Towards Gradual Multiparty Session Typing

Sung-Shik Jongmans
Open University / Centrum Wiskunde & Informatica (CWI)
Heerlen / Amsterdam, The Netherlands

ABSTRACT

To make concurrent programming easier, languages (e.g., Go, Rust, Clojure) have started to offer core support for message passing through channels in shared memory. However, channels also have their issues. Multiparty session types (MPST) constitute a method to make channel usage simpler. In this paper, to consolidate the best qualities of "static MPST" (early feedback, fast execution) and "dynamic MPST" (high expressiveness), we present a project that reinterprets the MPST method through the lens of gradual typing.

ACM Reference Format:
Sung-Shik Jongmans. 2022. Towards Gradual Multiparty Session Typing. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3551349.3561167

1 INTRODUCTION

Background. To take advantage of modern multi-core processors, concurrent programming with shared memory—a notoriously difficult enterprise—has become increasingly important. In the wake of this development, languages have started to offer core support for high-level *communication primitives*, in the form of message passing through channels (e.g., Go, Rust, Clojure), in addition to lower-level *synchronisation primitives*. The idea is that channels can also serve as a programming abstraction for shared memory.

Supposedly, channels are less prone to concurrency bugs than locks, semaphores, and the like. Yet, a growing body of evidence suggests that channels in shared memory also have their issues [38].

A premier source of channel-related concurrency bugs arises out of the following challenge. Suppose that we have:

- A specification S of a *communication session* among *processes* (e.g., a sequence diagram), to be implemented as sends/receives through a shared channel. Typical specifications rule out common concurrency bugs [38], such as sends without receives, receives without sends, or data type mismatches.
- An implementation I that should fulfil S.

How can we ensure that the channel actions in I are indeed *safe* relative to S? Safety means that "bad" channel actions never happen: if a channel action happens in I, then it is allowed to happen in S.

State of the art. *Multiparty session types* (MPST) [19] constitute a method to automatically prove safety. The idea is to specify sessions as *behavioural types* [1, 23] against which processes are statically checked. The MPST method ensures that *well-typedness at compile-time implies safety at run-time.* Over the past decade, major progress has been made, both in theory (e.g., extensions with time [7, 30], security [9–12], and parametrisation [13, 17, 32]), and in practice (e.g., tools in F# [31], F* [40], Go [13], Java [21, 22], OCaml [39], PureScript [25], Rust [26, 27], Scala [15, 34], and TypeScript [29]).

Notwithstanding the achievements, one of the open problems in the MPST literature has remained unsolved: *limited expressiveness.* That is, there continue to be many sessions that cannot be specified, implemented, and checked statically using the MPST method. Essentially, the inescapable complication is that the type checker needs to predict at compile-time how processes will dynamically behave at run-time. Doing so is hard and undecidable in general. As a result, static checks are unduly conservative in the MPST method.

In a recent attempt to sidestep this long-standing open problem, exploratory studies have been conducted to replace static checks with dynamic ones [18, 20], through *run-time monitoring* (based on operational models of behavioural types as state machines). While successful in terms of high expressiveness, it suffers from typical static-vs.-dynamic disadvantages: later feedback, slower execution.

Towards gradual MPST. In this paper, we present a project to consolidate the best qualities of static checks (early feedback, fast execution) and dynamic ones (high expressiveness), by reinterpreting the MPST method through the lens of *gradual typing* [36].

The aim of gradual typing is to allow a program to be *partially* annotated with types. Annotated segments of the program are checked at compile-time, while unannotated segments are checked at run-time. When data flow from a dynamically checked segment to a statically checked one, they are explicitly *cast* to the expected type. If a cast fails, a run-time error is reported *before the statically checked segment is executed.* Several languages have adopted a form of gradual typing (e.g., C# [4], Racket [37], and TypeScript [3]).

To illustrate the key novelties of our project, the main contribution of this paper is the design of a core calculus of *monitored multiparty sessions with casts.* It serves as a first foundation of "gradual MPST" by consolidating early feedback and high expressiveness.

