λ-Symphony: A Concise Language Model for MPC

DAVID DARAIS, University of Vermont
DAVID HEATH, Georgia Institute of Technology
RYAN ESTES, University of Vermont
WILLIAM HARRIS, Galois Inc.
MICHAEL HICKS*, University of Maryland

Secure multi-party computation (MPC) allows mutually distrusting parties to cooperatively compute, using a cryptographic protocol, a function over their private data. This paper presents λ-Symphony, an expressive yet concise domain-specific language for expressing MPCs. λ-Symphony starts with the standard, simply-typed λ calculus extended with integers, sums, products, recursive types, and references. It layers on two additional elements: (1) A datatype representing *secret shares*, which are the basis of multiparty computation, and (2) a simple mechanism called *par blocks* to express which scopes are to be executed by which parties, and which determine to whom various data values are visible at any given time. The meaning of a λ-Symphony program can be expressed using a *single-threaded semantics* even though in actuality the program will be run by multiple communicating parties in parallel. Our *distributed semantics* is expressed naturally as piecewise, single-threaded steps of *slices* of the program among its parties. We prove that these two semantic interpretations of a program coincide, and we prove a standard type soundness result. To demonstrate λ-Symphony's expressiveness, we have built an interpreter for it (with some extensions) and used it to implement a number of realistic MPC applications, including sorting, statistical and numeric computations, and private information retrieval.

Additional Key Words and Phrases: Secure Multiparty Computation, Distributed Semantics, Type Systems

1 INTRODUCTION

Secure Multiparty Computation (MPC) is a subfield of cryptography that allows mutually untrusting parties to compute arbitrary functions over their private inputs while revealing nothing except the function output. That is, MPC allows parties to work together to run programs *under encryption*.

Approaches to MPC have improved significantly over the years. The first full implementation, FairPlay [Malkhi et al. 2004], could evaluate only a few hundred Boolean gates per second. Modern implementations on custom setups evaluate *billions* of Boolean gates per second [Araki et al. 2016], and cryptographers continue to reduce the cost of MPC. Despite increasing efficiency and compelling applications (secure auctions, secure databases, collaborative machine learning, and any other imaginable application with security concerns), MPC has not been widely adopted. Perhaps the most significant barrier to adoption is a lack of appropriate infrastructure. Today, it is difficult for non-experts to understand and work in the complex distributed model that MPC requires.

High quality languages tailored specifically to MPC are required such that programmers can easily develop, debug, and streamline their secure applications. *Tailored* languages are required because, although MPC supports arbitrary computation, it fundamentally differs from cleartext computation in two key ways. First, MPC is a fundamentally **distributed**. Second, MPC is **more expensive** than cleartext computation. This cost manifests in two ways. First, basic operations like multiplication are more expensive to compute in MPC w.r.t. cleartext computation by orders of magnitude, in addition to requiring synchronized communication between parties. Second, and more importantly, concretely efficient protocols represent computed functions as *circuits*, which

*Work done in part as a consultant for Stealth Software, Inc.

Authors' addresses: David Darais, University of Vermont, david.darais@uvm.edu; David Heath, Georgia Institute of Technology, heath.davidanthony@gmail.comza; Ryan Estes, University of Vermont, restes@uvm.edu; William Harris, Galois Inc., wharris@galois.com; Michael Hicks, University of Maryland, mwh@cs.umd.edu.

increases cost because nontrivial control flow and random memory accesses do not have efficient circuit representations.

These key differences make a developer's workflow more difficult. In particular, the programmer must consider the following three questions:

- **Who knows what?** MPC is used to keep values private. In this context, parties typically should not learn secret values belonging to any other party, yet they are allowed to see encrypted versions of these secrets. Sometimes sharing secret values is explicitly allowed, e.g., between two mutually trusting parties. To help deal with the inherent distributed nature of MPC, a suitable language should help the programmer both to understand which parties can access which values, and how to share values with other parties.
- **Who computes what?** Often, parties have different resources and trust relationships. Some parties might have powerful hardware and fast networks, while other parties are limited. To increase efficiency it may be desirable to, for example, entrust a small yet powerful group of parties to run computations on behalf of a larger group. To help deal with the expense of MPC, a suitable language should make it easy to control which parties compute what.
- **How do the parties compute?** Given that MPC is expensive, it is helpful to "mix" MPC with cleartext computation. Where possible, the parties should avoid MPC, so the programmer should be empowered to mix in cleartext computation. MPC computations should be clearly distinguished such that experienced developers can estimate the cost of programs. The benefits of mixing in cleartext computation are significant enough to warrant manual control.

In this work, we introduce λ-Symphony, a domain-specific language for MPC. λ-Symphony is a concise extension to the lambda calculus that helps answer the three questions listed above.

