟧)
```

The data type definition above introduces four constants, namely `Proto₀`, `end`, `send`, and `recv`. The function `⟦_⟧` maps `end` to the unit type `End`, maps `send` to a Σ-type, and maps `recv` to a Π-type. Because of the quantification on message types, the type `Proto₀` is actually of type $\star_1$ and not $\star_0$ (we shall return to this shortly). The constructor `send` generalizes `send′` and is used for sending, while `recv` generalizes `recv′` and is used for receiving. The actions `send` and `recv` take the type `M` as an implicit argument, as it usually looks nicer to annotate the variable at its introduction. These protocols are richer as the next steps of a protocol might depend on the *value* of the sent or received message. We shall use these dependencies soon, but first we recover `send′` and `recv′`.

```
send′ recv′ : (M : ★)(P : M → Proto₀ ) → Proto₀
send′ M P = send λ (_ : M) → P
recv′ M P = recv λ (_ : M) → P
```

The type `M → Proto₀` is about computation: Observe that the continuations `M → Proto₀` in both `send` and `recv` can potentially analyze the message of type `M` to compute the remainder of the protocol. This is not 'higher-order abstract syntax' used in the Logical Framework (Harper et al. 1993) as here the `_→_` denotes the full computational function space. The rest of this section gives more examples as why we want the type `M → Proto₀` to potentially permit any function. The type `Proto₀` is not a syntax isolated from the type theory and its λ-calculus.

Sinks and Sources: We call a *sink* (respectively a *source*) a protocol only made of `recv` actions (respectively `send` actions). Sinks receive data without sending anything back, at least not through the same "channel". Sources only send data. Pure sinks have no observable value. Intuitively, since the type `A → 𝟙` is equivalent to `𝟙`, the same applies to sinks.

Dual and log: For any protocol we define its dual and its source. Dualizing amounts to exchanging the `send` and `recv` actions in a protocol. The `source-of` function returns the source of a protocol by replacing all the `recv` actions by `send` actions. The `sink-of` function will produce the sink of a protocol by dualizing its source. We call `⟦_⊥⟧` the composition of `⟦_⟧` and `dual`. A process following a source protocol can be thought of as a logger. We thus call the result of decoding a source the *transcript* or the *log*. The decoding function `Log` is the composition of `⟦_⟧` and `source-of`.

```
dual source-of sink-of : Proto₀ → Proto₀
dual end       = end
dual (recv P) = send (λ m → dual (P m))
dual (send P) = recv (λ m → dual (P m))

source-of end       = end
source-of (recv P) = send (λ m → source-of (P m))
source-of (send P) = send (λ m → source-of (P m))

sink-of P = dual (source-of P)

⟦_⊥⟧ Log : Proto₀ → ★₀
⟦ P ⊥⟧   = ⟦ dual P ⟧
Log P    = ⟦ source-of P ⟧
```

Properties of dualization, sinks and sources: Dualization is its own inverse (`dual ∘ dual ≡ id`) and thus is an *equivalence* on protocols. This implies in particular that for two protocols `P` and `Q` if (`dual P ≡ dual Q`) then (`P ≡ Q`). Both `source-of` and `sink-of` are idempotent (`source-of ∘ source-of ≡ source-of`), where a source is a fix point for `source-of` and a sink is a fix point for `sink-of`. Finally, both `source-of` and `sink-of` are oblivious to dualization (`source-of ∘ dual ≡ source-of`).

Communication, how and why? Assuming we have two processes of matching protocols (`⟦ P ⟧` and `⟦ P ⊥⟧`), can we make them communicate and what do we get out of it? By an induction on the protocol we proceed by cases. If the protocol ends then both processes have to end. If the protocol is a `recv` then the first process is a function `p` and the second is a pair `(m , q)`, the first projection of the pair, `m`, which matches the argument for the function (`p`),

finally `p m` and `q` are valid processes ready to continue with protocol `P m`. If the protocol is a `send` its dual is a `recv` and this is the symmetric situation where the first is a pair and second is a function. In short, yes they can communicate all the way through until the end of the protocol. Moreover this is guaranteed to terminate as this is defined by induction on the protocol. What is the result though? We propose a solution which does not make a fixed choice on what is returned and does not incur a change to the notion of protocols. Communication can return a transcript of the communication using our type `Log`. The result of the communication is then a nested pair of messages all the way to the end of the protocol.

```
telecom : ∀ P → ⟦ P ⟧ → ⟦ P ⊥⟧ → Log P
telecom end      end   end = end
telecom (recv P) p (m , q) = m , telecom (P m) (p m) q
telecom (send P) (m , p) q = m , telecom (P m) p (q m)
```

Remark: Our design choices are inspired by linear logic as explained section 3. Communication between two processes can be seen as an application of the *cut* rule. The result of a cut usually is a $\bot$ (the one from linear logic, not to be confused with the empty type $\mathbb{O}$). However one cannot get much information out of a $\bot$ while our log can be used as an observable result of process communication, the choice of returning the transcript of the communication is departing from linear logic. This transcript is a key tool for reasoning about the processes in our system.

2.2 Dependent Protocols: Illustrative Examples

With all the basic blocks in place, we can now turn to a sequence of examples, each illustrating how to reuse standard dependent type techniques. Along the way, we will present abstractions and library parts which captures recurring patterns involving dependent protocols.

Indexed Types: Dependent protocols enables us to precisely index the type of messages depending on the value of previous messages. For example, consider a process which receives a natural number n and must send back a vector of size n containing natural numbers:

```
P₂ : Proto₀
P₂ = recv (λ (n : ℕ) → {- Receive n : ℕ -}
     send (λ (v : Vec ℕ n) →
     end)) {- Parentheses around λ are
     unnecessary when the scope extends completely
     to the right. We shall now omit them. -}
```

Sending Proofs: Indexed types can be used to make the type of messages more precise and thereby establish the properties they should have. For instance, by using singleton types we can refine our protocol `P₀` to make sure the process returns the product of the two received numbers. As a bonus the types of `m` and `n` are now inferred by their use in `_*_`.

```
P₀' : Proto₀
P₀' = recv λ m →                  {- m : ℕ -}
      recv λ n →                  {- n : ℕ -}
      send λ (r : S⟨ m * n ⟩) → {- r ≡ m * n -}
      end

t₀' : ⟦ P₀' ⟧
t₀' = λ m n → (S[ m * n ] , end)
```

Protocols can be so precise that the participants are uniquely defined from their type. This extreme situation can be used to specify the two sides of a communication as a single definition. The code for the participants can then be automatically extracted from these precise protocols. For instance the process `t₀'` is uniquely defined. The weakening from a precise protocol (such as `P₀'`) to a less precise protocol (such as `P₀`) can be expressed as a function from `⟦ P₀' ⟧` to `⟦ P₀ ⟧` which simply erases the proofs.

Phantom messages using shape-polymorphism: Making the protocols precise and detailed about what is communicated also has its drawback. For instance some data might have to remain secret. In the Diffie-Hellman key exchange protocol, for example, each of the two participants has a secret exponent (`x` and `y`) and they publicly exchange g^x and g^y (where `g` is the generator of the multiplicative group `G` of order `q`) they finally agree on a common secret $g^{x\,y}$ (as they can both raise the received message to the power of their secret exponent). We can model this key exchange directly using our protocols. The value $g^{x\,y}$ being agreed on is supposed to be kept secret and be used for the continuation of the protocol. As an example, we make it insecure by making one of the participants finish by sending $g^{x\,y}$.

