**WYSTERIA: A Programming Language for Generic, Mixed-Mode Multiparty Computations**


Aseem Rastogi  
University of Maryland, College Park  
Matthew A. Hammer  
University of Maryland, College Park  
Michael Hicks  
University of Maryland, College Park

**Abstract**—In a Secure Multiparty Computation (SMC), mutually distrusting parties use cryptographic techniques to cooperatively compute over their private data; in the process each party learns only explicitly revealed outputs. In this paper, we present WYSTERIA, a high-level programming language for writing SMCs. As with past languages, like Fairplay, WYSTERIA compiles secure computations to circuits that are executed by an underlying engine. Unlike past work, WYSTERIA provides support for mixed-mode programs, which combine local, private computations with synchronous SMCs. WYSTERIA complements a standard feature set with built-in support for secret shares and with wire bundles, a new abstraction that supports generic $n$-party computations. We have formalized WYSTERIA, its refinement type system, and its operational semantics. We show that WYSTERIA programs have an easy-to-understand single-threaded interpretation and prove that this view corresponds to the actual multi-threaded semantics. We also prove type soundness, a property we show has security ramifications, namely that information about one party’s data can only be revealed to another via (agreed upon) secure computations. We have implemented WYSTERIA, and used it to program a variety of interesting SMC protocols from the literature, as well as several new ones. We find that WYSTERIA’s performance is competitive with prior approaches while making programming far easier, and more trustworthy.

I. INTRODUCTION

Secure multi-party computation (SMC) protocols [1], [2], [3] enable two or more parties $p_1,...,p_n$ to cooperatively compute a function $f$ over their private inputs $x_1,...,x_n$ in a way that every party directly sees only the output $f(x_1,...,x_n)$ while keeping the variables $x_i$ private. Some examples are

- the $x_i$ are arrays of private data and $f$ is a statistical function (e.g., median) [4], [5];
- (private set intersection) the $x_i$ are private sets (implemented as arrays) and $f$ is the $\cap$ set-intersection operator [6], [7]; one use-case is determining only those friends, interests, etc. that individuals have in common;
- (second-price auction) the $x_i$ are bids, and $f$ determines the winning bidders [8], [9]

Of course, there are many other possibilities.

An SMC for $f$ is typically implemented using garbled circuits [1], homomorphic encryption [10], or cooperative computation among a group of servers (e.g., using secret shares) [8], [11], [2], [12]. Several libraries and intermediate languages have been designed that provide efficient building blocks for constructing SMCs [13], [14], [15], [16], but their use can be tedious and error prone. As such, there have been several efforts, led by the Fairplay project [17], to define higher-level languages from which an SMC protocol can be compiled. In Fairplay, a compiler accepts a Pascal-like imperative program and compiles it to a garbled circuit. More recent efforts by Holzer et al [18] and Kreuter et al [16] support subsets of ANSI C, and follow-on work has expanded Fairplay’s expressiveness to handle $n > 2$ parties [19].

We are interested in making SMC a building block for realistic programs. Such programs will move between “normal” (i.e., per-party local) computations and “secure” (i.e., joint, multi-party secure) modes repeatedly, resulting overall in what we call mixed mode computations. For example, we might use SMCs to implement the role of the dealer in a game of mental poker [20]—the game will be divided into rounds of local decision-making and joint interaction (shuffling, dealing, bidding, etc.). Mixed-mode computations are also used to improve performance over monolithic secure computations. As one example, we can perform private set intersection by having each party iteratively compare elements in their private sets, with only the comparisons (rather than the entire looping computation) carried out securely [7]. Computing the joint median can similarly be done by restricting secure computations only to comparisons, which Kerschbaum [5] has shown can net up to a $30\times$ performance improvement.

Existing projects, like Fairplay, focus on how to compile normal-looking code into a representation like a boolean circuit that can be run by an SMC engine. In most cases, such compilation is done in advance. As such, mixed-mode programming is implemented by writing local computations in some host language (like C or Java) that call out to the SMC engine to evaluate the generated code. However, writing mixed-mode computations in a single language has some compelling benefits over the multi-lingual approach. First, it is easier to write mixed-mode programs since the programmer can see all of the interactions in one place, and not have to navigate foreign function interfaces, which are hard to use [21]. Second, for the same reason, programs are easier to understand, and thus answers to security concerns (could the overall computation leak too much information about my secrets?) will be more evident. Third, there is an opportunity for more dynamic execution models, e.g., compiling secure blocks on the fly where the participants or elements of the computation may be based on the results of prior interactions. Finally, there is an opportunity for greater code reuse, as a single language can encapsulate mixed mode protocols as reusable library functions.
This paper presents a new programming language for writing mixed-mode secure computations called \textsc{Wysteria} that realizes all of these benefits. \textsc{Wysteria} is the first language to support writing mixed-mode, multiparty computations in a generic manner, supporting any number of participants. It does this by offering several compelling features:

\textbf{Conceptual single thread of control:} All \textsc{Wysteria} programs operate in a combination of \textit{parallel} and \textit{secure} modes, where the former identifies local computations taking place on one or more hosts (in parallel), and the latter identifies secure computations occurring jointly among parties. Importantly, \textsc{Wysteria} mixed-mode computations can be viewed as having a single thread of control, with all communication between hosts expressed as variable bindings accessed within secure computations. Single threadness makes programs far easier to write and reason about (whether by humans or by automated analyses [5], [22]). We formalize \textsc{Wysteria}'s single-threaded semantics and prove a simulation theorem from single- to multi-threaded semantics. We also prove a theorem for correspondence of final configurations in the other direction, for successfully terminating multi-threaded executions. In both semantics, it is evident that all communication among parties occurs via secure blocks, and thus, information flows are easier to understand.

\textbf{Generic support for more than two parties:} \textsc{Wysteria} programs may involve an arbitrary number of parties, such that which parties, and their number, can be determined dynamically rather than necessarily at compile-time. To support such programs, \textsc{Wysteria} provides a notion of \textit{principals as data} which can be used to dynamically determine computation participants or their outcomes (e.g., to identify winners in a tournament proceeding to the next round). \textsc{Wysteria} also implements a novel feature called \textit{wire bundles} that are used to represent the inputs and outputs of secure computations such that a single party's view of a wire bundle is his own value, while the shared view makes evident all possible values. A secure computation, having the shared view, may iterate over the contents of a bundle. The length of such a bundle may be unspecified initially. The \textsc{Wysteria} compiler employs \textit{dynamic circuit generation} to produce circuits when unknowns (like wire bundle lengths) become available. Despite this dynamism, \textsc{Wysteria}'s meta-theoretical properties guarantee that participants proceed synchronously, i.e., they will always agree on the protocol they are participating in.

\textbf{Secret shares:} Many interesting programs interleave local and secure computations where the secured outcomes are revealed later. For example, in mental poker, each party must maintain a representation of the deck of cards whose contents are only revealed as cards are dealt. To support such programs, \textsc{Wysteria} provides secret shares as first-class objects. Secret shares resemble wire bundles in that each party has a local view (copy) and these views are combined in secure blocks to recover the original value. The \textsc{Wysteria} type system ensures shares are used properly; e.g., programs cannot inadvertently combine the shares from different objects.

\textbf{Refinement type system:} \textsc{Wysteria} is a functional programming language that comes equipped with a \textit{refinement type system} to express the expectations and requirements of computations in the language. In particular, the types for wire bundles and shares are \textit{dependent}, and directly identify the parties involved. For example, suppose we have a function \texttt{is_richer} that takes a list of principals and their net worths and returns who is richest. The logical refinement on the function's return type will state that the returned principal is one from the original set. Our type system also provides \textit{delegation effects} for expressing in which context a function can be called; e.g., a function that computes in parallel mode cannot be called from within a secure computation, while the reverse is possible in certain circumstances. In general, our type system ensures the standard freedom from type errors: there will be no mistake of Alice communicating a string to a secure computation which expects an integer. \textsc{Wysteria}'s mixed-mode design enables such reasoning easily: separating the host and SMC languages would make a proof of type soundness for mixed-mode programs far more difficult.

We have implemented a \textsc{Wysteria} interpreter which executes secure blocks by compiling them to boolean circuits, executed by Choi et al.'s implementation [23] of the Goldreich, Micali, and Wigderson protocol [2]. We have used \textsc{Wysteria} to build a broad array of mixed-mode programs proposed in the literature, along with some new ones. Our experimental results demonstrate three key points. First \textsc{Wysteria}'s performance is competitive with prior approaches; e.g., we can reproduce the mixed-mode performance gains reported previously. Second, generic protocols for \textit{n}-principals can be expressed with ease in \textsc{Wysteria}, and executed efficiently. Finally, \textsc{Wysteria}'s novel high-level abstractions, e.g. secure state, enables expressing novel protocols not present in the existing literature.

\textbf{Related work:} \textsc{Wysteria} is not the first language for mixed mode SMC, but is unique in its high-level design, generality, and formal guarantees. For example, languages like L1 [24] and SMCL [8] permit some mixed-mode computations to be expressed directly. However, these languages lack \textsc{Wysteria}'s single-threaded semantics, exposing more low-level details, e.g., for performing communication or constructing secret shares. As such, there are more opportunities for mistakes; e.g., one party may fail to always receive a sent message (or may receive the wrong one), or may not provide the right protocol shares. L1 is also limited to only two parties, and neither language has a type system expressing the requirements for well-formed mixed-mode compositions (which is handled by our delegation effects). No prior system of which we are aware has formalized its operational semantics and type system and shown them to be sensible (with the notable exception of Liu et al. [25], discussed later). Similar pitfalls exist for other languages (Section IX surveys related work).

The next section begins by presenting an overview of \textsc{Wysteria}'s design and features by example. Section III presents $\lambda_{W}$, a formalization of \textsc{Wysteria}, and Sections IV and V formalize $\lambda_{W}$'s type system and operational semantics, both single- and multi-threaded versions. Section VI proves the type system is sound and that the two semantics correspond. Sections VII and VIII describe our implementation and experimental results, and we finish with related work and conclusions.

\textsc{Wysteria} implementation is available at \url{http://bitbucket.org/aseemr/wysteria}.
II. OVERVIEW OF WYSTERIA

WYSTERIA is a functional programming language for performing mixed-mode secure multiparty computations. It has many features inherited directly from functional programming, such as first-class (higher order) functions, variable binding with let, tuples (aka records), sums (aka variants or tagged unions), and standard primitive values like integers and arrays. In this section we introduce WYSTERIA’s novel features.

Computation Modes: WYSTERIA defines two computation modes: secure mode in which secure (multiparty) computations take place, and parallel mode in which one or more parties compute locally, in parallel. Here is a version of the so-called millionaires’ problem that employs both modes:

```
1 let a =par({Alice})= read() in
2 let b =par({Bob})= read() in
3 let out =sec({Alice,Bob})= a>b in
4 out
```

Ignoring the par() and sec() annotations, this program is just a series of let-bindings: it first reads Alice’s value, then reads Bob’s value, computes which is bigger (who is richer?), and returns the result. The annotations indicate how and where these results should be computed. The par({Alice}) annotation indicates that the read() (i.e., the rhs computation) will be executed locally (and normally) at Alice’s location, while the par({Bob}) annotation indicates that the second read() will be executed at Bob’s location. The sec({Alice,Bob}) annotation indicates that a > b will be executed as a secure multiparty computation between Alice and Bob. Notice communication from local nodes (Bob and Alice) to secure computation is done implicitly, as variable binding: the secure block “reads in” values a and b from each site. Our compiler sees this and compiles it to actual communication. In general, WYSTERIA programs, though they will in actuality run on multiple hosts, can be viewed as having the apparent single-threaded semantics, i.e., as if there were no annotations; we have proved a simulation theorem from single- to multi-threaded semantics.

In the example we used parallel mode merely as a means to acquire and communicate values to a secure-mode computation (which we sometimes call a wire). As we show throughout the paper, there are many occasions in which parallel mode is used as a substantial component of the overall computation, often as a way to improve performance. On these occasions we specify the mode par(w) where w is a set of principals, rather than (as above) a single principal. In this case, the rhs of the let binding executes the given code at every principal in the set. Indeed, the code in the example above is implicitly surrounded by the code let result =par({Alice,Bob})= e in (result where e is the code given above). This says that Alice and Bob both, in parallel, run the entire program. In doing so, they will delegate to other computation modes, e.g., to a mode in which only Alice performs a computation (to bind to a) or in which only Bob does one, or in which they jointly and securely compute the a>b. The delegation rules stipulate that parallel mode computations among a set of principals may delegate to any subset of those principals for either secure or parallel computations. We will see several more examples as we go.

Wires: In the above example, we are implicitly expressing that a is Alice’s (acquired in par({Alice}) mode) and b is Bob’s. But suppose we want to make the computation of out into a function: what should the function’s type be, so that the requirements of input sources are expressed? We do this with a novel language feature we call wires, as follows:

```
1 is_richer = λa: W {Alice} nat. λb: W {Bob} nat. let out =sec({Alice,Bob})= a[Alice] > [b][Bob] in
2 out
```

Here, the is_richer function takes two arguments a and b, each of which is a wire. The wires express that the data “belongs to” a particular principal: a value of type W{Alice} t is accessible only to Alice, which is to say, inside of par({Alice}) computations or sec({Alice}∪w) computations (where w can be any set of principals); notably, it is not accessible from within computations par({Alice}∪w) where w is a nonempty set. Note that wires are given dependent types which refer to principal values, in this case the values Alice and Bob; we will see interesting uses of such types shortly. Here is how we can call this function:

```
1 let a =par({Alice})= read() in
2 let b =par({Bob})= read() in
3 let out =is_richer (wire {Alice} a) (wire {Bob} b) in
4 out
```

This code is creating wires from Alice and Bob’s private values and passing them to the function. Note that the output is not a wire, but just a regular value, and this is because it should be accessible to both Alice and Bob.

Delegation effects: Just as we use types to ensure that the inputs to a function are from the right party, we can use effects on a function’s type to ensure that the caller is in a legal mode. For example, changing the third line in the above program to let out =par({Alice})= is_richer ··· would be inappropriate, as we would be attempting to invoke the is_richer function only from Alice even though we also require Bob to participate in the secure computation in the function body. The requirement of joint involvement is expressed as a delegation effect in our type system, which indicates the expected mode of the caller. The delegation effect for is_richer function is sec({Alice,Bob}), indicating that it must be called from a mode involving at least Alice and Bob (e.g., par({Alice,Bob})). The effect annotates the function’s type: the type of is_richer is thus W(Alice) nat → W(Bob) nat → sec({Alice,Bob}) → bool; i.e., is_richer takes Alice’s wire and Bob’s wire, delegates to a secure block involving the two of them, and produces a boolean value. Delegation effects like par({Alice,Bob}) are also possible.

Wire bundles: So far we have used single wires, but WYSTERIA also permits bundling wires together, which (as we will see later) is particularly useful when parties are generic over which and how many principals can participate in a secure computation. Here is our example modified to use bundling:

```
1 is_richer = λv: W {Alice,Bob} nat. let out =sec({Alice,Bob})= v[Alice] > v[Bob] in
2 out
```

This code says that the input is a wire bundle whose values are from both Alice and Bob. We extract the individual values from the bundle v inside of the secure block using array-like projection syntax. To call this function after reading the inputs a and b we write is_richer ((wire {Alice} a) ++ (wire {Bob} b)).

2This program does not actually type check in WYSTERIA, but it is useful for illustration; the corrected program (using “wires”) is given shortly.
Here the calling code concatenates together, using $\rightarrow$, the two wires from Alice and Bob into a bundle containing both of their inputs. Of course, this is just an abstraction, and does not literally represent what is going on at either party’s code when this is compiled. For principal $p$, an empty wire bundle · (“dot”) is used for a wire bundle when $p$ is not in its domain, so that for Alice the above wire bundle would be represented as (Alice:a) $\rightarrow$ · while for Bob it would be · $\rightarrow$ {Bob:b}. When the secure computation begins, each party contributes its own value for every input bundle, and receives only its own value for every output bundle. The type system’s accessibility rules for bundles generalize what was stated above: if $v$ has type $W \ (\ A \cup \ b \ ) \ nat$, where $w$ may or may not be empty, then $v[A]$ is only allowed in $\text{par}((A))$ mode or in $\text{sec}((A) \cup w2)$ mode (s.t. $v$ is accessible to $\text{sec}((A) \cup w2)$, and nowhere else.

First-class principals and $n$-party computation: Now suppose we would like to generalize our function to operate over an arbitrary number of principals. At the moment we would have to write one function for two principals, a different one for three, and yet another one for four. To be able to write just one function for $n$ parties we need two things, (1) a way to abstract which principals might be involved in a computation, and (2) a way to iterate over wire bundles. Then we can write a function that takes a wire bundle involving multiple arbitrary principals and iterate over it to find the highest value, returning the principal who has it. Here is the code to do this:

```
richest_of = \ lambdas:ps. \ lambda: v : W ms nat.
          let out = sec ms =
              wfold (None, v).
          \ richest: \ l. \ lambda: \ n. :
              match richest with
              | None => Some p
              | Some q => if n > v[q] then Some p
                          else Some q
          in (wire ms out)
```

The idea is that $ms$ abstracts an unknown set of principals (which has type $ps$), and $wfold$ permits iterating over the wire bundle for those principals: notice how $ms$ appears in the type of $v$. The $wfold$ construct takes three arguments. Its first argument $None$ is the initial value for the loop’s accumulator ($None$ is a value of optional type). The second argument $v$ is the wire bundle to iterate over. The $wfold$’s body above is an anonymous function with three parameters: richest is the current accumulator, whose value is the richest principal thus far (or $None$), $p$ is the current principal under consideration, and $n$ is $p$’s wire value, a $nat$. On line 5, no principal has yet been considered so the first becomes a candidate to be richest. Otherwise, on line 6, the protocol compares the current maximum with the present principal’s worth and updates the accumulator. When the loop terminates, it yields the value of the accumulator, which is placed in a wire bundle and returned.

In addition to $wfold$, Wysteria also provides a way to apply a function to every element of a wire bundle, producing a new bundle (like the standard functional map).

```
richest_of = \ lambdas:ps. \ lambda: v : W ms nat.
          let out = sec ms =
              wfold (None, v).
          \ richest: \ l. \ lambda: \ n. :
              match richest with
              | None => Some p
              | Some q => if n > v[q] then Some p
                          else Some q
          in (wire ms out)
```

The $richest_of$ function illustrates that first-class principals are useful as the object of computation: the function’s result $r$ is a wire bundle carrying principal options. The type for values in $r$ is a refinement type of the form $t\phi$ where $\phi$ is a formula that refines the base type $t$. The particular type states that every value in $r$ is either None or Some(s) where not only $s$ has type $ps$ (a set of principals), but that it is a singleton set (the $s$ part), and this set is a subset of all (the $\subseteq$ all part); i.e., $s$ is exactly one of the set of principals involved in the computation. Wysteria uses refinements to ensure delegation requirements, e.g., to ensure that if $ps0$ is the set of principals in a nested parallel block, then the set of principals $ps1$ in the enclosing block is a superset, i.e., that $ps1 \supseteq ps0$. Refinements capture relationships between principal sets in their types to aid in proving such requirements during type checking.

Secret shares: Secure computations are useful in that they only reveal the final outcome, and not any intermediate results. However, in interactive settings we might not be able to perform an entire secure computation at once but need to do it a little at a time, hiding the intermediate results until the very end. To support this sort of program, Wysteria provides secret shares. Within a secure computation, e.g., involving principals $A$ and $B$, we can encode a value of type $t$ into shares having type $Sh \ w t$ where $w$ is the set of principals involved in the computation (e.g., $(A,B)$). When a value of this type is returned, each party gets its own encrypted share (similar to how wire bundles work, except that the contents are abstract). When the principals engage in a subsequent secure computation their shares can be recombined into the original value.

To illustrate the utility of secret shares in the context of mixed mode protocols, we consider a simple two-player, two-round bidding game. In each of the two rounds, each player submits a private bid. A player “wins” the game by having the higher average of two bids. The twist is that after the first round, both players learn the identity of the higher bidder, but not their bid. By learning which initial bid is higher, the players can adjust their second bid to be either higher or lower, depending on their preferences and strategy.

The protocol below implements the game as a mixed-mode computation that consists of two secure blocks, one per round. In order to force players to commit to their initial bid while not revealing it directly, the protocol stores the initial bids in secret shares. In the second secure block, the protocol recovers these bids in order to compute the final winning bidder:

```
/* Bidding round 1 of 2: */
let a1 = par ({ Alice }) = read () in
let b1 = par ({ Bob }) = read () in
let in1 = (wire { Alice }) a1 \rightarrow (wire { Bob }) b1 in
let (higher1, sa, sb) = sec ({ Alice, Bob }) =
let c = if in1 [Alice] > in2 [Bob] then Alice else Bob in
(c, makshe [in1 [Alice]], makshe [in1 [Bob]])
in
print higher1 ;

/* Bidding round 2 of 2: */
let a2 = par ({ Alice }) = read () in
let b2 = par ({ Bob }) = read () in
let in2 = (wire { Alice }) a2 \rightarrow (wire { Bob }) b2 in
let higher2 = sec ({ Alice, Bob }) =
let (a1, b1) = (combsb sa, combsb sb) in
let bid_a = (a1 + in2 [Alice]) / 2 in
let bid_b = (b1 + in2 [Bob]) / 2 in
if bid_a > bid_b then Alice else Bob in
print higher2
```

The first secure block above resembles the millionaires’ protocol, except that it returns not only the principal \( e \) with the higher input but also secret shares of both inputs: \( sa \) has type \( Sh (Alice, Bob) int \), and both Alice and Bob will have a different value for \( sa \), analogous to wire bundles; the same goes for \( sb \). The second block recovers the initial bids by combining the players’ shares and computes the final bid as two averages.

Unlike past SMC languages that expose language primitives for secret sharing (e.g., [24]), in WYSTERIA the type system ensures that shares are not misused, e.g., shares for different underlying values may not be combined.

**Expressive power:** We have used WYSTERIA to program a wide variety of mixed-mode protocols proposed in the literature, e.g., all of those listed in the introduction, as well as several of our own invention. In all cases, WYSTERIA’s design arguably made writing these protocols simpler than their originally proposed (often multi-lingual) implementations, and required few lines of code. We discuss these examples in more detail in Section VIII and in Appendix A.

### III. Formal Language

In this section we introduce \( \lambda_w \), the formal core calculus that underpins the language design of WYSTERIA. Sections IV and V present \( \lambda_w \)’s type system and operational semantics, respectively, and Section VI proves type soundness and a correspondence theorem.

Figure 1 gives the \( \lambda_w \) syntax for values \( v \), expressions \( e \), and types \( \tau \). \( \lambda_w \) contains standard values \( v \) consisting of variables \( x \), natural numbers \( n \) (typed as \( nat \)), sums \( inj \), \( v \) (typed by the form \( t_1 + t_2 \)), and products \( (v_1, v_2) \) (typed by the form \( t_1 \times t_2 \)). In addition, \( \lambda_w \) permits principals \( p \) to be values, as well as sets thereof, constructed from singleton principal sets \( \{w\} \) and principal set unions \( w_1 \cup w_2 \). These are all given type \( ps \), where \( p \) is a type refinement; the type system ensures that if a value \( w \) has type \( ps \), then \( \phi \) is valid, and \( \phi \) is a type refinement; the type system ensures that if a value \( w \) has type \( ps \), then \( \phi \) is valid.

Refinements in \( \lambda_w \) are relations in set theory. Refinements \( \nu \subseteq w \) and \( \nu = w \) capture subset and equality relationships, respectively, with another value \( w \). The refinement \( sing(\nu) \) indicates that the principal set is a singleton.

\( \lambda_w \) expressions \( e \) are, for simplicity, in so-called administrative normal form (ANF), where nearly all sub-expressions are values, as opposed to arbitrary nested expressions. ANF form can be generated from an unrestricted WYSTERIA program with simple compiler support.

Expressions include arithmetic operations \((v_1 + v_2)\), case expressions (for computing on sums), and \( \text{fst} \) and \( \text{snd} \) for accessing elements of a product. Expressions \( \lambda x.e \) and \( v_1 v_2 \) denote abstractions and applications respectively. \( \lambda_w \) also includes standard \( \text{fix} \) point expressions (which encode loops) and mutable arrays: \( \text{array}(v_1, v_2) \) creates an array (of type \( \text{Array} \tau \)) whose length is \( v_1 \), and whose elements (of type \( \tau \)) are each initialized to \( v_2 \); array accesses are written as \( \text{select}(v_1, v_2) \) where \( v_1 \) is an array and \( v_2 \) is an index; and array updates are written as \( \text{update}(v_1, v_2, v_3) \), updating array \( v_1 \) at index \( v_2 \) with value \( v_3 \).

