Value-Dependent Session Design in a Dependently Typed Language

Jan de Muijnck-Hughes Wim Vanderbauwhede
University of Glasgow
Jan.deMuijnck-Hughes@glasgow.ac.uk
Wim.Vanderbauwhede@glasgow.ac.uk

Edwin Brady
University of St Andrews
ecb10@st-andrews.ac.uk

Session Types offer a typing discipline that allows protocol specifications to be used during type-checking, ensuring that implementations adhere to a given specification. When looking to realise global session types in a dependently typed language care must be taken that values introduced in the description are used by roles that know about the value.

We present `Sessions`, a Resource Dependent Embedded Domain Specific Language (EDSL) for describing global session descriptions in the dependently typed language Idris. As we construct session descriptions the values parameterising the EDSLs' type keeps track of roles and messages they have encountered. We can use this knowledge to ensure that message values are only used by those who know the value. `Sessions` supports protocol descriptions that are computable, composable, higher-order, and value-dependent. We demonstrate `Sessions` expressiveness by describing the TCP Handshake, a multi-modal server providing echo and basic arithmetic operations, and a Higher-Order protocol that supports an authentication interaction step.

1 Introduction

Multi-Party Session Types (MPSTs) are a well-known typing discipline for describing sequences of message exchanges over communication channels [20]. Global session types present an overview of the interactions made between each component, and local session types describe the interactions respective to each participant. When realising MPSTs both global and local types are commonly presented either as bespoke Domain Specific Languages (DSLs) [39]; EDSLs that extend an existing programming language [24]; or a language extension [26].

MPSTs are, however, limited in their expressiveness. Session designers cannot specify value dependencies between sent messages, or reason about message contents. Recent work has extended the theory of MPST to incorporate value based reasoning on messages [37, 38, 5, 27].

Dependent types allow for more precise reasoning on programs by allowing types to depend on values. Existing work has shown how dependent types provide greater control over programs, and how they can be used to reason about communication in concurrent programs [12, 11, 10, 38] and communicating systems [19]. More generally, earlier work [8, 13] has shown how dependent types can provide resource-based type-level reasoning about programs.

Idris is a general purpose functional language with dependent types [7], Figure 1a presents a naïve implementation of a global session description as an EDSL within Idris. The type for expressions, `Session`, is indexed by a type describing participants (`role`) and type associated with expressions—`type`. Message passing is described using `Send` in which the sending and receiving roles are specified, together with the type of message being sent. Expressions are sequenced using a standard `Let` construct. This is possible by indexing `Session` by the type associated with individual expressions. The `Let` expression

F. Martins, D. Orchard (Eds.): Programming Language Approaches to Concurrency- & Communication-cEntric Software (PLACES 2019)
EPTCS 291, 2019, pp. 47–59, doi:10.4204/EPTCS.291.5

```
data Session : (type, role : Type)
            -> Type where
  Let  : (this : Session a role)
      -> (into : a -> Session b role)
      -> Session b role

  Send : (from, to : role)
      -> (type : Type)
      -> Session type role
  End  : Session () role
```

(a) Specification.

```
data Packet = Syn | SynAck | Ack

tcpHandshakeN : (alice,bob : role)
             -> Session () role
tcpHandshakeN alice bob = do
   Send alice bob (Packet, Nat)
   Send bob alice (Packet, Nat, Nat)
   Send alice bob (Packet, Nat, Nat)
   End
```

(b) Example

Figure 1: Naïve Realisation of an EDSL for describing Global Session Types.

also leads to Do-notation through implementation of Idris' monadic bind operator—`(>>=)`. Sessions descriptions are concluded using `End`.

Figure 1b demonstrates use of `Session` to describe the well-known TCP-handshake [33]. The roles of "Alice" and "Bob" describe the client and server, and `Packet` describes the TCP messages themselves. Each send operation describes how Alice and Bob send the packet and sequence numbers in a tuple. However, our session description does not describe the value dependencies between the sequence numbers, nor which constructor of `Packet` should be sent. Specifically, we should be able to provide guarantees that the sequence numbers are incremented by one when they are returned to their originators.

```
tcpHandshakeDep : (alice,bob : role) -> Session () role
tcpHandshakeDep alice bob = do
   (_,x) <- Send alice bob (Packet, Nat)
   (_,x',y) <- Send bob alice (Packet, (xplus1 ** xplus1 = S x), Nat)
   Send alice bob (Packet, (samex ** samex = x'), (yplus1 ** yplus1 = S y))
   End
```

(a) Value Dependent Specification

```
tcpHandshakeErr : (alice, bob, charlie : role) -> Session () role
tcpHandshakeErr alice bob charlie = do
   (_,x) <- Send alice bob (Packet, Nat)
   (_,x',y) <- Send bob alice (Packet, (xplus1 ** xplus1 = S x), Nat)
   Send charlie alice (Packet, (xplus1 ** xplus1 = S x), Nat)
   Send alice bob (Packet, (samex ** samex = x'), (yplus1 ** yplus1 = S y))
   End
```

(b) Errorful Value Dependent Specification

Figure 2: Global Session Types described using EDSL presented in Figure 1a.