Who knows what? λ-Symphony allows the programmer to move values between parties via simple communication primitives. To help the programmer reason about which parties hold which values, λ-Symphony formalizes value *location*. That is, values are *located* at certain parties. This notion is both formalized in the language semantics and is explicit in λ-Symphony's type system. Locatedness allows the programmer to reason about which parties know which values.

Who computes what? λ-Symphony's central term-level construction is the `par` expression, a mnemonic for "parallel". `par` "blocks" allow the programmer to control which parties compute which parts of the program. The expression `par` $[p]$ e instructs the parties in the set p to compute the expression e *in parallel* each on their respective host. Parties who are not in p just skip the subcomputation: they simply proceed past the `par` block without waiting

How do the parties compute? At its core, λ-Symphony is a lambda calculus extended with `par` blocks, located values, and located types. To this, we add MPC primitives that 'build up a circuit' which can be dispatched using high efficiency MPC backends. The programmer can refer to wires in circuits via *shares*, the central concept in λ-Symphony's model of MPC. These shares are encrypted values held between various parties. The parties work together to construct, compute over, and reveal shares and hence perform MPC. λ-Symphony ensures that shares are first class. This status allows the programmer to freely mix encrypted and cleartext computation. For example, λ-Symphony can encode complex MPC circuits as recursive programs, a technique that is difficult to emulate in existing languages (see discussion in Section 8). First class shares also drive much of the simplicity of λ-Symphony's formalism.

Manipulating which parties are in scope, which values they know, and how they compute is complex. To ensure that the developer can control this complexity, λ-Symphony provides a powerful model that allows for effective reasoning: λ-Symphony supports a **single-threaded interpretation** of distributed programs. That is, the developer can understand her program as if it

runs on a single thread of execution on a single machine, and λ-Symphony guarantees that this is a faithful understanding of the program's semantics when executed in a distributed setting.

The idea of a single-threaded view for MPC was established in Wysteria [Rastogi et al. 2014]. Compared to Wysteria, λ-Symphony provides a far simpler formalism and addresses several key practical design problems. One specific improvement is with respect to shares: Wysteria provides shares only as a second class constructs. Shares can only be manipulated in special contexts where other language features, like recursion, are disallowed. This restricted usage essentially means that Wysteria is two languages glued together, and the programmer must work hard to weave programs written in both languages. Additionally, Wysteria offers less flexibility about which parties provide inputs and which compute on them. In constrast, λ-Symphony's 'share-centric' stance means that weaving MPC with cleartext computations is simple, even when moving between different parties' hosts, and the formalism also benefits in concision. We compare λ-Symphony and Wysteria, and other related work, further in Section 8

In summary, this paper presents λ-Symphony, a concise extension of the lambda calculus for programming MPCs. The paper's specific contributions are the following:

- We present λ-Symphony informally with examples (Section 2).
- We present λ-Symphony in formal detail, including its syntax (Section 3), small-step *single-threaded* semantics (Section 4), and typing (Section 5). We prove type soundness.
- We introduce a small-step *distributed* semantics (Section 6). Whereas the single-threaded semantics represents the language's *single-threaded interpretation*, the distributed semantics models how distributed parties actually execute. We prove forward simulation and confluence, showing that terminating MPC programs agree on outcomes in both semantics.
- We describe our prototype interpreter for λ-Symphony and discuss how we have used it to implement several challenging benchmarks (Section 7). We have written more than two thousand lines of λ-Symphony, including code for standard data structures, private numeric computations, private information retrieval scenarios, and oblivious sorting.

2 BACKGROUND AND OVERVIEW

This section begins by reviewing the basics of MPC. We then present λ-Symphony informally and explain the advantage of our design over competing approaches.

2.1 Secure Multiparty Computation (MPC) and the GMW Protocol

MPC is a set of techniques for operating over data that is 'shared' between parties: each party holds some *share* of the program values. These shares have the following key properties:

- Shares can be combined to reveal the cleartext value.
- One party's share cannot reveal a cleartext value, so parties learn only jointly revealed values.
- The parties can work together to compute over shares. Many MPC protocols allow parties to compute XOR and AND over Boolean shares. By composing many XOR and AND operations, often referred to as *gates* to allude to the circuit-like structure of such computations, we compute arbitrary functions. We assume that more expressive functions, like integer addition and comparison, are built from XOR and AND gates and are available as black box functions.

Many MPC protocols exist, differing in security, efficiency, and flexibility. λ-Symphony is relatively agnostic to the choice of protocol: we equip λ-Symphony with black-box functions that delegate to a cryptographic backend. Despite this, it is convenient to consider a specific protocol. We choose the classic GMW protocol, because it is typical, simple, and flexible [Goldreich et al. 1987].

