Formalizing a Consensus Protocol using Dependent Session Types

Harlan Kringen, Zachary Sisco

December 12, 2020

1 Introduction

In applied cryptography research, consensus protocols serve a primary role, describing how multiple agents can communicate dynamically. However, consensus protocols tend to be communicated as pseudo-code algorithms. They are presented in a traditional pencil and paper style influenced heavily by set theory. Unfortunately this has a few negative consequences. Overall, it results in a large gap between the specification and implementation of the protocol. More specifically, it creates proofs that are hard to follow, especially for newcomers to the field, and additionally leads to errors at the implementation level. This paper investigates if type theory, a ubiquitous tool for describing program behavior, can aid in formally stating, verifying, and implementing consensus protocols.

We focus specifically on two flavors of type theory, session types and dependent types. Session types are used to describe concurrent processes. Practically, it gives types for sending and receiving data, notions of actors or processes, as well as channels connecting them. The modern incarnation has received a lot of attention under the banner "multiparty session types." [2]. The second extension is inspired by dependent type theory, an extremely general system built by adding to a simpler type theory the quantifiers from first-order logic, $\forall$ and $\exists$.

The existing research on such a type system is still very theoretical. While dependent types are well-represented in theorem provers like Coq and Agda, session types are much less so. To solve this, we focus exclusively on a prototype of this language, postponing an actual implementation until more work can be done. In this paper, we derive a type system based on multiparty session types with a small extension to allow dependent types in a particular component, namely message transfer. We further use our type system to give a description of the Dolev-Strong protocol and evaluate the advantages and disadvantages of writing code in this language. Finally, we suggest how this type system can lead to practical code, and consider impediments to making this approach more common in the applied cryptography and blockchain communities.

2 Background

Type theory itself is a foundational approach to computer science. Despite its abstraction, most programmers and researchers are in fact familiar with a simplified form of it. This

is because type theory underlies most statically typed languages, for instance Java and Haskell, and guards against errors like `2 + 'a'`. But whereas common types are `string` and `integer`, they could be arbitrarily complicated entities, such as "will never go below zero" or even "can be composed with a process on the left then terminate". We explain briefly the two major branches of type theory that we will use to express our prototype language, namely session types and dependent types.

2.1 Session Types

Session types describe concurrent communication between processes [2]. Session types capture the notion of a protocol by giving operations for sending and receiving data over channels, and internal and external choice. Because session types describe concurrent process, much of the focus is on properties like safety, termination, and deadlock freedom. Network protocols like HTTP and SMTP can naturally be expressed as session types.

Figure 1 gives a visualization of an example two-party session type. This example describes a "buyer-seller" interaction between a book seller and a customer. The visualization shows the back-and-forth interaction of each actor sending and receiving messages. To start, the buyer sends a book title to the seller. Upon receiving a title, the seller sends the buyer a quote. The buyer then has a choice to accept the quote—sending their address to the seller—or quit the interaction. The syntax for the types of the buyer and seller are at the bottom of Figure 1. The terms $!x$ and $?x$ stand for sending or receiving some data x, respectively. Note that the session types for the buyer and seller are duals of each other. That is, whenever the buyer sends something (!String), the seller must receive it (?String).