```
Diffie-Hellman-Key-Exchange : Proto₀
Diffie-Hellman-Key-Exchange =
  send λ (gˣ   : G) →
  recv λ (gʸ   : G) →
  send λ (gˣʸ : G) →
  end
```

If we wish to be more precise about the third message $g^{x\,y}$ and assert it is equal to `g ^(x * y)` we would need a way to mention `x` and `y`. To do so we make use of the shape-polymorphism modality available in Agda (wrapped as a data type, we call it $\Box$). This modality enforces that no elimination can be done on values of this type outside of the modality itself. However, these values can still be passed around and be used in data type indices such as an equation or a singleton type. Using this modality, a precise version of the protocol can be given. We choose to send `x` and `y` using two extra messages of type $\Box\ \mathbb{Z}_q$, where $\mathbb{Z}_q$ is the additive group of integers modulo `q`. The rest is of the protocol is made precise through the use of singleton types. Finally to implement a process for this protocol we need the following property about our group structure: `exp-law : ∀ x y → g ^(x * y) ≡ (g ^ y) ^ x`.

```
Precise-Diffie-Hellman-Key-Exchange : Proto₀
Precise-Diffie-Hellman-Key-Exchange =
  send λ (x   : □ ℤq) →
  send λ (gˣ  : S⟨ g ^ x ⟩) →
  recv λ (y   : □ ℤq) →
  recv λ (gʸ  : S⟨ g ^ y ⟩) →
  send λ (gˣʸ : S⟨ g ^(x * y) ⟩) →
  end
```

Sending types: To send types and still comply with the predicativity requirements of our type theory, we give a universe polymorphic definition of the type `Proto`. With this, sending a type such as $\mathbb{N}$ will result in a protocol of type $\texttt{Proto}_1$ instead of $\texttt{Proto}_0$. The notions presented above, such as decoding, are actually universe-polymorphic. For instance, $[\![\_]\!]$ actually has type $\forall\ \{\ell\}\ (\texttt{P} : \texttt{Proto}_\_\ \ell) \to \bigstar_\_\ \ell$. Here is a contrived way of asking for a natural number `m` as a sequence of function applica-

tions (`s` and `z`), receiving a standard natural number `n` together with a proof that $\mathtt{s}^{\mathtt{m}}(\mathtt{z})$ equals `n`:

```
P₃ : Proto₁
P₃ = recv λ (A : ★₀ {- A is a type -}) →
     recv λ (z : A) →
     recv λ (s : A → A) →
     send λ (m : A) →
     recv λ (n : ℕ) →
     recv (ℕ-rec s z n ≡ m) λ p →
     end{- ℕ-rec is the recursor on ℕ -}

t₃ : ⟦ P₃ ⟧
t₃ A z s {- Receive A : ★₀, z : A, s : A → A -}
  = (s (s (s (s z))) , {- Sends s⁴(z) -}
     _ {- This can be omitted as it is
          uniquely defined from its type -})

t₃⊥ : ⟦ P₃ ⊥⟧
t₃⊥ = ℕ , zero , suc , λ m → m , refl , _
      {- Pairs are right associated.
         No parentheses needed here. -}
```

Offering and making a choice: Given two protocols `P` and `Q` it is common to combine them as a *choice*. We reuse here a standard notation used for linear logic and session types. A process $\mathtt{t}_0$ following the protocol `P ⊕ Q` is making a choice between protocols `P` and `Q` and will thus continue with one of them. Whereas a process $\mathtt{t}_1$ following the protocol `(dual P) & (dual Q)` is responding to a choice between protocols `dual P` and `dual Q`, and thus needs to be able to continue both. The processes $\mathtt{t}_0$ and $\mathtt{t}_1$ should both communicate safely. We support `_⊕_` and `_&_` without the need to extend our type for protocols. Indeed a choice is no more than a single bit of information, so offering a choice is receiving one bit and making a choice is sending one bit. The point is then to analyze this bit to define the rest of the protocol. To do so we use the elimination for $\mathbb{2}$. The standard interpretation of $\&$ as a product $\times$ and of $\oplus$ as a coproduct $\uplus$ is recovered through these two type isomorphisms: $\mathtt{A} \times \mathtt{B} \cong \Pi\, \mathbb{2}\, [\mathtt{0}\!: \mathtt{A}\, \mathtt{1}\!: \mathtt{B}\,]$ and $\mathtt{A} \uplus \mathtt{B} \cong \Sigma\, \mathbb{2}\, [\mathtt{0}\!: \mathtt{A}\, \mathtt{1}\!: \mathtt{B}\,]$. Here are the definitions for choices ($\&$ and $\oplus$), their units (`P⊤` and `P0`), and an example.

```
_&_ _⊕_ : (P Q : Proto₀) → Proto₀
P & Q = recv {2} [0: P  1: Q ]
P ⊕ Q = send {2} [0: P  1: Q ]

P⊤ P0 : Proto₀
P⊤ = recv′ 𝟘 end
P0 = send′ 𝟘 end

p₄ : ⟦ P₀' & P₃ ⟧
p₄ = [0: t₀'  1: t₃ ]

{- Two processes making different choice -}
p₄⊥₀  p₄⊥₁ : ⟦ dual P₀' ⊕ dual P₃ ⟧
p₄⊥₀ = 0₂ , t₀
p₄⊥₁ = 1₂ , 2 , 4 , λ r → _
```

The unit for $\&$ has to be equivalent to $\mathbb{1}$ (the unit for $\times$) and the unit for $\oplus$ has to be equivalent to $\mathbb{0}$ (the unit for $\uplus$). Another way of viewing this is that the actions `send` and `recv` are n-ary choices and the units are of arity zero. Therefore the unit for $\&$ is defined as a receive in the empty type. The protocol `P⊤` decodes to $\mathbb{0} \to \mathtt{End}$ which is equivalent to $\mathbb{1}$. The unit for $\oplus$ is defined as a send of a value from the empty type it decodes to $\mathbb{0} \times \mathtt{End}$ which is equivalent to $\mathbb{0}$. Π-types and Σ-types can therefore be used to recover the additive fragment of linear logic. Choices of any arity are available by choosing the message type to be of a particular size. The elimination of the message can then permit as many continuations for the protocol as there are choices available.

Protocols which can be aborted: It is sometime convenient to extend a protocol such that one side has the ability to abort the communication instead of being forced to continue sending messages. The solution is to rewrite all the `recv{M}` into `recv{M ⊎ Abort}` (where `Abort` is another custom unit type) which gives the process the ability to give up rather than perform a send action. The remainder of the protocol is extended such that if the message is tagged with `inl` the protocol goes as usual otherwise if it is tagged `inr` the protocol ends.

Sequencing and replicating protocols: Protocols should be composable and reusable. If `P` and `Q` are two protocols, the sequencing of them is expressible by induction on `P`. If we consider protocols as trees, then sequencing should amount to substituting the `end` by `Q` in `P`. However, using `send` and `recv` gives the ability to depend on the message. We therefore need a dependent version of sequencing which provides `Q` with all the messages from protocol `P`. We call this function `_>>=_` as it resembles monadic sequencing (although $\mathtt{Proto}_0$ is not even an endo-functor). Thereafter we define `_>>_` which forgets about the log from `P`.