GMW allows any number of parties to securely compute an arbitrary Boolean function. GMW shares can be easily XORed by local computation on behalf of each party, but AND requires

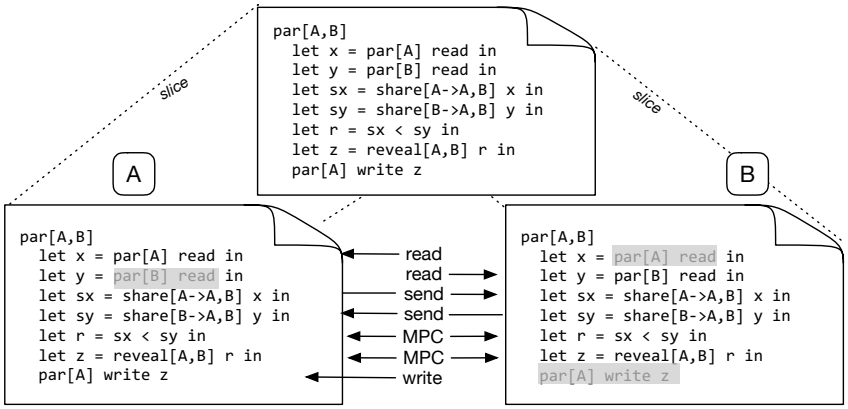

Fig. 1. Millionaires in λ-Symphony: Single-threaded and Distributed Views

communication. Under typical security parameters, each pair of parties exchanges ≈ 32 bytes per AND gate. Furthermore, to evaluate an AND gate whose input wire depends on the output of another AND gate, the parties must first evaluate the dependency. Thus, GMW incurs multiple *rounds* of communication, proportional to the length of the longest path of AND gates in the circuit's dependency graph. On slow networks, GMW is expensive.

In sum, it is usually unrealistic to push an entire application into an MPC protocol. Instead, the programmer must design her application with protocol costs in mind and avoid MPC overhead as much as possible. She is encouraged to reduce the number of parties in scope, both to reduce the amount of communication and to move the computation onto a powerful sub-network. This means that the programmer must control the answers to the three core questions λ-Symphony is designed to help answer: **who knows what**, **who computes what**, and **how the parties compute**.

2.2 λ-Symphony

λ-Symphony's presents a concise language model for MPC that affords a single-threaded interpretation. In this section, we demonstrate λ-Symphony's design by example.

Example: Millionaires in λ-Symphony. We begin with an encoding of the the classic Yao's Millionaires' problem [Yao 1982]. Here, two parties Alice (A) and Bob (B) wish to know who is richer but without revealing to the other (or a third party) his/her net worth. The top box in Figure 1 lists the λ-Symphony code for this problem.

The program is set inside a `par` block on line 1 that indicates the computation takes place between players A and B. On the next two lines, the two parties individually, as dictated by nested `par` blocks, read in their net worth as input. On line 2, since B is not named by the `par` block, he simply skips the `read` command and continues to the next instruction (and symmetrically A skips the `read` on line 3). The parties use the primitive `share` on lines 4 and 5 to distribute GMW secret shares of their respective input values. The parties compute $sx < sy$ on line 6 ($<$ is computed by the GMW backend) and reveal the cleartext output on line 7. Finally B writes out the answer.

We note several interesting aspects of this program. First, the programmer easily controls which parties perform which actions using `par` blocks. Second, note the mix of cryptographic and non-cryptographic operations: lines 2, 3, and 8 all perform cleartext `read`/`write` operations while lines 4 through 7 perform cryptographic operations.

A program	*B* program

`let x = read in`
`let sx = make-share x in`
`send-share B sx;`
`let sy = recv-share B in`
`let r = sx < sy in`
`let z = reveal r in`
`()`
	`let y = read in`
	`let sy = make-share y in`
	`let sx = recv-share A in`
	`send-share A sy;`
	`let r = sx < sy in`
	`let z = reveal r in`
	`write z`

Fig. 2. Millionaires in a typical MPC framework, as two programs, one each for *A* and *B*

Finally, we note λ-Symphony's single-threaded interpretation. In our explanation of the program, we inspected the code from top to bottom. In reality, this program is intended to execute on two different hosts that communicate, so it is not a priori clear that a top-to-bottom reading is sensible. Indeed, in some sense the code contains two programs, each to be executed on a different host. Figure 1 depicts the two program *slices* that separately emerge for party *A* and party *B*. Parties simply skip `par` blocks that do not explicitly mention them: Figure 1 grays out skipped computations. Thus, the parties execute different code. Despite this, λ-Symphony's formal properties ensure that the top-to-bottom single-threaded interpretation of the program is sensible.

Alternative approach: Multiple (parallel) programs. A typical alternative to language-based MPC (Section 8 says more) requires a different program for each participating party. For example, if we use the EMP toolkit [Wang et al. 2016] to write Millionaires', then the programs for *A* and *B* would be as shown in Figure 2 (using higher-level syntax, rather EMP's native C++).