Let bindings in \( \lambda_w \) can optionally be annotated with a mode \( M \), which indicates that expression \( c_1 \) should be executed in mode \( M \) as a delegation from the present mode. Modes are either secure (operator \( s \)) or parallel (operator \( p \)), among a set of principals \( w \). Mode \( T \) represents is a special parallel mode among all principals; at run-time, \( T \) is replaced with \( p(w) \) where \( w \) is the set of all principals participating in the computation. Once the execution of \( c_1 \) completes, \( c_2 \) then executes in the original mode. Unannotated let bindings execute in the present mode. \( \lambda_w \) has dependent function types, written \( x : t_1 \rightarrow t_2 \), where \( x \) is bound in \( e \) and \( t_2 \); the \( \epsilon \) annotation is an effect that captures all the delegations inside the function body. An effect is either empty, a mode, or a list of effects.

Wire bundle creation, concatenation, and folding are written \( \text{wire}_w(v), w_1 + w_2, \) and \( \text{wfold}_w(v_1, v_2, v_3) \), respectively (the \( w \) annotation on \( \text{wfold} \) and other combinators denotes the domain of the wire bundle being operated on). Wire bundles carrying a value of type \( \tau \) for each principal in a set \( w \) are given the (dependent) type \( W \tau \). We also support mapping a wire bundle by a either a single function \( \text{wapp}_w(v_1, v_2) \), or another wire bundle of per-principal functions \( \text{wapp}_w(v_1, v_2) \). Finally, the form \( \text{wcopy}_w(v) \) is a coercion that allows wire bundles created in delegated computations to be visible in computations that contain them (operationally, \( \text{wcopy}_w(v) \) is a no-op). \( \lambda_w \) also models support for secret shares, which have type \( Sh \, w \, \tau \), analogous to the type of wire bundles. Shares of value \( v \) are created (in secure mode) with \( \text{makesh}(v) \) and reconstituted (also in secure mode) with \( \text{combsh}(v) \).

### IV. Type System

At a high level, the \( \lambda_w \) type system enforces the key invariants of a mixed-mode protocol: (a) each variable can only be used in an appropriate mode, (b) delegated computations require that all participating principals are present in the current mode, (c) parallel local state (viz., arrays) must remain consistent across parallel principals, and (d) code in the secure blocks must be restricted so that it can be compiled to a boolean circuit in our implementation. In this section, we present the typing rules, and show how these invariants are maintained.
Γ ⊢ M v : τ

**Value typing**: Figure 2 shows the value typing judgement Γ ⊢ M v : τ, read as under Γ and in current mode M, value v has type τ. Variable bindings in Γ are of two forms: the usual x : τ, and x : M τ where M is the mode in which x is defined. Rule T-VAR looks up the binding of x in Γ, and checks that either x is bound with no mode annotation, or matches the mode M in the binding to the current mode. It uses an auxiliary judgement for type well-formedness, Γ ⊢ τ, which enforces invariants like, for a wire bundle type W w τ, w should have a ps φ type. The rule T-INJ uses another auxiliary judgement τ IsFlat which holds for types τ that lack wire bundles and shares. Both auxiliary judgments are defined in the Appendix (Figure 12). WySERTIA disallows wire bundles and shares in sum values, since it hides their precise sizes; the size information of wires and shares is required for boolean circuit generation. The rules T-PROD, T-PRINC, T-PSONE, T-PSUNION, and T-PSVAR are unsurprising.

**Delegations**: Rule T-MSUB is used to permit reading a variable in the present mode N, though the variable is bound in mode M. This is permitted under two conditions: when mode M is well-formed N = s(\_), which reads as under type environment Γ, mode M can delegate computation to mode N. The former condition enforces that variables accessible in secure blocks do not include functions known only to some parties as they can’t be compiled to circuits by all the parties. As such, the condition excludes wire bundles that carry functions. The condition Γ ⊢ M ⊃ N captures the intuitive idea that a variable defined in a larger mode can be accessed in a smaller mode. It is defined at the top of Figure 3. In addition to capturing valid variable accesses across different modes, the same relation also checks when it is valid for a mode to delegate computation to another mode (let \(x \overset{\tau}{\rightarrow} w_2 \in \phi_2\)). The rule D-REFL type checks the reflexive case, where the refinement \(\nu = w_1\) captures that \(w_2 = w_1\) at run-time. The special mode T, that we use to type check generic library code written in WySERTIA, can delegate to any mode (rule D-TOP). Recall that at run-time, T is replaced with p(w), where w is the set of all principals. A parallel mode p(w) can delegate computation to another parallel mode p(w) only if all the principals in w are present at the time of delegation, i.e. \(w_2 \subseteq w_1\). The rule D-RENZ forces this check by typing w2 with the \(\nu \subseteq w_1\) refinement. Finally, principals in parallel mode can begin a secure computation; rule D-SEC again uses refinements to check this delegation. We note that rules of type D-RENZ and rule D-SEC can be combined to effectively delegate from parallel mode to secure block consisting of a subset of the ambient principal set. Secure modes are only allowed to delegate to themselves (via rule D-REFL), since secure blocks are implemented using monolithic boolean circuits.

**Subtyping**: Rule T-SUB is the (declarative) subsumption rule, and permits giving a value of type \(\tau_1\) the type \(\tau\) if the \(\tau\) is a subtype of \(\tau_1\). More precisely, the subtyping judgement Γ ⊢ \(\tau_1 \triangleleft \tau_2\) is read as under Γ type \(\tau_1\) is a subtype of \(\tau_2\). The rules for this judgement are given at the bottom of Figure 3. Rules S-REFL, S-TRANS, S-SUM, S-PROD, S-PRINC, S-ARRAY, S-ARROW are standard. Rule S-PRINC offloads reasoning about refinements to an auxiliary judgement written \([\Gamma] \vdash \phi_1 \Rightarrow \phi_2\), which reads assuming variables in \(\Gamma\) satisfy their refinements, the refinement \(\phi_1\) entails \(\phi_2\). We elide the details of this ancillary judgement, as it can be realized with an SMT solver; our implementation uses Z3 [26], as described in Section VII. For wire bundles, the domain of the supertype, a principal set, must be a subset of domain of the subtype, i.e. type W w1 \(\tau_1\) is a subtype of W w2 \(\tau_2\) if \(w_2 \subseteq w_1\), and \(\tau_1 \triangleleft \tau_2\) (rule S-WIRE). As mentioned earlier, value w2 is typed with no mode annotation. Rule S-SHARE is similar but requires \(\tau_1\) and \(\tau_2\) to be invariant since circuit generation requires an fixed size (in bits) for the shared value.
Fig. 4. Expression typing judgements. Judgment $\Gamma \vdash v : \tau$, for typing values appearing in dependent types, is defined in the appendix (Figure 11).

**Expression typing:** Figure 4 gives the typing judgement for expressions, written $\Gamma \vdash_M e : \tau ; \epsilon$ and read under $\Gamma$ at mode $M$, the expression $e$ has type $\tau$ and delegation effects $\epsilon$. The rules maintain the invariant that $\Gamma \vdash_M M \triangleright \epsilon$, i.e. if $\epsilon$ is well-typed, then $M$ can perform all the delegation effects in $\epsilon$.

The rules T-BINOP, T-FST, and T-SND are standard. Rule T-CASE is mostly standard, except for two details. First, the effect in the conclusion contains the effects of both branches. The second detail concerns secure blocks. We need to enforce that case expressions in secure blocks do not return functions, since it won’t be possible to inline applications of such functions when generating circuits. So, the premise $(M = p(\_)) \land \epsilon = (p \triangleright \epsilon) \lor (\tau \text{IsFO} \land \epsilon = \_)$ enforces that either the current mode is parallel, in which case there are no restrictions on $\tau$, or we add an effect $p(\_)$ so that secure blocks cannot reuse this code, or the type returned by the branches is first order (viz., not a function).

Rule T-LAM is the standard rule for typing a dependent effectful function: the variable $x$ may appear free in the type of function body $\tau_1$ and its effect $\epsilon$, and $\epsilon$ appears on the function type. Rule T-APP is the function application rule. It checks that $v_1$ is a function type, $v_2$ matches the argument type of the function,\footnote{We restrict the values appearing in dependent types ($v_2$ in this case) to be typed without any mode annotation. We use an auxiliary judgment (similar to value-typing) $\Gamma \vdash v : \tau$ to type such values (see Appendix Figure 11). This judgment can only access those variables in $\Gamma$ that are bound without any mode.} and that the application $\tau_2[v_2/x]$ is well-formed in $\Gamma$. The rule performs two additional checks. First, it ensures that the current mode $M$ can perform the delegation effects inside the function body ($\Gamma \vdash_M M > [v_2/x]$). Second, it disallows partial applications in secure blocks ($M = s(\_) \Rightarrow \tau \text{IsFO}$) to prevent a need to represent functional values as circuits; our implementation of secure blocks inlines all function applications before generating circuits.
Rule **T-LET1** is the typing rule for regular let bindings. Rule **T-LET2** type checks the let bindings with delegation annotations. The rule differs from rule **T-LET1** in several aspects. First, it checks that the delegation is legal (premise $\Gamma \vdash M : N$). Second, it checks $e_1$ in mode $N$, instead of current mode $M$. Third, it checks $e_2$ with variable $x$ bound in mode $m(w)$ in $\Gamma$. This mode consists of the outer modal operator $m$ and the delegated party set $w$, meaning that $x$ is available only to principals in $w$, but within the ambient (secure or parallel) block.

Rule **T-FIX** type checks unguarded recursive loops (which are potentially non-terminating). The rule is standard, but with the proviso that such loops are only permitted in parallel blocks $(M = p(\_))$ and the current mode can perform all the effects of the loop $(\nu \vdash M \triangleright e)$. It also adds an effect $p(\_)$ to the final type, so that secure blocks cannot use such loops defined in parallel blocks.

The rules **T-ARRAY**, **T-SELECT**, and **T-UPDATE** type array creation, reading, and writing, respectively, and are standard with the proviso that the first and third are only permitted in parallel blocks. For array writes, the premise $\text{mode}(v_1, \Gamma) = M$ also requires the mode of the array to exactly match the current mode, so that all principals who can access the array do the modification. We elide the definition of this function, which simply extracts the mode of the argument from the typing context (note that since the type of $v_1$ is an array, and since programmers do not write array locations in their programs directly, the value $v_1$ must be a variable and not an array location).

Rule **T-WIRE** introduces a wire bundle for the given principal set $w_1$, mapping each principal to the given value $v$. The first premise $\Gamma \vdash w_1 : \text{ps}(\nu \subseteq w_2)$ requires that $w_1 \subseteq w_2$, i.e., all principals contributing to the bundle are present in the current mode. The mode under which value $v$ is typed is determined by the modal operator $m$ of the current mode. If it is $p$, $v$ is typed under $p(w_1)$. However, if it is $s$, $v$ is typed under the current mode itself. In parallel blocks, where parties execute locally, the wire value can be typed locally. However, in secure blocks, the value needs to be typable in secure mode. The next premise $m = s \Rightarrow \tau \text{IsF0}$ ensures that wire bundles created in secure mode do not contain functions, again to prevent private code execution in secure blocks. Finally, the premise $\tau \text{IsF1At}$ prevents creation of wire bundles containing shares.

Rule **T-WPROJ** types wire projection $v[w_2]$. The premise $\Gamma \vdash m(w_1) : v : W w_2 \tau$ ensures that $v$ is a wire bundle having $w_2$ in its domain. To check $w_2$, there are two subcases, distinguished by current mode being secure or parallel. If the latter, there must be but one participating principal, and that principal value needs to be equal to the index of wire projection: the refinement $\phi$ enforces that $w_2 = w_1$, and $w_2$ is a singleton. If projecting in a secure block, the projected principal need only be a member of the current principal set. Intuitively, this check ensures that principals are participants in any computation that uses their private data.

The rule **T-WIREUN** concatenates two wire bundle arguments, reflecting the concatenation in the final type $W(w_1 \cup w_2) \tau$. The rules **T-WFOLD**, **T-WAPP**, **T-WAPS**, and **T-WCOPY** type the remaining primitives for wire bundles. In rule **T-WFOLD**, the premise enforces that $\text{wfold}$ is only permitted in secure blocks, that the folding function is pure (viz., its set of effects is empty), and that the types of the wire bundle, accumulator and function agree. As with general function application in secure blocks, the rule enforces that the result of the fold is first order. The rules **T-WAPP** and **T-WAPS** are similar to each other, and to rule **T-WFOLD**. In both rules, a function $v_2$ is applied to the content of a wire bundle $v_1$, rule **T-WAPP** handles parallel mode applications where the applied functions reside in a wire bundle (i.e., each principal can provide their own function). Rule **T-WAPS** handles a more restricted case where the applied function is not within a wire bundle; this form is required in secure mode because to compile a secure block to a boolean circuit at run-time each principal must know the function being applied. As with rule **T-WFOLD**, the applied functions must be pure. Rule **T-WCOPY** allows copying of a wire bundle value $v$ from $p(w_2)$ to $p(w_1)$ provided $w_2 \subseteq w_1$. This construct allows principals to carry over their private values residing in a wire bundle from a smaller mode to a larger mode while maintaining the privacy – principals in larger mode that are not in the wire bundle see an empty wire bundle $(\_)$.

Rules **T-MAKE** and **T-COMB** introduce and eliminate secret share values of type $Sh w \tau$, respectively. In both rules, the type of share being introduced or eliminated carries the principal set of the current mode, to enforce that all sharing participants are present when the shares are combined. Values within shares must be flat and first-order so that their bit-level representation size can be determined. Finally, rule **T-SUBE** is the subsumption rule for expressions.

**V. OPERATIONAL SEMANTICS**

We define two distinct operational semantics for $\lambda_{W}$ programs, each with its own role and advantages. The single-threaded semantics of $\lambda_{W}$ provides a deterministic view of execution that makes apparent the synchrony of multi-party protocols. The multi-threaded semantics of $\lambda_{W}$ provides a non-deterministic view of execution that makes apparent the relative parallelism and privacy of multi-party protocols. In Section VI, we state and prove correspondence theorems between the two semantics.

### A. Single-threaded semantics

**Wysteria**’s single-threaded semantics defines a transition relation for machine configurations (just configurations for short). A configuration $C$ consists of a designated current mode $M$, and four additional run-time components: a store $\sigma$, a stack $\kappa$, an environment $\psi$ and a program counter $e$ (which is an expression representing the code to run next).

- **Configuration** $C ::= M\{\sigma; \kappa; \psi; e\}$
- **Store** $\sigma ::= \cdot | \sigma(\ell : M v_1, \ldots, v_k)$
- **Stack** $\kappa ::= \cdot | \kappa :: (M; \psi; x.e) | \kappa :: (\psi; x.e)$
- **Environment** $\psi ::= \cdot | \psi(\ell \mapsto M v) | \psi(x \mapsto v)$

A store $\sigma$ models mutable arrays, and consists of a finite map from (array) locations $\ell$ to value sequences $v_1, \ldots, v_k$. Each entry in the store additionally carries the mode $M$ of the allocation, which indicates which parties have access. A stack $\kappa$ consists of a (possibly empty) list of stack frames, of which there are two varieties. The first variety is introduced by delegations (let bindings with mode annotations) and consists of a mode, an environment (which stores the values of local variables), and
Fig. 5. $\lambda_{W}$: operational semantics of single-threaded configurations
a return continuation, which is an expression containing one free variable standing in for the return value supplied when the frame is popped. The second variety is introduced by regular let-bindings where the mode is invariant; it consists of only an environment and a return continuation.

An environment \( \psi \) consists of a finite map from variables to closed values. As with stores, each entry in the environment additionally carries a mode \( M \) indicating which principals have access and when (whether in secure or in parallel blocks). Note that we extend the definition of values \( v \) from Figure 1 to include several new forms that appear only during execution. During run-time, we add forms to represent empty party sets (written \( \cdot \)), empty wire bundles (written \( \cdot \)), single-principal wire bundles (\( \{ p : v \} \)), wire bundle concatenation (\( v_1 \cdot v_2 \)), array locations (\( I \)), and closures (written \( \text{clos} (\psi; \lambda x.e) \) and \( \text{clos} (\psi; \text{fix} f . \lambda x.e) \)).

Our “environment-passing” semantics (in contrast to a semantics based on substitution) permits us to directly recover the multi-threaded view of each principal in the midst of single-threaded execution. Later, we exploit this ability to show that the single and multi-threaded semantics enjoy a precise correspondence. If we were to substitute values for variables directly into the program text these distinct views of the program’s state would be tedious or impossible to recover.

Figure 5 gives the single-threaded semantics in the form of three judgements. The main judgement \( C_1 \rightarrow C_2 \) is located at the figure’s top and can be read as saying configuration \( C_1 \) steps to \( C_2 \). Configuration stepping uses an ancillary judgement for local stepping (lowest in figure). The local stepping judgement \( \sigma_1 : \psi_1 
\rightarrow \sigma_2 ; \psi_2 \) covers all (common) cases where neither the stack nor the mode of the configuration change. This judgement can be read as under store \( \sigma_1 \) and environment \( \psi_1 \), expression \( e_1 \) steps at mode \( M \) to \( \sigma_2, \psi_2 \) and \( v_2 \). Most configuration stepping rules are local, in the sense that they stay within one mode and do not affect the stack.

Configurations manage their current mode in a stack-based discipline, using their stack to record and recover modes as the thread of execution enters and leaves nested parallel and secure blocks. The rules of \( C_1 \rightarrow C_2 \) handle eight cases: Local stepping only (\( \text{STPC-LOCAL} \)), regular let-binding with no delegation annotation (\( \text{STPC-LET} \)), delegation to a parallel block (\( \text{STPC-DELPAR} \)), delegation to secure block from a secure or parallel block (\( \text{STPC-DELPSEC} \) and \( \text{STPC-DELPSEC} \), respectively), entering a secure block (\( \text{STPC-SECENTER} \)) and handling return values using the two varieties of stack frames (\( \text{STPC-POSTK1} \) and \( \text{STPC-POSTK2} \)). We discuss local stepping shortly, considering the other rules first.

Both delegation rules move the program counter under a let, saving the returning context on the stack for later (to be used in \( \text{STPC-POSTK} \)). Parallel delegation is permitted when the current principals are a superset of the delegated principal set; secure delegation is permitted when the sets coincide. Secure delegation occurs in two phases, which is convenient later when relating the single-threaded configuration semantics to the multi-threaded view; \( \text{STPC-SECENTER} \) handles the second phase of secure delegation.

The standard language features of \textsc{Wysteria} are covered by the first thirteen local stepping rules, including five rules to handle primitive array operations. We explain the first rule in detail, as a model for the rest. Case analysis (\( \text{STPL-Case} \)) branches based on the injection, stepping to the appropriate branch and updating the environment with the payload value of the injected value \( v' \). The incoming environment \( \psi \) closes the (possibly open) scrutinee value \( v \) of the case expression using value bindings for the free variables accessible at the current mode \( M \). We write the closing operation as \( \psi[v]_M \) and show selected rules in Figure 6 (complete rules are in the Appendix Figure 13). Note the second rule makes values bound in larger modes available in smaller modes. The rule \( \text{STPL-CASE} \) updates the environment using the closed value, adding a new variable binding at the current mode. The remainder of the rules follow a similar pattern of environment usage.

Projection of pairs (\( \text{STPL-FST,STPL-SND} \)) gives the first or second value of the pair (respectively). Binary operations close the operands before computing a result (\( \text{STPL-BINOP} \)). Lambdas and fix-points step similarly. In both cases a rule closes the expression, introducing a closure value with the current environment (\( \text{STPL-LAMBDA} \) and \( \text{STPL-FIX} \)). Their application rules restore the environment from the closure, update it to hold the argument binding, and in the case of the fix-points, a binding for the fix expression. The mode \( p(w) \) enforces that (potentially unguarded) recursion via \( \text{STPL-FIX} \) may not occur in secure blocks.

The array primitives access or update the store, which remained invariant in all the cases above. Array creation (\( \text{STPL-ARRAY} \)) updates the store with a fresh array location \( l \); the location maps to a sequence of \( v_1 \) copies of initial value \( v_2 \). The fresh location is chosen (deterministically) by the auxiliary function \( \text{next}_M(\sigma) \). Array selection (\( \text{STPL-SELECT,STPL-SEL-ERR} \)) projects a value from an array location by its index. The side conditions of \( \text{STPL-SELECT} \) enforce that the index is within range. When out of bounds, \( \text{STPL-SEL-ERR} \) applies instead, stepping the program text to \( \text{error} \), indicating that a fatal (out of bounds error) has occurred. We classify an \( \text{error} \) program as \textit{halted} rather than stuck. As with (“successfully”) halted programs that consist only of a return value, the \( \text{error} \) program is intended to lack any applicable stepping rules. Array updating (\( \text{STPL-UPDATE,STPL-UPD-ERR} \)) is similar to projection, except that it updates the store with a new value at one index (and the same values at all other indices). As with projection, an out-of-bounds array index results in the \( \text{error} \) program.

For secret sharing, the two rules \( \text{STPL-MAKESH} \) and \( \text{STPL-COMBSH} \) give the semantics of \textit{makesh} and \textit{combsh}, respectively. Both rules require secure computation, indicated by the mode above the transition arrow. In \( \text{STPL-MAKESH} \), the argument value is closed and “distributed” as a share value \( \text{sh} \; w \; v' \) associated with the principal set \( w \) of the current mode \( s(w) \). \( \text{STPL-COMBSH} \) performs the reverse operation, “combining” the share values of each principal of \( w \) to recover.
The remaining local stepping rules support wire bundles and their combinator. STPL-WIRE introduces a wire bundle for given party set \( w \), mapping each principal in the set to the argument value (closed under the current environment). STPL-WCOPY is a no-op: it yields its argument wire bundle. STPL-PARPROJ projects from wire bundles in a parallel mode when the projected principal is present and alone. STPL-SECPROJ projects from wire bundles in a secure mode when the projected principal is present (alone or not).

The wire combinator rules follow a common pattern. For each of the three combinator, there are two cases for the party set \( w \) that indexes the combinator: either \( w \) is empty, or it consists of at least one principal. In the empty cases, the combinator reduce according to their respective base cases: Rules STPL-WAPP1 and STPL-WAPS1 both reduce to the empty wire bundle, and STPL-WFOLD1 reduces to the accumulator value \( v_2 \). In the inductive cases there is at least one principal \( p \). In these cases, the combinators each unfold once for \( p \) (we intentionally keep the order of this unfolding non-deterministic, so all orderings of principals are permitted). Rule STPL-WAPP2 unfolds a parallel-mode wire application for \( p(\{p\}) \), creating let-bindings that project the argument and function from the two wire bundle arguments, perform the function application and recursively process the remaining principals; finally, the unfolding concatenates the resulting wire bundle for the other principals with that of \( p \). Rule STPL-WAPS2 is similar to STPL-WAPP2, except that the function being applied \( v_2 \) is not carried in a wire bundle (recall that secure blocks forbid functions within wire bundles). Finally, STPL-WFOLD2 projects a value for \( p \) from \( v_2 \) and applies the folding function \( v_3 \) to the current accumulator \( v_2 \), the principal \( p \), the projected value. The result of this application is used as the new accumulator in the remaining folding steps.

### B. Multi-threaded semantics

Whereas the single-threaded semantics of \( \lambda_{\text{w}} \) makes synchrony evident, in actuality a WYSISTERIA program is run as a distributed program involving distinct computing principals. We make this multi-agent view apparent in a multi-threaded semantics of \( \lambda_{\text{w}} \) which defines the notion of a protocol:

\[
\begin{align*}
\text{Protocol} & \quad \pi ::= \varepsilon | \pi_1 \cdot \pi_2 | A \\
\text{Agent} & \quad A ::= p\{\sigma; \kappa; \psi; e\} | S^{(w_2)}\{\sigma; \kappa; \psi; e\}
\end{align*}
\]

A protocol \( \pi \) consists of a (possibly empty) sequence of agents \( A \). There are two varieties of agents. First, principal agents are written \( p\{\sigma; \kappa; \psi; e\} \) and correspond to the machine of a single principal \( p \). Second, secure agents are written \( S^{(w_2)}\{\sigma; \kappa; \psi; e\} \) and correspond to a secure block for principals \( w_2 \), where \( w_1 \) is the subset of these principals still waiting for their result. Both varieties of agents consist of a store, stack, environment and expression—the same components as the single-threaded configurations described above. We note that in the rules discussed below, we treat protocols as commutative monoids, meaning that the order of composition does not matter, and that empty protocols can be freely added and removed without changing the meaning.

