This is a peer-reviewed, accepted author manuscript of the following conference paper: Muijnck-Hughes, JD, Urlea, C, Voinea, L & Vanderbauwhede, W 2023, 'Capable: a mechanised imperative language with native multiparty session types', ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, Lisbon, Portugal, 22/10/23 - 27/10/23.

Capable: A Mechanised Imperative Language with Native Multiparty Session Types

Jan de Muijnck-Hughes Cristian Urlea Laura Voinea Wim Vanderbauwhede
University of Glasgow, Glasgow, UK
{Jan.deMuijnck-Hughes,Cristian.Urlea,Laura.Voinea,Wim.Vanderbauwhede}@glasgow.ac.uk

CAPABLE is lightweight mechanised imperative language that provides native support for Multiparty Session Types (MPSTs). Through mechanisation, we can explore and catalogue the changes required to extend similar languages with native support for MPSTs, as well as the interplay between the existing type-system and other novel extensions. Principally, our demo shows CAPABLE in action and what a language with native MPSTs can look like. We also look beneath the surface syntax and offer insight over how we created intrinsically typed sessions (and session types) within a dependently typed language. We show a compact well-scoped encoding of session types, mechanised proofs of soundness and completeness for projection, and how dependent types help with bidirectional type checking of typed sessions.

1 Introduction

Session types, first introduced in [6, 7, 12], are an important typing discipline to reason statically about the communication behaviours of our programs. While originally session types addressed communication between two parties (binary), they were later generalised to express communication between any number of participants (multiparty) [8]. When looking to incorporate Session Types (STs) within an existing programming language many expressive solutions tend towards code generation using a pre-existing toolchain [16] and integration using bespoke Embedded Domain Specific Languages (eDSLs) [5, 3]. Ultimately, these solutions are limited in their expressiveness by the host language being targeted, and how idioms from Multiparty Session Type (MPST) theory are translated. For instance, code generation *à la* Scribble [16] generates an API endpoint that separates code for communication from code that operates on values. What if, instead, we sought to *extend* a language and incorporate STs as native, and perhaps future first-class, code constructs?

CAPABLE is a lightweight imperative language that we have mechanised within Idris 2, to ensure that the language's design and implementation is *correct-by-construction*. Through the mechanised implementation we can now explore how novel language extensions can be incorporated correctly without affecting the language's type-safety. With a mechanised language implementation we can also document the changes required to the language's syntax and semantics when extending similar language's with novel extensions.

We are particularly interested in exploring how MPSTs help make interprocess communication (IPC) more reliable and type-driven, and what language changes are required. We have extended CAPABLE with MPSTs derived from an existing concise formalism [11], combined with communication context managers to realise typed communication. As such CAPABLE is the first mechanised language implementation in a dependently-typed language to support intrinsically *multiparty session-typed* terms.

Our demo provides an overview of CAPABLE, how ill-typed IPC can occur, and how extending the language with MPSTs enables well-typed IPC to occur.

Submitted to:
30 Years of Session Types (ST30)

2 A Tour of CAPABLE

We begin our demo by exploring CAPABLE and its support for session types. Specifically our tour of the language is structured as follows:

1. *Basics of (Imperative) Programming in* CAPABLE: provides an overview of the core language features, concentrating datatypes, functions, memory, and interactive editing with typed-holes;
2. *IPC Behaving Badly*: demonstrates how, through a lack of behavioural typing, we can specify correctly and incorrectly behaving clients for the well-known RFC 862 `Echo` program [10];
3. *IPC Behaving Correctly*: explores CAPABLE's support for session types by describing various global protocols, and realising specific endpoints for these protocols, we will also demonstrate how typed-holes act as a guide during endpoint development;

3 Mechanisation of CAPABLE

The second part of our demo will delve into the salient aspects behind the incorporation of Multiparty Session Types into the mechanisation of CAPABLE. Specifically we will highlight:

1. *Efficient Encoding of Session Types*: showcases the advantages of dependent types for efficient and compact representations of session types as a single well-scoped parameterised datatype;
2. *Proofs (of Projection) as Programs*: details how a mechanised proof of soundness and completeness of plain global type projection is used during elaboration to not only construct local types from global types, but to construct evidence that global types are well-formed;
3. *Intrinsically Typed Sessions*: will explore our modelling of intrinsically typed sessions, and how through the auspices of bi-directional typing we can design our typed-sessions to have inferable local types for better error reporting;

We provide an overview of the code that will walked-through during both pars of the demo, highlighting different aspects, in Appendix A.

4 Conclusion

In this demo we have showcased CAPABLE, the first intrinsically-typed imperative language implementation with native support for Multiparty Session Types. CAPABLE is available freely online:

`https://github.com/DSbD-AppControl/capable-lang`

Other programming languages with native session types include Links [9], SePi [4], Sill [14], C0 [15], FreeST [1] to name just a few. These languages, however, tend to focus on binary communication, and while backed by powerful type systems, they lack mechanisation. Closer to our work is Theimann's [13], who presents a *functional pearl* [13] for Agda in which he demonstrates how to intrinsically session-type functional programs with binary session types. While both approaches look at constructing intrinsically-session-typed languages they compliment each other. Theimann's work takes a more theoretically accurate term language in which there are typed-channels that are linearly used, while we take a simpler approach with a managed runtime but have multiparty sessions. Within Theimann's and our work the complete meta-theory of session-types has not been studied, such study is an obvious area of future investigation. Fortunately, the meta-theory of binary session-types has been *engineered* in Coq [2], and will help provide us with insight in our own future directions.

References

[1] Bernardo Almeida, Andreia Mordido, Peter Thiemann & Vasco T. Vasconcelos (2022): *Polymorphic lambda calculus with context-free session types*. *Inf. Comput.* 289(Part), p. 104948, doi:10.1016/j.ic.2022.104948. Available at https://doi.org/10.1016/j.ic.2022.104948.

[2] David Castro-Perez, Francisco Ferreira & Nobuko Yoshida (2020): *EMTST: Engineering the Meta-theory of Session Types*. In Armin Biere & David Parker, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, *Lecture Notes in Computer Science* 12079, Springer, pp. 278–285, doi:10.1007/978-3-030-45237-7_17. Available at https://doi.org/10.1007/978-3-030-45237-7_17.

[3] Simon Fowler (2020): *Model-View-Update-Communicate: Session Types Meet the Elm Architecture (Artifact)*. *Dagstuhl Artifacts Ser.* 6(2), pp. 13:1–13:2, doi:10.4230/DARTS.6.2.13. Available at https://doi.org/10.4230/DARTS.6.2.13.

[4] Juliana Franco & Vasco Thudichum Vasconcelos (2013): *A Concurrent Programming Language with Refined Session Types*. In Steve Counsell & Manuel Núñez, editors: *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers*, *Lecture Notes in Computer Science* 8368, Springer, pp. 15–28, doi:10.1007/978-3-319-05032-4_2. Available at https://doi.org/10.1007/978-3-319-05032-4_2.

[5] Paul Harvey, Simon Fowler, Ornela Dardha & Simon J. Gay (2021): *Multiparty Session Types for Safe Runtime Adaptation in an Actor Language (Artifact)*. *Dagstuhl Artifacts Ser.* 7(2), pp. 08:1–08:2, doi:10.4230/DARTS.7.2.8. Available at https://doi.org/10.4230/DARTS.7.2.8.

[6] Kohei Honda (1993): *Types for Dyadic Interaction*. In Eike Best, editor: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, *Lecture Notes in Computer Science* 715, Springer, pp. 509–523, doi:10.1007/3-540-57208-2_35. Available at https://doi.org/10.1007/3-540-57208-2_35.