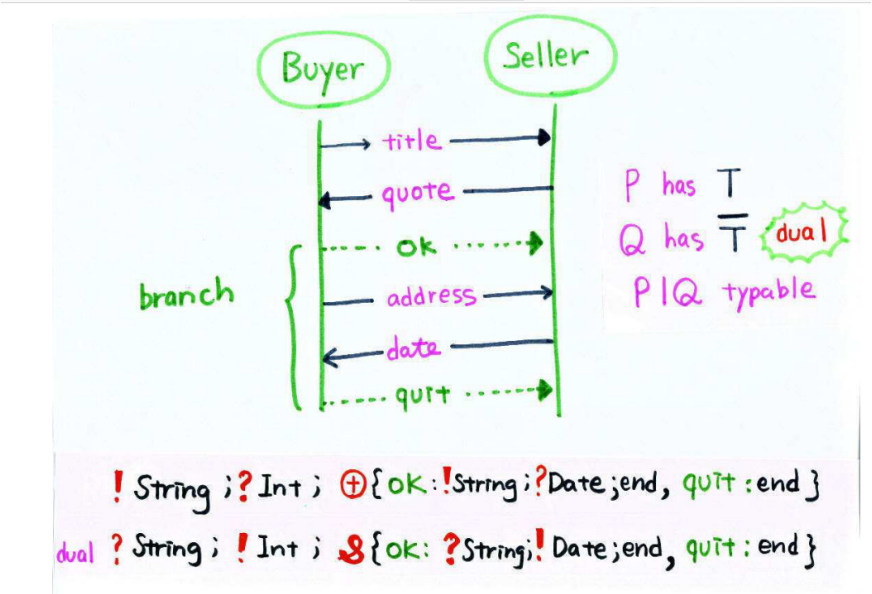

Figure 1: Binary session types, buyer-seller example [8].

Given the similarities between session types and protocols, we choose session types as the basis of our type system for describing consensus. However, alone we found session

types are not expressive enough to represent more detailed reasoning about the data being sent over channels. To address this, we augment multiparty session types with dependent types.

2.2 Dependent Types

Dependent types include the quantifiers from first-order logic $\forall$ and $\exists$. In the field of type theory, they are recognized as very expressive and general. Algorithmically, adding quantifiers allows us to create types where the consequent *depends* on the antecedent. This is best seen in an example:

$$\forall x \in \mathbb{Z}, x < 5$$

The above is the type of numbers that are integers (the consequent) that *depend* on the antecedent, a specific integer x, being less than 5. For our purposes, allowing our types to depend on certain terms allows us to keep track of where the messages came from, and whether or not they have specific qualities, such as length being less than a certain constraint. In fact, much of the Dolev-Strong protocol trades on these sorts of dependencies between senders and receivers. Actions like, "given Player 1 received message from Player 2, check length, proceed to send," are standard, and without dependent types, would be very difficult to describe. Most likely, they would force the creation of internal state and bookkeeping.

Given the high degree of generality of dependent types, it is possible to describe sessions with dependent types alone [1]. To this end, it would be interesting to look into ways of constraining the expressiveness, or alternatively, being clearer about how session types fit into the more abstract setting. Our approach is a first pass at combining the two concepts and we should look at minimizing the differences in the future.

3 A Prototype Language

3.1 DuSTy

We derived inspiration for our language DuSTy (Dependent Session Types) from [1,4,5,7]. It is worth noting only [7] combines both dependent and session types. However, we primarily used [2] as the basis for DuSTy, borrowing the same structure of a *global* type syntax that projects onto a *local* type syntax. In this organization, the local types are most similar to the language's type system, while the global types work like macros which are converted to local types at compile time. The global types offer the ability to describe the protocol from a top-down perspective, including all information absolutely needed. The global type is then specialized per actor to the type of that actor. Code that conforms to these local types will then be formally verified. We changed very little about the local type syntax—only adding Π and Σ types, universal and existential quantification, respectively—and reproduce it in Appendix A Figure 7. The local type syntax must reflect a term language, which is also more or less unchanged from [2]. We reproduce this term language in Appendix B.

From [2] we make a small addition as seen in Figure 2 by adding dependent message transfer (*bind*). This builds on the existing message transfer mechanism by allowing the

type of the transfer to depend on the message itself, e.g. where it originates, its length, etc.

$$
\begin{array}{lll}
\langle G \rangle ::= & p \to p' : k\langle U \rangle.G' & \text{message transfer} \\
\mid & p \to p' : k\{l_j : G_j\} \forall j \in J & \text{external choice} \\
\mid & p \to p' : k.bind(\lambda t : S.G)G' & \text{dependent transfer} \\
\mid & G, G' & \text{parallel} \\
\mid & t & \text{variable} \\
\mid & ite\ B\ G_1\ G_2 & \text{internal choice} \\
\mid & end & \text{end}
\end{array}
$$

$$\langle U \rangle ::= S \mid T@p$$

$$\langle S \rangle ::= \mathsf{bool} \mid \mathsf{nat} \mid \mathsf{Vec} \mid \ldots \mid \langle G \rangle$$

Figure 2: Global type syntax.