Figure 7 defines the stepping judgement for protocols \( \pi_1 \rightarrow \pi_2 \), read as protocol \( \pi_1 \) steps to protocol \( \pi_2 \). Rule STPP-PRIVATE steps principal \( p \)’s computing agent in mode \( p(\{p\}) \) according to the single-threaded semantics. We note that this rule covers nearly all parallel mode code, by virtue of parallel mode meaning “each principal does the same thing in parallel.” However, single-threaded rules involving delegation effects, STPP-Delpar and STPP-Sccenter are different in multi-threaded semantics.

Parallel delegation reduces by case analysis on the agent’s principal \( p \). Rule STPP-Present applies when \( p \) is a member of the delegated set. In that case, \( p \) simply reduces by pushing \( e_2 \) on the stack and continue to \( e_1 \). When \( p \) is not a member of the delegated set, rule STPP-Absent entirely skips the first nested expression and continues with \( e_2 \). The type system ensures that in this case, \( p \) never uses \( x \), and hence does not needs its binding in the environment.

To see the effect of these rules, consider the following code, which is like the millionaires’ example we considered earlier.

\[
\begin{align*}
& e = \begin{cases} \\
& \text{let } x_1 \equiv p(\{\text{Alice}\}) \equiv \text{read}() \text{ in} \\
& \text{let } x_2 \equiv p(\{\text{Bob}\}) \equiv \text{read}() \text{ in} \\
& \text{let } x_3 = (\text{wire}_{\{\text{Alice}\}}(x_1)) \oplus (\text{wire}_{\{\text{Bob}\}}(x_2)) \text{ in} \\
& \text{let } x_4 = s(\{\text{Alice, Bob}\}, x_3[\text{Alice}] > x_3[\text{Bob}]) \text{ in}\\
& x_4
\end{cases}
\end{align*}
\]

To start, both Alice and Bob start running the program (call it \( e \)) in protocol \( \{\cdots; \cdots; e\} \cdot \text{Bob} \{\cdots; \cdots; e\} \). Consider evaluation for Bob’s portion. The protocol will take a step according to STPP-Absent (via STPP-Frame) since the first let binding is for Alice (so Bob is not permitted to see the result). Rule STPP-Present binds the \( x_2 \) to whatever is read from Bob’s console; suppose it is 5. Then, Bob will construct the wire bundle \( x_3 \), where Alice’s binding is absent and Bob’s binding \( x_2 \) is 5.

At the same time, Alice will evaluate her portion of the protocol similarly, eventually producing a wire bundle where her value for \( x_1 \) is whatever was read in (6, say), and Bob’s binding is absent. (Of course, up to this point each of the steps of one party might have been interleaved with steps of the other.) The key is that elements of the joint protocol that are private to one party are hidden from the others, in fact totally absent from others’ local environments. Now, both are nearly poised to delegate to a secure block.

Secure delegation reduces in a series of phases that involve multi-agent coordination. In the first phase, the principal agents involved in a secure block each reduce to a secure expression \( \text{secure}_{\pi}(e) \), using STPP-Delpsec via STPP-Private. At any point during this process, rule STPP-Begin (nondeterministically) creates a secure agent with a matching principal set \( w \) and expression \( e \). After which, each principal agent can enter their input into the secure computation via STPP-Sccenter, upon which they begin to wait, blocking until the secure block completes. Their input takes the form of their current store \( \sigma \) and environment \( \psi \), which the rule combines with that of the secure agent. We explain the combine operation below. Once all principals have entered their inputs into the secure agent, the secure agent can step via STPP-Scstep. The secure agent halts when its stack is empty and its program is a value.

Once halted, the secure block’s principals can leave with the output value via STPP-Scleave. As values may refer to variables defined in the environment, the rule first closes the
Protocol stepping: “Protocol $\pi_1$ steps to $\pi_2$.”

- **STPP-PRIVATE** $p\{\sigma_1;\psi;\epsilon;e_1\} \rightarrow p\{\sigma_2;\psi;\epsilon;e_2\}$ when $p(\{\psi\})\{\sigma_1;\psi;\epsilon;e_1\} \rightarrow p(\{\psi\})\{\sigma_2;\psi;\epsilon;e_2\}$
- **STPP-PRESENT** $p\{\sigma;\psi;\epsilon;e\}$ with $e=p(\epsilon)$ $x \in \epsilon$ in $e_2$ $p\{\sigma;\psi;\epsilon;e\}$ when $p(\{\psi\}) \subseteq \psi[w]$ and $\kappa_1 = \kappa = \psi[p(\{\psi\});\epsilon;x,e_2]$
- **STPP-ABSENT** $p\{\sigma;\psi;\epsilon;e\}$ with $e=p(\epsilon)$ $x \in \epsilon$ in $e_2$ $p\{\sigma;\psi;\epsilon;e\}$ when $p(\{\psi\}) \subseteq \psi[w]$
- **STPP-FRAME** $\pi_1 \cdot \pi_2 \rightarrow \pi'_1 \cdot \pi_2$ when $\pi_1 \rightarrow \pi'_1$
- **STPP-SECBEGIN** $e \rightarrow s(\{e\}) \{\ell_1;\ell_2;\ell_3\}$
- **STPP-SECENTER** $s(\{v\})\{\sigma;\epsilon;\psi;\epsilon\} \cdot p\{\sigma';\psi;\epsilon\} \cdot p\{\sigma;\psi;\epsilon;\epsilon\} \rightarrow s(\{v\})\{\sigma;\epsilon;\psi;\epsilon\} \cdot p\{\sigma';\psi;\epsilon\}$
- **STPP-SECSTEP** $s(\{v\})\{\sigma_1;\psi;\psi_1;e_1\} \rightarrow s(\{v\})\{\sigma_2;\psi;\psi_2;e_2\}$
- **STPP-SECLEAVE** $s(\{v\})\{\sigma';\psi;\psi;\psi\} \cdot p\{\sigma;\psi;\epsilon;\epsilon\} \rightarrow s(\{v\})\{\sigma';\psi;\psi;\psi\} \cdot p\{\sigma;\psi;\epsilon;\epsilon\}$ when $\slice_p(\psi;\psi;\psi;\psi;\psi) \sim v'$
- **STPP-SECEND** $s(\{v\})\{\sigma;\epsilon;\psi;\psi\} \rightarrow e$

---

**Value slicing:** “Value $v_2$ sliced for $p$ is $v_2$”

- $\slice_p((v_1, v_2)) \sim (v'_1, v'_2)$ when $\slice_p(v_1) \sim v'_1$
- $\slice_p((p \cdot v) \parallel v_1) \sim (p \cdot v)$ when $v \not\subseteq \dom(v_1 \parallel v_2)$
- $\slice_p(v_1 \parallel v_2) \sim -$ when $v \not\subseteq \dom(v_1 \parallel v_2)$

**Environment slicing:** “Environment $\psi$ sliced for $p$ is $\psi'$”

- $\slice_p(\psi(x \rightarrow \psi'(v))) \sim \slice_p(\psi)(x \rightarrow \psi'(v)), v \in \psi$ if $x \not\subseteq \dom(\psi'(v))$
- $\slice_p(\psi(x \rightarrow \psi'(v))) \sim \slice_p(\psi)(x \rightarrow \psi'(v)), v \not\subseteq \dom(\psi'(v))$
- $\slice_p(\psi(x \rightarrow \psi'(v))) \sim \slice_p(\psi)(x \rightarrow \psi'(v))$

---

**VI. METATHEORY**

We prove several meta-theoretical results for $\lambda_{W}$ that are relevant for mixed-mode multi-party computations. Proofs for these results can be found in Appendix B. First, we show that well-typed WYSTERIA programs always make progress (and stay well-typed). In particular, they never get stuck: they either complete with a final result, or reach a well-defined error state (due to an out of bounds array access or update, but for no other reason).

**Theorem 6.1 (Progress):** If $\Sigma \vdash C_1 : \tau$ then either $C_1 \text{ halted}$ or there exists configuration $C_2$ such that $C_1 \rightarrow C_2$.

**Theorem 6.2 (Preservation):** If $\Sigma \vdash C_1 : \tau$ and $C_1 \rightarrow C_2$, then there exists $C_2 \supseteq \Sigma$ s.t. $C_2 \vdash C_2 : \tau$.

The premise $\Sigma \vdash C_1 : \tau$ appears in both theorems and generalizes the notion of well-typed expressions to that of well-typed configurations; it can be read as under store typing $\Sigma$, configuration $C_1$ has type $\tau$. This definition involves giving a notion of well-typed stores, stacks and environments, which we omit here for space reasons (see appendix).

Next, turning to the relationship between the single- and multi-threaded semantics, the following theorem shows that every transition in the single-threaded semantics admits corresponding transitions in the multi-threaded semantics:

**Theorem 6.3 (Sound forward simulation):** Suppose that $\Sigma \vdash C_1 : \tau$ and that $C_1 \rightarrow C_2$. Then there exist $\pi_1$ and $\pi_2$ such that $\pi_1 \rightarrow^{*} \pi_2$ and $\slice_{\w}(C_i) \sim \pi_i$ (for $i \in \{1, 2\}$), where $w$ is the set of all principals.

The conclusion of the theorem uses the auxiliary slicing judgement to construct a multi-threaded protocol from a (single-threaded) configuration.

Turning to the multi-threaded semantics, the following theorem states that the non-determinism of the protocol semantics always resolves to the same outcome, i.e., given any two pairs of protocol steps that take a protocol to two different configurations, there always exists two more steps that bring these two intermediate states into a common final state:

**Theorem 6.4 (Confluence):** Suppose that $\pi_1 \rightarrow \pi_2$ and $\pi_1 \rightarrow \pi_3$, then there exists $\pi_4$ such that $\pi_2 \rightarrow \pi_4$ and $\pi_3 \rightarrow \pi_4$.

A corollary of confluence is that every terminating run of the (non-deterministic) multi-threaded semantics yields the same result.

For correspondence in the other direction (multi- to single-threaded), we can prove the following lemma.
Lemma 6.1 (Correspondence of final configurations): Let \( \Sigma \vdash \mathcal{C} : \tau \) and \( \text{slice}_w(\mathcal{C}) \sim \pi \), where \( w \) is the set of all principals. If \( \mathcal{C} \rightarrow^* \pi' \), where \( \pi' \) is an error-free terminated protocol, then there exists an error-free terminated \( \mathcal{C}' \) s.t. \( \mathcal{C} \rightarrow^* \mathcal{C}' \) and \( \text{slice}_w(\mathcal{C}') \sim \pi' \).

One of the most important consequences of these theorems is that principals running parallel to one another, and whose computations successfully terminate in the single-threaded semantics, will be properly synchronized in the multi-threaded semantics; e.g., no principal will be stuck waiting for another one that will never arrive.

We would like to prove a stronger, backward simulation result that also holds for non-terminating programs, but unfortunately it does not hold because of the possibility of errors and divergence. For example, when computing \texttt{let } x \equiv e_1 \texttt{ in } e_2 \texttt{, the single-threaded semantics could diverge or get an array access error in } e_1, \text{ and therefore may never get to compute } e_2. \text{ However, in multi-threaded semantics, principals not in } M \text{ are allowed to make progress in } e_2. \text{ Thus, for those steps in the multi-threaded semantics, we cannot give a corresponding source configuration. We plan to take up backward simulation (e.g., by refining the semantics) as future work.}

Security: As can be seen in Figure 7, the definition of the multi-threaded semantics makes apparent that all inter-principal communication (and thus information leakage) occurs via secure blocks. As such, all information flows between parties must occur via secure blocks. These flows are made more apparent by WYSERIA’s single-threaded semantics, and are thus easier to understand.

VII. Implementation

We have implemented a tool chain for WYSERIA, including a frontend, a type checker, and a run-time interpreter. Our implementation is written in OCaml, and is roughly 6000 lines of code. Our implementation supports the core calculus features (in gentler syntax) and also has named records and conditionals. To run a WYSERIA program, each party invokes the interpreter with the program file and his principal name. If the program type slice is made apparent by the interpreter dynamically generates boolean circuits from secure blocks it encounters, running them using Choi et al.’s implementation [23] of the Goldreich, Micali, and Wigderson (GMW) protocol [2], which provably simulates (in the semi-honest setting) a trusted third party. Figure 9 gives a high-level overview of WYSERIA running for four clients (labeled A, B, C and D). The remainder of this section discusses the implementation in detail.

Type checker: The WYSERIA type checker uses standard techniques to turn the declarative type rules presented earlier into an algorithm (e.g., inlining uses of subsumption to make the typing rules syntax-directed). We use the Z3 SMT solver [26] to discharge the refinement implications, encoding sets using Z3’s theory of arrays. Since Z3 cannot reason about the cardinality of sets encoded this way, we add support for \texttt{singl}(\nu) as an uninterpreted logical function \texttt{single} that maps sets to booleans. Facts about this function are added to Z3 in the rule T-PRINC.

Interpreter: When the interpreter reaches a secure block it compiles that block to a circuit in several steps. First, it must convert the block to straight-line code. It does this by expanding \texttt{wfold} and \texttt{waps} expressions (according to the now available principal sets), inlining function calls, and selectively substituting in non-wire and non-share variables from its environment. The type system ensures that, thanks to synchrony, each party will arrive at the same result.

Next, the interpreter performs a type-directed translation to a boolean circuit, taking place in two phases. In the first phase, it assigns a set of wire IDs to each value and expression, where the number of wires depends on the corresponding type. The wires are stitched together using high-level operators (e.g., ADD r1 r2 r3, where r1, r2, and r3 are ranges of wire IDs). As usual, we generate circuits for both branches of \texttt{case} expressions and feed their results to a multiplexer switched by the compiled guard expression’s output wire. Records and wire bundles are simply an aggregation of their components. Wire IDs are also assigned to the input and output variables of each principal—these are the free wire and share variables that remained in the block after the first step.

In the second phase, each high-level operator (e.g. ADD) is translated to low-level AND and XOR gates. Once again, the overall translation is assured to produce exactly same circuit, including wire ID assignments, at each party. Each host’s interpreter also translates its input values into bit representations. Once the circuit is complete it is written to disk.

At this point, the interpreter signals a local server process originally forked when the interpreter started. This server process implements the secure computation using the GMW library. This process reads in the circuit from disk and coordinates with other parties’ GMW servers using network sockets. At the end of the circuit execution, the server dumps the final output to a file, and signals back the interpreter. The interpreter then reads in the result, converts it to the internal representation, and carries on with parallel mode execution (or terminates if complete).

Secure computation extensions and optimizations: The GMW library did not originally support secret shares, but they were easy to add. We extended the circuit representation to designate particular wires as making shares (due to \texttt{makesh}(\nu) expressions) and reconstituting shares (due to \texttt{combsh}(\nu) expressions). For the former, we modified the library to dump the designated (random) wire value to disk—already a share—and for the latter we do the reverse, directing the result into the circuit. We also optimized the library’s Oblivious Transfer (OT) extension implementation for the mixed-mode setting.
We conduct two sets of experiments to study WYSTERIA’s empirical performance. First we measure the performance of several \( n \)-party example programs of our own design and drawn from the literature. We find that these programs run relatively quickly and scale well with the number of principals. Second, we reproduce two experiments from the literature that demonstrate the performance advantage of mixed-mode vs. monolithic secure computation.\(^6\)

**Secure computations for \( n \) parties**: We have implemented several \( n \)-party protocols as WYSTERIA functions that are *generic* in the participating principal set (whose identity and size can both vary). The Richest protocol computes the richest principal, as described in Section II. The GPS protocol computes, for each participating principal, the other principal that is nearest to their location; everyone learns their nearest neighbor without knowing anyone’s exact location. The Auction protocol computes the high bidder among a set of participating principals, as well as the second-highest bid, which is revealed to everyone; only the auction holder learns who is the winning bidder. Finally, we have implemented the two-round bidding game from Section II for multiple principals. Recall that this example crucially relies on WYSTERIA’s notion of secret shares, a high-level abstraction that existing SMC languages lack.

Figure 10(a) shows, for varying numbers of principals, the elapsed time to compute these functions. We can see each of these computations is relatively fast and scales well with increasing numbers of parties.

**Mixed-mode secure computations**: To investigate the performance advantages of mixed-mode secure computations, we study two functions that mix modes: two-party median computes the median of two principals’ elements, and two-party intersect is a PSI protocol that computes the intersection of two principals’ elements. In both cases, we compare the mixed-mode version of the protocol with the secure-only versions, which like FairPlayMP only use a single monolithic secure block. We chose these protocols because they have been studied in past literature on secure computation [5], [7], [22]; both protocols enjoy the property that by mixing modes, certain computation steps in the secure-only version can either be off-loaded to local computation (as in median) or avoided altogether (as in intersect), while providing the same privacy guarantees.

The participating principals A and B store their (sorted, distinct) input pairs in wire bundles \( w_1 \) and \( w_2 \) such that \( w_1 \) contains A’s smaller numbers and \( w_2 \) contains B’s larger ones. First, the protocol compares the smaller numbers. Depending on this comparison, the protocol discards one input for each principal. Then, it compares the remaining two numbers and the smaller one is chosen as the median (thus preferring the lower-ranked element when there is an even number).

Under certain common assumptions [22], the following mixed-mode version equivalently computes median with the same security properties.

\[
\begin{align*}
&\text{let } m = \text{sec}((A,B)) = \\
&\quad \text{let } x_1 = (\text{fst } w_1[A]) \quad \text{let } x_2 = (\text{snd } w_1[A]) \quad \text{let } y_1 = (\text{fst } w_2[B]) \quad \text{let } y_2 = (\text{snd } w_2[B]) \\
&\quad \text{let } b_1 = x_1 \leq y_1 \quad \text{let } b_2 = x_2 \leq y_2 \\
&\quad \text{let } b_1 = b_2 \quad \text{let } x_3 = \text{if } b_1 \text{ then } x_2 \text{ else } x_1 \quad \text{let } y_3 = \text{if } b_2 \text{ then } y_1 \text{ else } y_2 \\
&\quad \text{let } b_2 = x_3 \leq y_3 \quad \text{let } b = \text{if } b_2 \text{ then } x_3 \text{ else } y_3 \\
&\quad \text{in } m
\end{align*}
\]

The key difference compared with the secure-only version is that the conditional assignments on lines 3 and 4 need not be done securely. Rather, the protocol reveals \( b_1 \) and \( b_2 \), allowing each principal to perform these steps locally. Past work as shown that this change still preserves the final *knowledge profile* of each party, and is thus equally secure in the semi-honest setting [22].

Figure 10(b) compares the performance of mixed-mode median over secure-only median for varying sizes of inputs (generalizing the program above).

The elapsed time for mixed-mode median remains comparatively fixed, even as input sizes increase exponentially. By comparison, secure-only median scales poorly with increasing input sizes. This performance difference illustrates the (sometimes dramatic) benefit of supporting mixed-mode computations.

\(^6\)We ran all our experiments on Mac OS X 10.9, with 2.8 GHz Intel Core Duo processor and 4GB memory. To isolate the performance of WYSTERIA from that of I/O, all the principals run on the same host, and network communication uses local TCP/IP sockets.
Private set intersection: In intersect, two principals compute the intersection of their private sets. The set sizes are assumed to be public knowledge. As with median, the intersect protocol can be coded in two ways: a secure-only pairwise comparison protocol performs \( n_1 \times n_2 \) comparisons inside the secure block which result from the straight-line expansion of two nested loops. Huang et al. [7] propose two optimizations to this naïve pairwise comparison protocol. First, when a matching element is found, the inner loop can be short circuited, avoiding its remaining iterations. Second, once an index in the inner loop is known to have a match, it need not be compared in the rest of the computation. We refer the reader to their paper for further explanation. We note that WYSTERIA allows programmers to easily express these optimizations in the language, using built-in primitives for expressing parallel-mode loops and arrays.

Figure 10(c) compares the secure-only and mixed-mode versions of intersect. For the mixed-mode version, we consider three different densities of matching elements: 0.5, 0.75, and 0.95 (where half, three-quarters, and 95% of the elements are held in common). For the unoptimized version, these densities do not affect performance, since it always executes all program paths, performing comparisons for every pair of input elements. As can be seen in the figure, as the density of matching elements increases, the mixed-mode version is far more performant, even for larger input sizes. By contrast, the optimization fails to improve performance at lower densities, as the algorithm starts to exhibit quadratic-time behavior (as in the secure-only version).

All the examples presented here, and more (including a prototype WYSTERIA program to deal cards for mental card games), are given in full in Appendix A.

IX. Related Work

Several research groups have looked at compiling multiparty programs in high-level languages to secure protocols. Our work is distinguished from all of these in several respects.

Support for multi-party \((n > 2)\) computations: Our language design has carefully considered support for secure computations among more than two parties. Most prior work has focused on the two-party case. Fairplay [17] compiles a garbled circuit from a Pascal-like imperative program. The entry point to this program is a function whose two arguments are players, which are records defining each participant’s expected input and output type. More recently, Holzer et al. [18] developed a compiler for programs with similarly specified entry points, but written in (a subset of) ANSI C. Alternatively, at the lowest level there are libraries for building garbled circuits directly, e.g., those developed by Malka [13], Huang et al. [14], and Mood et al. [15]. These lines of work provide important building blocks for the back end of our language (in the two party case).

The only language of which we are aware that supports \(n > 2\) parties is FairplayMP [19]. Its programs are similar to those of FairPlay, but now the entry point can contain many player arguments, including arrays of players, where the size of the array is statically known. Our wire bundles have a similar feel to arrays of players. Just as FairPlayMP programs can iterate over (arbitrary-but-known-length) arrays of players in a secure computation, we provide constructs for iterating over wire bundles. Unlike the arrays in FairplayMP, however, our wire bundles have the possibility of representing different subsets of principals’ data, rather than assume that all principals are always present; moreover, in WYSTERIA these subsets can themselves be treated as variable.

Support for mixed-mode computations: All of the above languages specify secure computations in their entirety, e.g., a complete garbled circuit, but much recent work (including this work) has focused on the advantage of mixed-mode computations.

As first mentioned in Section I, L1 [24] is an intermediate language for mixed-mode SMC, but is limited to two parties. Compared to L1, WYSTERIA provides more generality and ease of use. Further, WYSTERIA programmers need not be concerned with the low-level mechanics of inter-party communication and secret sharing, avoiding entire classes of potential misuse (e.g., when two parties wait to receive from each other at the same time, or when they attempt to combine shares of distinct objects).

PCF [16] is a circuit format language for expressing mixed-mode secure computations for two parties. As with L1, it allows programmers to distinguish secure computation from surrounding computation that is run locally, in the clear. It also suffers from limitations that are similar to those of L1, in that it is (by design) very low level, and in that it lacks abstractions for supporting multiple parties, as well as a formal semantics.

SMCL [8] is a language for secure computations involving a replicated client and a shared “server” which represents secure multiparty computations. Our approach is less rigid in its specification of roles: we have secure and parallel computations involving arbitrary numbers of principals, rather than all of them, or just one, as in SMCL. SMCL provides a type system that aims to enforce some information flow properties modulo declassification. SMCL’s successor, VIFF [27], reflects the SMCL computational model as a library/DSL in Python, but lacks type-based guarantees.

Liu et al. define a typed intermediate language for mixed-mode SMC protocols [25]. However, their approach is limited to two parties and their proposed language is simplistic in comparison to WYSTERIA, e.g., it lacks function abstractions and is thus not suitable for expressing reusable library code. Given each variable’s classification as public, secret (e.g., used only within an SMC), or private, their compiler can produce a mixed mode protocol guaranteed to have the same security as a monolithic secure protocol. They use an information flow control-style type system to prohibit illegal flows of information between modal computations. By contrast, WYSTERIA makes no attempt to relate the security properties of mixed-mode and non-mixed-mode versions of a protocol; instead, one must use a separate analysis for this purpose (e.g. [22]). We note that WYSTERIA could safely use Liu et al.’s novel “RAM-model secure” computation protocol as a means to implement secure blocks among two parties.