The figure illustrates two key downsides to separating programs. The first is that the flow of control is hard to discern—there is no clear, single-threaded series of steps as there is in the λ-Symphony program. Instead, we must consider the programs together and assemble a mental model of the interaction. In this case, the programs are small enough to understand. For larger programs with many parties communicating in different sub-groups, doing so is much more difficult.

The second downside is that there is more opportunity for mistakes, whose likelihood is greater because of a program's reduced understandability. In particular, because making, sending, and receiving a share constitute three distinct operations, the programmer may now introduce race conditions and deadlocks. For example, if we swapped lines 3 and 4 of *A*'s program then it would be waiting for *B*'s share before sending its own, but *B* would be trying to do the same thing. Or, perhaps *A* will think it is sharing with *B* and another party *C* while those two might think they are only sharing with each other. The single-program structure offered by λ-Symphony assures this cannot happen, even when loops and other control constructs are involved.

Advanced Millionaires', Delegating Computation. The example thus far is limited in its use of distributed features. In particular, we only include two parties, and those parties are included in all parts of the computation, save reads and writes. Figure 3 presents a modified Millionaires' example that demonstrates λ-Symphony's flexibility in managing who knows what, who computes what, and how they compute. This program still has two input parties, *A* and *B*, but now also involves three more parties: *C* and *D* are *compute parties*, and *E* is the *output party*. Compute parties are particularly useful when computation is very expensive (e.g., suppose encrypted < operation is very slow), or when many input parties are involved (which would require excessive *N*-way communication). Output parties are useful when even the final result is sensitive.

```
par[A, B, C, D, E]
  let x = par[A] read in
  let y = par[B] read in
  let sx = par[A, C, D] share[A → C, D] x in
  let sy = par[B, C, D] share[B → C, D] y in
  let r = par[C, D] sx < sy in
  let z = par[C, D, E] reveal[E] r in
  par[E] write z
```

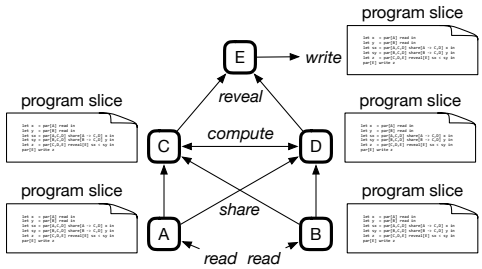

Fig. 3. λ-Symphony Advanced Millionaires': code (left) and distributed view (right)

As before, lines 2 and 3 read input from A and B, as designated by par blocks; variables x and y are visible (as cleartext) only at A and B, respectively, having types $\texttt{int@}\{A\}$ and $\texttt{int@}\{B\}$. Lines 4–5 convert A's and B's inputs into secret shares, which are then sent to parties C and D. Variables sx and sy have type $\texttt{int}^{\texttt{enc\#}\{C,D\}}@\{C, D\}$, indicating that they are secret shares split between C and D (the $\texttt{enc\#}\{C, D\}$ part) and also accessible at those hosts. Line 6 performs the secure comparison, and the $\texttt{reveal}[E]$ construct on line 7 converts the resulting share to cleartext and reveals it to E; the variable z has type $\texttt{int@}\{E\}$.

GCD in λ-Symphony. As a last example, consider a λ-Symphony version of Euclid's greatest common divisor (GCD) algorithm, shown in Figure 4. Each party provides one of the inputs, read in on lines 8 and 9. Then, both parties convert their inputs to shares and call *gcd* with a bound n, which determines the number of recursive calls that *gcd* will make. Since n does not depend on a or b, the iteration count reveals nothing about them. The check on n occurs in a normal (short circuiting) conditional, since n is not a share, but the check whether a is non-zero on line 7 occurs in a multiplexer, in which both branches are always executed—here, even if a was 0, and thus we should return b, the code will still compute b' via recursive call (line 6). Type annotations are not strictly needed, and are provided for clarity. Notice the ease with which this recursive *gcd* program is written. This simplicity is owed to λ-Symphony's first class treatment of shares.

In sum, λ-Symphony packages a powerful interface to MPC in a concise language model. λ-Symphony emphasizes control of the distributed nature of computation and a mixing of cleartext and encrypted computation. This control is largely achieved by arbitrary nesting of par blocks and by first class MPC shares. To aid the programmer, λ-Symphony provides expressive types

```
let gcd = λ_gcd n : int. λa : int^enc#{A,B}. λb : int^enc#{A,B}.
  if n < 1 then − 1
  else
    let r = b % a in
    let b′ = gcd (n − 1) r a in
    a ? b′ ⋄ b in
let x = par[A] read in
let y = par[B] read in
let z = par[A, B] reveal[A, B] gcd 93 (share[A → A, B] x) (share[B → A, B] y)
```

Fig. 4. GCD in λ-Symphony

that include locatedness information and a powerful single-threaded intepretation. Together these features make it possible to naturally write a large class of useful programs.