Beyond MPST, our general approach (i.e., combining static/dynamic analysis by integrating operational models into gradual typing) can be applied more broadly in automated run-time monitoring [2], to enable mixed compile-time/run-time verification.

2 CORE CALCULUS

Types. Let $\mathbb{N} = \{\mathbf{a}, \mathbf{b}, \ldots\}$ and $\mathbb{T} = \{\texttt{Bool}, \texttt{Nat}, \ldots\}$ denote the sets of all *process names* and *data types*, ranged over by p, q and by t. Let $\mathbb{B}$ denote the set of all *(behavioural) channel types*:

$$B ::= \varepsilon \mid pq?t \mid \circledast \mid B_1 + B_2 \mid B_1 \cdot B_2 \mid B^*$$

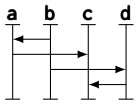
Informally, $pq?t$ specifies the communication of a value of data type t from p to q, while $\circledast$ specifies any communication (wildcard). The remaining primitives specify choice, concatenation, and repetition

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3561167

(cf. regular expressions). Formally, operational models of channel types as state machines are defined using *transition rules* (omitted). We write $B \xrightarrow{pq\dagger t} B'$ to denote a transition from B to B' with $pq\dagger t$.

Example 2.1. For any t, the following channel type specifies a session in which numbers are communicated from Bob to Alice, from Alice to Carol, from Bob to Dave, and from Dave to Carol:



$$B = \mathbf{ba}\dagger t \cdot \mathbf{ac}\dagger t \cdot \mathbf{bd}\dagger t \cdot \mathbf{dc}\dagger t \xrightarrow{\mathbf{ba}\dagger t} B' \xrightarrow{\mathbf{ac}\dagger t} B'' \xrightarrow{\mathbf{bd}\dagger t} B''' \xrightarrow{\mathbf{dc}\dagger t} \varepsilon$$

Terms (syntax). Let $\mathbb{X} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \_, \ldots\}$, $\mathbb{V} = \{\mathtt{true}, \mathtt{false}, \mathtt{0}, \mathtt{1}, \ldots\}$, and $\mathbb{E} = \mathbb{X} \cup \mathbb{V} \cup \{\mathtt{!true}, \mathtt{x+1}, \ldots\}$ denote the sets of all *data variables*, *data values*, and *data expressions*, ranged over by x, by v, and by e.

Let $\mathbb{C}$ and $\mathbb{P}$ denote the sets of all *channel terms* and *process terms*:

$$C ::= k \mid C\langle \mathcal{B} \rangle \qquad P ::= C[pq!e].P \mid C[pq?x{:}t].P \mid \mathbf{if}\ e\ P_1\ P_2 \mid \cdots$$

Channel term k implements the *access* of the channel identified by k. Channel term $C\langle \mathcal{B} \rangle$ implements a *cast* of the type of C from B to $\mathcal{B}(B)$, where $\mathcal{B} : \mathbb{B} \rightharpoonup \mathbb{B}$. Process term $C[pq!e].P$ implements the *send* of e from p to q through C, followed by P'. Process term $C[pq?x{:}t].P'$ implements the *receive* of a value of type t into x from p to q through C, followed by P'. Let $\mathbb{P}^*$ denote the set of all *concurrent programs* (i.e., lists of process terms), ranged over by $\vec{P}$.

Example 2.2. For $t = \mathtt{Nat}$, the following process terms implement a session among Alice, Bob, Carol, and Dave over shared channel k:

$$P_1 = k[\mathbf{ba}?\mathtt{x}{:}t].k[\mathbf{ac}!\mathtt{x}] \qquad P_3 = k[\mathbf{ac}?\_{:}t].k[\mathbf{dc}?\_{:}t]$$
$$P_2 = k[\mathbf{ba}!\mathtt{fast()}].k[\mathbf{bd}!\mathtt{slow()}] \qquad P_4 = k[\mathbf{bd}?\mathtt{y}{:}t].k[\mathbf{dc}!\mathtt{y}{:}t]$$

Terms (static semantics). Let $\mathbb{X} \rightharpoonup \mathbb{T}$ and $\{k_1, k_2, \ldots\} \rightharpoonup \mathbb{B}$ denote the sets of all *data type environments* and *channel type environments*, ranged over by Γ and by Δ. We write $\Gamma[x \mapsto t]$ and $\Delta[k \mapsto B]$ to update Γ and Δ with new mappings.