SMCs as cloud computations: Another line of research in SMCs deals with a client-server setting, where client wants to run a function over his private input using untrusted servers (e.g. in a cloud). To protect confidentiality of his data, the client distributes secret shares of his input among the servers. The servers run same function, but use their own shares. Finally, they send the output shares to the client, who then recovers
the clear output value. Launchbury et al. [12] present a table-lookup based optimization for such SMC protocols, that aims at minimizing the cost incurred by expensive operations such as multiplication and network communication between servers. Mitchell et al. [11] give an expressive calculus for writing such functions. Their calculus is mixed-mode, but only in terms of data—the programs can use both encrypted (private) and non-encrypted (public) values. They give an extended information flow type system that rejects programs that cannot be run on a secure computation platform (such as homomorphic encryption).

In WYSTERIA, the above client-server setting can be expressed as a monolithic secure block to be run by the servers, each of which holds secret shares of client’s input. As we have shown in the paper, we can express more general mixed-mode SMCs.

Other language proposals: The TASTY compiler produces secure computations that may combine homomorphic encryption and garbled circuits [28]. Its input language, TASTYL, requires explicit specification of communication between parties, as well as the means of secure computation, whereas in our approach, such concerns are handled automatically (during runtime compilation of generated circuits). Kerschbaum et al. [29] explore automatic selection of mixed protocols consisting of garbled circuits and homomorphic encryption. Jif/Split enables writing multi-party computations in Java as (conceptually) single-threaded programs [30]. It offers compiler support for dividing Java programs into pieces to run on different hosts, based on information-flow analysis and explicit declassifications. Unlike our work, Jif/Split runs statically (at compile time), depends on real-life trusted third parties, and lacks language abstractions and run-time techniques for employing secure computations without a trusted third party.

X. Conclusion

This paper presents WYSTERIA, the first programming language designed for expressing mixed-mode computations for multiple parties. In contrast to prior work, multi-party protocols in WYSTERIA can be expressed generically, and may perform dynamic decisions about each principal’s role within a protocol. WYSTERIA’s type system ensures that well-typed programs never misuse the language’s abstractions. Further, WYSTERIA programs are concise and readable, since they can be interpreted through a (conceptually simple) single-threaded semantics, as well as a (more realistic) multi-threaded semantics. We show formally that these two views coincide. We present writing multi-party computations in Java as (conceptually) well as the means of secure computation, whereas in our purposes notwithstanding any copyright notation hereon.

REFERENCES

[27] “VIFF, the virtual ideal functionality framework.” http://viff.dk/.
A. Additional Code Examples and Formal Definitions