```
_>>=_ : (P : Proto₀)(Q : Log P → Proto₀ ) → Proto₀
end    >>= Q = Q _
send P >>= Q = send λ m →
               P m >>= λ log → Q (m , log)
recv P >>= Q = recv λ m →
               P m >>= λ log → Q (m , log)

_>>_ : (P Q : Proto₀ ) → Proto₀
P >> Q = P >>= λ _ → Q
```

Sequencing behaves like a monad, with `end` acting as a unit and `_>>=_` is associative even though dependent types introduce a twist. For instance sequencing is needed to express the (dependent) concatenation of two logs. This concatenation is then needed to express associativity of sequencing. Thus, sequencing is associative and has `end` as a left and right unit.

Replicating a protocol `P` n times is achieved by recursively iterating the sequencing of `P` and finally using `end`.

```
replicate : ℕ → Proto₀ → Proto₀
replicate 0       P = end
replicate (suc n) P = P >> replicate n P
```

Clients and Servers: A typical form of protocol involves a sequence of query/response rounds between a client and a server. Our type $\mathtt{Proto}_0$ provides no support for (co)inductively defined protocols but we can still use a natural number to drive the number of rounds. We first describe one round for the client side. The server

round protocol is the dual of the client round. Finally we replicate the client round to get the client protocol using `replicate`.

```
module Client/Server (Query : ★₀)
                     (Resp : Query → ★₀) where
  ClientRound ServerRound : Proto₀
  ClientRound = send λ (q : Query)  →
                recv λ (r : Resp q) →
                end
  ServerRound = dual ClientRound

  Client Server : ℕ → Proto₀
  Client n = replicate n ClientRound
  Server = dual ∘ Client
```

Dependent types for fueling: Even though our protocols are limited to be of finite depth, dependent types can be used to push these limits further. For instance, a server could first wait to receive a number n, and then be a server for n rounds. Similarly, a server could itself choose the number of rounds:

```
  DynamicServer StaticServer : Proto₀
  DynamicServer = recv λ n →
                  Server n
  StaticServer  = send λ n →
                  Server n
```

Sending protocols and processes: Although this remains to be studied more thoroughly, a form of process mobility is already possible here. Indeed, for any protocol `P` of type $\texttt{Proto}_0$, the type `⟦ P ⟧` can be used as a message type. As an example, consider a "cloud computing provider" for a protocol `P`, which receives two matching processes and sends back the transcript:

```
module Sky (P : Proto₀) where
  Cloud : Proto₀
  Cloud = recv λ (p   : ⟦ P   ⟧) →
          recv λ (q   : ⟦ P ⊥ ⟧) →
          send λ (log : Log P)   →
          end

  cloud : ⟦ Cloud ⟧
  cloud p q = telecom P p q , end
```

The protocol and process above can be made more generic by receiving the protocol `P` as the first message. Predicativity causes a slight complication here as the resulting protocol is then a $\texttt{Proto}_1$. Moreover, the lack of cumulativity forces us to lift the protocol and types manually. Furthermore we need a process lifter which has the following type : $\forall\ \texttt{P} \to \llbracket\ \texttt{P}\ \rrbracket \to \llbracket\ \texttt{lift}_0\texttt{to}_1\ \texttt{P}\ \rrbracket$ where $\texttt{lift}_0\texttt{to}_1$ is the protocol lifter.

3. Concurrent protocols

In the examples shown so far, the communication, has been between two processes following some protocol `P`. From this, one might get the impression that we need to extend the type $\texttt{Proto}_0$ in order to communicate with several processes. This is not the case — we can already describe processes communicating with multiple processes by interleaving protocols.

The interleaved protocol between `P` and `Q` is `P ⅋ Q` (pronounced par) and is described below. Informally `P ⅋ Q` is a protocol for a process that, at each step, can decide to make progress on either `P` or `Q`. It is important to note that a process `⟦ P ⅋ Q ⟧` can do more than what one could do with two processes `⟦ P ⟧` and `⟦ Q ⟧`. This is best illustrated by an example let `P` be `recv′ 𝟘 end` and `Q` be its dual (namely `send′ 𝟘 end`). There exists no process of type `⟦ Q ⟧` since it would require us to find a value of an empty type. However there is a process of type `⟦ P ⅋ Q ⟧` which would first receive a value on the left and then send it back on the right. We define `⅋` as a binary operation on protocols, combining the two given protocols to yield one representing their interleaving.

```
_⅋_ : Proto₀ → Proto₀ → Proto₀
end      ⅋ Q        = Q
recv P   ⅋ Q        = recv λ m → P m ⅋ Q
P        ⅋ end      = P
P        ⅋ recv Q   = recv λ n → P ⅋ Q n
send{M}P ⅋ send{N}Q = send{M ⊎ N}
  [inl: (λ m → P m ⅋ send Q)
  ,inr: (λ n → send P ⅋ Q n) ]
```

The order of equations in our definition makes it left biased. This definition makes `end` a left and right unit. Moreover, if any of the protocols is a `recv P` the whole protocol is going to also start with the same `recv` action. The process has to receive the message at some point, so it might as well do it immediately. The interesting case is when both of the protocols are sending: here the process will have the choice to make progress in one of the two protocols first, and the rest of the protocol which made progress will still be interleaved with the protocol which did not. The decision is made by sending a message of type `M ⊎ N`, so to progress on the left by sending `m : M` the process can send `inl m` and then continue.

Tensors: A process `t` of the interleaved protocol `P ⅋ Q` is able to decide the order in which to send messages. In order to communicate with `t` one needs to be able to communicate through any interleaving. Such a process will follow a dual protocol, namely `dual P ⊗ dual Q` (pronounced tensor). The definition for `⊗` exchanges the sends and receives from the definition of `⅋`.