We write $\Gamma \vdash e : t$ to denote that e is of type t (as usual [33]). We write $\Delta \vdash C : B$ to denote that C is of type B. We write $\Gamma, \Delta \vdash P : \mathbf{1}$ and $\vec{\Gamma}, \Delta \vdash \vec{P} : \mathbf{1}$ to denote that P and $\vec{P}$ are *one-step safe* (i.e., every next action is safe). We write $\vec{\Gamma}, \Delta \vdash \vec{P} : *$ to denote that $\vec{P}$ is *any-step safe* (i.e., every next sequence of actions is safe). Formally (excerpt):

$$\frac{\Delta(k) = B}{\Delta \vdash k : B} \qquad \frac{\Delta \vdash C : B_1 \qquad \mathcal{B}(B_1) = B_2}{\Delta \vdash C\langle \mathcal{B} \rangle : B_2}$$

$$\frac{\Gamma \vdash e : t \qquad \Delta \vdash C : B \qquad B \xrightarrow{pq\dagger t} B'}{\Gamma, \Delta \vdash C[pq!e].P' : \mathbf{1}} \qquad \frac{\forall i \left(\vec{\Gamma}_i, \Delta \vdash \vec{P}_i : \mathbf{1} \right)}{\vec{\Gamma}, \Delta \vdash \vec{P} : \mathbf{1}}$$

$$\frac{\vec{\Gamma}, \Delta \vdash \vec{P} : \mathbf{1} \qquad \forall \left(\vec{\Gamma}, \Delta, \vec{P} \to \vec{\Gamma}', \Delta', \vec{P}' \right) \left(\vec{\Gamma}', \Delta' \vdash \vec{P}' : * \right)}{\vec{\Gamma}, \Delta \vdash \vec{P} : *}$$

The send of e from p to q through C is one-step safe if the type of C allows a communication of a value of type t from p to q, and e is of type t. A concurrent program is one-step safe if every process is. A concurrent program is any-step safe if it is both one-step safe and any-step safe after one step (i.e., the rule is defined in terms of auxiliary *type-level reductions* [28, 35], which are finite/decidable abstractions of value-level reductions in the dynamic semantics).

The key novelty of the static semantics is highlighted in the third rule: processes are checked using the operational models of types (transitions of B) instead of using their syntax. The aim is to mimic run-time monitoring-with-state-machines, *but at compile-time*; it enables us to formalise static/dynamic analysis in a unified fashion.

Example 2.3. $[P_1, P_2, P_3, P_4]$ in Example 2.2 is not any-step safe relative to B in Example 2.1: informally, the type forbids the communication from Bob to Dave (in red font in Example 2.2) to happen before the communication from Alice to Carol (in blue font), but the terms allow it; formally, B' cannot reduce with $\mathbf{bd}\dagger t$. To make the program any-step safe, we insert casts at the problematic actions (in red/blue font) and defer their checks to run-time. For instance, we replace P_4 with $k\langle B' \mapsto \circledast^*, B'' \mapsto B'' \rangle[\mathbf{bd}?\mathtt{y}{:}t].k[\mathbf{dc}!\mathtt{y}{:}t]$.