Adding infrastructure to the global type syntax requires specifying what can be represented down in the local type syntax. This is done by means of a projection function. The projection function takes a global type and an actor (or process) and determines what the global type looks like from that actor's perspective. To ensure we could make use of dependent types, we created the *bind* construct at the global level and project it onto Π and Σ at the local level. This requires adding the dependent type constructors Π and Σ to the local language used in [2]. However, this addition is fairly standard and we follow [6]. We give the projection function in Figure 3.

$$
\begin{array}{l}
(p_1 \to p_2 : k.bind(\lambda t.G')) \setminus p ::= \\
\qquad k.(\Pi(\lambda.(G' \setminus p)) \qquad \text{if } p = p_1 \neq p_2 \\
\qquad k.(\Sigma(\lambda.(G' \setminus p)) \qquad \text{if } p = p_2 \neq p_1 \\
\qquad (G' \setminus p) \qquad \text{if } p \neq p_2 \text{ and } p \neq p_1
\end{array}
$$

$$
\begin{array}{l}
(ite\ b\ g_1\ g_2) \setminus p ::= \\
\qquad (g_1 \setminus p) \qquad \text{if } b = \mathsf{true} \\
\qquad (g_2 \setminus p) \qquad \text{if } b = \mathsf{false}
\end{array}
$$

Figure 3: Global-to-local projection function.

To build the Dolev-Strong protocol we need a few extra operators, such as the *repeat* function, which runs a global protocol for some finite number of times. We show this function in Equation 1.

$$repeat(n : \mathsf{Int}, g : \mathsf{G}) ::= ite\ (n > 0)\ (g.repeat(n - 1, g))\ (end) \tag{1}$$

- **Round** 0: Sender sends $\langle b \rangle_1$ to every node.
- **For each round** $r = 1$ **to** $f + 1$: For every message $\langle \bar{b} \rangle_{1,j_1,j_2,\ldots,j_{r-1}}$ node i receives with r signatures from distinct nodes including the sender:
  - If $\bar{b} \notin \mathsf{extr}_i$: add $\bar{b}$ to extr_i and send $\langle b \rangle_{1,j_1,\ldots,j_{r-1},i}$ to every node.
- **At the end of round** $f + 1$: If $|\mathsf{extr}_i| = 1$: node i outputs extr_i; else node i outputs $\perp$.

Figure 4: The conventional definition of the Dolev-Strong protocol [3].

3.2 The Dolev-Strong Protocol

With our expanded type language, we are able to express the Dolev-Strong Protocol.

3.2.1 Classical Formulation

To review, the conventional formulation of Dolev-Strong is given in Figure 4. The Dolev-Strong protocol proceeds by round, where in the initial round, a designated sender broadcasts a message to all nodes in the network. In the following rounds, whenever a node receives a message it checks if the message is valid (inspecting the number of distinct signatures on the message chain), adds the bit to their own "extracted set", and then signs the message themselves and rebroadcasts it. After the last round, all nodes output the bit in their final message (if there is only one unique element in their extracted set), otherwise they output a default value $\perp$—indicating an indeterminate result. Figure 4 reproduces the Dolev-Strong Protocol as presented in [3].

3.2.2 DuSTy Version

Writing up the Dolev-Strong Protocol in DuSTy results in a very long type. For the sake of presentation, we present only the most interesting component in Figure 5. This corresponds to the second step in the classical formulation, and captures the fact that each actor receives a message, and to forward it along, must check that the message has a specified length and does not express more than one unique bit. We produce an expanded version of the protocol in Appendix C.

In Figure 5, the type wraps the core step of Dolev-Strong in a *repeat* over k, where k is the number of rounds. The proceeding line presents an interaction between $\mathsf{General}_0$ and $\mathsf{General}_1$ as a dependent message transfer. The *bind* term introduces a message m received by $\mathsf{General}_1$ and the conditional operator *ite* evaluates the condition checking whether m is valid. If it is, $\mathsf{General}_1$ signs and rebroadcasts the message. This is enforced at the type level by asserting that the length of the message sent must be 1 longer than the original received message.