```
_⊗_ : Proto₀ → Proto₀ → Proto₀
end      ⊗ Q        = Q
send P   ⊗ Q        = send λ m → P m ⊗ Q
P        ⊗ end      = P
P        ⊗ send Q   = send λ n → P ⊗ Q n
recv{M}P ⊗ recv{N}Q = recv{M ⊎ N}
  [inl: (λ m → P m ⊗ recv Q)
  ,inr: (λ n → recv P ⊗ Q n) ]
```

Fixing the left bias: As already mentioned, our definitions for `⅋` and `⊗` are left biased. This means for instance that `P ⅋ end` is not equivalent to `P` by definition. From this result we establish the commutativity of `⅋` and `⊗`. To resolve this issue we show that the following equations hold:

```
P ⊗ end ≡ P  {- equal protocols -}
P ⅋ end ≡ P  {- equal protocols -}

{- equal process types -}
⟦ P ⊗ send Q ⟧ ≡ (Σ M λ m → ⟦ P ⊗ Q m ⟧)
⟦ P ⅋ recv Q ⟧ ≡ (Π M λ m → ⟦ P ⅋ Q m ⟧)
```

Working with tensor: Any two processes for protocols `P` and `Q` can be put in conjunction as a process for protocol `P ⊗ Q`. Such a process can then be taken apart into two separate processes again. However processes for tensor are not uniquely defined as a pair of processes, as there are more processes of tensor than there are pairs of processes. Intuitively our definition for tensor allows the process to observe the order in which processes are queried. Our

definitions therefore do not completely conflate & and ⊗ as they are co-provable but the process types are not equivalent. These operations have the following types and computation rules:

```
⊗-pair : ⟦ P ⟧ → ⟦ Q ⟧ → ⟦ P ⊗ Q ⟧
⊗-fst  : ⟦ P ⊗ Q ⟧ → ⟦ P ⟧
⊗-snd  : ⟦ P ⊗ Q ⟧ → ⟦ Q ⟧

⊗-β-fst : ∀ t u → ⊗-fst (⊗-pair t u) ≡ t
⊗-β-snd : ∀ t u → ⊗-snd (⊗-pair t u) ≡ u
```

3.1 Communication with a ⅋:

A process `s` following protocol `P ⅋ Q` can communicate with a process `t` following protocol `dual P` and another process `u` following protocol `dual Q`. However, the communication can be set up in two ways. One could use `⊗-pair t u` to build a compound process following `dual P ⊗ dual Q` which is equal to `dual (P ⅋ Q)` and thus we can use the function `telecom` to obtain a transcript for `P ⅋ Q`. A second way is to combine `s` with `t`, for instance, to get a compound process following `Q`. This compound process can then communicate with `u` to get a transcript for `Q`. This second way is actually hiding the details of the communication between `s` and `t`, and is implemented through a function we call `⅋-apply`. In figure 1 we show the full code for `⅋-apply` together with the two ways of setting up the communication.

Linear implication: `⅋-apply` can be given another type more appealing from a functional viewpoint. By introducing the standard notation for implication in linear logic `_⊸_`, the apply function has the following type:

```
_⊸_ : Proto₀ → Proto₀ → Proto₀
P ⊸ Q = dual P ⅋ Q

⊸-apply : ∀ P Q → ⟦ P ⊸ Q ⟧ → ⟦ P ⟧ → ⟦ Q ⟧
```

Composition: Given two processes `s : ⟦ P ⊸ Q ⟧` and `t : ⟦ Q ⊸ R ⟧` we can construct a composite process `t ∘ s : ⟦ P ⊸ R ⟧`. As with `⅋-apply`, the internal communication on the protocol `Q`, is now hidden. (We omit the code as it requires nine cases and therefore is not as readable as `telecom` or `⅋-apply`.) Notice, however, that by picking `P` to be `end` we gets `⊸-apply`.

```
_∘_ : ⟦ Q ⊸ R ⟧ → ⟦ P ⊸ Q ⟧ → ⟦ P ⊸ R ⟧
```

3.2 Building processes for ⅋:

The forwarder: For every protocol `P` it is possible to construct a process `fwd P : ⟦ P ⊸ P ⟧`. Recall that `P ⊸ P` is equal to `dual P ⅋ P`, and since either `P` or `dual P` is going to receive (unless both are `end`), the forwarder process can choose to receive from that side and forward this message to the other side. In linear logic this process corresponds to the axiom rule.

Absorption: Recall `P⊤`, the unit for & defined in section 2. It intuitively means that someone can send a value of type `𝟘`, which is impossible so it is easy to disregard this case. However, it so happens receiving a value of type `𝟘` concurrently to any other protocol is an equally impossible scenario. This principle is called *absorption*:

```
⊤-⅋ : ∀ P → ⟦ P⊤ ⅋ P ⟧
⊤-⅋ P = λ() {- elimination of the empty type -}
```

Flexible left/right sending: Our definition of `P ⅋ Q` requires that a process following this is only allowed to send when either both `P` and `Q` are send actions or one of them is a send action and the other is `end`. This can be inconvenient when building a process. With the following combinators, as soon as one side is a `send`, it becomes possible to send a message. The combinator `⅋-sendR` is defined by induction on the protocol `P` and thus can send the message when `P` `end`s, and send the message with `inr` when `P` is a `send` action. Finally, when `P` is a `recv` action, the message is received and passed to the transformed process `p`, which is then recursively transformed. The converse operation is obtained from commutativity of ⅋.

```
⅋-sendR : ∀ {M}P{Q : M → Proto}
            (m : M) → ⟦ P ⅋ Q m ⟧ → ⟦ P ⅋ send Q ⟧
⅋-sendR end      m p = m , p
⅋-sendR (send P) m p = inr m , p
⅋-sendR (recv P) m p = λ n → ⅋-sendR (P n) m (p n)

⅋-sendL : ∀ {M P Q} → (m : M) → ⟦ P m ⅋ Q ⟧
                    → ⟦ send P ⅋ Q ⟧
⅋-sendL {M} {P} Q m pq =
   coe (⅋-comm Q (send P))
       (⅋-sendR Q m (coe (⅋-comm (P m) Q) pq))
```

3.3 Properties from Linear Logic:

Equality on protocols and processes: In order to assess whether our definitions for ⊕, &, ⅋, and ⊗ are valid, we established the results expected from linear logic. First there is no proof of `P0` as it is equivalent to the empty AGDA type `𝟘`. Second `_⊕_` enjoys the disjunctive property as it is equivalent to `_⊎_`. Any linear logic formula (multiplicative and additive fragments) can be expressed as a protocol `P`. A process type `⟦ P ⟧` is then the type of all *closed* proofs of the formula `P`.

There are several levels at which an equation about protocols `P` and `Q` can hold. The strongest level is when we can establish an equality between the protocols themselves `P ≡ Q`. From such an equality it automatically holds that the process types are equal, (i.e `⟦ P ⟧ ≡ ⟦ Q ⟧`). From a logical perspective, equal formulae imply equal collections of proofs. The next level is when we can establish an equality between the types of processes `⟦ P ⟧ ≡ ⟦ Q ⟧`. This amounts to directly proving a equality on collections of proofs. The next level is when we can only establish the co-provability of the two formulas as `⟦ P ⟧ ↔ ⟦ Q ⟧` (where `A ↔ B` is `(A → B) × (B → A)`). The weakest level is when we can only establish an implication between process types: `⟦ P ⟧ → ⟦ Q ⟧`.

A note on Univalence: We make good use of the Univalence principle (Univalent Foundations Program 2013) available as a variant of the type theory used by AGDA. Univalence enables us to treat equivalent types (or isomorphic types) as equal. This principle has far-reaching consequences such as function extensionality. Using the Univalence principle greatly simplifies our proofs as most of these equivalences would otherwise be 4-5 times longer.

Additive fragment: The required properties are straightforward to establish for this fragment. We first establish the equivalences of ⊕, `P0`, &, `P⊤` with ⊎, `𝟘`, ×, and `𝟙` respectively. Using these equalities, the laws (associativity, commutativity, and units) follow from these equivalent types. However, we can use the Univalence principle on the equivalence which swaps 0_2 and 1_2 in the type $\mathbb{2}$. This equivalence can then be used to show stronger statements for commutativity, namely the protocols themselves are equal. We did