The `Let` construct of `Session` allows binding of, and pattern matching on, message values to an identifier that can be reasoned about using standard Idris constructs. Figure 2a demonstrates how we can use these *bound* values to describe messages that rely on previously seen values. A dependent pair provides existential quantification that the type of Bob's response to Alice contains the sequence number `x` incremented by one. Standard equality types `(=)` ensure this relation is available at the type-level. Similar use of dependent pairs ensures that the same incremented sequence number is returned to Bob, together with their sequence number `y` also incremented by one. However, our EDSL does not reason about message origin, nor who is aware of these values. Figure 2b demonstrates how we can insert a third participant "Charlie" into the protocol who can send a message to Alice whose type depends on a value

that Charlie is not aware of. When specifying EDSLs for global session descriptions in a dependently typed language we must prevent roles from depending on values they are not aware of.

1.1 Contributions

Taking inspiration from Multi-Party Session Types we have been investigating how dependent types, as presented in Idris, can be used to design, implement, and reason about communicating systems. Existing work has presented a type-driven approach for designing and developing communicating concurrent systems [11]. A limitation in this work is not being able to specify value-dependencies on messages for multi-party protocols. This paper presents the initial outcomes of our research to address this limitation.

Resource-Dependent EDSLs are a common design pattern associated with EDSL construction in Idris [10, 9] derived from existing work on parameterised monads and Hoare monads [3, 6]. Such construction allows us to associate, and manage, an abstract state within the type of the EDSL itself. Using this construction we can index the type of our EDSL by a *knowledge index* that captures the messages seen by each participant. Using this index we ensure that value dependent messages are only sent by participants using values that the participant knows about.

Our main contributions are:

1. `Sessions`, a Resource-Dependent EDSL to specify global session descriptions. `Sessions` improves upon existing EDSL designs by introducing global session descriptions that are computable, composable, higher-order, and value-dependent.
2. Several example session descriptions that describe the TCP Handshake (Section 3.1), a multi-modal server providing echo and basic arithmetic operations (Section 3.2), and a higher-order session description that emulates an authentication interaction—Section 3.3.

`Sessions` has been realised using Idris, however, the construction techniques presented are agnostic to dependently typed languages. Other languages such as Agda [29] should be capable of realising this EDSL.

2 A Language for Describing Sessions

Figure 3 presents `Sessions`, our EDSL for specifying global session descriptions. Messages are represented using variables that have an associated abstract state that keeps track of roles that have seen the message. Central to the EDSLs operation is a parameterised monad that manages the set of variables and their abstract state: The *Knowledge Index*.

Knowledge Index Figure 4a presents the various type-level data structures that make the knowledge index. Roles within our EDSL are represented as an indexed data type tagged with a descriptive name. The type `Role` is indexed by the type associated with roles to ensure that all roles are from the same family. This representation of roles is a marked difference from the naïve implementation presented in Figure 1a. Tagging of roles with a string value provides role comparison and reuse of existing predicates for list quantification. Each message in a session description is represented by an indexed type `Var`, whose type is indexed by the type of the message being sent. During type checking Idris' elaborator allows us to distinguish between different instances of `Var` based on their names and type-level values. The knowledge index itself is a list of state items. Items within our knowledge index associate a message type, with a variable, and a list of roles that are aware of the message. Standard list quantifiers, such as

```
data Sessions : (type : Type)
             -> (roleType : Type)
             -> (participants : List (Role roleType))
             -> (old : List (Item roleType))
             -> (new : type -> List (Item roleType))
             -> Type where
  Let : (this : Sessions a roleType ps old ctxt_fn)
     -> (into : (value : a) -> Sessions b roleType ps (ctxt_fn value) ctxt_fn_new)
     -> Sessions b roleType ps old ctxt_fn_new

  NewMsg : (creator : Role roleType) -> (mType : Type) -> (prfInvolved : Elem creator ps)
        -> Sessions (Var (MSG mType)) roleType ps old
                    (\lbl => MkItem (MSG mType) lbl ([creator]) :: old)

  NewDepMsg : (creator : Role roleType) -> (dep : Var (MSG mType))
           -> (prfKnows : Any (KnowsData creator dep) old) -> (pred : mType -> Type)
           -> Sessions (Var (MSG (x ** pred x))) roleType ps old
                       (\lbl => MkItem (MSG (x ** pred x)) lbl ([creator]) :: old)

  Send : (sender, receiver : Role roleType) -> (msg : Var (MSG mType))
      -> (senderInvolved   : Elem sender ps) -> (receiverInvolved : Elem receiver ps)
      -> (prf : Any (KnowsData sender msg) old)
      -> Sessions () roleType ps old
                  (const $ update old prf (Learn receiver))

  Rec : Inf (Sessions () roleType ps Nil (const Nil))
      -> Sessions () roleType ps ctxt (const Nil)

  Call : Sessions () roleType ss Nil (const Nil) -> (prf : Overlapping ps ss)
      -> Sessions () roleType ps ctxt (const ctxt)

  End : Sessions () roleType ps old (const Nil)

  Read : (msg : Var (MSG mType)) -> (prf : Any (AllKnow ps msg) old)
      -> Sessions mType roleType ps old (const old)
```