Below, we list the Wysteria code for computing the private set intersection of two parties’ integer array inputs:

```plaintext
let eq = sec(x,y) = ( testeq w[x] w[y] ) in
if eq then ( out [ i ]
2 nat
9
let w = wire x xv
let yv = par(y) = inp [ y ] [ j ] in
7
for j = 0 to ( n )
1
psi_cnt : ( x: princ )
1
1
let xv = par(x) = inp [ x ] [ i ] in
7
let w = wire x xv + wire y yv in
11
let csh = c cnt in
11
c
let c = sec(x,y) = begin
7
let eq = ( testeq w[x] w[y] ) in
14
if eq then makesh ( c + 1 )
15
else makesh c
16
end
17
in cnt := c
20
done

The first two lines give the type of the function; the following line gives names to its arguments. It is parameterized by two principals x and y, an equivalence-testing function for integers testeq of type int \rightarrow int \rightarrow bool, a natural number n, a wire bundle of integer arrays inp of type W(x,y)[int], and an array of integer-valued wire bundles out to hold the output of type W(x,y)[int]. The final arrow indicates that the two principals must be present to compute the protocol; the final return type unit is like void in C in that no additional return value is given as output. Rather, the output of the protocol is given by its side-effects to the wire bundle array out, which it writes each time a pair of elements from inp are deemed equivalent by the testing function testeq.

The computation’s structure mixes modes in a synchronous looping pattern that consists of two doubly-nested for loops. The indices of the loops i and j index each pairing of the two principals’ n input elements. The loop body projects x’s i-th element and y’s j-th element, creating a wire bundle of the pair to use as input in a secure test for equivalence. The secure computation uses testeq to produce a boolean-valued output that the protocol reveals to both principals. When equivalent, the principals side-effect out, updating its i-th entry to contain the equivalent pair. After filling the i-th entry, they advance the outer loop, continuing to compare the (i+1)th element of x with all the elements of y. When not equivalent, they update nothing and continue to compare by checking the i-th element of x with the (i+1)th element of y.

The protocol above is mixed-mode in that it exploits both parallel and secure modes. The two loops of the protocol occur in parallel mode and consist of the two principals doing the same thing locally at the same time; in this mode they are in sync, but not directly communicating. The secure computation on line 14 requires the principals to synchronize requiring communication and direct coordination.

Throughout the course of the psi protocol above, both parties learn which elements of theirs are present in the intersection. Instead, the parties may want to be more conservative, only revealing (at the end of the protocol), the cardinality of the intersection, i.e., a count of how many elements they have in common, but not the precise identity of those elements. This variation can also easily be expressed in Wysteria:

psi_cnt : ( x: princ ) \rightarrow ( y: princ ) \rightarrow ( int \rightarrow int \rightarrow bool ) \rightarrow nat \rightarrow W [ x, y ] [ int ] \rightarrow \text{Sh}(x,y) nat

As can be seen, psi_cnt is similar in structure to psi, but also slightly different. Starting with its type and arguments, we note that it does not output an array to update like psi. Rather, it conceals the elements of the intersection and reveals only a final count as a share, typed by the Sh connective. The returned share can be combined in a subsequent protocol, or revealed, as determined by the calling context. Like psi, the function body of psi_cnt consists of two nested loops. As before, the secure block in the loop body performs equivalence tests; but unlike psi, the result of each test is not revealed. Rather, the count cnt consists of shares, and serves as an additional parameter to both loops. Each test is used to securely update cnt; this consists of combining secret shares of the prior count (using combsh), and creating new shares for the updated count, either incremented by one or unchanged (using makesh). Since the output of the tests are not directly revealed, the loops both proceed to completion; there is no short-circuiting in the inner loop, as there was for psi.

/* Nearest neighbors */
nns : ( all: ps ) \rightarrow ( ps( \nu \subseteq all \land single \nu ) ) →
W all int \rightarrow ( par(all) ) → W all ps( \nu \subseteq all \land single \nu )

let out = sec(all) =
let answers =
  waps(coords, \lambda pcoord.
  wfold(coords, (defaultnn MAX),
  \lambda nn:(ps x int). \lambda q. \lambda pcoord.
  if pcoord = qcoord then nn
  else
    let dist = \nu (qcoord) in
    if dist < snd(nn) then (q, dist)
    else
      (q, dist)
    ))
  in
  wcopy out

This is the code for generic nearest neighbor application. The function takes as input a wire bundle mapping each party to its location, and returns another wire bundle mapping each party to its nearest neighbor (exact locations are kept secret).

The function consists of two loops: one using waps, and the other using wfold. The waps loop iterates over the input wire bundle, and performs a wfold operation for each mapping. The wfold operation folds over the input wire bundle and updates the accumulator, which is a record containing the nearest party name and its location. The location is filtered from the output using another waps loop at the end. Since this function is written generically, the returned wire bundle needs to be copied to T using wcopy. Other examples (second-price auction, etc.) are available online: http://bitbucket.org/aseemr/wysteria.
\begin{align*}
\Gamma \vdash v : \tau \\
\text{(Value typing)}
\end{align*}

\begin{align*}
\text{TN-VAR} & \quad \begin{array}{c}
x : \tau \in \Gamma \quad \Gamma \vdash \tau \\
\end{array} \\
\text{TN-UNIT} & \quad \begin{array}{c}
j \in \{1, 2\} \\
\end{array} \\
\text{TN-INJ} & \quad \begin{array}{c}
\Gamma \vdash v : \tau_1 \\
\end{array} \\
\text{TN-PROD} & \quad \begin{array}{c}
\tau_j \text{ isFlat} \\
\end{array} \\
\text{TN-PRINC} & \quad \begin{array}{c}
\tau_j \text{ isFlat} \\
\end{array} \\
\text{TN-PSONE} & \quad \begin{array}{c}
\Gamma \vdash w : \text{ps} (\text{singl}(\nu)) \\
\end{array} \\
\text{TN-PSUNION} & \quad \begin{array}{c}
\Gamma \vdash \{w\} : \text{ps} (\nu = \{w\}) \\
\end{array} \\
\text{TN-PSVAR} & \quad \begin{array}{c}
\Gamma \vdash x : \phi \\
\end{array} \\
\text{TN-SUB} & \quad \begin{array}{c}
\tau' < \tau \\
\end{array} \\
\end{align*}

\begin{align*}
\text{Fig. 11.} & \quad \text{Value typing with no place.}
\end{align*}

\begin{align*}
\text{EFFDEL-EMPTY} & \quad \begin{array}{c}
\Gamma \vdash M > \epsilon \\
\end{array} \\
\text{EFFDEL-MODE} & \quad \begin{array}{c}
\Gamma \vdash M > \epsilon, N \\
\end{array} \\
\end{align*}

\begin{align*}
\text{First order types} & \quad \begin{array}{c}
\tau \text{ IsFO} \\
\end{array} \\
\text{F-UNIT} & \quad \begin{array}{c}
\text{unit} \text{ isFO} \\
\end{array} \\
\text{W-UNIT} & \quad \begin{array}{c}
\text{unit} \text{ isFlat} \\
\end{array} \\
\end{align*}

\begin{align*}
\text{Wire and Share free types} & \quad \begin{array}{c}
\text{SIN-FO} \\
\end{array} \\
\text{SIN-ARROW} & \quad \begin{array}{c}
\tau_1 \text{ IsSecIn} \\
\end{array} \\
\end{align*}

\begin{align*}
\text{Valid input types for secure blocks} & \quad \begin{array}{c}
\text{W-SUM} \\
\text{W-PROD} \\
\text{W-ARR} \\
\end{array} \\
\end{align*}

\begin{align*}
\text{(Well formed type)} & \quad \begin{array}{c}
\text{WF-UNIT} \\
\text{WF-PROD} \\
\end{array} \\
\end{align*}

\begin{align*}
\text{(Well formed refinement)} & \quad \begin{array}{c}
\text{WFREF-TRUE} \\
\text{WFREF-SINGL} \\
\text{WFREF-SUB} \\
\text{WFREF-EQ} \\
\text{WFREF-CONJ} \\
\end{array} \\
\end{align*}

\begin{align*}
\text{(Well formed place)} & \quad \begin{array}{c}
\text{WF-UNIT} \\
\text{WF-PROD} \\
\text{WF-PRINC} \\
\end{array} \\
\end{align*}

\begin{align*}
\text{Fig. 12.} & \quad \text{Auxiliary judgements used in the type system.}
\end{align*}
Fig. 13. Environment lookup.
Environment slicing: “Environment $\psi$ sliced for $p$ is $\psi'$”

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sliceEnv~}(\psi) \sim \psi'$</td>
<td>$\text{sliceEnv~}(\psi \cdot x \mapsto p(w)) \sim \psi'(x \mapsto p(w)) \text{ slice } (\psi(v))$ when $p \in w$ and $\text{slice}_p(\psi) \sim \psi'$</td>
</tr>
<tr>
<td>$\text{sliceEnv~}(\psi) \sim \psi'$</td>
<td>$\text{sliceEnv~}(\psi \cdot x \mapsto p(w)) \sim \psi'$ when $p \not\in w$ and $\text{slice}_p(\psi) \sim \psi'$</td>
</tr>
<tr>
<td>$\text{sliceEnv~}(\psi) \sim \psi'$</td>
<td>$\text{sliceEnv~}(\psi \cdot x \mapsto v') \sim \psi'(x \mapsto v') \text{ slice } (\psi(v))$ when $\text{slice}_p(\psi) \sim v'$ and $\text{slice}_p(\psi) \sim \psi'$</td>
</tr>
</tbody>
</table>

Stack slicing: “Stack $\kappa$ sliced for $p$ is $\kappa'$”

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sliceSTK~}(\psi) \sim \psi'$</td>
<td>$\text{sliceSTK~}(\psi \cdot p(w) \cup \psi(x)) \sim \psi'(x \mapsto p(w)) \text{ slice } (\psi(v))$ when $\text{slice}_p(\psi) \sim \psi'$ and $\text{slice}_p(\psi) \sim \psi'$</td>
</tr>
<tr>
<td>$\text{sliceSTK~}(\psi) \sim \psi'$</td>
<td>$\text{sliceSTK~}(\psi \cdot p(w) \cup \psi(x)) \sim \psi'(x \mapsto \psi(v)) \text{ slice } (\psi(v))$ when $\text{slice}_p(\psi) \sim \psi'$ and $\text{slice}_p(\psi) \sim \psi'$</td>
</tr>
</tbody>
</table>

Store slicing: “Store $\sigma$ sliced for $p$ is $\sigma'$”

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sliceST~}(\psi) \sim \sigma'$</td>
<td>$\text{sliceST~}(\psi \cdot \ell(p(w), v_1, \ldots, v_k) \cup \psi(x)) \sim \sigma'(x \mapsto \ell(p(w), v_1, \ldots, v_k)) \text{ slice } (\psi(v))$ when $p \in w$ and $\text{slice}_p(\psi) \sim \sigma'$ and $\text{slice}_p(v_i) \equiv v'_i$ when $p \not\in w$ and $\text{slice}_p(\psi) \sim \sigma'$</td>
</tr>
</tbody>
</table>

Configuration slicing: “Configuration $C$ sliced for $w$ is $\pi$”

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sliceCfg~}(\psi) \sim \pi$</td>
<td>$\text{sliceCfg~}(\psi) \sim \pi$ when $\text{slice}_p(\psi) \sim \pi$ and $\text{slice}_p(\psi) \sim \pi$</td>
</tr>
<tr>
<td>$\text{sliceCfg~}(\psi) \sim \pi$</td>
<td>$\text{sliceCfg~}(\psi) \sim \pi$ when $p \in w$ and $\text{slice}_p(\psi) \sim \sigma'$ and $\text{slice}_p(\psi) \sim \kappa'$</td>
</tr>
<tr>
<td>$\text{sliceCfg~}(\psi) \sim \pi$</td>
<td>$\text{sliceCfg~}(\psi) \sim \pi$ when $p \not\in w$ and $\text{slice}_p(\psi) \sim \sigma'$</td>
</tr>
</tbody>
</table>

Value slicing: “Value $v_1$ sliced for $p$ is $v_2$”

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sliceVal~}(\psi) \sim v_2$</td>
<td>$\text{sliceVal~}(\psi) \sim v_2$ when $\text{slice}_p(\psi) \sim v_2$</td>
</tr>
<tr>
<td>$\text{sliceVal~}(\psi) \sim v_2$</td>
<td>$\text{sliceVal~}(\psi) \sim v_2$ when $p \not\in w$ and $\text{slice}_p(\psi) \sim v_2$</td>
</tr>
<tr>
<td>$\text{sliceVal~}(\psi) \sim v_2$</td>
<td>$\text{sliceVal~}(\psi) \sim v_2$ when $p \not\in w$ and $\text{slice}_p(\psi) \sim v_2$</td>
</tr>
</tbody>
</table>

Value composing: “Value $v_1$ composed with $v_2$ is $v_3$”

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{composeVal~}(\psi) \sim v_3$</td>
<td>$\text{composeVal~}(\psi) \sim v_3$ when $\text{compose}_p(\psi) \sim v_3$</td>
</tr>
<tr>
<td>$\text{composeVal~}(\psi) \sim v_3$</td>
<td>$\text{composeVal~}(\psi) \sim v_3$ when $p \not\in w$ and $\text{compose}_p(\psi) \sim v_3$</td>
</tr>
<tr>
<td>$\text{composeVal~}(\psi) \sim v_3$</td>
<td>$\text{composeVal~}(\psi) \sim v_3$ when $p \not\in w$ and $\text{compose}_p(\psi) \sim v_3$</td>
</tr>
</tbody>
</table>

Environment composing: “Environment $\psi_1$ composed with $\psi_2$ is $\psi_3$”

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{composeEnv~}(\psi) \sim \psi_3$</td>
<td>$\text{composeEnv~}(\psi) \sim \psi_3$ when $\text{compose}_p(\psi) \sim \psi_3$</td>
</tr>
<tr>
<td>$\text{composeEnv~}(\psi) \sim \psi_3$</td>
<td>$\text{composeEnv~}(\psi) \sim \psi_3$ when $p \not\in w$ and $\text{compose}_p(\psi) \sim \psi_3$</td>
</tr>
<tr>
<td>$\text{composeEnv~}(\psi) \sim \psi_3$</td>
<td>$\text{composeEnv~}(\psi) \sim \psi_3$ when $p \not\in w$ and $\text{compose}_p(\psi) \sim \psi_3$</td>
</tr>
</tbody>
</table>

Fig. 14. $\lambda_W$: slicing and composing judgments.
\(\Gamma \vdash_M \nu : \tau\)

T-PS-EMP
\(\Gamma \vdash_M \text{ps}(\nu = \cdot)\)

T-EMP
\(\vdash \tau\)
\(\vdash \text{M} \vdash_M \text{ps}(\nu = \cdot)\)

T-SINGLWISE
\(M = \text{m}(w_1)\)  \(m = s \Rightarrow N = \text{s}(w_1)\)
\(m = p \Rightarrow N = \text{p}(\{p\})\)

T-WIRECAT
\(\vdash_M \text{v1 : W}_w \tau\)
\(\vdash_M \text{v2 : W}_w \tau\)
\(\Gamma \vdash_M \text{v1} \oplus \text{v2} : \text{W}_w (w_1 \cup w_2) \tau\)

T-LOC
\(\Sigma (\ell) = \tau\)
\(\vdash \tau\)
\(\vdash \text{IsFlat}\)
\(\vdash_M \ell : \text{Array} \tau\)

T-LOC
\(\vdash_M \ell : \text{Array} \tau\)

T-SH
\(\vdash_M \nu : \tau\)
\(\vdash_M \nu : \text{IsFU}\)
\(\vdash_M \nu : \text{IsFlat}\)

T-CLOS
\(\Sigma \vdash \psi \sim \Gamma\)
\(\Gamma, x : t_1 \vdash_M e : t_2; \epsilon\)
\(\vdash (x : t_1 \rightarrow t_2)\)

T-CLOS
\(\Gamma, f : y : t_1 \rightarrow t_2 \vdash_M \lambda x : e : y : t_1 \rightarrow t_2; \epsilon\)
\(\vdash (y : t_1 \rightarrow t_2)\)

T-FIXCLOS
\(\Gamma \vdash_M \; \text{fix}(\lambda x : e) : y : t_1 \rightarrow t_2\)

T-SECBLK
\(\vdash_M \; \text{ps}(\nu = w')\)
\(\vdash_M \; \text{w} \; \text{secure}(e): \tau; \epsilon\)

TSTORE-EMP
\(\Sigma \vdash \sigma \; \text{wf}\)

TSTORE-LOC
\(\Sigma \vdash \sigma \; \text{wf}\)
\(\Sigma ; \vdash_M \; \text{v1 : } \tau\)

TSTORE-LOC
\(\Sigma ; \vdash_M \; \text{v1 : } \tau\)

TSTK-FRAME1
\(M = \text{m}(w)\)
\(N = \text{m}(w)\)
\(\Sigma \vdash \psi \sim \Gamma\)
\(\Gamma \vdash t_1\)
\(\Sigma ; \vdash_M \; x : t_1; \epsilon\)
\(\vdash \text{m}(w) \; \text{w} \; \text{secure}(\epsilon): \tau; \epsilon\)

TSTK-FRAME2
\(\Sigma \vdash \psi \sim \Gamma\)
\(\Gamma \vdash t_1\)
\(\Sigma ; \vdash_M \; x : t_1; \epsilon\)
\(\vdash \text{m}(w) \; \text{w} \; \text{secure}(\epsilon): \tau; \epsilon\)

TENV-EMP
\(\Sigma \vdash \psi \sim \Gamma\)
\(\Sigma ; \vdash_M \; v : \psi[\tau]\)
\(\vdash \text{M} \; \text{v} \; \text{secure}(\epsilon): \tau; \epsilon\)

TENV-MAPP1
\(\Sigma \vdash \psi \sim \Gamma\)
\(\Sigma ; \vdash_M \; v : \psi[\tau]\)
\(\vdash \text{M} \; \text{v} \; \text{secure}(\epsilon): \tau; \epsilon\)

TENV-MAPP2
\(\Sigma \vdash \psi \sim \Gamma\)
\(\Sigma ; \vdash_M \; v : \psi[\tau]\)
\(\vdash \text{M} \; \text{v} \; \text{secure}(\epsilon): \tau; \epsilon\)

TENV-MUNK
\(\Sigma \vdash \psi \sim \Gamma\)
\(\Gamma \vdash \tau\)
\(\Sigma \vdash \psi(x : \text{M}) : \psi[\tau]\)

TENV-MUNK
\(\Sigma \vdash \psi \sim \Gamma\)
\(\Gamma \vdash \tau\)
\(\Sigma \vdash \psi(x : \text{M} \; \text{M}) : \psi[\tau]\)

C halted

HALTED-ANSWER
\(\nu(w) \{\sigma; \psi; v\} \; \text{halted}\)

HALTED-ERROR
\(\text{M} \{\sigma; \psi; \epsilon\} \; \text{halted}\)

C st

STOK-SEC
\(\kappa = \kappa' : (\nu(w); \psi; x; e_2) :: \kappa_1\)
\(s(w) \{\sigma; \kappa; \psi; e\} \; \text{st}\)

STOK-SEC
\(\kappa = \kappa' : (\nu(w); \psi; x; e_2) :: \kappa_1\)
\(s(w) \{\sigma; \kappa; \psi; \text{secure}(\epsilon)\} \; \text{st}\)

STOK-PAR1
\(\nu(w) \{\sigma; \psi; e\} \; \text{st}\)

STOK-PAR3
\(p(w) \{\sigma; \kappa; \psi; e\} \; \text{st}\)

STOK-PAR2
\(w \subseteq w'\)
\(\nu(w) \{\sigma; \psi; e\} \; \text{st}\)

STOK-PAR2
\(p(w) \{\sigma; \psi; e\} \; \text{st}\)

STOK-PAR2
\(p(w) \{\sigma; \kappa; \psi; e\} \; \text{st}\)

STOK-PAR2
\(p(w) \{\sigma; \kappa; \psi; e\} \; \text{st}\)

STOK-PAR2
\(p(w) \{\sigma; \kappa; \psi; e\} \; \text{st}\)

C halted

Fig. 15. Runtime configuration typing.
Fig. 16. Runtime configuration typing.
B. Proofs

We first present several auxiliary lemmas. Main theorems are proved towards the end. This section is best read electronically, as it has several hyperlinks to aid navigation (such as for skipping long proofs, etc.).

Lemma A.1: (Weakening of type environment)

Let $x \notin \text{dom}(\Gamma)$ and $\Gamma_1 = \Gamma, x : M, \tau'$.

1. If $\Gamma \vdash_M v : \tau$, then $\Gamma_1 \vdash_M v : \tau$.
2. If $\Gamma \vdash v : \tau$, then $\Gamma_1 \vdash v : \tau$.
3. If $\Gamma \vdash \tau$, then $\Gamma_1 \vdash \tau$.
4. If $\Gamma \vdash \phi$, then $\Gamma_1 \vdash \phi$.
5. If $\Gamma \vdash \tau_1 <: \tau_2$, then $\Gamma_1 \vdash \tau_1 <: \tau_2$.
6. If $\Gamma \vdash N$, then $\Gamma_1 \vdash N$.
7. If $\Gamma \vdash M \triangleright N$, then $\Gamma_1 \vdash M \triangleright N$.
8. If $\Gamma \vdash \epsilon$, then $\Gamma_1 \vdash \epsilon$.

Proof: (Skip) By simultaneous induction.

(1.) Induction on derivation of $\Gamma \vdash_M v : \tau$, case analysis on the last rule used.

Rule T-VAR. We have,

(a) $v = y$
(b) $y : M \tau \in \Gamma \vee y : \tau \in \Gamma$
(c) $\Gamma \vdash \tau$

(b) means $y$ is different from $x$.

From (b) we have,

From (b) we have,

(d) $y : M \tau \in \Gamma_1 \vee y : \tau \in \Gamma_1$

Use I.H. (3.) on (c) to get,

(e) $\Gamma_1 \vdash \tau$

Use I.H. (1.) on (a), I.H. (3.) on (c), and with (b) use rule T-INJ.

Rule T-UNIT. Use rule T-UNIT with $\Gamma_1$.

Rule T-INJ. We have,

(a) $\Gamma \vdash_M v : \tau_i$
(b) $\tau_j \text{IsFlat}$
(c) $\Gamma \vdash \tau_j$

Use I.H. (1.) on (a), I.H. (3.) on (c), and with (b) use rule T-INJ.

Rule T-PROD. Use I.H. (1.) on premises.

Rule T-PRINC. Use rule T-PRINC with $\Gamma_1$.

Rule T-PSONE. Use I.H. (1.) on rule premise, and then use rule T-PSONE.

Rule T-PSUNION. Use I.H. (1.) on rule premises, and then use rule T-PSUNION.

Rule T-PSVAR. Use I.H. (1.) on rule premise, and then use rule T-PSVAR (note that $v = y_i$, different from $x$).

Rule T-MSUB. Use I.H. (6.), (1.), and (7.), and then use rule T-MSUB.

Rule T-SUB. Use of I.H. (1.), (5.), and (3.), and then use rule T-SUB.

Rule T-SINGL. given premises, use rule T-SINGL with $\Gamma_1$.

Rule T-WIRECAT. given premises, use rule T-WIRECAT with $\Gamma_1$.

Rule T-LOC. given premises, use rule T-LOC with $\Gamma_1$.

Rule T-SH. given premises, use rule T-SH with $\Gamma_1$.

Rule T-CLOS. given premises, use rule T-CLOS with $\Gamma_1$.

Rule T-FIXCLOS. given premises, use rule T-FIXCLOS with $\Gamma_1$.

(2.) Similar to proof above.

(3.) Induction on derivation of $\Gamma \vdash \tau$, case analysis on the last rule used.

Rule WF-UNIT. use rule WF-UNIT with $\Gamma_1$.

Rule WF-SUM. use I.H. (3.) on premises, and then use rule WF-SUM.

Rule WF-PROD. use I.H. (3.) on premises, and then use rule WF-PROD.

Rule WF-PRINC. use I.H. (4.) on rule premise, and then use rule WF-PRINC.

Rule WF-ARROW. we have,

(a) $\tau = y : \tau_1 \rightarrow \tau_2$
(b) $\Gamma \vdash \tau_1$
(c) $\Gamma, y : \tau_1 \vdash \epsilon$
(d) $\Gamma, y : \tau_1 \vdash \tau_2$

Use I.H. (3.) on (b), I.H. (8.) on (c), and I.H. (3.) on (d) to get

(e) $\Gamma_1 \vdash \tau_1$
(f) $\Gamma, y : \tau_1, x : M, \tau' \vdash \epsilon$
(g) $\Gamma, y : \tau_1, x : M, \tau' \vdash \tau_2$

Use type environment permutation lemma on (f) and (g) and then use rule WF-ARROW with (e) to get the derivation.

Rule WF-WIRE. use I.H. (2.) and (3.), and then use rule WF-WIRE.

Rule WF-ARRAY. use I.H. (3.), and then use rule WF-ARRAY.

Rule WF-SHARE. use I.H. (2.) and (3.), and then use rule WF-SHARE.

(4.) Induction on derivation of $\Gamma \vdash \phi$, case analysis on the last rule used.

Rule WFREF-TRUE. use rule WFREF-TRUE with $\Gamma_1$.

Rule WFREF-SINGL. use rule WFREF-SINGL with $\Gamma_1$.

Rule WFREF-SUB. use I.H. (2.) on the premise, and then use rule WFREF-SUB.
Rule WFREF-EQ. use I.H. (2.) on the premise, and then use rule WFREF-EQ.

Rule WFREF-CONJ. use I.H. (4.) on the premises, and then use rule WFREF-CONJ.

(5.) Induction on derivation of $\Gamma \vdash \tau_1 <: \tau_2$, case analysis on the last rule used.

Rule S-REFL. use rule S-REFL with $\Gamma_1$.

Rule S-TRANS. use I.H. (5.) on premises, and then use rule S-TRANS.

Rule S-SUM. use I.H. (5.) on premises, and then use rule S-SUM.

Rule S-PROD. use I.H. (5.) on premises, and then use rule S-PROD.

Rule S-PRINCS. we have,

(a) $[[\Gamma]] \models \phi_1 \Rightarrow \phi_2$

We can derive

(b) $[[\Gamma_1]] \models \phi_1 \Rightarrow \phi_2$

Use rule S-PRINCS again.

Rule S-WIRE. use I.H. (2.) and (5.) on premises, and then use rule S-WIRE.

Rule S-ARRAY. use I.H. (5.) on premises, and then use rule S-ARRAY.

Rule S-SHARE. use I.H. (2.) and (5.) on premises, and then use rule S-SHARE.

Rule S-ARROW. use I.H. (5.) on premises, permutation lemma, and then use rule S-ARROW.