```
⅋-apply : ∀ P Q → ⟦ P ⅋ Q ⟧ → ⟦ dual P ⟧ → ⟦ Q ⟧
⅋-apply end      Q        s          end     = s
⅋-apply (recv P) Q        s          (m , p) = ⅋-apply (P m) Q (s m) p
⅋-apply (send P) end      s          p       = end
⅋-apply (send P) (recv Q) s          p       = λ n → ⅋-apply (send P) (Q n) (s n) p
⅋-apply (send P) (send Q) (inl m , s) p      = ⅋-apply (P m) (send Q) s (p m)
⅋-apply (send P) (send Q) (inr m , s) p      = m , ⅋-apply (send P) (Q m) s p

telecom₂-⊗-pair : ∀ P Q → ⟦ P ⅋ Q ⟧ → ⟦ P ⊥⟧ → ⟦ Q ⊥⟧ → Log (P ⅋ Q)
telecom₂-⊗-pair P Q pq p q = telecom (P ⅋ Q) pq (⊗-pair p q)

telecom₂-⅋-apply : ∀ P Q → ⟦ P ⅋ Q ⟧ → ⟦ P ⊥⟧ → ⟦ Q ⊥⟧ → Log Q
telecom₂-⅋-apply P Q pq p q = telecom Q (⅋-apply P Q pq p) q
```

Figure 1. Communication with a ⅋

not define & as the dual of ⊕ but as a separate definition. We must therefore establish that they are dual to each other.

```
{- Equivalences -}
⟦ P ⊕ Q ⟧ ≡ (⟦ P ⟧ ⊎ ⟦ Q ⟧)
⟦ P0    ⟧ ≡ 𝟘
⟦ P & Q ⟧ ≡ (⟦ P ⟧ × ⟦ Q ⟧)
⟦ P⊤    ⟧ ≡ 𝟙

{- Commutativity -}
P ⊕ Q ≡ Q ⊕ P
P & Q ≡ Q & P
{- Associativity -}
⟦ P ⊕ (Q ⊕ R) ⟧ ≡ ⟦ (P ⊕ Q) ⊕ R ⟧
⟦ P & (Q & R) ⟧ ≡ ⟦ (P & Q) & R ⟧
{- Units -}
⟦ P⊤ & P ⟧ ≡ ⟦ P ⟧
⟦ P0 ⊕ P ⟧ ≡ ⟦ P ⟧

{- Duality -}
dual (P ⊕ Q) ≡ dual P & dual Q
dual (P & Q) ≡ dual P ⊕ dual Q
```

The so-called *additive mix* happens to hold in our setting. This is likely to come from the lack of linearity in type theory:
`amix : ⟦ P & Q ⟧ → ⟦ P ⊕ Q ⟧`.

Multiplicative fragment: The main discrepancy between linear logic and our shallow embedding of it in type theory is to be found in the multiplicative units (`1` and ⊥). They both are played by our constructor `end`. (With our definitions, `end` is a unit for ⅋, ⊗, and also &.)

Recall that duality is not immediate as we are defining these connectives in terms of sends and receives. We therefore prove it. Notice these equalities are on protocols directly and can therefore apply more widely equalities on process types (using ⟦_⟧).

```
dual (P ⅋ Q) ≡ dual P ⊗ dual Q
dual (P ⊗ Q) ≡ dual P ⅋ dual Q
```

Unit laws between `end` and ⅋ or ⊗ hold by definition at the level of protocols. Associativity for ⅋ and ⊗ also hold at the level of protocols thanks to Univalence. Commutativity for ⅋ and ⊗ are established as equalities between the corresponding process types which is a strong statement on the identity between the different processes of a given protocol (or proofs of a given statement). As an example of a proof, the commutativity for ⅋ is given in figure 2.

```
{- Unit -}
end ⊗ P ≡ P {- by definition -}
end ⅋ P ≡ P {- by definition -}

{- Associativity -}
P ⊗ (Q ⊗ R) ≡ (P ⊗ Q) ⊗ R
P ⅋ (Q ⅋ R) ≡ (P ⅋ Q) ⅋ R

{- Commutativity -}
⟦ P ⊗ Q ⟧ ≡ ⟦ Q ⊗ P ⟧
⟦ P ⅋ Q ⟧ ≡ ⟦ Q ⅋ P ⟧
```

Distributivity of ⊗ over ⊕ and ⅋ over & holds directly at the level of protocols. It suffices to unfold the definitions and to use function extensionality.

```
(Q ⊕ R) ⊗ P ≡ (Q ⊗ P) ⊕ (R ⊗ P)
(Q & R) ⅋ P ≡ (Q ⅋ P) & (R ⅋ P)
```

As for the additive fragment, our multiplicative connectives supports the *mix rule*, which states that one can put two independently-defined processes in parallel. The strategy used for interleaving them is arbitrary; in our case it favors the left. This situation with additive and multiplicative mix together with a common unit for the multiplicative fragment also appear in Abramsky and Jagadeesan model for linear logic (Abramsky and Jagadeesan 1994).

```
mmix : ⟦ P ⊗ Q ⟧ → ⟦ P ⅋ Q ⟧
```

3.4 Examples from cryptography

One method for proving security used by cryptographers is to use *semantic security*. Security notions are represented as probabilistic games which usually involve guessing a secret random bit. The security is then established by showing that no "efficient" adversary is better than a random guess of the secret. Each game is defined as the protocol that an adversary follows. Through these interactions, the adversary is supposed to figure out a secret bit `b`, and by the end of the protocol the adversary will send their guess for the value of `b`.

The first game we describe is the Decisional Diffie-Hellman (DDH) game, which is the underlying hardness problem for Diffie-

```
⅋-comm : ∀ P Q → ⟦ P ⅋ Q ⟧ ≡ ⟦ Q ⅋ P ⟧
⅋-comm end      Q        = ! ap ⟦_⟧ (⅋-endR Q)
⅋-comm (recv P) Q        = (Π=′ _ λ m → ⅋-comm (P m) Q) • ! ⅋-recvR Q P
⅋-comm (send P) end      = refl
⅋-comm (send P) (recv Q) = Π=′ _ λ m → ⅋-comm (send P) (Q m)
⅋-comm (send P) (send Q) = Σ≃ ⊎-comm-equiv [inl: (λ m → ⅋-comm (P m) (send Q))
                                           ,inr: (λ m → ⅋-comm (send P) (Q m)) ]
```

Figure 2. Commutativity of ⅋

Hellman key exchange. An adversary for this game is a process of the following protocol:

```
DDH : Proto₀
DDH = recv λ (gˣ : G) →
      recv λ (gʸ : G) →
      recv λ (gᶻ : G) →
      send λ (b  : 𝟚) →
      end
```

The adversary receives three random values: g^x, g^y and g^z of the group `G`. The question to the adversary is whether `z` was random in which case they should send 0_2 as the guess or whether `z` $\equiv$ `x * y`, in which case they should guess 1_2. The communication is happening with a process that is traditionally called the *challenger*. Here, the challenger is represented by a function which takes a bit and three random values of the type $\mathbb{Z}q$.