Figure 3: Type definition for Sessions.

`Any` and `All`, help reason about the knowledge context itself, and such quantifiers allow construction of operations acting on knowledge index instances—see Figure 4b.

Parameterised Type `Sessions` is a resource dependent EDSL in which the program's abstract state is the knowledge index. The parameters of `Sessions` form a Hoare monad [3, 6] where the resulting type-level state is dependent on the expression value: `type`—the expression's return type; `roleType`—the underlying type for roles; `participants`—the set of roles involved in the protocol; `old`—the knowledge index in which the expression must operate; and `new`—a function that computes the resulting knowledge index that is dependent upon the result associated with the expression. A Let-binding allows us to sequence expressions, and bind message values to identifiers. Each expression describes how the knowledge index is affected by the result of the expression. When the result is not dependent on the expression value the function `const` (which drops its first parameter) allows one to state the new value. When sequencing expressions, together with result dependent changes, the value of the knowledge index will change dynamically as we step through a session description.

```
data Role : (type : Type) where
  MkRole : (tag : String) -> Role type

data Ty = MSG Type

data Var : (metaType : Ty) -> Type where
  MkVar : Var metatype

data Item : (type : Type) -> Type where
  MkItem : (msgtype : Ty)
        -> (label : Var msgtype)
        -> (value : List (Role type))
        -> Item type
```

(a) Definition.

```
update : (ctxt : List (Item roleType))
      -> (idx : Any p ctxt)
      -> (f : (item : Item roleType)
           -> (prf : p item)
           -> Item roleType)
      -> List (Item roleType)
update (x :: xs) (Here y) f = f x y :: xs
update (x :: xs) (There y) f =
        x :: update xs y f
```

(b) Operations.

Figure 4: Knowledge Index Definition and Helpers.

```
data KnowsData : (role : Role roleType)
              -> (msg : Var (MSG typeM))
              -> (item : Item roleType)
              -> Type where
  WhoKnows : Elem r rs
          -> KnowsData r l (MkItem (MSG ty) l rs)
```

(a) Predicate stating that a role knows the message.

```
Learn : Role roleType
     -> (i : Item roleType)
     -> (prf : KnowsData s l i)
     -> Item roleType
Learn r (MkItem (MSG ty) l rs) (WhoKnows prf) =
  MkItem (MSG ty) l (r::rs)
```

(b) Function to add role to abstract state.

```
data AllKnow : (roles : List (Role roleType))
            -> (msg : Var (MSG typeM))
            -> (item : Item roleType) -> Type where
  LastToKnow : (prfKnows : Elem r rs) -> AllKnow [r] l (MkItem (MSG ty) l rs)

  NextToKnow : Elem r rs
            -> AllKnow rs' lbl (MkItem (MSG ty) lbl rs)
            -> AllKnow (r::rs') lbl (MkItem (MSG ty) lbl rs)
```

(c) Predicate stating all the roles know the message.

Figure 5: Predicates reasoning on Items within the Knowlege Context, and operations that use these predicates.

Message Creation `NewMsg` and `NewDepMsg` are responsible for introducing message descriptions. The constructor `NewMsg` will introduce a message of type `mType` created by `creator` if `creator` was in the global set of participants `ps`. A list quantifier provides this guarantee. Creation of the message will extend the knowledge context with a new `Item` instance that populates the list of roles with `creator`. The constructor `NewDepMsg` will introduce a dependently typed message that depends on a previously seen value if said value is known to the message creator. This proof is provided by `prfKnows`, an instance of a list quantification that the predicate `KnowsData` holds for at least one element in the knowledge index. Figure 5 presents the definition for `KnowsData` that states the given role is an element in the list of roles associated with the given message. The parameter `pred` is a function that constructs the type of the message to be sent, with the actual returned message type being a dependent pair that details the value associated with `dep` that is passed to `pred`.