Terms (dynamic semantics). We write $\Delta, C \Downarrow B$ to denote that C is monitored using B. We write $\Delta, P \xrightarrow{\pi} \Delta', P'$ and $\Delta, \vec{P} \to \Delta', \vec{P}'$ to denote that P and $\vec{P}$ can reduce to P' and $\vec{P}'$. Formally (excerpt):

$$\frac{\Delta(k) = B}{\Delta, k \Downarrow B} \qquad \frac{\Delta, C \Downarrow \mathcal{B}(B) = B}{\Delta, C\langle \mathcal{B} \rangle \Downarrow B} \qquad \frac{\Delta, C \Downarrow \mathcal{B}(B) \neq B}{\Delta, C\langle \mathcal{B} \rangle \Downarrow \varepsilon}$$

$$\frac{\vdash e : t \qquad \Delta, C \Downarrow B \qquad B \xrightarrow{pq\dagger t} B' \qquad \mathrm{id}(C) = k}{\Delta, C[pq!e].P' \xrightarrow{k[pq!e]} \Delta[k \mapsto B'], P'}$$

$$\frac{P = C[pq!e].P' \qquad P \not\to}{\Delta, P \xrightarrow{\mathrm{id}(C)[pq!\mathtt{err}]} \Delta, \mathtt{err}} \qquad \mathrm{id}(C) = \begin{cases} k & \text{if } C = k \\ \mathrm{id}(\hat{C}) & \text{if } C = \hat{C}\langle \mathcal{B} \rangle \end{cases}$$

The send of e from p to q through C is successful if the type of C allows it; it fails with an error otherwise. In particular, a failed cast yields ε, which always results in an error (i.e., ε has no transitions).

The key novelty of the dynamic semantics is the usage of channel type environments in (value-level) reductions, to leverage *the same* operational models of types as in the static semantics. As a result, we also get a natural formalisation of run-time monitoring with state machines in dynamic MPST [18] (i.e., without casts and the static semantics, the core calculus simplifies to dynamic MPST).

The safety guarantee of the core calculus is that if $\vec{\Gamma}, \Delta \vdash \vec{P} : *$, then every process in $\vec{P}$ never reduces to $\mathtt{err}$, unless a cast fails.

Example 2.4. If P_2 for Bob (Example 2.2) computes the value to send to Dave `slow()`-ly indeed, then "coincidentally", Alice and Carol always communicate before Bob and Dave, as required by B (Example 2.1). Without casts, the "coincidentally safe" actions are checked statically (fail); with casts, they are checked dynamically (success). For instance, the cast inserted in P_4 for Dave (Example 2.3) fails only if k is monitored using the operational model of B' when Dave receives, but as Bob computes `slow()`-ly, this never happens.

3 CONCLUSION

Related work. Igarashi et al. [24] studied gradual typing for binary sessions (vs. multiparty in our work); in their work, processes are checked using the syntax of types (vs. operational models in our work). Others [5, 6, 16, 30] studied mixed static/dynamic analysis at the granularity of processes (vs. actions in our work).

Future work. (1) We aim to build a higher-level language (HLL) on top of the core calculus (CC) in which channel actions can be marked as statically/dynamically checked without writing casts explicitly (as usual with gradual typing [14]). Through *automated cast insertion*, HLL can be translated to CC. **(2)** To also consolidate fast execution, we aim to develop a technique to optimise away dynamic checks for channel actions that have already passed static checks. **(3)** We aim to implement CC and HLL in Clojure (which has a form of gradual typing [8]) on top of Discourje (which offers dynamic MPST in Clojure [18]).

REFERENCES

[1] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages* 3, 2-3 (2016), 95–230.

[2] Ezio Bartocci and Yliès Falcone (Eds.). 2018. *Lectures on Runtime Verification - Introductory and Advanced Topics*. Lecture Notes in Computer Science, Vol. 10457. Springer.

[3] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP (Lecture Notes in Computer Science, Vol. 8586)*. Springer, 257–281.

[4] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C#. In *ECOOP (Lecture Notes in Computer Science, Vol. 6183)*. Springer, 76–100.

[5] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. 2017. Monitoring networks through multiparty session types. *Theor. Comput. Sci.* 669 (2017), 33–58.

[6] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *CONCUR (Lecture Notes in Computer Science, Vol. 6269)*. Springer, 162–176.

[7] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. 2014. Timed Multiparty Session Types. In *CONCUR (Lecture Notes in Computer Science, Vol. 8704)*. Springer, 419–434.

[8] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure. In *ESOP (Lecture Notes in Computer Science, Vol. 9632)*. Springer, 68–94.