(6.) Induction on derivation of $\Gamma \vdash N$, case analysis on the last rule used.

Rule WFPL-TOP. use rule WFPL-TOP with $\Gamma_1$.

Rule WFPL-OTHER. use I.H. (2.) on premise, and then use rule WFPL-OTHER.

(7.) Induction on derivation of $\Gamma \vdash M \gg N$, case analysis on the last rule used.

Rule D-REFL. use I.H. (2.) on premise, and then use rule D-REFL.

Rule D-TOP. use I.H. (2.) on premise, and then use rule D-TOP.

Rule D-PAR. use I.H. (2.) on premise, and then use rule D-PAR.

Rule D-SEC. use I.H. (2.) on premise, and then use rule D-SEC.

(8.) Induction on derivation of $\Gamma \vdash \epsilon$, case analysis on the last rule used.

Rule WFEFF-EMPTY. use rule WFEFF-EMPTY with $\Gamma_1$.

Rule WFEFF-MODE. use I.H. (8.) and I.H. (6.) on premises, and then use rule WFEFF-MODE.

Lemma A.2: (Weakening of type environment)

Let $x \notin \text{dom}(\Gamma)$ and $\Gamma_1 = \Gamma, x : \tau'$.

1) If $\Gamma \vdash_M v : \tau$, then $\Gamma_1 \vdash_M v : \tau$.

2) If $\Gamma \vdash v : \tau$, then $\Gamma_1 \vdash v : \tau$.

3) If $\Gamma \vdash \tau$, then $\Gamma_1 \vdash \tau$.

4) If $\Gamma \vdash \phi$, then $\Gamma_1 \vdash \phi$.

5) If $\Gamma \vdash \tau_1 <: \tau_2$, then $\Gamma_1 \vdash \tau_1 <: \tau_2$.

6) If $\Gamma \vdash N$, then $\Gamma_1 \vdash N$.

7) If $\Gamma \vdash M \gg N$, then $\Gamma_1 \vdash M \gg N$.

8) If $\Gamma \vdash \epsilon$, then $\Gamma_1 \vdash \epsilon$.

Proof: Similar to the proof of Lemma A.1.

Lemma A.3: (Weakening of type environment under subtyping)

Let $\Gamma \vdash \tau' <: \tau$ and $\Gamma \vdash \tau'$. Let $\Gamma_1 = \Gamma, x : M \tau'$. For $\Gamma_2 = \Gamma, x : M \tau'$,

1) If $\Gamma_1 \vdash_N v : \tau$, then $\Gamma_2 \vdash_N v : \tau$.

2) If $\Gamma_1 \vdash v : \tau$, then $\Gamma_2 \vdash v : \tau$.

3) If $\Gamma_1 \vdash \tau_1 <: \tau_2$, then $\Gamma_2 \vdash \tau_1 <: \tau_2$.

4) If $\Gamma_1 \vdash \tau$, then $\Gamma_2 \vdash \tau$.

5) If $\Gamma_1 \vdash \phi$, then $\Gamma_2 \vdash \phi$.

6) If $\Gamma_1 \vdash N \gg M_1$, then $\Gamma_2 \vdash N \gg M_1$.

7) If $\Gamma_1 \vdash M_1$, then $\Gamma_2 \vdash M_1$.

8) If $\Gamma_1 \vdash \epsilon$, then $\Gamma_2 \vdash \epsilon$.

9) If $\Gamma_1 \vdash_N e : \tau; \epsilon$, then $\Gamma_2 \vdash_N e : \tau; \epsilon$.

Proof: (Skip)

By simultaneous induction.

(1.) Induction on derivation of $\Gamma_1 \vdash_N v : \tau$, case analysis on the last rule used.

Rule T-VAR. We have,

(a) $v = y$

(b) $y : N \tau \in \Gamma_1 \lor y : \tau \in \Gamma_1$

(b') $\Gamma_1 \vdash \tau$

We have two cases now,

(i) $y = x$

This means,

(c) $M = N$

With lemma premise $\Gamma \vdash \tau'$, use Lemma A.1 to get,

(d) $\Gamma_1 \vdash \tau'$

Use I.H. (4.) on (d) to get,

(e) $\Gamma_2 \vdash \tau'$

With (d), use rule T-VAR on $\Gamma_2$ to get,
(f) $\Gamma_2 \vdash_M x : \tau'$
With $\Gamma \vdash \tau' <: \tau$, use Lemma A.1 to get.

(g) $\Gamma_2 \vdash \tau' <: \tau$
Use I.H. (4.) on (b') to get.

(h) $\Gamma_2 \vdash \tau (M = N)$
With (f), (g) and (h), use rule T-SUB.

(ii) $y$ is different from $x$, in which case rule T-VAR still holds on $\Gamma_2$ (with use of I.H. (4.)).

Rule T-UNIT. Use rule T-UNIT with $\Gamma_2$.

Rule T-INJ. Use I.H. (1.) on (c) and use rule T-INJ.

Rule T-FRAC. Use I.H. (1) on premises, and then use rule T-FRAC.

Rule T-UNIT. Use rule T-UNIT with $\Gamma_2$.

Rules T-PSONE, T-PSUM, and T-PSVAR. Use I.H. (1.) on premises, and then use respective rule.

Rule T-MSUB. We have,

(a) $v = y$
(b) $\Gamma_1 \vdash M_1$
(c) $\Gamma_1 \vdash M_1 y : \tau$
(d) $\Gamma_1 \vdash M_1 \triangleright N$
Use I.H. (7.) on (b),

(f) $\Gamma_2 \vdash M_1$
Use I.H. (1.) on (c)
(g) $\Gamma_2 \vdash M_1 y : \tau$
Use I.H. (6.) on (d)

(h) $\Gamma_2 \vdash M_1 \triangleright N$

With (f), (g), (h), use rule T-MSUB (the fact about $\tau$ isSecIn carries)

Rule T-SUB. Use I.H. (1.), (3.), and (4.) on premises, and then use rule T-SUB.

Runtime value typing rules are similar to proof of Lemma A.1 (they don’t depend on $\Gamma$).

(2.) Induction on derivation of $\Gamma \vdash v : \tau$, case analysis on the last rule used.

Rule TN-VAR. We have,

(a) $v = y$
(b) $y : \tau \in \Gamma_1$
(c) $\Gamma_1 \vdash \tau$
From (b) it follows that $y$ is different from $x$, and so,

(d) $y : \tau \in \Gamma_2$
Use I.H. (4.) on (c) and with (b), use rule TN-VAR.
Other cases are similar to (1.).

(3.) Induction on derivation of $\Gamma_1 \vdash \tau_1 <: \tau_2$, case analysis on the last rule used.

Rule S-REFL. Use rule S-REFL with $\Gamma_2$.


Rule S-PRINC. We have,

(a) $\|\Gamma_1\| \vdash \phi_1 \Rightarrow \phi_2$
We need to prove $\|\Gamma_2\| \vdash \phi_1 \Rightarrow \phi_2$. Informally, only principal types in $\Gamma$ matter when deciding logical implications. And, a more precise type in the typing environment means stronger assumption.

Rule S-WIRE. Use I.H. (2.) and I.H. (3.) on premises, and then use rule S-WIRE.

Rule S-ARRAY. Use I.H. (3.) on premises, and then use rule S-ARRAY.

Rule S-SHARE. Use I.H. (2.) and I.H. (3.) on premises, and then use rule S-SHARE.

Rule S-ARROW. Use I.H. (3.) on premises with permutation lemma for second premise, and then use rule S-ARROW.

(4.) Induction on derivation of $\Gamma_1 \vdash \tau$, case analysis on the last rule used.

Rule WF-UNIT. Use rule WF-UNIT with $\Gamma_2$.

Rule WF-SUM. Use I.H. (4.) on premises, and then use rule WF-SUM.

Rule WF-PROD. Use I.H. (4.) on premises, and then use rule WF-PROD.

Rule WF-PRINC. Use I.H. (5.) on premise, and then use rule WF-PRINC.

Rule WF-ARROW. Use I.H. (4.) and (8.) with permutation lemma on typing environment, and then use rule WF-ARROW.

Rule WF-WIRE. Use I.H. (2.) and (3.) on premises, and then use rule WF-WIRE.

Rule WF-ARRAY. Use I.H. (4.) on premise, and then use rule WF-ARRAY.

Rule WF-SHARE. Use I.H. (2.) and (4.) on premises, and then use rule WF-SHARE.

(5.) Induction on derivation of $\Gamma_1 \vdash \phi$, case analysis on the last rule used.

Rule WREF-TRUE. Use rule WREF-TRUE with $\Gamma_2$.

Rule WREF-SINGL. Use rule WREF-SINGL with $\Gamma_2$.

Rules WREF-SUB and WREF-EQ. Use I.H. on premises, and then use respective rule.

Rule WREF-CONJ. Use I.H. on premises, and then use rule WREF-CONJ.

(6.) Induction on derivation of $\Gamma_1 \vdash N \triangleright M_1$, case analysis on the last rule used.

Rule D-REFL. Use I.H. on premise, and then use rule D-REFL.
Rule D-TOP. Use I.H. on premise, and then use rule D-TOP.

Rules D-PAR and D-SEC. Similar use of I.H. and then respective rule.

(7.) Induction on derivation of $\Gamma \vdash M_1$, case analysis on the last rule used.

Rule WFPL-TOP. Use rule WFPL-TOP with $\Gamma_2$.

Rule WFPL-OTHER. Use I.H. on premise, and then use rule WFPL-OTHER.

(8.) Induction on derivation of $\Gamma_1 \vdash \epsilon$, case analysis on the last rule used.

Rule WFEFF-EMPTY. Use rule WFEFF-EMPTY with $\Gamma_2$.

Rule WFEFF-MODE. Use I.H. on premises, and then use rule WFEFF-MODE.

(9.) Proof by induction on derivation of $\Gamma_1 \vdash \tau; \tau$.

Lemma A.4: (Can derive self equality in refinements)
If $\Gamma \vdash v : \phi$, then $\Gamma \vdash v : \phi' (\nu = v)$.

Proof: Structural induction on $v$.

- $v = x$. We have,
  - (a) $\Gamma \vdash x : \phi$

With (a) use rule TN-PSVAR to get $\Gamma \vdash x : \phi (\nu = x)$.

- $v = p$. Use rule TN-PRINC.

- $v = \{w\}$. Use rule TN-PSONE.

- $v = w_1 \cup w_2$. Use rule TN-PSUNION.

No other form of $v$ is possible.

Lemma A.5: (Can derive self subset in refinements)
If $\Gamma \vdash v : \phi$, then $\Gamma \vdash v : \phi (\nu \subseteq v)$.

Proof:
Using Lemma A.4, we have
- (a) $\Gamma \vdash v : \phi (\nu = v)$

Also,
- (b) $[\Gamma] \vdash (\nu = v) \Rightarrow (\nu \subseteq v)$

With (b) use rule S-PRINCS to get
- (c) $\Gamma \vdash \phi (\nu = v) <:\phi (\nu \subseteq v)$

With lemma premise $\Gamma \vdash \phi$, we can use rule WFREF-SUB and rule WF-PRINC to get,
- (d) $\Gamma \vdash \phi (\nu \subseteq v)$

With (a), (c), and (d), use rule T-SUB.

Lemma A.6: (Transitivity of Delegation)
If $\Gamma \vdash M \triangleright M_1$, and $\Gamma \vdash M_1 \triangleright M_2$, then $\Gamma \vdash M \triangleright M_2$.

Lemma A.7: (Secure place can only delegate to self)
If $\Gamma \vdash s(w_1) \triangleright m(w_2)$, then
1) $m = s$
2) $\Gamma \vdash w_2 : \phi (\nu = w_1)$
3) $\Gamma \vdash p(w_1) \triangleright m(w_2)$

Proof: $\Gamma \vdash s(w_1) \triangleright m(w_2)$ can only be derived using rule D-REFL, which immediately gives us (1.) and (2.). For (3.), use rule D-SEC and then Lemma A.6.

Lemma A.8: (Delegation implies well-formedness)
If $\Gamma \vdash M$, $\Gamma \vdash M \triangleright N$, then $\Gamma \vdash N$.

Proof:
Proof by induction on derivation of $\Gamma \vdash M \triangleright N$, case analysis on the last rule used.

Rule D-REFL. We have,
- (a) $M = m(w_1)$
- (b) $N = m(w_2)$
- (c) $\Gamma \vdash w_2 : \phi (\nu = w_1)$

With (c), use rule WFPL-OTHER on $m(w_2)$.

Rule D-TOP. We have,
- (a) $M = \top$
- (b) $N = m(w)$
- (c) $\Gamma \vdash \top w : \phi (\nu = w_1)$

With (c), use rule WFPL-OTHER to get $\Gamma \vdash m(w)$.

Rules D-PAR and D-SEC. Similar to rule D-TOP.

Lemma A.9: (Effect delegation implies well-formedness)
If $\Gamma \vdash M$, $\Gamma \vdash M \triangleright \epsilon$, then $\Gamma \vdash \epsilon$.

Proof: Straightforward extension of Lemma A.8.

Lemma A.10: (Typing results in well-formed types)
Let $\Gamma \vdash M$.

1) If $\Gamma \vdash_M \tau$, then $\Gamma \vdash \tau$.
2) If $\Gamma \vdash v : \tau$, then $\Gamma \vdash \tau$.
3) If $\Gamma \vdash_M e : \tau; \epsilon$, then $\Gamma \vdash \tau$ and $\Gamma \vdash M \triangleright \epsilon$.

Proof: (Skip)
Proof by induction on derivation of $\Gamma \vdash_M v : \tau$, case analysis on the last rule used.

Rule T-VAR. Follows from rule premise.

Rule T-UNIT. Use rule WF-UNIT.

Rule T-INJ. With rule premises, use rule WF-SUM.
Rule T-PRINC. We have
(a) $\Gamma \vdash_M p : \text{ps} \ (\nu = \{p\})$
With (a), use rule WFREF-EQ and rule WF-PRINC.

Rules T-PSONE, T-PSUNION, and T-PSVAR. Use rule premise in rule WFREF-EQ and rule WF-PRINC.

Rule T-MSUB. We have,
(a) $\Gamma \vdash_N x : \tau$
(b) $\Gamma \vdash N \triangleright M$
Use I.H. on (a) to get $\Gamma \vdash \tau$

Rule T-SUB. Follows from rule premise.

Rule T-SINGWIRE. We have,
(a) $v = \{p : v'\}$
(c) $\tau = W \{p\} \tau$
(d) $\vdash \tau$
(d') $\tau$ IsFlat
Use rule TN-PRINC to get,
(e) $\vdash p : \text{ps} \ (\nu = \{p\})$
With (e), use rule T-PSONE to get,
(f) $\vdash \{p\} : \text{ps} \ (\nu = \{p\})$
Use weakening on (d) and (f), and then with (d') use rule WF-WIRE.

Rule T-WIRECAT. We have,
(a) $v = v_1 \uplus v_2$
(b) $\tau = W (w_1 \cup w_2) \tau$
(c) $\vdash_M v_1 : W w_1 \tau$
(d) $\vdash_M v_2 : W w_2 \tau$
Use I.H. on (c) and (d) to get,
(e) $\vdash W w_1 \tau$
(f) $\vdash W w_2 \tau$
Invert rule WF-WIRE on (e) and (f) to get,
(g) $\vdash \tau$
(h) $\vdash w_1 : \text{ps} \phi_i$
Use weakening on (g) and (h) (with $\Gamma$), and then use rule WF-WIRE.

Rule T-LOC. Follows from the premise.

Rule T-SH. We have,
(a) $\vdash M$
(b) $M = _\text{w}$
(c) $\vdash \tau$
With (a) and (b), invert rule WFPL-OTHER to get,
(d) $\vdash w : \text{ps} \phi$
Use weakening on (c) and (d), and then use rule WF-SH.

Rules T-CLOS and T-FIXCLOS. Follows from the premises (with weakening).

(2.) Induction on derivation of $\Gamma \vdash_M e : \tau ; e$, case analysis on the last rule used.

Rule T-FST. We have,
(a) $e = \text{fst} (v)$
(b) $\epsilon = \cdot$
(c) $\Gamma \vdash_M v : \tau_1 \times \tau_2$ (rule premise)
(d) $\tau = \tau_1$
With (c), use I.H. to get
(e) $\Gamma \vdash \tau_1 \times \tau_2$
With (e), invert rule rule WF-PROD to get,
(f) $\Gamma \vdash \tau_1$
With (f), and rule EFFDEL-EMPTY, we have the proof.

Rule T-SND. Similar to rule T-FST.

Rule T-CASE. Follows from the premises.

Rule T-APP.

Rules T-LET1, T-LET2, and T-FIX. Follows directly from rule premises.

Rule T-ARRAY. We have,
(a) $e = \text{array} (v_1, v_2)$
(b) $\tau = \text{Array} \tau_2$
(c) $\Gamma \vdash_M v_2 : \tau_2$ (rule premise)
With (c), use Lemma A.10, to get
(d) $\Gamma \vdash \tau_2$
With (d), use rule WF-ARRAY to get $\Gamma \vdash \text{Array} \tau_2$. $\Gamma \vdash M \triangleright \cdot$ follows from rule EFFDEL-EMPTY.

Rule T-SELECT. Use inversion on $\Gamma \vdash \text{Array} \tau$ (from rule premise and I.H.).

Rule T-UPDATE. Use rule WF-UNIT and rule EFFDEL-EMPTY.

Rule T-WIRE. We have,
(a) $\Gamma \vdash w_1 : \text{ps} \ (\nu \subseteq w_2)$
(b) $\Gamma \vdash_N v : \tau$
Use I.H. on (b) and then with (a) use rule WF-WIRE.

Rule T-WPROJ. Use I.H. on premise $\Gamma \vdash_M v_2 : W w_2 \tau$, and then invert rule WF-WIRE.

Rule T-WIREUN. Use I.H. on premises and then rule WF-WIRE.

Rule T-WFOLD. Follows from rule premise $\Gamma \vdash_M v_2 : \tau_2$.

Rule T-WAPP. We have,
Lemma A.11: (Subtyping inversion)

1) If \( \Gamma \vdash \tau <: \text{unit} \), then \( \tau = \text{unit} \).

2) If \( \Gamma \vdash \tau <: \tau_1 \times \tau_2 \), then \( \tau = \tau_3 \times \tau_4 \) s.t. \( \Gamma \vdash \tau_3 <: \tau_1 \) and \( \Gamma \vdash \tau_4 <: \tau_2 \).

3) If \( \Gamma \vdash \tau <: \tau_1 \rightarrow \tau_2 \), then \( \tau = \tau_3 \rightarrow \tau_4 \) s.t. \( \Gamma \vdash \tau_3 <: \tau_1 \) and \( \Gamma \vdash \tau_4 <: \tau_2 \).

4) If \( \Gamma \vdash \tau <: \text{ps} \phi \), then \( \tau = \text{ps} \phi_2 \) s.t. \( \Gamma \vdash \phi_1 \rightarrow \phi_2 \).

5) If \( \Gamma \vdash \tau <: W \tau_1 \) and \( \Gamma \vdash \tau <: \text{ps} \phi \), then \( \tau = W \tau_1 \) s.t. \( \Gamma \vdash \phi \rightarrow \text{ps} \phi_2 \) s.t. \( \Gamma \vdash \tau_1 <: \tau_2 \).

6) If \( \Gamma \vdash \tau <: \text{Array} \tau_2 \), then \( \tau = \text{Array} \tau_1 \) s.t. \( \Gamma \vdash \tau_1 <: \tau_2 \) and \( \Gamma \vdash \tau_2 <: \tau_1 \).

7) If \( \Gamma \vdash \text{Sh} \tau w_2 \) and \( \Gamma \vdash \tau <: \text{ps} \phi \), then \( \tau = \text{Sh} \tau w_1 \) s.t. \( \Gamma \vdash \tau_1 <: \tau_2 \) and \( \Gamma \vdash \tau_2 <: \tau_1 \).

8) If \( \Gamma \vdash \tau <: x : \tau_3 \) s.t. \( \Gamma \vdash \tau_1 <: \tau_2 \).

Proof: (Skip)

(1) Only possible last rules in derivation of \( \Gamma \vdash \tau <: \text{unit} \) are rule S-REFL (immediate) and rule S-TRANS (Use I.H. twice)

(2) Induction on derivation of \( \Gamma \vdash \tau <: \tau_1 \times \tau_2 \), case analysis on the last rule used.

Rule S-REFL. We get \( \tau = \tau_1 \times \tau_2 \), use rule S-REFL on \( \tau_1 \) and \( \tau_2 \) to complete the proof.

Rule S-TRANS. We have,

(a) \( \Gamma \vdash \tau <: \tau' \)

(b) \( \Gamma \vdash \tau' <: \tau_1 \times \tau_2 \)

Use I.H. on (b) to get,

(c) \( \tau' = \tau_1' \times \tau_2' \)

(d) \( \Gamma \vdash \tau_1' <: \tau_1 \)

(e) \( \Gamma \vdash \tau_2' <: \tau_2 \)

Substitute \( \tau' \) from (c) in (a), and then use I.H. on (a) to get,

(f) \( \tau = \tau_3 \times \tau_4 \)

(g) \( \Gamma \vdash \tau_3 <: \tau_1' \)

(h) \( \Gamma \vdash \tau_1' <: \tau_2' \)

Use rule S-TRANS on (g) and (d), and then (h) and (e), with (f) this completes the proof.

Rule S-PROD. Read from the rule.

(3.) Similar to (2.)

(4.) Induction on derivation of \( \Gamma \vdash \tau <: \text{ps} \phi \), case analysis on the last rule used.

Rule S-REFL. \( \tau = \text{ps} \phi \), and \( \Gamma \vdash \phi \rightarrow \phi \) is trivially true.

Rule S-TRANS. We have,

(a) \( \Gamma \vdash \tau <: \tau_1 \)

(b) \( \Gamma \vdash \tau_1 <: \text{ps} \phi \)

Use I.H. on (b), we get

(c) \( \tau_1 = \text{ps} \phi_1 \)

(d) \( \Gamma \vdash \phi_1 \rightarrow \phi \)

Substitute (c) in (a) to get

(e) \( \Gamma \vdash \tau <: \text{ps} \phi_1 \)

Use I.H. on (e), we get

(f) \( \tau = \text{ps} \phi_2 \)

(g) \( \Gamma \vdash \phi_2 \rightarrow \phi_1 \)
From (f) and transitivity of implication on (g) and (d), we have the proof.

**Rule S-PRINCS.** Read from the rule.

(5.) Proof by induction on derivation of \( \Gamma \vdash \tau < \mathbf{W} w_2 \tau_2 \), case analysis on the last rule used.

**Rule S-REFL.** We have \( \tau = \mathbf{W} w_2 \tau_2 \), thus \( \tau_1 = \tau_2 \) and \( w_1 = w_2 \). To prove \( \Gamma \vdash \tau_1 < \tau_2 \), use rule S-REFL. To prove \( \Gamma \vdash \tau_M w_2 : \mathbf{ps} \ (\nu \subseteq w_2) \), use Lemma A.5 (we have \( \Gamma \vdash w_2 : \mathbf{ps} \phi \) from premise.)

**Rule S-TRANS.** We have,

(a) \( \Gamma \vdash \tau < \tau' \)

(b) \( \Gamma \vdash \tau' < \mathbf{W} w_2 \tau_2 \)

Using I.H. (6.) on (b) to get,

(c) \( \tau' = \mathbf{W} w_3 \tau_3 \)

(d) \( \Gamma \vdash w_2 : \mathbf{ps} \ (\nu \subseteq w_3) \)

(e) \( \Gamma \vdash \tau_3 < \tau_2 \)

Using Lemma A.10 on (d), we get

(f) \( \Gamma \vdash \mathbf{ps} \ (\nu \subseteq w_3) \)

Inverting rule WF-PRINC on (f),

(g) \( \Gamma \vdash \nu \subseteq w_3 \)

Inverting rule WF-SUB, we get

(h) \( \Gamma \vdash w_3 : \mathbf{ps} \phi \)

Use I.H. on (a) (now that we have (h)) (substitute \( \tau' \) from (c))

(i) \( \tau = \mathbf{W} w_1 \tau_1 \)

(j) \( \Gamma \vdash w_3 : \mathbf{ps} \ (\nu \subseteq w_1) \)

(k) \( \Gamma \vdash \tau_3 < \tau_2 \)

From (j), we can use rule S-PRINCS to derive:

(l) \( \Gamma \vdash \mathbf{ps} \ (\nu \subseteq w_3) < \mathbf{ps} \ (\nu \subseteq w_1) \)

From (j), use Lemma A.10 and inversions on rule WF-PRINC and rule WF-SUB to get

(m) \( \Gamma \vdash w_1 : \mathbf{ps} \phi \)

With (d), (l), and (m), use rule T-SUB to derive

(n) \( \Gamma \vdash w_2 : \mathbf{ps} \ (\nu \subseteq w_1) \)

With (e) and (k), use rule S-TRANS to complete the proof.

**Rule S-WIRE.** Read from the rule.

(6.) Straightforward using I.H.

(7.) Similar to (5.)

(8.) Interesting case is rule S-TRANS. We have,

(a) \( \Gamma \vdash \tau < \tau' \)

(b) \( \Gamma \vdash \tau' < x : \tau_1 <_\tau \tau_2 \)

Using I.H. (9.) on (b),

(c) \( \tau' = x : \tau_1 <_\tau \tau_2 \)

(d) \( \Gamma \vdash \tau_1 < \tau_1 \)

(e) \( \Gamma, x : \tau_1 < \tau_2 \)

Using I.H. on (a) now (with (c))

(f) \( \tau = x : \tau_1 < \tau_4 \)

(g) \( \Gamma \vdash \tau_1 < \tau_3 \)

(h) \( \Gamma, x : \tau_1 < \tau_4 < \tau_2 \)

With (d), (h), use Lemma A.3, to get

(i) \( \Gamma, x : \tau_1 < \tau_4 < \tau_2 \)

Use rule S-TRANS on (d) and (g), and then on (i) and (e) to complete the proof.

**Lemma A.12: (Canonical forms)**

1) If \( \vdash \_M v : \text{unit} \), then \( v = (\_). \)

2) If \( \vdash \_M v : \tau_1 \times \tau_2 \), then \( v = (v_1,v_2) \) s.t. \( \vdash \_M v_1 : \tau_1 \) and \( \vdash \_M v_2 : \tau_2 \).

3) If \( \vdash \_M v : \tau_1 + \tau_2 \), then \( v = \text{inj}_i v' \) s.t. \( \vdash \_M v' : \tau_i \).

4) If \( \vdash \_M v : \mathbf{ps} \phi \), then \( v = w_1 \cup w_2 \) s.t. \( [\_] \vdash \phi[w_1 \cup w_2 / \nu] \).

5) If \( \vdash \_p(w) v : \mathbf{W} w \tau \), then \( v = v_1 + v_2 \) s.t. \( w \subseteq \text{dom}(v_1 + v_2) \) and for all \( p \in \text{dom}(v_1 + v_2) \), \( \vdash \_p(p) v[p] : \tau \).

6) If \( \vdash \_s(w_1) v : \mathbf{W} w \tau \), then \( v = v_1 + v_2 \) s.t. \( w \subseteq \text{dom}(v_1 + v_2) \) and for all \( p \in \text{dom}(v_1 + v_2) \), \( \vdash \_s(w_1) v[p] : \tau \).

7) If \( \vdash \_M v : \text{Array} \tau, \) then \( v = \ell \) s.t. \( \Sigma(\ell) = \tau_1, \vdash \tau_1 < \tau, \) and \( \vdash \tau < \tau_1. \)

8) If \( \vdash \_M v : \text{Sh} w \tau, \) then \( v = \text{sh} w v' \) s.t. \( \vdash \_M v' : \tau. \)

9) If \( \vdash \_M v : x : \tau_1 \rightarrow \tau_2, \) then either \( v = \text{clo}(\psi; \lambda x. e) \) s.t. \( \Sigma \vdash \psi \rightarrow \Gamma \) and \( \Gamma, x : \tau_1 \vdash \_M e : \tau_2 < \epsilon, \) or \( v = \text{clo}(\psi; \text{fix} f.\lambda x. e) \) s.t. \( \Sigma \vdash \psi \rightarrow \Gamma \) and \( \Gamma, f : y : \tau_1 \rightarrow \tau_2 \vdash \_M \lambda x. e : y : \tau_1 \rightarrow \tau_2. \)

10) If \( \vdash N, \) then \( N = \top \) or \( N = \_M(w_1 \cup w_2). \)

**Proof: (Skip) By simultaneous induction.**

(1.) Proof by induction on derivation of \( \vdash \_M v : \text{unit} \), case analysis on the last rule used.

**Rule T-UNIT.** Follows from the rule.

**Rule T-SUB.** We have,

(a) \( \vdash \_M v : \tau \)

(b) \( \vdash \tau < \_\text{unit} \)

With (b), use Lemma A.11 to get

(c) \( \tau = \_\text{unit} \)

Use I.H. on (a).

(2.) Proof by induction on derivation of \( \vdash \_M v : \tau_1 \times \tau_2 \), case analysis on the last rule used.

**Rule T-PROD.** We have,
(a) \( v = (v_1, v_2) \)
(b) \( \vdash_M v_1 : \tau_1 \)
Proof follows.