Sending Messages The expression `Send` allows one to specify the type of message that is to be sent and which roles are sending and receiving the message. As with all the other expressions evidence must be given to ensure expression correctness. We require that both the sender and receiver are elements of `ps`, and that the sender knows the message to be sent—cf. predicates associated with dependent message creation. If a send is successfully described the knowledge index will be updated to add `receiver` to the list of roles that know about `msg` using the function `update`.

Accessing Values By design `Sessions` forbids message descriptions to be bound to a value explicitly, as not all participants will know the value. However, if all participants in the protocol are aware of the message, then we can depend on the value. The expression `Read` facilitates this. The predicate `AllKnow` (presented in Figure 5) constructs an inductive proof that each role in the session has knowledge of the message. Section 3 presents an example of `Read` in action.

Recursion Not all sessions comprise of a linear sequence of actions, interactions between roles may repeat. `Rec` and `Call` allow for recursively calling an already described session, and calling an external session. These functions are restricted such that the knowledge of the called session must begin and end with an empty context i.e. no knowledge is learned about the supplied session. A further restriction is on the set of participants respective to the calling function. For `Rec` the caller and callee must have the same set, while a call to `Call` must have overlapping sets. Here `Overlapping` is a thinning [1] that ensures elements of `ps` appear in order within `ss`. A thinning allows for structures to be weakened respective to some decision procedure [15, 2].

```
send : (sender, receiver : Role roleType)
    -> (msg : Var (MSG mType))
    -> {auto prfSender : Elem sender ps}
    -> {auto prfReceiver : Elem receiver ps}
    -> {auto prf : Any (KnowsData sender msg) ctxt_old}
    -> Session () roleType ps ctxt_old (const $ update ctxt_old prf (Learn receiver))
send s r msg {prfSender} {prfReceiver} {prf} =
    Send s r msg prfSender prfReceiver prf
```

Figure 6: Example API for `Send`.

A Clean API Separately, we provide a high-level API for protocol designers to use when specifying protocols. Without this API the proofs required for each expression have to be explicitly presented. We can use Idris' auto implicit feature to automatically construct the proofs associated with an expression. Idris' compiler will attempt to search and combine values, found in the expressions context, that together satisfy the type of the presented predicate. If a suitable value cannot be constructed the expression will fail to type-check. Figure 6 demonstrates this approach for the `Send` expression. Further, we can provide a function (`Session`) to describe the expected initial and final state of the knowledge index together with the final expression type such that the only expression that will satisfy the end conditions would be the `End` expression.

3 Example Sessions

This section considers the expressiveness of our EDSL by considering three protocol descriptions. The first example presents the TCP Handshake that demonstrates how we can construct value dependent message descriptions. The second example is a multi-modal server that demonstrates how we can compose protocol descriptions together and make interaction decisions based of message values. The final example demonstrates how we can construct higher-order protocols.

3.1 TCP Handshake

```
tcpHandShakeDep : Session roleType [Alice, Bob]
tcpHandShakeDep = do
  m1 <- newMsg Alice (Packet, Nat)
  send Alice Bob m1
  m2 <- newDepMsg Bob m1 (\v => (Packet, Next (snd v), Nat))
  send Bob Alice m2
  m3 <- newDepMsg Alice m2 (\v => (Packet,
     (Literal $ fst $ snd $ (snd v)), (Next $ (snd $ snd $ (snd v)))))
  send Alice Bob m3
  end
```

Figure 7: A Dependently Typed Global Session Description for the TCP Handshake.

Figure 7 illustrates how the TCP handshake (presented earlier in Section 1) can be specified using `Sessions`. Central to operation of the handshake is the sending of two sequence numbers that have been correctly transformed. Within this example, Alice sends a non-dependently typed message (`m1`) to Bob that contains the initial packet and Alice's sequence number. In response, Bob constructs a value dependent message (`m2`) that depends on the content of `m1`. The anonymous function states that `m2` must have a type in which the second postion in the tuple is the sequence number from `m1` incremented by one. Here the type `Next` is a type synonym for a dependent pair stating the transformation on the sequence number from `m1`. The final message sent by Alice (`m3`) is dependent on `m2`. Alice must send a packet together with the incremented sequence number from `m2`, and the second sequence number incremented by one. Recall that the type of `m2` is a dependent pair in which the message type is in the second position and the dependend upon value is in the first. Therefore to access the underlying values we have to project into these pairs, and their contents, accordingly. Use of anonymous functions here does complicate the session description. Such complication can be resolved with named functions.