```
DDH-challenger : 𝟚 → ℤq → ℤq → ℤq → ⟦ DDH ⊥⟧
DDH-challenger 0₂ x y z = (x , (y , (z     , _)))
DDH-challenger 1₂ x y _ = (x , (y , (x * y , _)))
```

If the communication between `DDH-chal b x y z`, for random `x`, `y` and `z`, ends with the adversary sending `b` then the adversary wins. An adversary has an advantage if the distance between pure guessing, i.e probability one half, and the probability of winning is non-negligible.

ElGamal is an public key encryption system that is used to encrypt messages in a group `G`, as in Diffie-Hellman key exchange. The secret key will be a number `x : ℤq` and the public key is `g ^ x`. The encryption function will also take a random number `y : ℤq`, and a message `m : G` and will output a pair of elements in the group `G`. Decryption is the left inverse of encryption and relies on the same law about exponentials as we saw in section 2 for Diffie-Hellman key exchange.

```
exp-law : ∀ x y → (g ^ x) ^ y ≡ (g ^ y) ^ x

enc : (pk : G)(rₑ : ℤq) → G → G × G
enc gˣ y m = (g ^ y , gˣ ^ y · m)

dec : (sk : ℤq) → G × G → G
dec x (gʸ , m) = m / (gʸ ^ x)
```

One of the games used to prove security for encryption schemes is IND-CPA, which stands for "ciphertext indistinguishability". In this game the adversary receives the public key and gets to pick two messages of his liking. One of these messages will be encrypted and sent back to the adversary, who must then guess if this is an encryption of the first or the second message. Once again the adversary is a process of the following protocol:

```
IND-CPA : Proto₀
IND-CPA = recv λ (publickey  : G) →
          send λ (messages   : G × G) →
          recv λ (ciphertext : G × G) →
          send λ (b? : 𝟚) →
          end
```

Similar to the Decisional Diffie Hellman game, we will have a challenger that will complete the communication with an adversary. This challenger takes as input the secret bit `b`, the secret key `x` and a random value `y` to use in the encryption.

```
IND-CPA-challenger : 𝟚 → ℤq → ℤq → ⟦ IND-CPA ⊥⟧
IND-CPA-challenger b x y =
  g ^ x , λ ms →
  enc (g ^ x) y (case b 0: fst ms 1: snd ms) ,  _
```

One can now establish that IND-CPA security for ElGamal can be reduced to the security of DDH. A proof of this fact will transform any adversary `A` in the IND-CPA game into one in the DDH game, such that if `A` had an advantage in IND-CPA, then the transformed adversary will have an advantage in the DDH game. Seen another way, if we can find a bound for the advantage of adversaries in DDH, we can find a bound for the IND-CPA. Since the semantic security proof is out of scope for this paper, we will only focus on the transformation part and leave out the proof about advantages.

The transformation will be a process `sim` following the protocol `IND-CPA ⊸ DDH`, since, given `A : ⟦ IND-CPA ⟧`, we can have a process `apply sim A : ⟦ DDH ⟧`. Here `sim` will pretend to be a challenger for `A` and we call these kinds of processes *simulators*. A complete proof will proceed by game hopping and will actually use two simulators. One of the simulators will simulate the case when `b` $\equiv 0_2$ and the other for the case `b` $\equiv 1_2$. Below we give a simulator that will simulate the case when `b` $\equiv 0_2$:

```
sim₀ : ⟦ IND-CPA ⊸ DDH ⟧
sim₀ gˣ gʸ gᶻ = ⅋-sendL gˣ
                (λ m₀m₁ →
                ⅋-sendL (gʸ , gᶻ · fst m₀m₁ )
                (λ b → ⅋-sendR b _))
```

4. Multiparty protocols

In the previous section we described how a process could communicate with several other processes concurrently. However, each of the communications is following a protocol involving only two participants. When multiple participants have to follow a common protocol a global view is more suitable. This has been studied as

multiparty session types also known as *global types* or *choreographies* (Honda et al. 2008).

For two party communications, it is enough to define a single protocol for one participant, since the protocol for the other participant is obtained by dualization. When multiple participants follow a common protocol, we cannot use this technique. We therefore define a type analogous to the type `Proto` but which instead describes "who sends to who", such as "A sends m of type M to B". (This approach is also reminiscent of the notation used in security protocols, also known as *Alice-Bob notation*.) We name these global types $\texttt{MProto}_0$ for multiparty protocols. The participants will be represented by an index type `I`. For multiparty protocols we expose two methods of communication: a participant `A : I` can either `send` a message `m : M` to participant `B : I`, or `broadcast` the message.

```
data MProto₀ (I : ★) : ★₁ where
  send : (A B : I){M : ★}(P : □ M → MProto₀ I)
       → MProto₀ I
  broadcast : (A : I){M : ★}(P : M → MProto₀ I)
            → MProto₀ I
  end : MProto₀ I
```

When a participant `broadcast`s a message, everyone else receives it, and therefore the proceeding $\texttt{MProto}_0$ may strongly depend on the message. This situation corresponds to the `send` case of the type `Proto`, since there is only one other participant. However, if the constructor `send` is used to describe that `A` (Alice) sends a message to `B` (Bob), then a third participant `C` (Charlie) will only receive a phantom message, and therefore cannot depend on the value of this message. Our protocols are sequential (or synchronous), so Charlie does know that some message has been sent, but not its content. These phantom messages use shape-polymorphism (the $\Box$ modality) to describe the rest of the $\texttt{MProto}_0$. This ensures that any participant will still be able to follow the protocol: Everyone knows when a message is being sent, but only the recipient learns the content. For instance, Charlie might learn that Alice sent a message `n : ℕ` to Bob, but for Charlie this is of type `n : □ ℕ`. Later on Charlie might receive a message `m : S⟨ n / 2 ⟩`, which would tell him that `n` is either equal to `2 * n` or `1 + 2 * n` but nothing more.

Projecting local views: We can move to a more local perspective where we describe for each participant which `Proto` they must follow. The function `P / ϕ` will map any multiparty protocol $\texttt{P} : \texttt{MProto}_0\ \texttt{I}$ to a protocol of type `Proto`. A participant `A : I` should follow the protocol `P / ϕ` if `ϕ A` is true (i.e. it returns $1_2 : \mathbb{2}$). Our projection function `_/_` is thus defined not only for single participants, but for any decidable subset of the type `I` of participants.