Rule T-SUB. We have,
(a) \( \vdash_M v : \tau \)
(b) \( \vdash \tau <: \tau_1 \times \tau_2 \)
(c) \( \vdash \tau_1 \times \tau_2 \)
With (b), use Lemma A.11 to get,
(d) \( \vdash \tau_3 <: \tau_1 \)
(e) \( \vdash \tau_4 <: \tau_2 \)
Use I.H. on (a) (substituting from (d)) to get,
(f) \( \vdash \tau \subseteq \tau_1 \times \tau_2 \)
Inverting rule WF-PROD on (c),
(g) \( \vdash \tau_1 \)
(h) \( \vdash \tau_2 \)
Use rule T-SUB on (h), (e) and (k), and then (i), (f), and (l) to get rest of the proof.

(3.) Similar to rule T-PROD.

(4.) Induction on derivation of \( \vdash_M v : \text{ps} \phi \), case analysis on the last rule used.

Rule T-PRINC. We have,
(a) \( v = p \)
(b) \( \phi = (\nu = \{ p \}) \)
Choose \( w_1 = \{ p \}, w_2 = \cdot \) and \( \llbracket . \rrbracket \models (\nu = \{ p \})[\{ p \}/\nu]. \)

Rule T-PSONE. We have,
(a) \( v = \{ w \} \)
(b) \( \phi = (\nu = \{ w \}) \)
Choose \( w_1 = \{ w \}, w_2 = \cdot \) and then similar to rule T-PRINC.

Rule T-PSUNION. Follows similarly.

Rule T-SUB. We have,
(a) \( \vdash_M v : \tau \)
(b) \( \vdash \tau <: \text{ps} \phi \)
With (b), use Lemma A.11 to get,
(c) \( \tau = \text{ps} \phi_1 \)
(d) \( \llbracket . \rrbracket \models \phi_1 \Rightarrow \phi \)
Use I.H. on (a) (substituting from (c)),
(e) \( v = w_1 \cup w_2 \)
(f) \( \phi_1[\nu/\nu] \)
With (e), (d), and (f), we have the proof.

(5.) Induction on derivation of \( \vdash_{p(w')} v : W w \tau \), case analysis on the last rule used.

Rule T-SINGLWIRE. We have,
(a) \( v = \{ p : v' \} \)
(b) \( w = \{ p \} \)
Choose \( v_1 = \{ p : v' \}, v_2 = \cdot \), clearly \( w \subseteq \text{dom}(v_1 \cup v_2) \).
We need to show \( \vdash_{p\{p\}} v' : \tau \), it follows from premise of the rule.

Rule T-WIRECAT. We have,
(a) \( v = v_1 \cup v_2 \)
(b) \( w = w_1 \cup w_2 \)
(c) \( \vdash_{p\{p\}} v_1 : W w_1 \tau \)
(d) \( \vdash_{p\{p\}} v_2 : W w_2 \tau \)
Use I.H. on (c) to get,
(e) \( w_1 \subseteq \text{dom}(v_1) \)
(f) \( \text{for all } p \in \text{dom}(v_1), . \vdash_{p\{p\}} v_1[p] : \tau \)
Use I.H. on (d) to get,
(g) \( w_2 \subseteq \text{dom}(v_2) \)
(h) \( \text{for all } p \in \text{dom}(v_2), . \vdash_{p\{p\}} v_2[p] : \tau \)
From (e) and (g), we get
(i) \( \text{for all } p \in \text{dom}(v_1 \cup v_2), . \vdash_{p\{p\}} (v_1 \cup v_2)[p] : \tau \)
From (a), (g), and (i), we have the proof.

Rule T-SUB. We have,
(a) \( \vdash_{p\{p\}} v : \tau' \)
(b) \( \vdash \tau' <: : W w \tau \)
(c) \( \vdash W w \tau \)
Invert rule WF-WIRE on (c) to get,
(c') \( \vdash \tau \)
(d) \( \vdash w : \text{ps} \phi \)
With (b) and (d) use Lemma A.11 to get,
(e) \( \tau' = W w_1 \tau_1 \)
(f) \( \vdash w : \text{ps} (\nu \subseteq w_1) \)
(g) \( \vdash \tau_1 <: \tau \)
Use I.H. (5.) on (a) (substituting \( \tau \) from (c)) to get,
(h) \( v = v_1 \cup v_2 \)
(i) \( w_1 \subseteq \text{dom}(v_1 \cup v_2) \)
(j) \( \text{for all } p \in \text{dom}(v_1 \cup v_2), . \vdash_{p\{p\}} v[p] : \tau_1 \)
Use I.H. (f) to get,
(k) $\llbracket \tau \rrbracket = \nu \subseteq w_1 \cup [w/\nu]$
From (i) and (k) we get,
(l) $w \subseteq \text{dom}(v_1 \cup v_2)$

Use rule T-SUB with (j), (g) and (c'), we have the proof.
(6.) Induction on derivation of $\vdash s(v_1) \cdot W w \tau$, case analysis on the last rule.

**Rule T-SINGLWIRE.** We have,
(a) $v = \{ \tau : v' \}$
(b) $w = \{ \nu \}$
(c) $\vdash s(v_1) \cdot v' : \tau$

Choose $v_1 = \{ \tau : v' \}$, $v_2 = \cdot$. We have $w \subseteq \text{dom}(v_1 \cup v_2)$, and (c) completes rest of the proof.

**Rule T-WIRECAT.** We have,
(a) $v = v_1 \cup v_2$
(b) $\vdash s(v_1) \cdot v_1 : W w_1 \tau$
(c) $\vdash s(v_1) \cdot v_2 : W w_2 \tau$
(d) $w = w_1 \cup w_2$

Use I.H. (6.) on (b), and on (c) to get,
(e) $w_1 \subseteq \text{dom}(v_1)$
(f) $w_2 \subseteq \text{dom}(v_2)$
(g) for all $p \in \text{dom}(v_1)$, $\vdash s(v_1) \cdot v_1[p] : \tau$.
(h) for all $p \in \text{dom}(v_2)$, $\vdash s(v_1) \cdot v_2[p] : \tau$.

Using (d), (e), and (f), we get,
(i) $w \subseteq \text{dom}(v_1 \cup v_2)$
(g), and (h) complete rest of the proof.

**Rule T-SUB.** We have,
(a) $\vdash s(v_1) \cdot v : \tau''$
(b) $\vdash \tau'' \cdot W w \tau$
(c) $\vdash W w \tau$

With (b), use Lemma A.11 to get,
(d) $\tau'' = W w' \tau'$
(e) $\vdash w : p (\nu \subseteq w')$
(f) $\vdash \tau' <: \tau$

Use I.H. (6.) on (a) (substituting (d) in (a)),
(g) $v = v_1 \cup v_2$
(h) $w' \subseteq \text{dom}(v_1 \cup v_2)$
(i) for all $p \in \text{dom}(v_1 \cup v_2)$, $\vdash s(v_1) \cdot v[p] : \tau'$.

Use I.H. on (e) to get,
(j) $\llbracket \nu \rrbracket \vdash (\nu \subseteq w')[w/\nu]$

Use (h) and (j) to get,
(k) $w \subseteq \text{dom}(v_1 \cup v_2)$

Invert rule WF-WIRE on (c) to get,
(l) $\vdash \tau$

With (i), (f), and (l), use rule T-SUB to complete rest of the proof.
(7.) Induction on derivation of $\vdash M v : \text{Array} \tau$, case analysis on the last rule.

**Rule T-LOC.** Read from the rule, use rule S-REFL.

**Rule T-SUB.** We have,
(a) $\vdash M v : \tau'$
(b) $\vdash \tau'' <: \text{Array} \tau$
(c) $\vdash \text{Array} \tau$

With (b) use Lemma A.11 to get,
(d) $\tau'' = \text{Array} \tau''$
(e) $\vdash \tau'' <: \tau$
(f) $\vdash \tau <: \tau''$

Use I.H. (7.) on (a) (substituting $\tau'$ from (d)),
(g) $v = \ell$
(h) $\Sigma(\ell) = \tau_1$
(i) $\vdash \tau_1 <: \tau''$
(j) $\vdash \tau'' <: \tau_1$

Use rule S-TRANS on (i) and (e), and then (f) and (j) to complete the proof.
(8.) Induction on derivation of $\vdash_M v : \text{Sh} w \tau$, case analysis on the last rule used.

**Rule T-SH.** Read from the rule.

**Rule T-SUB.** We have,
(a) $\vdash_M v : \tau'$
(b) $\vdash \tau' <: \text{Sh} w \tau$
(c) $\vdash \text{Sh} w \tau$

With (b), use Lemma A.11 to get,
(d) $\tau' = \text{Sh} w_2 \tau_2$
(e) $\vdash w_2 : p (\nu = w)$
(f) $\vdash \tau_2 <: \tau$
(g) $\vdash \tau <: \tau_2$

Use I.H. (8.) on (a) (substituting $\tau'$ from (d)),
(h) $v = \text{sh} w v'$
(j) $\vdash_M v' : \tau_2$

Invert rule WF-SH on (c) to get,
(l) $\vdash \tau$

With (j), (f), and (l), use rule S-TRANS to complete rest of the proof.
(9.) Induction on derivation of . ⊢_M v : ρ → τ_2, case analysis on the last rule used.

**Rule T-CLOS.** Read from the rule.

**Rule T-FIXCLOS.** Read from the rule.

**Rule T-SUB.** We have,

(a) . ⊢_M v : τ
(b) . ⊢ τ' < x : τ_1 → τ_2
(c) . ⊢ (x : τ_1 → τ_2)

With (b), use Lemma A.11 to get,

(d) τ' = x : τ_1 → τ_2
(e) . ⊢ τ_1 < τ_1'
(f) . x : τ_1 ⊢ τ_2' < τ_2

Use I.H. on (a) (substituting τ' from (d)), we have 2 cases,

(i) v = clos(ψ; λx.e)
(j) Σ ⊢ ψ ↷ Γ
(k) Γ, x : τ_1 ⊢_M e : τ_2'; ϵ

Use weakening and permutation of type environment on (f) and (e) to get,

(ii) Γ, x : τ_1 ⊢_M e : τ_2'; ϵ
(iii) Γ ⊢ τ_1 < τ_1'

With (k) and (i), use rule T-SUBE to complete the proof.

Second case is,

(ii) v = clos(ψ; fix f.λx.e)
(j) Σ ⊢ ψ ↷ Γ
(k) Γ, f : y : τ_1' → τ_2' ⊢_M λx.e : y : τ_1' → τ_2'; ϵ

Use weakening of type environment on (b) to get,

(i) Γ ⊢ y : τ_1' → τ_2' = x : τ_1' → τ_2

With (h) and (i), use Lemma A.3, and then rule T-SUBE to complete the proof (similar to above).

(10.) Induction on derivation of . ⊢ N, case analysis on the last rule used.

**Rule WFPL-GLOBAL.** Follows.

**Rule WFPL-PC.** Use I.H. on premise.

**Lemma A.13:** (Delegation among closed places)

Let w be a closed principal set. Let . ⊢ m(w) ⊢ N. Then, N = m_1(w_1) s.t. one of the following holds:

1) m = p, m_1 = s, and w_1 = w.
2) m = p, m_1 = p, and w_1 ⊆ w.
3) m = p, m_1 = s, and w_1 = w.

**Proof:** Induction on derivation of . ⊢ m(w) ⊢ N, case analysis on the last rule used, using Lemma A.12.

**Lemma A.14:** (Environment lookup same across delegation)

If Γ ⊢ M ⊢_N, then ψ[x]_N = ψ[x]_M.

**Proof:** Case analysis on ψ[x]_M.

**Rule VL-VAR1.** We have,

(a) ψ[x]_M = v
(b) x ⊢_M, v ∈ ψ
(c) Γ ⊢ M ⊢_N

With (c) use Lemma A.6 to get,

(d) Γ ⊢_N

With (b) and (d), we get ψ[x]_N = v

**Rule VL-VAR2.** Use rule VL-VAR2 again with N.

**Rule VL-VAR3.** Use rule VL-VAR3 again with N.

**Lemma A.15:** (Well-formedness of runtime environment)

Let Σ ⊢ ψ ↷ Γ, and Γ' ⊢ M. Also, let dom(Γ) ∩ dom(Γ') = φ.

1) If Γ, Γ' ⊢_M v : τ, then ψ[Γ'] ⊢_φ ψ[Γ] : ψ[τ].
2) If Γ, Γ' ⊢ v : τ, then ψ[Γ'] ⊢ ψ[v] : ψ[τ].
3) If Γ, Γ' ⊢ τ_1 < τ_2, then ψ[Γ'] ⊢ ψ[τ_1] < ψ[τ_2].
4) If Γ, Γ' ⊢ τ, then ψ[Γ'] ⊢ ψ[τ].
5) If Γ, Γ' ⊢ φ, then ψ[Γ'] ⊢ ψ[φ].
6) If Γ, Γ' ⊢ N, then ψ[Γ'] ⊢ ψ[N].
7) If Γ, Γ' ⊢ ϵ, then ψ[Γ'] ⊢ ψ[ϵ].
8) If Γ, Γ' ⊢ N_1 ⊢_N_2, then ψ[Γ'] ⊢_ψ ψ[N_1] ⊢_ψ ψ[N_2].

**Proof:** (Skip) By simultaneous induction.

(1.) By induction on derivation of Γ, Γ' ⊢_M v : τ, case analysis on the last rule used.

**Rule T-VAR.** We have two cases here,

(a) x : M τ ∈ Γ, Γ'
(b) Γ, Γ' ⊢ τ

If x : M τ ∈ Γ, then using rule TENV-MAPP,

(c) x ⊢_M v ∈ ψ
(d) . ⊢_M v : τ
(e) M is closed.

Since M is closed,

(f) ψ[M] = M
And so, we want to prove, since \( \psi \) is closed,

(i) \( \psi[\tau] = \tau \)

With (c) and (g) use rule \texttt{VL-VAR} to get,

(j) \( \psi[[x]]_{\psi[[\Gamma]]} = v \)

We want to prove,

(k) \( \psi[[\Gamma']] \vdash \psi[x]_{\psi[[\Gamma']]} : \psi[[\tau]] \)

i.e. (substitute using (f)),

(l) \( \psi[[\Gamma']] \vdash \psi[x]_{\psi[[\Gamma']]} : \psi[[\tau]] \)

i.e. (substitute using (j) and (i)),

(m) \( \psi[[\Gamma']] \vdash \psi[\tau] \).

which is true by weakening (d).

If \( x : \tau \in \Gamma' \), then

(c) \( x : \psi[\Gamma] \psi[\tau] \in \psi[[\Gamma']] \)

Using I.H. on (b), we get,

(d) \( \psi[[\Gamma']] \vdash \psi[\tau] \)

Since \( \text{dom}(\Gamma) \cap \text{dom}(\Gamma') = \phi \), we have

(e) \( x \notin \text{dom}(\psi) \)

And so,

(f) \( \psi[[x]]_{\psi[[\Gamma']]} = x \)

We want to prove,

(g) \( \psi[[\Gamma']] \vdash \psi[x]_{\psi[[\Gamma']]} : \psi[[\tau]] \)

Substitute (f), we want

(h) \( \psi[[\Gamma']] \vdash \psi[\Gamma] \psi[x] \psi[\tau] \)

Derive using rule \texttt{T-VAR} on (c) and (d).

The second case is,

(a) \( x : \tau \in \Gamma, \Gamma' \)

(b) \( \Gamma \vdash \tau \)

Split on two cases \( \Gamma \) and \( \Gamma' \) as above.

\textbf{Rule T-UNIT.} We have,

(a) \( v = () \)

(b) \( \tau = \text{unit} \)

Using rule \texttt{VL-UNIT} and rule \texttt{T-UNIT},

(c) \( \psi[[()]]_{\psi[[\Gamma]]} = () \)

(d) \( \psi[[\text{unit}]] = \text{unit} \)

So, now we need to prove \( \psi[[\Gamma']] \vdash \psi[[\text{unit}]] : \text{unit} \).

Follows from rule \texttt{T-UNIT}.

\textbf{Rules T-INJ and T-PROD.} Use I.H. on premises and then corresponding typing rule.

\textbf{Rule T-PRINC.} Similar to rule \texttt{T-UNIT}.

\textbf{Rules T-PSONE, T-PSUNION, and T-PSVAR.} Use I.H. on premises and then corresponding typing rule.

\textbf{Rule T-MSUB.} We have,

(a) \( \Gamma, \Gamma' \vdash N \)

(b) \( \Gamma, \Gamma' \vdash N \vdash M \)

(c) \( \Gamma, \Gamma' \vdash N \vdash \tau \)

Use I.H. on (a) and (c) to get,

(d) \( \psi[[\Gamma']] \vdash \psi[N] \psi[x]_{\psi[[\Gamma']]} : \psi[[\tau]] \)

Use I.H. on (b),

(e) \( \psi[[\Gamma']] \vdash \psi[N] \vdash \psi[M] \)

Use I.H. on (a),

(f) \( \psi[[\Gamma']] \vdash \psi[N] \)

With (d), (e), and (f), use rule \texttt{T-MSUB} to get,

(g) \( \psi[[\Gamma']] \vdash \psi[N] \psi[x]_{\psi[[\Gamma']]} : \psi[[\tau]] \)

With (e), use Lemma A.14 to get,

(h) \( \psi[[x]]_{\psi[[\Gamma']]} = \psi[x]_{\psi[[\Gamma']]} \)

Substitute in (g) to get the proof.

(2.) Similar to proof of (1.)

(3.) Induction on derivation of \( \Gamma, \Gamma' \vdash \tau_1 <: \tau_2 \), case analysis on the last rule.

\textbf{Rule S-REFL.} Use Rule \texttt{S-REFL} on \( \psi[[\tau]] \)

\textbf{Rule S-TRANS.} Use I.H. on premises, and then rule \texttt{S-TRANS}.

\textbf{Rule S-SUM.} Use I.H. on premises, and then rule \texttt{S-SUM}.

\textbf{Rule S-PROD.} Use I.H. on premises, and then rule \texttt{S-PROD}.

\textbf{Rule S-PRINCS.}

\textbf{Rules S-WIRE, S-ARRAY, and S-SHARE.} Use I.H. on premises and then corresponding rule.

\textbf{Rule S-ARROW.} We have,

(a) \( \Gamma, \Gamma' \vdash \tau_1 <: \tau_1' \)

(b) \( \Gamma, \Gamma', x : \tau_1' \vdash \tau_2 <: \tau_2' \)

Use I.H. on (a),

(c) \( \psi[[\Gamma']] \vdash \psi[[\tau_1]] <: \psi[[\tau_1]] \)

Use I.H. on (b),

(d) \( \psi[[\Gamma']], x : \psi[[\tau_1]] : \psi[[\tau_2]] <: \psi[[\tau_2]] \)

With (c) and (d), use rule \texttt{S-ARROW}.

(4.) By induction on derivation of \( \Gamma, \Gamma' \vdash \tau \), case analysis on the last rule used.

\textbf{Rule WF-UNIT.} Since \( \psi[[\text{unit}]] = \text{unit} \), use rule \texttt{WF-UNIT}.
Rule WF-SUM. Use I.H. on the premises, and then use rule WF-SUM.

Rule WF-PROD. Use I.H. on the premises, and then use rule WF-PROD.

Rule WF-PRINC. Use I.H. on the premise, and then use rule WF-PRINC.

Rule WF-ARROW. We have,
(a) \( \tau = x : \tau_1 \arr \tau_2 \)
(b) \( \Gamma, \Gamma' \vdash \tau_1 \)
(c) \( \Gamma, \Gamma', x : \tau_1 \vdash \epsilon \)
(d) \( \Gamma, \Gamma', x : \tau_1 \vdash \tau_2 \)

Using I.H. on (b), (c), and (d), we get,
(e) \( \psi([\Gamma']) \vdash \psi([\tau_1]) \)
(f) \( \psi([\Gamma']), x : \psi([\tau_1]) \vdash \psi([\epsilon]) \)
(g) \( \psi([\Gamma']), x : \psi([\tau_1]) \vdash \psi([\tau_2]) \)

Use rule WF-ARROW on (e), (f), and (g).

Rule WF-WIRE. Use I.H. on the premises, and then use rule WF-WIRE.

Rule WF-ARRAY. Use I.H. on the premise, and then use rule WF-ARRAY.

Rule WF-SHARE. Similar to rule WF-WIRE.

(5.), (6.), (7.), (8.): Induction on respective derivations.

Lemma A.16: (Well-formedness of runtime environment)

Let \( \Sigma \vdash \psi \rightarrow \Gamma \) and \( \Gamma \vdash M \).

1) If \( \Gamma \vdash v : \tau \), then \( \vdash_{\psi[M]} \psi[v] : \psi[\tau] \).
2) If \( \Gamma \vdash v : \tau \), then \( \vdash \psi[v] : \psi[\tau] \).
3) If \( \Gamma \vdash \tau_1 \prec \tau_2 \), then \( \vdash \psi[\tau_1] \prec \psi[\tau_2] \).
4) If \( \Gamma \vdash \tau \), then \( \vdash \psi[\tau] \).
5) If \( \Gamma \vdash \phi \), then \( \vdash \psi[\phi] \).
6) If \( \Gamma \vdash \lambda \), then \( \vdash \psi[\lambda] \).
7) If \( \Gamma \vdash \epsilon \), then \( \vdash \psi[\epsilon] \).
8) If \( \Gamma \vdash N_1 \triangleright N_2 \), then \( \vdash \psi[N_1] \triangleright \psi[N_2] \).

Proof: Corollary of Lemma A.15 with \( \Gamma' = \).
With (c), use Lemma A.12 to get,
(d) $\psi\{v\}_M = (v_1, v_2)$

$C$ can now take a step using STPC-LOCAL and STPL-FST to
$M\{\sigma; \kappa; \psi; v_1\}$.

Rule T-SND. Similar to rule T-FST.

Rule T-CASE. We have,
(a) $e = \text{case } (v, x_1, e_2, x_2.e_2)$
(b) $\Gamma \vdash_M v : t_1 + t_2$
(c) $\Gamma, x_i : \tau_i \vdash_M e_i : \tau; \epsilon_i$
(d) $\Gamma \vdash \tau$
(e) $\Gamma \vdash M \triangleright \epsilon_i$

With (b), use Lemma A.16 to get,
(d). $\vdash_M \psi\{v\}_M : (\psi\[\tau_1\]) + (\psi\[\tau_2\])$

With (d), use Lemma A.12 to get,
(e) $\psi\{v\}_M = \mathbf{inj}_i.v'$

$C$ can now take a step using STPC-LOCAL and STPL-CASE to
$M\{\sigma; \kappa; \psi\{x_i \mapsto v'\}; \epsilon_i\}$.

Rule T-LAM. We have,
(a) $e = \lambda x. e$

$C$ can take a step using STPC-LOCAL and STPL-LAMBDA.

Rule T-APP. We have,
(a) $e = v_1 v_2$
(b) $\Gamma \vdash_M v_1 : x: t_1 \rightarrow t_2$
(c) $\Gamma \vdash t_2 : \tau_2$

With (b) and (c) use Lemma A.16 to get,
(d). $\vdash_M \psi\{v_1\}_M : x : \tau_1 \rightarrow \psi[t_2]$\]
(e). $\vdash \psi[v_2] : \psi[t_1]$

With (d) use Lemma A.12 to get,
Case 1: $\psi[v_1]_M = \mathbf{clo}_s (\psi; \lambda x. e)$

$C$ can take a step using STPC-LOCAL and STPL-APPLY.

Case 1: $\psi[v_1]_M = \mathbf{clo}_s (\psi; \mathbf{fix \ x. \ \lambda y. e})$

$C$ can take a step using STPC-LOCAL and STPL-APPLY.

Rule T-LET1. We have,
(a) $e = \text{let } x \equiv N \mathbf{in} e_2$
(b) $M = m(\_)$
(c) $N = \_w$
(d) $\Gamma \vdash M \triangleright N$
(e) $\Gamma \vdash N e_1 : \tau_1 ; \epsilon_1$
(f) $\Gamma, x : \tau \vdash_M e_2 : \tau_2 ; \epsilon_2$
(g) $\Gamma \vdash \tau_2$
(h) $\Gamma \vdash M \triangleright \epsilon_2$

With (d), use Lemma A.8 to get,
(i) $\Gamma \vdash N$

With (i), use Lemma A.16 to get,
(j) $\vdash \psi[N]$

With (d), use Lemma A.16 to get,
(k) $\vdash M \triangleright \psi[N]$

With (k), use Lemma A.13 to get,

Either
(l) $M = p(w), N = p(w_1)$, and $w_1 \subseteq w$

In this case, $C$ can take a step using STPC-Delpar to
$p(w_1)\{\sigma; \kappa :: (p(w); \psi; x.e_2); \psi; e_1\}$

Or
(l) $M = p(w)$ and $N = s(w)$.

In this case, $C$ can take a step using STPC-DELSEC to
$p(w)\{\sigma; \kappa :: (p(w); \psi; x.e_2); \psi; \mathbf{seu}_w(e_1)\}$

Or
(l) $M = m(w)$ and $N = m(w)$.

Depending on $m$, one of the above applies.

Rule T-LET2. We have,
(a) $e = \text{let } x = e_1 \mathbf{in} e_2$
(b) $\Gamma \vdash_M e_1 : \tau_1 ; \epsilon_1$
(c) $\Gamma, x : \tau_1 \vdash_M e_2 : \tau_2 ; \epsilon_2$
(d) $\Gamma \vdash \tau_2$
(e) $\Gamma \vdash M \triangleright \epsilon_2$

Case 1: $\psi[v_1]_M = \mathbf{clo}_s (\psi; \lambda x. e)$

$C$ can take a step using STPC-LOCAL to
$M\{\sigma; \kappa :: (\psi; x.e_2); \psi; e_1\}$.

Rule T-FIX. $C$ can take a step using STPC-LOCAL and STPL-FIX.

Rule T-ARRAY. We have,
(a) $e = \text{array}(v_1, v_2)$
(b) $\Gamma \vdash_M v_1 : \mathbf{nat}$
(c) $\Gamma \vdash_M v_2 : \tau$

With (b) and (c), use Lemma A.16 to get,
(d). $\vdash_M \psi[v_1]_M : \mathbf{nat}$
(e). $\vdash_M \psi[v_2]_M : (\psi[\tau])$

$C$ can take a step using STPC-LOCAL and STPL-ARRAY to
$M\{\ell :: \{\psi[v_2]_M\}_k \}; \kappa; \psi; \ell\}$.

Rule T-SELECT. We have,
(a) $e = \text{select}(v_1, v_2)$
(b) $\Gamma \vdash_M v_1 : \text{Array} \tau$
(c) $\Gamma \vdash_M v_2 : \mathbf{nat}$

With (b) and (c), use Lemma A.16 to get,
(d) \( \vdash_M \psi[v_2]_M : \text{Array} \psi[\tau] \)

(e) \( \vdash_M \psi[v_2]_M : \text{nat} \)

With (d) use Lemma A.12 to get,

(f) \( \psi[v_1]_M = \ell \)

(g) \( \Sigma(\ell) = \psi[\tau] \)

Since \( \Sigma \vdash \sigma \mathsf{wf} \), \( \ell \in \text{dom}(\sigma) \)

\( C \) can take a step using \( \mathsf{STPC-LOCAL} \) with \( \mathsf{STPL-SELECT} \) or \( \mathsf{STPL-SEL-ERR} \).

\textit{Rule T-UPDATE.} Similar to rule T-SELECT.

\textit{Rule T-WIRE.} We have,

(a) \( e = \text{wire}_{w_1}(\nu) \)

(b) \( M = m(w_2) \)

(c) \( \Gamma \vdash w_1 : \mathsf{ps} (\nu \subseteq w_2) \)

(d) \( m = s \Rightarrow N = M \) and \( m = p \Rightarrow N = p(w_1) \)

(e) \( \Gamma \vdash_N \nu : \tau \)

With (c), use Lemma A.16 to get,

(f) \( \vdash \psi[w_1] : \mathsf{ps} (\nu \subseteq w_2) \)

Case 1: \( m = s \Rightarrow N = s(w_2) \)

With (e), use Lemma A.16 to get,

(g) \( \vdash_M \psi[v]_M : \psi[\tau] \)

\( C \) can take a step using \( \mathsf{STPC-LOCAL} \) and \( \mathsf{STPL-WIRE} \) to \( M \{ \sigma; \kappa; \psi; (\psi[v])_N \}_M(w_{w_1}) \} \).

Case 2: \( m = p \Rightarrow N = p(w_1) \)

With (e), use Lemma A.16 to get,

(g) \( \vdash_M \psi[w_1] : \psi[\tau] \)

\( C \) can take a step using \( \mathsf{STPC-LOCAL} \) and \( \mathsf{STPL-WIRE} \) to \( M \{ \sigma; \kappa; \psi; (\psi[v])_N \}_M(w_{w_1}) \} \).

\textit{Rule T-WPROJ.} We have,

(a) \( e = w_{w_2} \)

(b) \( M = m(w_1) \)

(c) \( m = p \Rightarrow \phi = \nu = w_1 \) and \( m = s \Rightarrow \phi = \nu \subseteq w_1 \)

(d) \( \Gamma \vdash_M \nu : \mathsf{W} w_2 \tau \)

(e) \( \Gamma \vdash w_2 : \mathsf{ps} (\phi \land \mathsf{singl}(\nu)) \)

Case 1: \( m = p \Rightarrow \phi = \nu = w_1 \)

With (e), use Lemma A.16 to get,

(f) \( \vdash (\psi[w_2]) : \mathsf{ps} (\psi[(\nu = w_1)]) \land (\mathsf{singl}(\nu)) \)

With (f), use Lemma A.12 to get,

(g) \( \psi[w_2] = p, w_1 = p, \) and \( M = p(\{p\}) \)

With (d) use Lemma A.16 to get,

(h) \( \vdash_M \psi[v]_M : \mathsf{W} \psi[w_2] \psi[\tau] \)

With (h), use Lemma A.12 to get,

(i) \( \psi[v]_M = v_1 + v_2 \) and \( p \in \text{dom}(v_1 + v_2) \).

\( C \) can now take a step using \( \mathsf{STPC-LOCAL} \) and \( \mathsf{STPL-PARPROJ} \).

Case 2: \( m = s \Rightarrow \phi = \nu \subseteq w_1 \)

With (e), use Lemma A.16 to get,

(f) \( \vdash (\psi[w_2]) : \mathsf{ps} (\psi[(\nu \subseteq w_1)] \land (\mathsf{singl}(\nu))) \)

With (f), use Lemma A.12 to get,

(g) \( \psi[w_2] = p, w_1 = p \cup w'', \) and \( M = s(p \cup w'') \)

With (d), use Lemma A.16 to get,

(h) \( \vdash_M \psi[w]_M : \mathsf{W} \psi[w_2] \psi[\tau] \)

With (h), use Lemma A.12 to get,

(i) \( \psi[w]_M = v_1 + v_2 \) and \( p \in \text{dom}(v_1 + v_2) \).

\( C \) can now take a step using \( \mathsf{STPC-LOCAL} \) and \( \mathsf{STPL-PARPROJ} \).

\textit{Rule T-WIREUN.} We have,

(a) \( e = v_1 + v_2 \)

(b) \( \Gamma \vdash v_1 : \mathsf{W} w_1 \tau \)

(c) \( \Gamma \vdash v_2 : \mathsf{W} w_2 \tau \)

With (b) and (c) use Lemma A.16, and then Lemma A.12, and then \( C \) can take a step using \( \mathsf{STPC-LOCAL} \) and \( \mathsf{STPL-WIREUN} \).