3.2 Multi-Modal Server

Figure 8 presents the *complete* specification of a global session description for interacting with a server that offers simple arithmetic calculations, and an echo service. Figure 8a presents the messages sent. Within this example: Alice represents the client; Bob the server; and Charlie a third-party for performing simple arithmetic. Figure 8d presents the top-level interactions between Alice and Bob. Alice sends a message of type `CMD` to Bob. Once Bob has received the message all participants specified in the top-level protocol have seen the value. This enables the `Read` expression to access the value, and allow case-splitting (analogous to offer and choice from MPST) to change behaviour based on the message's value. If the command was: `Math` then we call the Maths protocol and loop—Figure 8b; `Echo` then we call the Echo protocol and loop—Figure 8c; or `Quit` then we end the interaction.

```
data MathsCMD = Add Nat Nat | Sub Nat Nat | Div Nat Nat | Mul Nat Nat
data CMD = Maths | Echo | Quit
```

(a) Message Types and Values.

```
doMaths : Session roleType [Alice, Bob, Charlie]
doMaths = do
  m1 <- newMsg Bob (Literal "Time for Maths!")
  send Bob Alice m1
  m2 <- newMsg Alice MathsCMD
  send Alice Bob m2
  send Bob Charlie m2
  m3 <- newMsg Charlie Nat
  send Charlie Bob m3
  send Bob Alice m3
  end
```

(b) A Maths Protocol.

```
doEcho : Session roleType [Alice, Bob]
doEcho = do
  m1 <- newMsg Bob (Literal "Time to Echo!")
  send Bob Alice m1
  m2 <- newMsg Alice String
  send Alice Bob m2
  m3 <- newDepMsg Bob m2 (Literal)
  end
```

(c) An Echo Protocol.

```
myServer : Session roleType [Alice, Bob]
myServer = do
  m1 <- newMsg Alice CMD
  send Alice Bob m1
  m1val <- read m1
  case m1val of
    Maths => do {call doMaths; Rec myServer} Echo => do {call doEcho; Rec myServer}; Quit => end}
```

(d) The Server Protocol.

Figure 8: A Global Session Type for a Server.

The doEcho protocol is a non-recursive implementation of RFC862/RFC347 [34, 32] in which Bob repeats the message Alice sent, back to Alice. We ensure this repetition of values using a dependent message with a predicate to reason about the sent value. Here Literal is a type-synonym for a dependent pair with an equality predicate. Note the use of Literal to ensure that the welcome message is the literal string value given. The doMath protocol allows Alice to send Bob simple arithmetic expressions (MathsCMD), that Bob sends to Charlie. Bob can use Charlie to generate a response to send to Alice.

3.3 Higher-Order Protocols

```
HoppyServer : Session roleType ss
           -> {auto prf : Overlapping [Alice,Bob] ss}
           -> Session roleType [Alice,Bob]
HoppyServer body = do
  m1 <- newMsg Bob (Literal "Who are you!")
  send Bob Alice m1
  m2 <- newMsg Alice String
  send Alice Bob m2
  m3 <- newMsg Bob Bool
  send Bob Alice m3
  res <- read m3
  case res of { True => do {call body; end}; False => end}
```

Figure 9: A Higher-Order Protocol.

Figure 9 presents a *Higher-Order Protocol*, in which we treat session descriptions as *first class* con-

structs. Rather than explicitly calling a named description (cf. Figure 8d) we pass in descriptions as parameters. The predicate `Overlapping` ensures that the participants of `body` (the called description) overlap with those specified in `HoppyServer`. The description presented in Figure 9 presents a decision procedure reminiscent of an authentication procedure. Alice sends some message that is checked by Bob who responds with a decision, and the protocol's next steps are based on that decision. This is not secure but nonetheless demonstrates the potential power of `Sessions`. We can construct protocol descriptions that are composable and higher-order.

4 Discussion

`Sessions` is an EDSL for describing global session descriptions within a dependently typed host language. We introduce value dependent messages and reason about messages within a paramterised monad. Construction of `Sessions` as a EDSL allows session descriptions to be first class, composable, and computable. Overriding `Do` notation facilitates use of Idris' control structures to describe decisions. Here choice differs from branching/selection as traditionally seen in session type implementations. `Sessions` supports value based choice using pattern-matching on message values or constant values.