[9] Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. 2014. Typing access control and secure information flow in sessions. *Inf. Comput.* 238 (2014), 68–105.

[10] Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. 2016. Information flow safety in multiparty sessions. *Mathematical Structures in Computer Science* 26, 8 (2016), 1352–1394.

[11] Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Tamara Rezk. 2010. Session Types for Access and Information Flow Control. In *CONCUR (Lecture Notes in Computer Science, Vol. 6269)*. Springer, 237–252.

[12] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Jorge A. Pérez. 2016. Self-adaptation and secure information flow in multiparty communications. *Formal Asp. Comput.* 28, 4 (2016), 669–696.

[13] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures. *PACMPL* 3, POPL (2019), 29:1–29:30.

[14] Matteo Cimini and Jeremy G. Siek. 2016. The gradualizer: a methodology and algorithm for generating gradual type systems. In *POPL*. ACM, 443–455.

[15] Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença. 2022. API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3. In *ECOOP (LIPIcs, Vol. 222)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:28.

[16] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2015. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *Formal Methods Syst. Des.* 46, 3 (2015), 197–225.

[17] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. 2012. Parameterised Multiparty Session Types. *Logical Methods in Computer Science* 8, 4 (2012).

[18] Ruben Hamers and Sung-Shik Jongmans. 2020. Discourje: Runtime Verification of Communication Protocols in Clojure. In *TACAS (1) (Lecture Notes in Computer Science, Vol. 12078)*. Springer, 266–284.

[19] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*. ACM, 273–284.

[20] Erik Horlings and Sung-Shik Jongmans. 2021. Analysis of specifications of multiparty sessions with dcj-lint. In *ESEC/SIGSOFT FSE*. ACM, 1590–1594.

[21] Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation. In *FASE (Lecture Notes in Computer Science, Vol. 9633)*. Springer, 401–418.

[22] Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *FASE (Lecture Notes in Computer Science, Vol. 10202)*. Springer, 116–133.

[23] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1 (2016), 3:1–3:36.

[24] Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. 2019. Gradual session types. *J. Funct. Program.* 29 (2019), e17.

[25] Jonathan King, Nicholas Ng, and Nobuko Yoshida. 2019. Multiparty Session Type-safe Web Development with Static Linearity. In *PLACES@ETAPS (EPTCS, Vol. 291)*. 35–46.

[26] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. 2020. Implementing Multiparty Session Types in Rust. In *COORDINATION (Lecture Notes in Computer Science, Vol. 12134)*. Springer, 127–136.

[27] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. 2022. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In *ECOOP (LIPIcs, Vol. 222)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:29.

[28] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A static verification framework for message passing in Go using behavioural types. In *ICSE*. ACM, 1137–1148.

[29] Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. 2021. Communication-safe web programming in TypeScript with routed multiparty session types. In *CC*. ACM, 94–106.

[30] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. 2017. Timed runtime monitoring for multiparty conversations. *Formal Asp. Comput.* 29, 5 (2017), 877–910.

[31] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 2018. A session type provider: compile-time API generation of distributed protocols with refinements in F#. In *CC*. ACM, 128–138.

[32] Nicholas Ng and Nobuko Yoshida. 2015. Pabble: parameterised Scribble. *Service Oriented Computing and Applications* 9, 3-4 (2015), 269–284.

[33] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

[34] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *ECOOP (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 24:1–24:31.

[35] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying message-passing programs with dependent behavioural types. In *PLDI*. ACM, 502–516.

[36] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP (Lecture Notes in Computer Science, Vol. 4609)*. Springer, 2–27.

[37] Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen. 2015. Towards Practical Gradual Typing. In *ECOOP (LIPIcs, Vol. 37)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4–27.

[38] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *ASPLOS*. ACM, 865–878.

[39] Nobuko Yoshida, Fangyi Zhou, and Francisco Ferreira. 2021. Communicating Finite State Machines and an Extensible Toolchain for Multiparty Session Types. In *FCT (Lecture Notes in Computer Science, Vol. 12867)*. Springer, 18–35.

[40] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 148:1–148:30.