3 FORMAL λ-SYMPHONY: SYNTAX

The next four sections present the formalization of λ-Symphony. This section contains its syntax; the subsequent ones outline its single-threaded semantics (Section 4), type system (Section 5), and distributed semantics (Section 6). The main theoretical results of type soundness and simulation are presented in sections 5 and 6.

3.1 Types

The syntax of λ-Symphony is in Fig. 5. λ-Symphony types comprise a series of standard types—integers, sums, functions, mutable references, pairs, and recursive types—augmented with three additional elements.

First: types are of two varieties, notated σ and τ. The former are *located types*, which classify integers, sums, functions, and references. Values of σ type are *located* in the sense that they can be manipulated only at particular hosts. The metavariable m represents a set of parties, and a located type is written $\sigma@m$. Types τ include located types and also pair types and recursive types. Pairs and recursive values are *not* located, in the sense that all parties can see their structure. All parties may *access* the contents of these types, however, they may not be able to compute with the results, which may be available only at their designated locations; we formalize these distinctions and discuss them in more detail in Section 4.1.

Second: integers are annotated with a *protocol* ψ that indicates whether they are either cleartext $(\cdot)$ or are *encrypted* and shared among parties p. For concreteness, we imagine the encryption protocol is Goldreich, Widgerson, and Micali (GMW) 1987, and values of this type are *secret shares*, but the formal language is indifferent to the cryptographic details. Note that we often just write `int` for cleartext integer types, to reduce clutter (i.e., eliding the $\cdot$).

Last, we annotate a reference type $\texttt{ref}^{\omega}\ \tau$ with a *ref-mode* ω. If ω is `RW#`p, then the reference is writeable by parties p; if ω is `RO` then it is read-only.

3.2 Expressions

To simplify the formal semantics, we simplify λ-Symphony's expression language to be in a kind of *administrative normal form* (ANF): Most expression forms operate directly on variables x, rather than subexpressions e. Thus, while we might be accustomed to writing program syntax like `ref (1 + read)`, λ-Symphony's formal syntax would require something like

`let` x `=` `read` `in` `let` y `=` `1` `in` `let` z `=` $y + x$ `in` `ref` z

The expressiveness of the language is not impacted by this restriction, and a direct-style syntax (with syntactically-recursive subexpressions) is easily transformed into our language without changing its meaning.

Standard expression constructs. Most of λ-Symphony's expression forms are standard. We isolate *atomic expressions* a as a sub-category of full expressions e. Atomic expressions evaluate to a final result in one "small" step, which simplifies the presentation of the semantics. They include variables x; integers i; binary integer operations $x \odot y$ (left abstract, but could be, e.g., addition and multiplication); ternary conditional $x\ ?\ y \diamond z$ (an "if" expression which evaluates both branches and returns one result); sum injection $\iota_i\ x$; pair creation $\langle x, y\rangle$ and projection $\pi_i\ x$; (recursive) function creation $\lambda_z x.\ e$; reference creation `ref` x dereference $!x$ and assignment $x := y$; and recursive type introduction `fold` x and elimination `unfold` x (in support of iso-recursive types). Sum elimination `case` $x\ \{y.\ e_1\}\ \{y.\ e_2\}$ is a full expression e because it does not reduce to a value in one step; here,