MPSTs can provide guarantees towards several protocol properties. Namely, session fidelity, communication safety, liveness, and progress [20, 21]. Many of these properties are for *complete systems*, `Sessions` describes global descriptions only. Existing work has shown a correct-by-construction approach to linking session descriptions to implementations within a dependently typed language [11]. Global descriptions are projected to compute the local type for the viewpoint of a protocol participant as a continuation. The type of the implementation is indexed by the continuation to ensure there is an intrinsic link between the specified global session description and its implementation. Thus providing *communication safety* and *session fidelity*. We are currently investigating how to build a similar framework that includes our value dependent session descriptions.

Further, Idris is a total language that checks for program termination and coverage of pattern matching clauses. Thus, our global session descriptions are checked for termination and coverage in the same way as regular Idris programs. We believe that implementing local types and implementations within a total language will help to provide guarantees towards *liveness* and *progress*.

5 Related Work

There are many implementations of Session Types available[1]. Generally, there are three approaches to implementation either: as a DSL; as an EDSL; or as a language extension.

DSL Scribble is a DSL for describing MPSTs [39]. The DSL is limited in describing non-value dependent message exchanges. There are various existing runtimes that generate code from these descriptions [28, 24, 14]. As `Sessions` is an EDSL the resulting specifications cannot not be directly resused by other projects[2]. However, the library can be extended to generate Scribble specifications directly from the `Session` specification. StMungo is a Java oriented tool for generating local types from Scribble descriptions [24].

[1] `http://simonjf.com/2016/05/28/session-type-implementations.html`

[2] Idris does, however, support multiple code generation targets, however, embedding compiled Idris code into other code projects is non-trivial.

Scribble-Refined Recent work [27] has extended Scribble specifically for F# to leverage the language's support for refinement types [17] and type providers [31]. The authors present an expressive refinement language to specify boolean refinements related to messages being sent. `Sessions` complements this work by showing an alternative realisation to reasoning about value messages.

EDSL EDSLs have been realised for: Haskell [36, 30, 35]; Go [25]; and Rust [22]. These approaches follow our approach, however, the host language chosen is not as expressive in describing global session description and thus do not allow for reasoning on message descriptions.

Language Extension Not all implementations follow the DSL approach. Others have extended existing programming languages to embedded MPST directly within the language. For example, Links and SIL [16, 26, 4]. However, these approaches require extending the language and compiler. `Sessions` presents an opportunity to develop the specification in the same language as the implementation without major changes required to the language itself.

MPST Theory Our work in using a dependently typed language to realise value dependent global session descriptions compliments existing theoretical work [38, 37, 5]. `Sessions` achieves value-dependent session descriptions by maintaining a *knowledge index*. This compliments existing work [37, 27] in which the authors define knowledge in much the same way. However, we have embedded the knowledge index directly within the type-system of the global session description ensuring that our EDSL instances are correct-by-construction with respect to value dependencies. A *Design-By-Contract* approach has been taken to extend MPST with message oriented assertions [5]. We remark that the assertions associated with messages are comparable to the dependent pair construct in which the message (first position) must satisfy the predicate in the second position. Our use of dependent pair's is different: the value in the first position presents evidence that the predicate (message) in the second position is valid.

6 Conclusion

Type systems in modern programming languages are expressive enough to support EDSLs describing session types. With the additional expressivity given by dependent types, we have shown that dependently typed languages such as Idris provide a natural setting to further enhance the expressiveness of EDSLs for describing global session descriptions.

`Sessions` demonstrates how we incorporate reasoning on value-dependencies between messages, and provide first-class global session descriptions. Idris' auto implicit mechanism has proven useful in presenting correct-by-construction guarantees towards protocol design. With this new setting for global session design we expect to be able to use Idris' type-system to verify additional correctness guarantees of our global descriptions. Especially properties required by *security* protocols [18, 23] where we also need to reason about the content of messages, and more importantly how messages are related.

`Sessions` is an EDSL for global session descriptions. We wish to compliment our EDSL with a complete system for protocol design, implementation, and verification such that we provide a self-contained system within a single language. Of importance will be how we can correctly project our descriptions to local types such that only correct local actions and knowledge are carried over.

References

[1] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride & James McKinna (2018): *A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs*. *Proc. ACM Program. Lang.* 2(ICFP), pp. 90:1–90:30, doi:10.1145/3236785.

[2] Thorsten Altenkirch, Martin Hofmann & Thomas Streicher (1995): *Categorical Reconstruction of a Reduction Free Normalization Proof*. In David H. Pitt, David E. Rydeheard & Peter T. Johnstone, editors: *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings*, *Lecture Notes in Computer Science* 953, Springer, pp. 182–199, doi:10.1007/3-540-60164-3_27.