\textit{Rule T-WFOLD.} We have,

(a) \( e = \mathsf{wfold}_p(v_1, v_2, v_3) \)

(b) \( M = s(\_2) \)

(c) \( \phi = \nu \subseteq w \land \mathsf{singl}(\nu) \)

(d) \( \Gamma \vdash_M v_1 : \mathsf{W} w \tau \)

(e) \( \Gamma \vdash_M v_2 : \tau_2 \)

(f) \( \Gamma \vdash_M v_3 : \tau_2 \Rightarrow \mathsf{ps} \phi \Rightarrow \tau \Rightarrow \tau_2 \)

With (d) use Lemma A.10 to get,

(g) \( \Gamma \vdash \mathsf{W} w \tau \)

With (g) invert rule \( \mathsf{WFOLD} \) to get,

(h) \( \Gamma \vdash w : \mathsf{ps} \phi \)

With (h) use Lemma A.16 to get,

(i) \( \vdash \psi[w] : \mathsf{ps} (\psi[\phi]) \)

With (i) use Lemma A.12 to get,

(j) \( \psi[w] = w_1 \cup w_2 \)

\( C \) can take a step using \( \mathsf{STPC-LOCAL} \) and either \( \mathsf{STPL-WFOLD1} \) or \( \mathsf{STPL-WFOLD2} \).

\textit{Rule T-WAPP.} We have,

(a) \( e = \mathsf{wapp}_p(v_1, v_2) \)

(b) \( M = p(\_2) \)

(c) \( \Gamma \vdash_M v_1 : \mathsf{W} w \tau_1 \)
(d) $\Gamma \vdash_M v_2 : W \ w \ (\tau_1 \rightarrow \tau_2)$

Similar to rule T-WFOLD now.

**Rule T-WAPL.** Similar to rule T-WFOLD.

**Rule T-WCOPY.** $C$ can take a step using STPL-LOCAL with STPL-WCOPY.

**Rule T-MAKESH.** We have,
(a) $e = \text{makesh}(v)$
(b) $M = s(w)$
(c) $\Gamma \vdash_M v : \tau$

With (c) use Lemma A.16 we get,
(d) $\vdash_M \psi[v]_M : \psi[\tau]$ $C$ can take a step using STPL-LOCAL with STPL-MAKESH.

**Rule T-COMBSH.** We have,
(a) $e = \text{combsh}(v)$
(b) $M = s(w)$ (therefore $w$ is closed)
(c) $\Gamma \vdash_M v : \text{Sh} \ w \ \tau$

With (c) use Lemma A.16 we get,
(d) $\vdash_M \psi[v]_M : \text{Sh} \ w \ \psi[\tau]$ $C$ can take a step using STPL-LOCAL with STPL-COMBSH.

**Rule T-SECBLK.** We have,
(a) $e = \text{secure}_w(e)$
(b) $M = m(w')$
(c) $\Gamma \vdash w : \text{ps} \ (\nu = w')$

Use Lemma A.16 and then Lemma A.12 on (c). $C$ can take a step using STPL-SECENTER.

**Rule T-VALUE.** We consider case when
(a) $e = v$
(b) $\Gamma \vdash_M v : \tau$

If $\kappa$ is empty then by rule HALTED-ANSWER $C$ is empty.

If $\kappa$ is not empty, then depending on top frame, $C$ can take a step using STPL-POPSTK1 or STPL-POPSTK2.

(After applying Lemma A.16 and then Lemma A.12 on (b).)

---

**Theorem A.2. (Preservation)** If $\Sigma_1 \vdash_C \Gamma : \tau$ and $C \rightarrow C_2$, then there exists $\Sigma_2 \supseteq \Sigma_1$ s.t. $\Sigma_2 \vdash_C : \tau$.

**Proof: (Skip)** Case analysis on $C_1 \rightarrow C_2$.

**STPL-CASE.** We have,
(a) $C_1 = M\{\sigma; \kappa; \psi; \text{case}(v, x_1, e_1, x_2, e_2)\}$
(b) $C_2 = M\{\sigma; \kappa; \psi; \{x_1 \rightarrow v'\}; e_1\}$
(c) $\psi[v]_M = \text{inj}_v v'$
(d) $\Sigma_1 \vdash \sigma \ \text{wf}$
(e) $\vdash_M$
(f) $\Sigma_1 \vdash_M \kappa : \tau_1 \rightarrow \tau$
(g) $\Sigma_1 \vdash \psi \rightarrow \Gamma$
(h) $\Sigma_1 ; \Gamma \vdash_M \text{case}(v, x_1, e_1, x_2, e_2) : \tau_1; \epsilon$
(i) $\Gamma \vdash M > \epsilon$

Inverting rule T-CASE on (h), we get,
(j) $\Gamma \vdash_M v : \tau_1 + \tau_2$
(k) $\Gamma, x_i : \tau_i \vdash_M e_i : \tau_1; \epsilon$
(l) $\Gamma \vdash \tau$
(m) $\phi = \epsilon, e_1, e_2$

With (j) use Lemma A.16 to get,
(n) $\vdash_M \psi[\tau]_2$ $C$ can take a step using STPL-CASE.

**STPL-CONFIG.** We have,(a) $C_1 = M\{\sigma; \kappa; \psi; \text{fst}(v)\}$
(b) $C_2 = M\{\sigma; \kappa; \psi; v_1\}$
(c) $\psi[v]_M = (v_1, v_2)$
(d) $\Sigma_1 \vdash \sigma \ \text{wf}$
(e) $\vdash_M$
(f) $\Sigma_1 \vdash_M \kappa ; \psi[\tau_1] \rightarrow \tau$
(g) $\Sigma_1 \vdash \psi \rightarrow \Gamma$
(h) $\Sigma_1 ; \Gamma \vdash_M \text{fst}(v) : \tau_1; \cdot$
(i) $\Gamma \vdash M > \epsilon$

Inverting rule T-FST on (h) to get,
(j) $\Gamma \vdash_M v : \tau_1 \times \tau_2$

With (j) and (c) use Lemma A.16 to get,
(k) $\vdash_M v_1 : \psi[\tau_1]$

Choose $\Sigma_2 = \Sigma_1$

With (d), (e), (f) (g), (k) (after weakening), we get,
\[
\Sigma_2 \vdash M\{\sigma; \kappa; \psi; v_1\} : \tau
\]
STPL-SND. Similar to STPL-FST.

STPL-LAMBDA. There is no change in the stack environment, and the closure has same type as lambda.

STPL-APPLY. We have, (a) \(C_1 = M\{\sigma; \kappa; \psi; v_1; v_2\}\)
(b) \(C_2 = M\{\sigma; \kappa; \psi_2; x \mapsto v'; \epsilon\}\)
(c) \(\psi_1[v_1] = w\) with \(\epsilon\)
(d) \(\psi_1[v_2] = w'\)
(e) \(\Sigma \vdash \kappa: \psi_1[t_1] \Rightarrow \tau\)
(f) \(\Gamma \vdash M_1: \psi_1: \tau_1: \epsilon\)
(g) \(\Gamma \vdash M \triangleright \epsilon\)

Invert rule T-APP on (f) to get,
(h) \(\Gamma \vdash M_1: \tau_2 \Rightarrow \tau_1\)
(i) \(\Gamma \vdash M_2: \tau_2\)
(j) \(\Gamma \vdash M \triangleright \epsilon_1[v_2/x]\)
(k) \(\epsilon = \epsilon_1[v_2/x]\)

With (h) and (c) use Lemma A.16 to get,
(l) \(\vdash_M \psi \circ (\psi_2; \lambda x. e) : (\psi [t_2]) (\psi [t_1])\)

Invert rule T-CLOS (on (i)) to get,
(m) \(\Sigma_1 \vdash \psi_2 \Rightarrow \Gamma_2\)
(n) \(\Gamma_2, x : \psi [t_2] \vdash M : \psi [t_1]; \psi [\epsilon_1]\)

With (i) and (d) use Lemma A.16 to get,
(o) \(\vdash_M v' : \psi [t_2]\)

i.e.
(p) \(\vdash_M v' : \psi [t_2]\)

From (m) and (o), use rule TENV-MAPP2 to derive,
(q) \(\Sigma_1 \vdash \psi_2 \{x \mapsto v'\} \Rightarrow \Gamma_2, x : \psi [t_2]\)

(e) can also be written as,
(r) \(\Sigma \vdash M : \psi_2 [(\psi [t_1])] \Rightarrow \tau\)

With (q), (n), (r), we get \(\Sigma_1 \vdash M \{\sigma; \kappa; \psi_2 \{x \mapsto v'\}; \epsilon\} : \tau\)

(effect delegation comes from Lemma A.16 on (j).)

STPL-FIX. Similar to STPL-LAMBDA.

STPL-FIXAPPLY. Similar to STPL-APPLY.

STPL-ARRAY. Standard proof, choose \(\Sigma_2\) as \(\Sigma_1\) with new \(\ell\). Other array cases are also standard.

STPL-MAKESH. We have, (a) \(C_1 = s(w)\{\sigma; \kappa; \psi; makesh(v')\}\)
(b) \(C_2 = s(w)\{\sigma; \kappa; \psi; sh w v'\}\)
(c) \(\psi[w] = w'\)
(d) \(\Sigma_1 \vdash_s w) \kappa : \psi [t_1] \Rightarrow \tau_2\)
(e) \(\Sigma_1 \vdash \psi \Rightarrow \Gamma\)
(f) \(\Gamma \vdash s(w) makesh(v) : \tau_1: \epsilon\)

Invert rule T-MAKESH on (f) to get,
(g) \(\tau_1 = Sh w \tau_2\)
(g') \(\Gamma \vdash s(w) v : \tau_2\)

Use Lemma A.16 on (g') and (c) to get,
(h) \(\vdash s(w) v' : \psi [t_2]\)

Use rule T-SH on (h) to get,
(i) \(\vdash s(w) sh w v' : Sh w \psi [t_2]\)

Observe that \(\psi [Sh w \psi [t_2]] = \psi [Sh w \tau_2]\)

Therefore, stack typing (d) still holds, and new configuration is well-typed.

STPL-COMBSH. Similar to STPL-MAKESH.

STPL-WIRE. We have, (a) \(C_1 = M\{\sigma; \kappa; \psi; wire_{w}(v)\}\)
(b) \(C_2 = M\{\sigma; \kappa; \psi; \{v'\}^{wires}_{w}\}\)
(c) \(\psi[w] = w'\)
(d) \(\psi[v] = v'\)
(e) \(M = m(\psi)\)
(f) \(\Gamma \vdash w : \psi [t_2]\)

Invert rule T-WIRE on (f) to get,
(h) \(\Gamma \vdash w : \psi [t_2]\)
(i) \(\Gamma \vdash m : \psi [t_2]\)

Case 1: \(m = \psi\)

Use Lemma A.16 on (j) to get,
(k) \(\vdash m \psi : \psi [t_2]\)

Using rule T-SINGLWIRE and rule T-WIRECAT, we can derive:
(l) \(\vdash s(w) \{v'\}^{wires}_{w} : W w' (\psi [t_2])\)

Observe that \(\psi [W w' (\psi [t_2])] = \psi [t_1]\).

Hence, stack typing (g) still holds, therefore \(C_2\) is well-typed.

Case 2: \(m = \psi\)

Use Lemma A.16 on (j) to get,
(k) \(\vdash m \psi : \psi [t_2]\)

Using rule T-SINGLWIRE and rule T-WIRECAT, we can derive:

Observe that \(\psi [W w' (\psi [t_2])] = \psi [t_1]\).

Hence, stack typing (g) still holds, therefore \(C_2\) is well-typed.
We have, 
\[
\Gamma \vdash M \{\sigma; \kappa; \psi; v_1[v_2]\}
\]
Using rule T-WIRECAT with (j) and (k) we get,
\[
\Gamma \vdash M \psi_{v_1}' + \psi_{v_2}' : W (\psi\{w_1]\} \cup (\psi\{w_2]\)) \psi\{\tau_2]\}
\] Proof now follows by showing that stack typing holds.

We have, (a) \(C_1 = M\{\sigma; \kappa; \psi; wapp_w(v_1, v_2)\}\)
\[
\Gamma \vdash M \{\sigma; \kappa; \psi; e\}
\]
Also, 
\[
RT, (\psi\{\tau_2]\}) \rightarrow \tau
\]
We have,
\[
\Gamma \vdash M : W (\psi\{\tau_2]\})
\]
With (h) and (i), we can derive stack typing in \(C_2\).

We have, (a) \(C_1 = M\{\sigma; \kappa; \psi; wapp_w(v_1, v_2)\}\)
\[
\Gamma \vdash M \{\sigma; \kappa; \psi; e\}
\]
Using rule T-WIRECAT we get,
\[
\Gamma \vdash M \psi_{v_1}' + \psi_{v_2}' : W (\psi\{w_1]\} \cup (\psi\{w_2]\)) \psi\{\tau_2]\}
\] We now consider typing of \(e\) from (f).

We have, (a) \(C_1 = M\{\sigma; \kappa; \psi; wapp_w(v_1, v_2)\}\)
\[
\Gamma \vdash M \{\sigma; \kappa; \psi; e\}
\]
Using rule T-WIRECAT with (j) and (k) we get,
(q) \vdash_M \text{wapp}_w(v'_1, v'_2) : W \; w' \; \psi[[\tau_2]]\

Using rule T-WIREUN we get,

(r) \vdash_M (\text{wire}_{(p)}(z_1 + z_4)) : W (\{p\} \cup w') \; \psi[[\tau_2]]

We also have,

(s) \psi[[W (\{p\} \cup w') \; \psi[[\tau_2]]]] = \psi[[W \; w \; \tau_2]]

and hence stack typing from (j) holds for \( C_2 \) as well.

**STPL-WAPS1.** Similar to STPL-WAPP1.

**STPL-WAPS2.** We have,

(a) \( C_1 = M \{\sigma; \kappa; \psi; \text{waps}_w(v_1, v_2)\} \)

(b) \( C_2 = M \{\sigma; \kappa; \psi; e\} \)

(b') \( e = \text{let} \ z_1 = v'_1[p] \text{ in } z_2 = v'_2 \; z_1 \text{ in } z_3 = \text{waps}_w(v'_1, v'_2) \text{ in } (\text{wire}_{(p)}(z_2 + z_3)) \)

(c) \( M = s((\{p\} \cup w') \cup w_1) \)

(d) \( \psi[[w]] = \{p\} \cup w' \)

(e) \( \psi[[v_1]]_M = v'_1 \)

(f) \( \psi[[v_2]]_M = v'_2 \)

(g) \( \Gamma \vdash_M \text{waps}_w(v_1, v_2) : W \; w \; \tau_2; \)

(h) \( \Sigma_1 \vdash_M \kappa : \psi[[\tau_2]] \equiv \tau \)

Inverting rule T-WAPS on (g) we get,

(i) \( \Gamma \vdash_M v_1 : W \; w \; \tau_1 \)

(j) \( \Gamma \vdash_M v_2 : \tau_1 \rightarrow \tau_2 \)

Using Lemma A.16 on (i) and (j),

(k) \( \vdash_M v'_1 : W \{p\} \cup w' \; (\psi[[\tau_1]]) \)

(l) \( \vdash_M v'_2 : (\psi[[\tau_1]]) \rightarrow (\psi[[\tau_2]]) \)

We now consider typing of \( e \) from (b')

Using rule T-WPROJ we get,

(m) \( \vdash_M v'_1[p] : \psi[[\tau_1]] \)

Using rule T-APP we get,

(n) \( \vdash_M v'_2 z_1 : \psi[[\tau_2]] \)

Using rule T-WIRE we get,

(o) \( \vdash_M \text{wire}_{(p)}(z_2) : W \{p\} \; (\psi[[\tau_2]]) \)

Using rule T-WAPS we get,

(p) \( \vdash_M \text{waps}_w(v'_1, v'_2) : W \; w' \; (\psi[[\tau_2]]) \)

Using rule T-WIREUN we get,

(q) \( \vdash_M (\text{wire}_{(p)}(z_2 + z_3)) : W (\{p\} \cup w') \; (\psi[[\tau_2]]) \)

Also,

(r) \( \psi[[W (\{p\} \cup w') \; (\psi[[\tau_2]])]] = \psi[[W \; w \; \tau_2]] \)

and hence stack typing (h) holds for \( C_2 \) also.

**STPL-WFOLD2.** We have, \( \text{wfold}_w(v'_1, v'_2) \)

(b) \( C_2 = M \{\sigma; \kappa; \psi; e\} \)

(c) \( \psi[[w]] = \{p\} \cup w' \)

(d) \( M = s((\{p\} \cup w') \cup w_1) \)

(e) \( \psi[[v_1]]_M = v'_1 \)

(f) \( \psi[[v_2]]_M = v'_2 \)

(g) \( \psi[[v_3]]_M = v'_3 \)

(h) \( e = \text{let} \; z_1 = v'_1[p] \text{ in } z_2 = v'_2 \; z_1 \text{ in } \text{wfold}_w(v'_1, v'_2, v'_3) \)

(i) \( \Gamma \vdash_M \text{wfold}_w(v_1, v_2, v_3) : \tau_2; \)

(j) \( \Sigma_1 \vdash_M \kappa : \psi[[\tau_2]] \equiv \tau \)

Inverting rule T-WFOLD on (i) we get,

(k) \( \Gamma \vdash_M v_1 : W \; w \; \tau \)

(l) \( \Gamma \vdash_M v_2 : \tau_2 \)

(m) \( \Gamma \vdash_M v_3 : \tau_2 \rightarrow \text{ps} (\nu \subseteq W \wedge \text{singl}(\nu)) \rightarrow \tau \rightarrow \tau_2 \)

Using Lemma A.16 on (k), (l), and (m), we get,

(n) \( \vdash_M v'_1 : W (\{p\} \cup w') \; (\psi[[\tau]]) \)

(o) \( \vdash_M v'_2 : (\psi[[\tau_2]]) \)

(p) \( \vdash_M v'_3 : (\psi[[\tau_2]]) \rightarrow \text{ps} (\nu \subseteq (\{p\} \cup w') \wedge \text{singl}(\nu)) \rightarrow (\psi[[\tau]]) \rightarrow (\psi[[\tau_2]]) \)

We now consider typing of \( e \) from (h).

Using rule T-WPROJ we get,

(q) \( \vdash_M v'_1[p] : \psi[[\tau]] \)

Using rule T-APP we get,

(r) \( \vdash_M v'_2 v'_2 p \; z_1 : \psi[[\tau_2]] \)

Using rule T-WFOLD we get,

(s) \( \vdash_M \text{wfold}_w(v'_1, v'_2, v'_3) : \psi[[\tau_2]] \)

And,

(t) \( \psi[[\psi[[\psi[[\tau_2]]]]]] = \psi[[\tau_2]] \)

and hence stack typing (j) remains valid for \( C_2 \).

**STPC-LET.** We have, \( \text{let} \; x = e_1 \in e_2 \)

(b) \( C_2 = M \{\sigma; \kappa : (\psi; x; e_2) ; \psi; e_1\} \)

(c) \( \Sigma_1 \vdash \sigma \text{ wf} \)

(d) \( \vdash_M \)

(e) \( \Sigma_1 \vdash_M \kappa : \psi[[\tau_1]] \equiv \tau \)

(f) \( \Sigma_1 \vdash \psi \rightarrow \Gamma \)

(g) \( \Sigma_1; \Gamma \vdash_M \text{let} \; x = e_1 \in e_2 : \tau_1; \epsilon \)

Invert rule T-LET on (g) to get,

(h) \( \Gamma \vdash_M e_1 : \tau_1; \epsilon \)

(i) \( \Gamma; x : \tau'_1 \vdash_M e_2 : \tau_1; \epsilon \)
Lemma A.10 on (h)), we have

\[ \Sigma \vdash M \{ \sigma; \kappa :: \langle \psi; x. e_2 \rangle; \psi; e_1 \} : \tau, \]

we need to prove

(e) \( \Sigma_2 \vdash M \ \kappa :: \langle \psi; x. e_2 \rangle; \psi[\tau_1] \Rightarrow \tau \)

i.e. (from rule rule \text{TSTK-FRAME2}) we need to prove,

(f) \( \Gamma; x : \tau_1 \vdash M \ \epsilon : \tau_1; \epsilon \)

and

(g) \( \Sigma_2 \vdash M \ \kappa :: \psi[\tau_1] \Rightarrow \tau \)

Choose \( \Sigma_2 = \Sigma_1 \), then (f) is same as (i) and (g) is same as (e).

Thus (e) holds.

With (c), (d), (e), (f), (h) (effect delegation follows from Lemma A.10 on (h)), we have

\[ \Sigma_2 \vdash M \{ \sigma; \kappa :: \langle \psi; x. e_2 \rangle; \psi; e_1 \} : \tau \]

**STPC-DELPAR.** We have, \( \text{(a')} \) \( M = p(w_1 \cup w_2) \)

(a) \( \mathcal{C}_1 = M \{ \sigma; \kappa :: \langle \psi; \psi \rangle; \psi; e_1 \} \)

(b) \( \mathcal{C}_2 = p(w_2) \{ \sigma; \kappa :: \langle M; \psi; x. e_2 \rangle; \psi; e_1 \} \)

(c) \( \psi[\psi] = w_2 \)

(d) \( \Sigma_1 \vdash M \ \kappa :: \psi[\tau_1] \Rightarrow \tau \)

(e) \( \Gamma \vdash_M \ let \ x : p(w_1) \ \epsilon \ \ : \ \epsilon \ \ : \ \epsilon \)

(f) \( \Gamma \vdash M \ \vdash \epsilon \)

Inverting rule \text{T-LET1} on (e) we get,

(g) \( \Gamma \vdash p(w_2) \ \epsilon \ : \ \epsilon \)

(h) \( \Gamma \vdash x : p(w_2) \ \tau_2 \vdash M \ \epsilon : \ \epsilon \)

(i) \( \epsilon = p(w_2) \)

(j) \( \Gamma \vdash p(w_1 \cup w_2) \ \vdash \epsilon \)

To prove \( \mathcal{C}_2 \) is well-typed, we need to prove:

(k) \( \Gamma \vdash p(w_2) \ \epsilon \ : \ \epsilon \)

(l) \( \Sigma_1 \vdash p(w_2) \kappa :: \langle M; \psi; x. e_2 \rangle; \psi[\tau_2] \Rightarrow \tau \)

(k) follows from (g)

To prove (l), we need to prove:

(m') \( \Gamma \vdash \tau_2 \) (follows from Lemma A.10 on (g)).

(m) \( \Gamma \vdash x : p(w_2) \ \tau_2 \vdash M \ \epsilon : \ \epsilon \)

(n) \( \Sigma_1 \vdash M \ \kappa :: \psi[\tau_1] \Rightarrow \tau \)

(m) follows from (h), (n) follows from (d).

**STPC-DELSSC.** We have, \( \text{(a') } \mathcal{C}_1 = M(w) \{ \sigma; \kappa :: \langle M; \psi; x. e \rangle; \psi; v \} \)

(b) \( \mathcal{C}_2 = M(w) \{ \sigma; \kappa :: \langle M(w); \psi; x. e_2 \rangle; \psi[\psi[\psi]] \} \)

(c) \( \Sigma_1 \vdash M(w) \kappa :: \psi[\tau] \Rightarrow \tau \)

(d) \( \Gamma \vdash M(w) \ \let \ \epsilon \ : \ \epsilon \)

(d') \( \Gamma \vdash \epsilon \Rightarrow \sigma \)

Inverting rule \text{T-LET1} on (d) we get,

(e) \( \Gamma \vdash \epsilon \) \( \epsilon \ : \ \epsilon \)

(f) \( \Gamma \vdash x : m(w) \ \tau_2 \vdash M(w) \ \epsilon : \ \epsilon \)

(g) \( \epsilon = \sigma \)

Inverting rule \text{D-REFL} or rule \text{D-SEC} on (d') we get,

(h) \( \Gamma \vdash w : \psi \ (\nu = w) \)

Using rule \text{T-SECBLK} on (h) and (e) we get,

(i) \( \Gamma \vdash m(w) \ \kappa :: \psi \Rightarrow \epsilon \)

To prove \( \mathcal{C}_2 \) is well-typed, we need to prove,

(j) \( \Sigma_1 \vdash m(w) \kappa :: \langle m(w); \psi; x. e_2 \rangle; \psi[\tau_2] \Rightarrow \tau \)

i.e.

(l) \( \Gamma \vdash x : m(w) \ \tau_2 \vdash M(w) \ \epsilon : \ \epsilon \)

and

(m) \( \Sigma_1 \vdash m(w) \kappa :: \psi[\tau_1] \Rightarrow \tau \)

(l) follows from (f) and (h), (m) follows from (c).

**STPC-SECENTER.** We have, \( \text{(a) } \mathcal{C}_1 = m(w) \{ \sigma; \kappa :: \langle M; \psi; x. e \rangle; \psi; v \} \)

(b) \( \mathcal{C}_2 = \psi \{ \sigma; \kappa :: \psi \} \)

(c) \( \psi[\psi] = w \)

(d) \( \Gamma \vdash m(w) \ \kappa :: \psi[\tau] \Rightarrow \tau \)

(e) \( \Sigma_1 \vdash m(w) \kappa :: \psi[\tau_1] \Rightarrow \tau \)

Inverting rule \text{T-SECBLK} on (d) we get,

(f) \( \Gamma \vdash w : \psi \ (\nu = w) \)

(g) \( \Gamma \vdash \epsilon \) \( \epsilon \)

Proof now follows from (g), (f), and (e).

**STPC-POPSBLK.** We have, \( \text{(a) } \mathcal{C}_1 = M \{ \sigma; \kappa :: \langle M; \psi; x. e \rangle; \psi; v \} \)

(b) \( \mathcal{C}_2 = M \{ \sigma; \kappa :: \langle M; \psi; x. e_2 \rangle; \psi[\psi[\psi]] \} \)

(c) \( \nu = \psi \)

(d) \( \Gamma \vdash m(w) \ \kappa :: \psi[\tau] \Rightarrow \tau \)

(e) \( \psi[\psi] = v' \)

(f) \( \Sigma_1 \vdash \psi[\psi] \Rightarrow \tau \)

(g) \( \Gamma \vdash v : \psi[\tau] \)

Inverting rule \text{TSTK-FRAME1} on (f) we get,

(h') \( \Gamma_1 \vdash \tau \)

(h) \( \Gamma_1 \vdash x : m(w) \ \epsilon : \ \epsilon \)

(i) \( \Sigma_1 \vdash m(w) \ \kappa :: \psi[\tau] \Rightarrow \tau \)

Using Lemma A.16 on (g) to get,

(j) \( \vdash \psi \) \( \psi[\tau] \)

Using Lemma A.17,
Theorem A.3: (Sound simulation)

Let $\Sigma \vdash C : \tau$, $C_1 \rightarrow C_2$, $C$ st, and $\text{slice}_w(C_1) \sim \pi_1$, where $w$ is the set of all parties. Then, there exists $\pi_2$ s.t. $\pi_1 \rightarrow^* \pi_2$ and $\text{slice}_w(C_2) \rightarrow \pi_2$.

Proof: Case analysis on $C_1 \rightarrow C_2$.

STPC-Delpar. We have,

(a) $C_1 = p(w_1 \cup w_2)\{\sigma; \kappa; \psi; \text{let } x \overset{p(w')}{=} e_1 \text{ in } e_2\}$

(b) $C_2 = p(w_2)\{\sigma; \kappa; \psi; \pi_1; x; e_2; \psi; e_1\}$

(c) $\psi[w'] = w_2$

(d) $\text{slice}_p(C_1) \sim p\{\sigma'; \kappa'; \psi'; \text{let } x \overset{p(w')}{=} e_1 \text{ in } e_2\}$, where $\text{slice}_p(\sigma) \sim \sigma'$, $\text{slice}_p(\kappa) \sim \kappa'$, and $\text{slice}_p(\psi) \sim \psi'$ when $p \in w_1 \cup w_2$

(e) $\text{slice}_p(C_1) \rightarrow \text{slice}_p(m(w)\{\sigma; \kappa'; \psi; \pi_1; x \overset{m(w_1 \cup w_2)}{=} \circ; e\}'\})$ when $\kappa = \kappa' \vdash (m(w); \psi'; x; e')$ or $\text{slice}_p(C_1) \rightarrow \text{slice}_p(p(w_1 \cup w_2)\{\sigma; \kappa'; \psi; \pi_1; x \overset{\circ}{=} \circ; e\}'\})$ when $\kappa = \kappa' \vdash (\psi'; x; e')$

Consider $p \in w_1 \cup w_2$. By Lemma A.21,

(f) $\psi[w'] = w_2$

Case 1. $p \in w_2$

Then it can take step using STPP-PRESENT to $p\{\sigma'; \kappa'; \psi'; \pi_1; x; e_2; \psi'; e_1\}'\}$ which is slice of $C_2$.

Case 2. $\{p\}$ not in $w_2$

Then it takes step using STPP-ABSENT to $p\{\sigma'; \kappa'; \psi'; e_2\}'\}$ which is slice of $C_2$ using SLICECFG-ABS1.

Consider $\{p\}$ not in $w_1 \cup w_2$. These parties do not take a step, and their slice remains same via SLICECFG-ABS1.

STPC-Delssec. In the protocol only secure agent takes a step per rule STPP-SECESTEP and STPC-DELSSEC. All other parties remain as is.

STPC-SECenter. We have,

(a) $C_1 = p(w)\{\sigma; \kappa; \psi; \text{secure}_{w'}(e)\}$

(b) $C_2 = s(w)\{\sigma; \kappa; \psi; e\}$

(c) $\psi[w'] = w$

(d) $\text{slice}_p(C_1) \rightarrow p\{\sigma'; \kappa'; \psi'; \text{secure}_{w'}(e)\}$ when $p \in w$

(e) For $\{p\}$ not in $w$ slice is by SLICECFG-ABS1 or SLICECFG-ABS2.

For parties in $w$, we first note that $\kappa = \kappa_1 \vdash \{p(w); \psi'; x; e'\}'\}$ (rule STOK-SECE).

Their slice in $C_2$ is $p\{\sigma'; \kappa_1; \psi; \text{wait}\}'\}$, where $\text{slice}_p(\kappa) \sim \kappa_1$.

The execution goes as: STPP-BEGIN, followed by STPP-SECECENTER for each $p \in w$.

STPC-POPSTK1. We have,

(a) $C_1 = N\{\sigma; \kappa; \vdash (M; \psi; x; e); \psi; v\}$

(b) $C_2 = M\{\sigma; \kappa; \psi; \pi_1; x \overset{m(w)}{=} \circ; e\}'\}$

(c) $M = m(\_)$

(d) $N = \_\{w\}$

If $p \in w$, slicing in $C_2$ follows easily (since parties in $N$ must be there in $M$, parties remain same or grow up the stack).

If $\{p\}$ not in $w$.

Depending on whether $\{p\}$ in $M$ or not, we can prove the slicing relation on $C_2$ (if $\{p\}$ in $M$ but not in $N$, it cannot be the case that either $M$ or $N$ is secure).

STPC-LET, STPC-LOCAL, STPC-Delpssec. Similar to STPC-Delpar. In protocol, parties in $w$ take same step, while others do not.

Theorem A.4: (Confluence)

Suppose that $\pi_1 \rightarrow \pi_2$ and $\pi_1 \rightarrow \pi_3$, then there exists $\pi_4$ such that $\pi_2 \rightarrow \pi_4$ and $\pi_3 \rightarrow \pi_4$.

Proof:

Proof sketch: From Lemma A.22, if same agent (a party or secure agent) takes step in $\pi_1 \rightarrow \pi_2$ and $\pi_1 \rightarrow \pi_3$, then $\pi_2 = \pi_3$.

If different agents take step, then they can take corresponding steps in $\pi_2$ and $\pi_3$ to reach $\pi_4$.

A complete formal proof can be derived using case analysis on $\pi_1 \rightarrow \pi_2$ and $\pi_1 \rightarrow \pi_3$. 

■