$i \in$	$\mathbb{Z}$			*integers*
$A, B, C \in$	party			*parties*
$m, p, q \in$	party-set	$\triangleq$	$\wp(\text{party}) ::= \{A, \ldots, A\}$	*sets of parties*
$\alpha \in$	tvar			*type variables*
$\psi \in$	prot	$::=$	$\cdot$	*cleartext*
		$\mid$	$\texttt{enc\#}m$	*encrypted*
$\omega \in$	ref-mode	$::=$	$\texttt{RW\#}m$	*read/write*
		$\mid$	$\texttt{RO}$	*read only*
$\sigma \in$	loc-type	$::=$	$\texttt{int}^{\psi}$	*integer type*
		$\mid$	$\tau + \tau$	*sums type*
		$\mid$	$\tau \xrightarrow{m} \tau$	*function type*
		$\mid$	$\texttt{ref}^{\omega}\ \tau$	*reference type*
$\tau \in$	type	$::=$	$\sigma@m$	*located type*
		$\mid$	α	*type variable reference*
		$\mid$	$\tau \times \tau$	*pair type*
		$\mid$	$\mu\alpha.\ \tau$	*recursive type*
$x, y, z \in$	var			*variables*
$\odot \in$	binop			*binary operations (e.g., plus, times)*
$a \in$	atomic	$::=$	x	*variable reference*
		$\mid$	i	*integer literal*
		$\mid$	$x \odot x$	*binary operation*
		$\mid$	$x\ ?\ x \diamond x$	*atomic conditional*
		$\mid$	$\iota_i\ x$	*sum injection ($i \in \{1, 2\}$)*
		$\mid$	$\langle x, x \rangle$	*pair creation*
		$\mid$	$\pi_i\ x$	*pair projection ($i \in \{1, 2\}$)*
		$\mid$	$\lambda_z x.\ e$	*(recursive) function creation*
		$\mid$	$\texttt{ref}\ x$	*reference creation*
		$\mid$	$!x$	*dereference*
		$\mid$	$x := x$	*reference assignment*
		$\mid$	$\texttt{fold}\ x$	*recursive type introduction*
		$\mid$	$\texttt{unfold}\ x$	*recursive elimination*
		$\mid$	$\texttt{read}$	*read int input*
		$\mid$	$\texttt{write}\ x$	*write output*
		$\mid$	$\texttt{embed}[p]\ x$	*encrypted a known constant*
		$\mid$	$\texttt{share}[p \rightarrow p]\ x$	*share encrypted value*
		$\mid$	$\texttt{reveal}[p]\ x$	*reveal encrypted value*
$e \in$	expr	$::=$	a	*atomic expression*
		$\mid$	$\texttt{case}\ x\ \{x.\ e\}\ \{x.\ e\}$	*tagged union elimination*
		$\mid$	$x\ x$	*function elimination*
		$\mid$	$\texttt{par}[p]\ e$	*parallel execution*
		$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e$	*let binding*

Fig. 5. λ-Symphony Syntax

y binds in the branch bodies e_i, only one of which is evaluated.[1] Function application $x\ y$ and local variable binding $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ describe full expressions for the same reason. The $\texttt{par}[p]\ e$ expression is special and specific to our distributed programming setting, which we describe next.

[1]We can encode $\texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ as $\texttt{let}\ z = 0\ \texttt{in}\ \texttt{let}\ y = x\ ?\ \iota_1\ z \diamond \iota_2\ z\ \texttt{in}\ \texttt{case}\ y\ \{\_.\ e_1\}\ \{\_.\ e_2\}$.

Distributed computing constructs. The λ-Symphony expression `par`$[p]$ e says that e may be computed at parties p in parallel (hence the syntax `par`). That is, every party $A \in p$ may evaluate e. We say "may" here because nesting such an expression in another `par` could shrink the set of parties. For example, e in `par`$[p]$ (`par`$[q]$ e) will be evaluated by $p \cap q$; if this intersection is $\varnothing$ then e is essentially dead code.

We call the set of parties m computing an expression in parallel the *mode*. We say the parties $A \in m$ are *present* for a computation. The semantics of many constructs depends on the mode. A number i created in mode m (having type `int`$@m$) is known only to parties $A \in m$. This means that adding two numbers located at p can only be done in a mode m such that $m \subseteq p$; if the mode contained additional parties $A \notin p$ then they wouldn't know what to do; such states will be stuck in our semantics. The same goes for functions, sums, and references. The `read` and `write` x expressions perform local I/O and so can only be run in a mode with a single party.

On the other hand, it is possible that a variable x is in scope for A, but maps to a value only usable by B. Party A can still manipulate a placeholder for x (e.g., to store it in a datastructure or pass it to a function) but may never compute with its contents (e.g., add to it or branch on it). This approach simplifies the design of the language at the cost of missing some (unlikely, but ultimately harmless) logic errors.

Generally speaking, λ-Symphony's design aims to ensure that any expression e of type $\sigma@m$ will have the *same run-time value* at each $A \in m$. This is a key invariant underpinning λ-Symphony's single threaded interpretation.

MPC-specific constructs. The λ-Symphony expression `share`$[p \to q]$ x directs p (required to be a singleton party set) to create secret shares of x, an integer, and distribute the shares to each party in q. All parties $p \cup q$ must be present. The resulting value has type $\texttt{int}^{\texttt{enc\#}q}@q$. This type reads "an integer, encrypted (i.e., secret shared) between parties q, and accessible to parties q". The duplication of q may seem redundant, but they may differ in other contexts. The first q represents *who has the shares* (determined when the share is created), and the second q represents *who has access to this value* (determined by the enclosing `par` blocks). If this encrypted value flows to a part of the program only executed by $q' \subset q$, then it will not be possible to recombine shares, since not all parties q will be present. We can also make a share of a constant usable by parties p via `embed`$[p]$ x; making shares of constants does not require the sharing parties to communicate.