[7] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In Chris Hankin, editor: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, *Lecture Notes in Computer Science* 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567. Available at https://doi.org/10.1007/BFb0053567.

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

[9] Sam Lindley & J Garrett Morris (2017): *Lightweight functional session types*. *Behavioural Types: from Theory to Tools. River Publishers*, pp. 265–286.

[10] J Postel (1983): *RFC0862: Echo Protocol*. Available at https://dl.acm.org/doi/pdf/10.17487/RFC0862.

[11] Alceste Scalas & Nobuko Yoshida (2019): *Less is more: multiparty session types revisited*. *Proc. ACM Program. Lang.* 3(POPL), pp. 30:1–30:29, doi:10.1145/3290343. Available at https://doi.org/10.1145/3290343.

[12] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou & Sergios Theodoridis, editors: *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, *Lecture Notes in Computer Science* 817, Springer, pp. 398–413, doi:10.1007/3-540-58184-7_118. Available at https://doi.org/10.1007/3-540-58184-7_118.

[13] Peter Thiemann (2023): *Intrinsically Typed Sessions With Callbacks*. *CoRR* abs/2303.01278, doi:10.48550/arXiv.2303.01278. arXiv:2303.01278. To appear at ICFP'23.

[14] Bernardo Toninho, Luís Caires & Frank Pfenning (2013): *Higher-Order Processes, Functions, and Sessions: A Monadic Integration*. In Matthias Felleisen & Philippa Gardner, editors: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, *Lecture Notes in Computer Science* 7792, Springer, pp. 350–369, doi:10.1007/978-3-642-37036-6_20. Available at https://doi.org/10.1007/978-3-642-37036-6_20.

[15] Max Willsey, Rokhini Prabhu & Frank Pfenning (2016): *Design and Implementation of Concurrent C0*. In Iliano Cervesato & Maribel Fernández, editors: *Proceedings Fourth International Workshop on Linearity, LINEARITY 2016, Porto, Portugal, 25 June 2016*, *EPTCS* 238, pp. 73–82, doi:10.4204/EPTCS.238.8. Available at https://doi.org/10.4204/EPTCS.238.8.

[16] Nobuko Yoshida, Raymond Hu, Rumyana Neykova & Nicholas Ng (2013): *The Scribble Protocol Language*. In Martín Abadi & Alberto Lluch-Lafuente, editors: *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, *Lecture Notes in Computer Science* 8358, Springer, pp. 22–41, doi:10.1007/978-3-319-05119-2_3. Available at https://doi.org/10.1007/978-3-319-05119-2_3.

A An Extended Look at "CAPABLE in Action"

A two page demo proposal doesn't leave much room to showcase precisely what we plan to showcase. We thus provide an overview of the code that will be walked-through during the first part of our demo. The second half of the demo, will walk-through parts of the code that we have already linked to.

A.1 Basics of (Imperative Programming in CAPABLE

CAPABLE is a simple imperative language (with ML-style references) with a concrete term syntax modelled after Rust in which functions contain expressions that can be sequenced. Although the term syntax is inspired by Rust we do not *borrow* the advanced behavioural type checking (specifically borrow checking and lifetimes) that Rust is well known for. Further, the set of base types are indicative containing integers, characters, string, fixed length (and dynamic) arrays, tuples, and data structures comprising of structures and unions. For this section we will walk-through the following code that highlights the core imperative code constructs, using typed-holes to guide the development of the `main` function.

```
struct point {x : Int; y : Int}
union either {left : String; right : Int}

func printLn (str : String) -> Unit { print(str); print(toString('\n')) }
func setVarX (p : &point, x : Int) -> Unit { mut(p, set_x(!p, x)) }

main (args : [String]) -> Unit
{ val p = point(1, 2) ;
  printLn(toString(get_x(!p)));
  setVarX(p, 2);
  printLn(toString(get_x(!p)));
  local x = left("Hello World") ;
  match x { when left(x) { printLn(x) } when right(y) { printLn(toString(y)) } }
}
```

A.2 IPC Behaving Badly

Within CAPABLE behaviourally untyped IPC is file-handle-oriented and presented to the developer in the form of bespoke base types and a corresponding API to create, read, write, and close handles. Using these handles, developers can interact with files (`FILE`), unidirectional processes (`PIPE`), and bi-directional processes (`PROC`). The process API itself avoids the *billion dollar mistake* and returns tagged unions indicating the success of the effectful operations.