```
_/_ : (P : MProto₀ I)(ϕ : I → 𝟚) → Proto
broadcast A {M} P / ϕ =
  case (ϕ A) 0: recv (λ (m : M) → P m / ϕ)
             1: send (λ (m : M) → P m / ϕ)
send A B {M} P / ϕ = case (ϕ A)
    0: (case (ϕ B)
          0: recv (λ (m : □ M) → P m / ϕ)
          1: recv (λ (m :   M) → P m / ϕ))
    1: send (λ (m : M) → P m / ϕ)
end / ϕ = end
```

Projection in details: The definition above proceeds as follows. If the multiparty protocol is broadcast from `A`, we proceed by cases on `ϕ A`. If `ϕ A ≡ 0₂`, it means that `A` is not part of the subset `ϕ` and thus participants in `ϕ` are not sending this messages, they can receive it as it is broadcast. If `ϕ A ≡ 1₂` then `A` is part of `ϕ` and thus the group `ϕ` has to send. If the multiparty protocol is a send from `A` to `B`, we first proceed by cases on `ϕ A`. If `ϕ A ≡ 0₂`, then `ϕ` does not contain the sender thus we proceed by cases on `ϕ B`. If `ϕ A ≡ 0₂` and `ϕ B ≡ 0₂`, then `ϕ` does not contain the receiver and thus the group only "receives" under the type $\Box$, which allow them to follow the protocol. If `ϕ A ≡ 0₂` and `ϕ B ≡ 1₂`, then `ϕ` contains the receiver and thus receives as a group. If `ϕ A ≡ 1₂`, then `ϕ` contains the sender and thus the group has to send. Finally if the protocol ends any group has to end as well. Notice that if Alice has to send a message to herself she has to send it. This ensure that Alice is not indefinitely waiting for herself.

Special projections: As a special case, we pick `ϕ = λ _ → 1₂`, the group with containing all the participants. The projection `P / λ _ → 1₂` then corresponds to a source, i.e this is a protocol which just sends. This projection being a source, any process implementing it acts as a process sending the transcript (or the log) of the communication. Furthermore if we pick `ϕ = λ _ → 0₂`, the group with no participants, then the projection `P / λ _ → 0₂` is a sink. It can be considered as an observer of the messages being sent. Similar to how the result of a communication with `Proto`, is a process of type `Log P`, the result of a multiparty communication is a process of type `⟦ P / λ _ → 1₂ ⟧`.

Merging disjoints groups: One possible way to get the complete transcript of all messages sent is to define for each participant a process. Any participant $\texttt{A}_i$ follows the protocol `P / λ i → i == A`$_i$. A group of participants `ϕ` can then be merged to appear as single "bigger" process representing the whole group. Groups can be merged until all participants have been accounted for. We define a function `merge` which merges two groups `ϕ` and `ψ`. The `merge` of two groups is only defined when the two group `ϕ` and `ψ` are disjoint. To account for this requirement, an extra predicate `Disjoint ϕ ψ` is used to guarantee that `ϕ` and `ψ` are never true (i.e. equal to 1_2) at the same values. The Boolean disjunction `_∨_` can be pointwise-lifted to predicates $(\texttt{I} \to \mathbb{2})$ as follows: `(p ∨° q) i = p i ∨ q i`. The definition of `merge` follows the structure of `_/_` and has to account for 11 different cases. There is only one case for `end`. For `broadcast`, there are three cases `ϕ` could send, `ψ` could send, and `ϕ` together with `ψ` could receive. For `send` there is 7 different cases. Without the disjointedness condition, we would have situations where the two groups are both claiming to send. This situation cannot be resolved by picking an arbitrary message as one of the group would not be well-typed anymore. The definition of `merge` can also be seen as a proof that disjoint subsets of participants can always communicate.

```
merge : (P : MProto₀ I) → Disjoint ϕ ψ
      → ⟦ P / ϕ ⟧ → ⟦ P / ψ ⟧ → ⟦ P / ϕ ∨° ψ ⟧
```

By iterating the `merge` until we have covered all participants, we end up with a transcript of the communication. Therefore, as long as the type for participants is finite, it is possible to iteratively merge all the participants starting from the empty set. The transcript can be seen as a certificate of the communication. As this iterative merge process is exhaustively defined and terminating, it is also a proof of deadlock-freedom and termination.

Properties of merging: Our merging operation has the empty group of participants (`ϕ = λ _ → 0₂`) as a unit. Merging is also commutative and associative. Finally, once the group is complete (`ϕ = λ → 1₂`), it can only be further merged with the empty group, moreover the process is only sending and thus acts as a transcript. While quite straightforward and intuitively simple, the number of cases quickly grows. Thus the formal AGDA proofs of these facts remain to be completed.

4.1 Examples

The Two-Buyer-Protocol: a standard example used to describe multiparty protocols is the `two-buyer-protocol`. In this protocol there is a `Seller` and two buyers ($\texttt{Buyer}_1$ and $\texttt{Buyer}_2$). The two buyers want to agree on a price to collectively buy a book from the seller. The first step is $\texttt{Buyer}_1$ sending the title of the book they want to purchase. In the second step, the `Seller` informs the buyers of the price of the book; we model this communication as a broadcast. In the third step, $\texttt{Buyer}_1$ sends to $\texttt{Buyer}_2$ what share of the price he wants to pay. The type of the share expresses that it is lower than the price. In the fourth step, $\texttt{Buyer}_2$ decides whether or not to buy the book and broadcast the choice (`ok` or `quit`). If $\texttt{Buyer}_2$ decides to pay they must now send to the seller the address for delivery, to which the seller responds with a delivery date. If $\texttt{Buyer}_2$ refuses to pay, using the `quit` message, then the protocol ends.

```
data Response : ★ where
  ok quit : Response

data Participant : ★ where
  Buyer₁ Buyer₂ Seller : Participant

{------------ The protocol -----------------}
two-buyer-protocol : MProto₀ Participant
two-buyer-protocol =
  send Buyer₁ Seller λ (title : String) →
  broadcast Seller   λ (price : Price) →
  send Buyer₁ Buyer₂ λ (share :
                        Σ Price (λ n → n ≤ price)) →
  broadcast Buyer₂   λ {
     ok → send Buyer₂ Seller λ (address  : String) →
          send Seller Buyer₂ λ (delivery : Date) →
          end
   ; quit → end {- Buyer₂ refuses to pay -} }

{---------------- Buyer₁ -------------------}
is-buyer₁ : Participant → 𝟚
is-buyer₁ Buyer₁ = 1₂
is-buyer₁ _      = 0₂

buyer₁ : ⟦ MProto₀ Participant / is-buyer₁ ⟧
buyer₁ = "Homotopy Type Theory" , λ price →
  (price /2 , half-is-lower-lemma price)
  , λ { ok   → λ address delivery → end
      ; quit → end }

{---------------- Buyer₂ -------------------}
is-buyer₂ : Participant → 𝟚
is-buyer₂ Buyer₂ = 1₂
is-buyer₂ _      = 0₂

buyer₂ : ⟦ MProto₀ Participant / is-buyer₂ ⟧
buyer₂ title price (share , share≤price) =
  case share == (price /2)
    0: (quit , end)
    {- share is half the price -}
    1: (ok , "Somewhere nice" , λ delivery → end)

{----------------- Seller --------------------}
is-seller : Participant → 𝟚
is-seller Seller = 1₂
is-seller _      = 0₂

seller : ⟦ MProto₀ Participant / is-seller ⟧
seller title = 42 , λ price share-share≤price →
  λ { ok → λ address → "2014/03/10" , end
    ; end → end }

module RunExample where
  P      = two-buyer-protocol
  disj₀  = ...
  dist₁  = ...
  buyers = merge P disj₀ buyer₁ buyer₂
  all    = merge P disj₁ seller
  transcript-ok :
    all ≡ ("Homotopy Type Theory" , 42 ,
           (21 , half-is-lower 42) , ok ,
           "Somewhere nice" , "2014/03/10" ,
           end)
  transcript-ok = refl {- by definition -}
```

Multiparty Diffie-Hellman: Another example is the multiparty generalization of Diffie-Hellman key exchange. In this example we will have three participants, represented by the type `Participant`.