We can see the use of *bind* in the global type. This will project down as a Π in the case of a dependent *send* and dually as a Σ in the case of a dependent *receive*. Because these are dependent types, they both allow their type to depend on terms in the language. In this case, the type of an actor sending a message depends on the qualities expressed under

$$
\begin{aligned}
&\mathsf{Dolev\text{-}Strong} ::= \\
&\mathsf{DS\text{-}Input}\,. \\
&\mathit{repeat}(k, \\
&\mathsf{General}_0 \to \mathsf{General}_1 : \mathsf{chan}_{0,1}.\mathit{bind}(\lambda m : \mathsf{Msg}. \\
&\quad \mathit{ite}\ (\mathit{len}(m) < k\ \wedge\ \mathit{size}(\mathit{set}(m)) > 1) \\
&\qquad \mathsf{General}_1 \to \mathsf{General}_2 : \mathsf{chan}_{1,2}\langle \mathit{msg}(\mathit{len}(m)+1)\rangle, \\
&\qquad\quad \mathsf{General}_1 \to \mathsf{General}_3 : \mathsf{chan}_{1,3}\langle \mathit{msg}(\mathit{len}(m)+1)\rangle, \\
&\qquad\quad \mathsf{General}_1 \to \mathsf{General}_0 : \mathsf{chan}_{1,0}\langle \mathit{msg}(\mathit{len}(m)+1)\rangle \\
&\qquad \bot \\
&\ldots . \mathit{end})
\end{aligned}
$$

Figure 5: This portion of the global type represents the second step in the Dolev-Strong Protocol.

$$
\begin{aligned}
&\mathsf{Dolev\text{-}Strong} \setminus \mathsf{General}_0 = \\
&\Sigma(\mathit{msg}(1), \mathsf{chan}_{0,1}).\Sigma(\mathit{msg}(1), \mathsf{chan}_{0,2}).\Sigma(\mathit{msg}(1), \mathsf{chan}_{0,3}). \\
&\Pi(\mathit{msg}(2), \mathsf{chan}_{0,1}).\Pi(\mathit{msg}(2), \mathsf{chan}_{0,2}).\Pi(\mathit{msg}(2), \mathsf{chan}_{0,3}). \\
&\Sigma(\mathit{msg}(3), \mathsf{chan}_{0,1}).\Sigma(\mathit{msg}(3), \mathsf{chan}_{0,2}).\Sigma(\mathit{msg}(3), \mathsf{chan}_{0,3}). \\
&\ldots \\
&\Pi(\mathit{msg}(j), \mathsf{chan}_{0,1}).\Pi(\mathit{msg}(j), \mathsf{chan}_{0,2}).\Pi(\mathit{msg}(j), \mathsf{chan}_{0,3}). \\
&\Sigma(\mathit{msg}(j+1), \mathsf{chan}_{0,1}).\Sigma(\mathit{msg}(j+1), \mathsf{chan}_{0,2}).\Sigma(\mathit{msg}(j+1), \mathsf{chan}_{0,3}) \\
&\ldots
\end{aligned}
$$

Figure 6: This demonstrates projecting the global Dolev-Strong type onto an actor.

the λ term. We use ellipses as a notational device to avoid writing every case for every actor. Figure 6 shows the projection of Dolev-Strong on $\mathsf{General}_0$.

3.3 Discussion

With the fully written Dolev-Strong protocol we are in a position to evaluate the utility of DuSTy. Writing down the specific type of the protocol yields a number of insights about the structure of consensus protocols.

In the first place, illustrating consensus as a type makes the network structure apparent. The types precisely capture who is talking to who and under what conditions. Encoding this at the type level provides the sender/receiver duality required of session types, and should guarantee deadlock freedom. Overall this enables easier reasoning about network synchronicity, as we can observe the beginning and end of the round structure more easily.