Parties q can all mutually compute on a shared, encrypted value using $x \odot y$ and $x \ ?\ y \diamond z$. Recall from above that the latter is a multiplexor: we select between y and z based on whether x is zero or non-zero. The former models binary operations over numeric types. When operating on encrypted values, both the multiplexor and binary operation expressions will necessitate *communication* in the distributed semantics, and an actual implementation will use an underling MPC protocol to implement the computation over the encrypted value. All parties to which a share was sent must all be present when computing on it. E.g., for numbers of type $\texttt{int}^{\texttt{enc\#}A,B}$ to be added, both A and B must be present. Indeed, it must be *exactly* these parties which are present; we do not allow more, since other parties would not be able to carry out the operation (they don't have access to the share).

An MPC is completed by invoking `reveal`$[q]$ x. This takes a share (among some set of parties p) and converts it to cleartext, sharing the result among parties q. Doing so requires that all of $p \cup q$ are present so that the shareholders can agree to send the value, and the result-receivers are ready to receive it.

$$
\begin{array}{lll}
\ell \in \text{loc} & & \textit{memory locations (i.e., pointers)} \\
\gamma \in \text{env} & \triangleq \text{var} \rightharpoonup \text{value} & \textit{value environment} \\
\delta \in \text{store} & \triangleq \text{loc} \rightharpoonup \text{value} & \textit{value store} \\
u \in \text{loc-value} & ::= i^{\psi} & \textit{integer/share value} \\
& \mid \iota_i\ v & \textit{tagged union injection} \\
& \mid \langle \lambda_z x.\ e, \gamma \rangle & \textit{closures} \\
& \mid \ell^{\omega} & \textit{located reference} \\
v \in \text{value} & ::= u@m & \textit{located value} \\
& \mid \texttt{fold}\ v & \textit{recursive value} \\
& \mid \langle v, v \rangle & \textit{pairs} \\
& \mid \bigstar & \textit{opaque value}
\end{array}
$$

$$\boxed{\_\swarrow_m \in \text{loc-value} \to \text{loc-value}\ ;\ \text{value} \to \text{value}}$$

$$
\begin{array}{rcl}
i^{\psi}\swarrow_m & \triangleq & i^{\psi} \\
(\iota_i\ v)\swarrow_m & \triangleq & \iota_i\ (v\swarrow_m) \\
\langle \lambda_z x.\ e, \gamma \rangle\swarrow_m & \triangleq & \langle \lambda_z x.\ e, \gamma \rangle \\
\ell^{\omega}\swarrow_m & \triangleq & \begin{cases} \ell^{\omega} & \textit{if}\quad \omega = \texttt{RW\#}m \\ \ell^{\texttt{RO}} & \textit{if}\quad \omega \neq \texttt{RW\#}m \end{cases}
\end{array}
\qquad
\begin{array}{rcl}
(u@p)\swarrow_m & \triangleq & \begin{cases} (u\swarrow_{p \cap m})@(p \cap m) & \textit{if}\quad p \cap m \neq \varnothing \\ \bigstar & \textit{if}\quad p \cap m = \varnothing \end{cases} \\
(\texttt{fold}\ v)\swarrow_m & \triangleq & \texttt{fold}\ (v\swarrow_m) \\
\langle v_1, v_2 \rangle\swarrow_m & \triangleq & \langle v_1\swarrow_m, v_2\swarrow_m \rangle \\
\bigstar\swarrow_m & \triangleq & \bigstar
\end{array}
$$

Fig. 6. λ-Symphony Semantics Metafunctions

4 SINGLE THREADED SEMANTICS

We define a small-step, *single-threaded operational semantics* for λ-Symphony. It is single-threaded in that we don't have independently executing parties; rather, we simulate the group of parties m executing in lockstep. Section 6 extends this small-step semantics to the distributed setting.

4.1 Located values

Evaluating a (terminating) λ-Symphony expression yields a *value*, of which their are two kinds. *Located* values u, defined in Figure 6, are only accessible to particular parties. The form $u@m$ indicates that u is accessible to parties $A \in m$. Values for numbers i^{ψ}, sums $\iota_i\ v$, closures $\langle \lambda_z x.\ e, \gamma \rangle$ (with an explicit environment γ), and references (pointers) ℓ^{ω} are located, and otherwise standard. When ψ on i^{ψ} is $enc\#p$ then i is a number that is secret-shared among parties in $B \in q$; otherwise (when $\psi = \cdot$) it is a cleartext number. Pointer values ℓ are annotated with ω to indicate either that they are read/write by parties p (ω is $\texttt{RW\#}p$) or that they are read-only (ω is $\texttt{RO}$). *Non-located* values $\langle v_1, v_2 \rangle$ for pairs and $\texttt{fold}\ v$ for recursive types are accessible to all parties (though their contents may not be). The *opaque value* $\bigstar$ stands in for any inaccessible value (examples below).