```
data Participant : ★ where
  A B C : Participant
```

The secret value this time is g^{xyz}. Participants are ordered, `A`, then `B`, then `C`. In order for the participants to be able to compute this secret, they first send their own public value to the next participant: g^x, g^y and g^z. After this first round, participant `A` can compute g^{xy} and g^{xz}, but not g^{yz}; similar restrictions apply to `B` and `C`. Therefore, to complete the protocol each participant sends to next participant the part he or she is missing. In order to make the protocol more precise, we add an initial round where the participants broadcast a $\Box$ version of their own secret. This enables us to precisely type all the forthcoming exchanges.

```
3-way-Diffie-Hellman : MProto₀ Participant
3-way-Diffie-Hellman =
  broadcast A λ (x : □ ℤq) →
  broadcast B λ (y : □ ℤq) →
  broadcast C λ (z : □ ℤq) →
  send A B  λ (gˣ  : S⟨ g ^ x ⟩) →
  send B C  λ (gʸ  : S⟨ g ^ y ⟩) →
  send C A  λ (gᶻ  : S⟨ g ^ z ⟩) →
  send A B  λ (gˣᶻ : S⟨ gᶻ ^ x ⟩) →
  send B C  λ (gˣʸ : S⟨ gˣ ^ y ⟩) →
  send C A  λ (gʸᶻ : S⟨ gʸ ^ z ⟩) →
  end
```

5. Related work and future work

Linear logic: The connectives $\oplus$, $\&$, $⅋$ and $\otimes$ we have shown in this paper have been inspired from linear logic (Girard 1987). The idea that linear logic and communication are connected goes back to the early 90s (Abramsky 1993; Bellin and Scott 1994). In recent years this connection has been extended between π-calculus session types and linear logic (Caires and Pfenning 2010; Wadler 2012). Dependent types have been used before in order to have more typesafe communication (Toninho et al. 2011), in which

later messages could depend on previous messages sent. But the dependency is only for messages, the protocol is still fixed and cannot evolve depending on the messages being sent.

The connectives in this paper correspond more closely to a linear logic with mix rules, one for the multiplicative fragment and one for the additive. One way of phrasing the multiplicative mix rule is the term `mmix : ⟦ P ⊗ Q ⟧ → ⟦ P ⅋ Q ⟧`, which was shown in section 3, and which picks an interleaving and acts as either of the processes it received. Similarly, the additive fragment can pick one side to continue on, as shown by the term `amix : ⟦ P & Q ⟧ → ⟦ P ⊕ Q ⟧`. Moreover, the units for the multiplicative fragment are conflated. Game semantics models for linear logic such as (Abramsky and Jagadeesan 1994) features similar peculiarities.

Our work does not give any account to the exponentials of linear logic. What is the corresponding protocols? One idea would be that weakening corresponds to protocols in which one can abort the communication, which were shown in section 2. How does this relate with earlier treatments (Caires and Pfenning 2010; Wadler 2012) as server replication and client requests?

Game semantics: One of the models of linear logic is to interpret formulas as games and valid derivations as strategy from a game semantics perspective (Blass 1992; Abramsky and Jagadeesan 1994). It might be worthwhile to study what kind of games are expressible using our protocols. As seen in section 3, the `end` protocol acts as a unit for both `_⅋_` and `_⊗_`, whereas from a linear logic perspective one might expect two different ones. By instead having two `end`, one representing winning and one representing losing the game, this discrepancy might disappear.

Session types: Session types were introduced to guarantee deadlock freedom for communication between two participants (Honda 1993). This work was later extended to work with multiple participants. The global types described in section 4 share similarity with global types in multiparty asynchronous session types (Honda et al. 2008). One difference is that in multiparty session types, the remainder of the protocol for a participant can only depend on a messages received by that participant. In our setting they are phantom messages models using the `□` modality, and use of `broadcast`.

It is customary in session types to include fixpoints in the protocols; this is something we have not investigated so far. We were interested in seeing how far we could go just with dependent versions of `send` and `recv`, and we believe that fixpoints are an orthogonal question that should be studied in the future.

6. Conclusion

We have presented a theory of communication described through the use of dependent types. The use of dependent types allows us to have an elegant, small yet expressive definition of protocols. Given two processes which can communicate we can produce a transcript of all the messages being exchanged throughout the communication. The type for both participants and the transcript are all functionally derived from the protocol. Using dependent types we are able to combine protocols in a similar manner as the connectives from linear logic. In particular, the connective `_⅋_` made it possible to communicate on several protocols concurrently. This is achieved even if the protocols themselves describe the view of a single participant.

Furthermore we have shown a version of global types for describing multiparty communication. These global types can then be projected out to a protocol for any subset of the participants. In a multiparty setting the dependencies are limited as everyone needs to be synchronized. We retain expressive dependent protocols through the use of phantom messages (shape-polymorphism, `□ A`) and the possibility to broadcast a message. Everything in this paper has been formalized in AGDA.

Acknowledgments: We would like to thank Nicolas Guenot, Taus Brock-Nannestad, Andreas Abel, Alec Faithfull for the interesting discussions, helpful comments, and suggestions. The authors were supported in part by grant 10-092309 from the Danish Council for Strategic Research, Programme Commission on Strategic Growth Technologies.

References

S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(12):3 – 57, 1993. ISSN 0304-3975.

S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. In *Journal of Symbolic Logic*, volume 59, pages 543–574, 1994.

G. Bellin and P. Scott. On the -calculus and linear logic. *Theoretical Computer Science*, 135(1):11 – 65, 1994. ISSN 0304-3975.

A. Blass. A game semantics for linear logic. *Ann. Pure Appl. Logic*, 56 (1-3):183–220, 1992.

L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In P. Gastin and F. Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15374-7. .

W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

D. Gustafsson and N. Pouillard. protocols, 2014. `https://github.com/crypto-agda/protocols`.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, pages 194–204, 1993.

K. Honda. Types for dyadic interaction. In *CONCUR*, pages 509–523, 1993.

K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. .

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

B. Toninho, L. Caires, and F. Pfenning. Dependent session types via intuitionistic linear type theory. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, PPDP '11, pages 161–172, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0776-5. .

T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Aug. 2013.

P. Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 273–286, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. .