[3] Robert Atkey (2009): *Parameterised notions of computation*. *Journal of Functional Programming* 19(3-4), pp. 335–376, doi:10.1017/S095679680900728X.

[4] Stephanie Balzer & Frank Pfenning (2015): *Objects as session-typed processes*. In Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci & Carlos Varela, editors: *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, ACM, pp. 13–24, doi:10.1145/2824815.2824817.

[5] Laura Bocchi, Kohei Honda, Emilio Tuosto & Nobuko Yoshida (2010): *A Theory of Design-by-Contract for Distributed Multiparty Interactions*. In Paul Gastin & François Laroussinie, editors: *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, *Lecture Notes in Computer Science* 6269, Springer, pp. 162–176, doi:10.1007/978-3-642-15375-4_12.

[6] Johannes Borgström, Juan Chen & Nikhil Swamy (2011): *Verifying stateful programs with substructural state and hoare types*. In Ranjit Jhala & Wouter Swierstra, editors: *Proceedings of the 5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011, Austin, TX, USA, January 29, 2011*, ACM, pp. 15–26, doi:10.1145/1929529.1929532.

[7] Edwin Brady (2013): *Idris, a general-purpose dependently typed programming language: Design and implementation*. *J. Funct. Program.* 23(5), pp. 552–593, doi:10.1017/S095679681300018X.

[8] Edwin Brady (2014): *Resource-Dependent Algebraic Effects*. In Jurriaan Hage & Jay McCarthy, editors: *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*, *Lecture Notes in Computer Science* 8843, Springer, pp. 18–33, doi:10.1007/978-3-319-14675-1_2.

[9] Edwin Brady (2016): *State Machines All The Way Down: An Architecture for Dependently Typed Applications*. Available at `https://eb.host.cs.st-andrews.ac.uk/drafts/states-all-the-way.pdf`. Unpublished Draft.

[10] Edwin Brady (2016): *Type-Driven Devlopment with Idris*, 1st edition. Manning. Available at `https://www.manning.com/books/type-driven-development-with-idris`.

[11] Edwin Brady (2017): *Type-driven Development of Concurrent Communicating Systems*. *Computer Science (AGH)* 18(3), doi:10.7494/csci.2017.18.3.1413.

[12] Edwin Brady & Kevin Hammond (2010): *Correct-by-Construction Concurrency: Using Dependent Types to Verify Implementations of Effectful Resource Usage Protocols*. *Fundam. Inform.* 102(2), pp. 145–176, doi:10.3233/FI-2010-303.

[13] Edwin Brady & Kevin Hammond (2012): *Resource-Safe Systems Programming with Embedded Domain Specific Languages*. In Claudio V. Russo & Neng-Fa Zhou, editors: *Practical Aspects of Declarative Languages - 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings*, *Lecture Notes in Computer Science* 7149, Springer, pp. 242–257, doi:10.1007/978-3-642-27694-1_18.

[14] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng & Nobuko Yoshida (2019): *Distributed Programming Using Role-parametric Session Types in Go: Statically-typed Endpoint APIs for*

Dynamically-instantiated Communication Structures. Proc. ACM Program. Lang. 3(POPL), pp. 29:1–29:30, doi:10.1145/3290342.

[15] James Maitland Chapman (2009): *Type checking and normalisation*. Ph.D. thesis, School of Computer Science, University of Nottingham. Available at `http://eprints.nottingham.ac.uk/10824/`.

[16] Simon Fowler, Sam Lindley, J. Garrett Morris & Sára Decova (2019): *Exceptional Asynchronous Session Types: Session Types Without Tiers*. Proc. ACM Program. Lang. 3(POPL), pp. 28:1–28:29, doi:10.1145/3290341.

[17] Timothy S. Freeman & Frank Pfenning (1991): *Refinement Types for ML*. In David S. Wise, editor: *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, ACM, pp. 268–277, doi:10.1145/113445.113468.

[18] Andrew D. Gordon & Alan Jeffrey (2003): *Authenticity by Typing for Security Protocols*. Journal of Computer Security 11(4), pp. 451–520, doi:10.3233/JCS-2003-11402. Available at `http://content.iospress.com/articles/journal-of-computer-security/jcs189`.

[19] Nicolas Guenot, Daniel Gustafsson & Nicola Pouillard (2015): *Dependent Communication in Type Theory*. Available at `https://nicolaspouillard.fr/publis/ptt1.pdf`.

[20] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. J. ACM 63(1), pp. 9:1–9:67, doi:10.1145/2827695.

[21] Raymond Hu & Nobuko Yoshida (2017): *Explicit Connection Actions in Multiparty Session Types*. In Marieke Huisman & Julia Rubin, editors: *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, Lecture Notes in Computer Science 10202, Springer, pp. 116–133, doi:10.1007/978-3-662-54494-5_7.