We will motivate the *badly behaving* IPC with the following code that shows incorrect interaction with an `Echo` clone written in Python. Our client does not read the response from the initial send and instead attempts to send another value to be returned.

```
main (args : [String]) -> Unit
{ printLn("My First Program with files");
  match popen2("python3 echoClone.py")
  { when left(e) { printLn("Error opening") }
    when right(fhs)
    { match write(fhs, "Hello")
      { when left(err) { printLn("Error sending line.") }
        when right(res)
        { match write(fhs, "Hello")
          { when left(err) { printLn("Error reading line.") }
```

```
            when right(res) { printLn(res); close(fhs) }}}}}}
}
```

A.3 IPC Behaving Correctly

In the final section of the demo in using CAPABLE we demonstrate how the support for MPSTs enables the developer to specify an Echo protocol, and write a session-typed client that adheres to said protocol. The following example will be walked-through and typed.

```
union MaybeInt {nout : Unit; this : Int}
union msg {msg : Int}

role Client role Server

protocol Echo = Client ==> Server [msg] { msg(Int) . Server ==> Client [msg] { msg(Int) . end } }

session echoClient <Echo as Client> (i : Int) -> MaybeInt
{ send [msg] Server msg(i) catch { printLn("Crashing on Send"); crash(nout(unit)) }
  recv [msg] Server  { when msg (i) { end(this(i)) } }
    catch { printLn("Crashing on Recv"); crash(nout(unit)) } }

main (args : [String]) -> Unit
{ match run echoClient(2) with [Server as "python3 pingping.py"]
  { when nout(i) { printLn("Oops we crashed!") }
    when this(i) { print("We echoed this, successfully: "); printLn(toString(i)) } } }
```

We will show how global types are manifested within the language and are *just* another regular old datatype, and a datatype that can also be projected in the REPL. We then will look at how typed-sessions are embedded in the language as a bespoke typed-function, offering a behaviourally safe subset of CAPABLE.

As part of this walk-through we will show how support for typed-holes allows a basic form of type-aware editing. Specifically we will walk-through the following example:

```
session echoClient <Echo as Client>
  (i : Int) -> MaybeInt
{ send [msg] Server msg(i)
    catch { printLn("Crashing on Send");
            crash(nout(unit)) }
  ?next
}
```

```
## Typing Context
i : Int
## Recursion Vars

## Roles
+ Server
+ Client
---
next : (recvFrom[Server] {msg(Int).End})
```

and demonstrates how our compiler, when encountering a typed-hole, displays the local typing context (including recursion variables for processes), the current active roles, and the type of the next step in the protocol.

We will also show the power of bi-directional typing and how we can synthesis partial local types (send is not a synthesisable term, but we have a solution we will talk about during discussion of the mechanisation) from typed-sessions when encountering an error during type-checking. Specifically we will discuss the following ill-typed session and generated error message.

```
session echoServer <Echo as Server>
  (i : Int) -> MaybeInt
{ send [msg] Client msg(i)
     catch { printLn("Crashing on Send");
             crash(nout(unit)) }
  end(nout(unit))
}
```

```
Type Checking Error
main.capable:23:2-0:
Expecting an expression of type:
  (recvFrom[Client] { msg(Int)
  . (sendTo[Client] {msg(Int).End})})
but given:
  (sendTo[Client] msg(Int) . End)
```