Importantly, however, we do not have enough information to reason about data flowing

through the protocol itself. To show larger proofs about consensus agreement and validity requires more involved reasoning about messages and *their* behavior. This makes it more difficult to determine the agreed upon value after running the protocol. We think addressing this could come from a finer-grained approach to representing message information at the type level, or representing the protocol type itself as an inductive object, which would make it more amenable to traditional functional programming techniques.

We also note that representing dishonest, colluding actors is non-trivial. While our current type allows implementing code where actors can perform dishonest actions, such as not sending messages, covering all possible colluding cases is difficult at the type level without deeper introspection into the messages carried over channels.

Finally, we note that writing the protocol as a type exposes some assumptions built into its presentation as a proof. For instance, the actors at the end of the protocol must reveal their chosen bits. However, in the context of a program, one must ask where this happens. Do the actors broadcast this information, or send it to a trusted third party? This effectively brings up the idea of a protocol as having input/output ports. Data must still make its way into a safe protocol and then make its way out. Writing protocols as types then reifies this notion and forces protocol designers to be explicit about the dataflow itself.

We think types are a good fit to capture the dynamic nature of protocols in applied cryptography. This is a fairly new area, however, and much work needs to be done to decide on a reasonable foundation for such a type system, as well as substantial engineering work to build a real-world language for implementing and testing these ideas. In the future we would be very interested in building some of these ideas in a theorem prover.

4 Conclusion

We presented a language with dependent multiparty session types to formally specify consensus protocols. We demonstrated our type system by showing how to express the Dolev-Strong protocol as a global type, projecting its local types on different actors. This is a novel approach to specifying consensus protocols that minimizes the gap between the specification and the implementation.

Our type system succeeds at capturing the "network topology" of a consensus protocol—that is, who communicates, and in what order. It does not, however, keep track of the messages sent through the system. This makes it difficult to reason about desired properties like validity and agreement, and is one drawback of a type-driven approach to specifying consensus protocols. To prove claims about these properties requires expressing a consensus protocol as some kind of inductive type—although it is not immediately clear how this would proceed.

Overall, types appear to be a promising candidate for protocols. However, as we found during this work it takes a sophisticated type system to just express the topology of the network. Further, general-purpose programming languages lack support for either dependent or session types. Some implementations exist as academic software, but only one combines both dependent and session types in the way we have [7]. In general, there needs to be more focus on moving session and dependent types into usable general-purpose programming languages. Going forward we would be like to move this work off paper and

into a mechanical theorem prover like Agda or Coq and pursue efficient code generation for consensus protocols.

References

[1] Daniel Gustafsson and Nicolas Pouillard. Dependent protocols for communication. Preprint at `https://nicolaspouillard.fr/publis/dep-proto.pdf`. Last accessed December 11, 2020.

[2] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1), March 2016.

[3] Elaine Shi. *Foundations of Distributed Consensus and Blockchains*. Preliminary draft, 2020.

[4] Peter Thiemann and Vasco T. Vasconcelos. Label-dependent session types. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[5] Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In *PPDP'11 - Proceedings of the 2011 Symposium on Principles and Practices of Declarative Programming*, pages 161–172, 01 2011.

[6] David Walker. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*. The MIT Press, 12 2004.

[7] Hanwen Wu and Hongwei Xi. Dependent session types. *CoRR*, abs/1704.07004, 2017.

[8] Nobuko Yoshida. Multiparty asynchronous session types. Accessed: 2020–11-01, 2015.

Appendix A Local Type Syntax

$$
\begin{array}{rl}
\langle T \rangle ::= & k \oplus \{l_i : T_i\} \forall i \in I \\
\mid & k \& \{l_i : T_i\} \forall i \in I \\
\mid & t \\
\mid & end \\
\mid & \Pi(t : T, k).T' \\
\mid & \Sigma(t : T, k).T'
\end{array}
$$

Figure 7: Local type syntax.