[22] Thomas Bracht Laumann Jespersen, Philip Munksgaard & Ken Friis Larsen (2015): *Session Types for Rust*. In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, ACM, New York, NY, USA, pp. 13–22, doi:10.1145/2808098.2808100.

[23] David Kaloper-Mersinjak, Hannes Mehnert, Anil Madhavapeddy & Peter Sewell (2015): *Not-Quite-So-Broken TLS: Lessons in Re-Engineering a Security Protocol Specification and Implementation*. In Jaeyeon Jung & Thorsten Holz, editors: *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, USENIX Association, pp. 223–238. Available at `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/kaloper-mersinjak`.

[24] Dimitrios Kouzapas, Ornela Dardha, Roly Perera & Simon J. Gay (2016): *Typechecking Protocols with Mungo and StMungo*. In: *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP '16, ACM, New York, NY, USA, pp. 146–159, doi:10.1145/2967973.2968595.

[25] Julien Lange, Nicholas Ng, Bernardo Toninho & Nobuko Yoshida (2017): *Fencing off go: liveness and safety for channel-based programming*. In Giuseppe Castagna & Andrew D. Gordon, editors: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, ACM, pp. 748–761, doi:10.1145/3009837. Available at `http://dl.acm.org/citation.cfm?id=3009847`.

[26] Sam Lindley & J Garrett Morris (2017): *Behavioural Types: from Theory to Tools*, chapter Lightweight Functional Session Types. River Publishers, doi:10.13052/rp-9788793519817.

[27] Rumyana Neykova, Raymond Hu, Nobuko Yoshida & Fahd Abdeljallal (2018): *A session type provider: compile-time API generation of distributed protocols with refinements in F#*. In Christophe Dubach & Jingling Xue, editors: *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, ACM, pp. 128–138, doi:10.1145/3178372.3179495.

[28] Nicholas Ng, Nobuko Yoshida & Kohei Honda (2012): *Multiparty Session C: Safe Parallel Programming with Message Optimisation*, pp. 202–218. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-30561-0_15.

[29] Ulf Norell (2009): *Dependently Typed Programming in Agda*. In: *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, ACM, New York, NY, USA, pp. 1–2, doi:10.1145/1481861.1481862.

[30] Dominic Orchard & Nobuko Yoshida (2016): *Effects As Sessions, Sessions As Effects*. *SIGPLAN Not.* 51(1), pp. 568–581, doi:10.1145/2914770.2837634.

[31] Tomas Petricek, Gustavo Guerra & Don Syme (2016): *Types from data: making structured data first-class citizens in F#*. In Chandra Krintz & Emery Berger, editors: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, ACM, pp. 477–490, doi:10.1145/2908080.2908115.

[32] J. Postel (1972): *Echo process*. RFC 347. Available at `http://www.ietf.org/rfc/rfc347.txt`.

[33] J. Postel (1981): *Transmission Control Protocol*. RFC 793 (INTERNET STANDARD). Available at `http://www.ietf.org/rfc/rfc793.txt`. Updated by RFCs 1122, 3168, 6093, 6528.

[34] J. Postel (1983): *Echo Protocol*. RFC 862 (INTERNET STANDARD). Available at `http://www.ietf.org/rfc/rfc862.txt`.

[35] Riccardo Pucella & Jesse A. Tov (2008): *Haskell Session Types with (Almost) No Class*. In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, ACM, New York, NY, USA, pp. 25–36, doi:10.1145/1411286.1411290.

[36] Matthew Sackman & Susan Eisenbach (2008): *Session Types in Haskell: Updating Message Passing for the 21st Century*. Technical Report, Imperial College London. Available at `http://pubs.doc.ic.ac.uk/session-types-in-haskell/`.

[37] Bernardo Toninho & Nobuko Yoshida (2016): *Certifying data in multiparty session types*. *Journal of Logical and Algebraic Methods in Programming*, doi:10.1016/j.jlamp.2016.11.005.

[38] Bernardo Toninho & Nobuko Yoshida (2018): *Depending on Session-Typed Processes*. In Christel Baier & Ugo Dal Lago, editors: *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, *Lecture Notes in Computer Science* 10803, Springer, pp. 128–145, doi:10.1007/978-3-319-89366-2_7.

[39] Nobuko Yoshida, Raymond Hu, Rumyana Neykova & Nicholas Ng (2014): *The Scribble Protocol Language*, pp. 22–41. Springer International Publishing, Cham, doi:10.1007/978-3-319-05119-2_3.