Appendix B Term Syntax

Appendix C Full Dolev-Strong Protocol

$\langle P \rangle ::= a[2..n](s).P$	multicast session request
$\mid a[p](s).P$	session acceptance
$\mid s!\langle e \rangle;\ P$	value sending
$\mid s?(x);\ P$	value reception
$\mid s!\langle\langle s \rangle\rangle;\ P$	session delegation
$\mid s?((s));\ P$	session reception
$\mid s \lhd l;\ P$	label selection
$\mid s \rhd l_i : P_{ii \in I}$	label branching
$\mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional branch
$\mid P \parallel Q$	parallel composition
$\mid \mathbf{0}$	inaction
$\mid (\nu n)P$	hiding
$\mid \mathsf{def}\ D\ \mathsf{in}\ P$	recursion
$\mid X\langle es \rangle$	process call
$\mid s : h$	message queue
$e ::= v \mid e \wedge e' \mid \neg e \ ...$	expressions
$v ::= a \mid \mathsf{true} \mid \mathsf{false}$	values
$h ::= l \mid v \mid s$	messages-in-transit
$D ::= \{X_i(x_i s_i) = P_i\}_{i \in I}$	declaration for recursion

Figure 8: Term syntax.

$$
\begin{aligned}
&\mathsf{DS\text{-}Input} ::= \\
&\qquad \mathsf{General}_0 \to \mathsf{General}_1 : \mathsf{chan}_{0,1}\langle \mathsf{Msg}\rangle. \\
&\qquad \mathsf{General}_0 \to \mathsf{General}_2 : \mathsf{chan}_{0,2}\langle \mathsf{Msg}\rangle. \\
&\qquad \mathsf{General}_0 \to \mathsf{General}_3 : \mathsf{chan}_{0,3}\langle \mathsf{Msg}\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Dolev\text{-}Strong} ::= \\
&\mathsf{DS\text{-}Input}\,. \\
&\mathit{repeat}(k, \\
&\mathsf{General}_0 \to \mathsf{General}_1 : \mathsf{chan}_{0,1}.\mathit{bind}(\lambda m : \mathsf{Msg}. \\
&\quad \mathit{ite}\ (\mathit{len}(m) < k\ \wedge\ \mathit{size}(\mathit{set}(m)) > 1) \\
&\qquad \mathsf{General}_1 \to \mathsf{General}_2 : \mathsf{chan}_{1,2}\langle \mathit{msg}(\mathit{len}(m)+1)\rangle, \\
&\qquad\quad \mathsf{General}_1 \to \mathsf{General}_3 : \mathsf{chan}_{1,3}\langle \mathit{msg}(\mathit{len}(m)+1)\rangle, \\
&\qquad\quad \mathsf{General}_1 \to \mathsf{General}_0 : \mathsf{chan}_{1,0}\langle \mathit{msg}(\mathit{len}(m)+1)\rangle \\
&\qquad \bot \\
&\mathsf{General}_0 \to \mathsf{General}_2 \ldots \\
&\ldots \\
&\mathsf{General}_1 \to \mathsf{General}_3 \ldots . \mathit{end})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{DS\text{-}Output} ::= \\
&\qquad \mathsf{General}_0 \to \mathsf{S} : \mathsf{chan}_{0,S}\langle \mathsf{Msg}\rangle. \\
&\qquad \mathsf{General}_1 \to \mathsf{S} : \mathsf{chan}_{1,S}\langle \mathsf{Msg}\rangle. \\
&\qquad \mathsf{General}_2 \to \mathsf{S} : \mathsf{chan}_{2,S}\langle \mathsf{Msg}\rangle. \\
&\qquad \mathsf{General}_3 \to \mathsf{S} : \mathsf{chan}_{3,S}\langle \mathsf{Msg}\rangle. \\
&\qquad \mathsf{S} \to \mathsf{General}_0 : \mathsf{chan}_{0,S}.\mathit{bind}(\lambda m : \mathsf{Msg}. \\
&\qquad\quad \mathit{ite}\ (\mathit{len}(\mathit{traverse}\ m) = 1) \\
&\qquad\qquad \mathsf{S} \to \mathsf{General}_0 : \mathsf{chan}_{0,S}\langle \mathit{first}(m)\rangle \\
&\qquad\qquad \bot
\end{aligned}
$$

Figure 9: The fully specified type for the Dolev-Strong protocol.