The same figure defines the function $\_\swarrow_m$, which is used to *relocate* a value to be accessible to (only) parties m. For located values $u@p$, function $\_\swarrow_m$ relocates them to $p \cap m$; if the intersection is $\varnothing$ then the value is inaccessible, so it becomes $\bigstar$. Relocating $u@p$ also may update the contents u. This step has no effect on numbers/secret shares i^{ψ} or closures (the latter's environment's variables get relocated when they are referenced, as we will see), but does recursively relocate the contents of a sum. It may also update the annotation on a reference value to be $\texttt{RO}$ (read-only), to ensure that

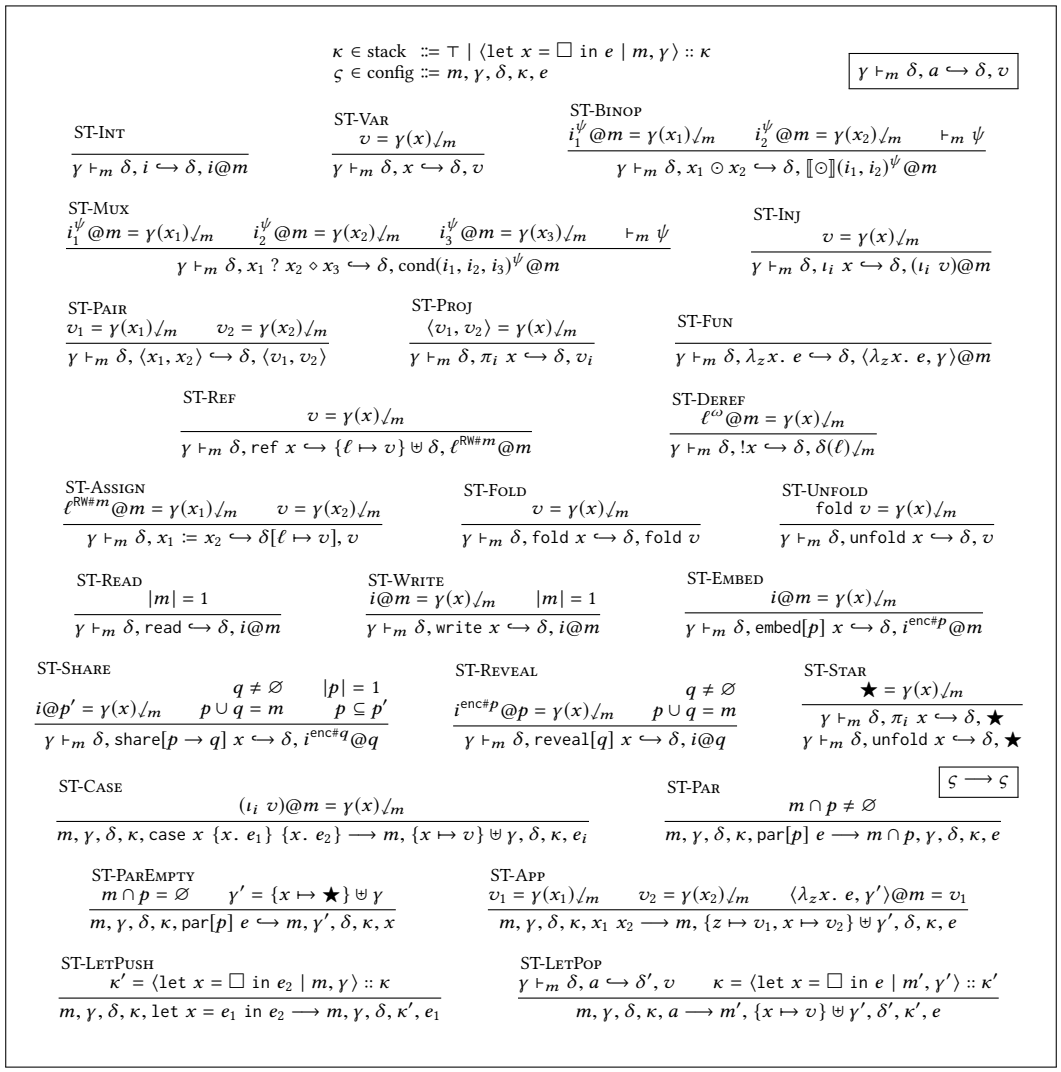

Fig. 7. λ-Symphony Semantics: Small-step, Global

all parties agree on the contents of the reference. As shown in an example in the next subsection, relocating a writeable reference to a subset of its current parties could allow those parties to get out of sync.

4.2 Operational rules

The small-step semantics is given in Figure 7. The main judgment, shown at the bottom, is expressed as rules that step between *configurations* ς, which are 5-tuples comprising the current mode m, environment γ, store δ, stack κ, and expression e. Per Figure 6, environments are partial maps from variables to values, and stores are partial maps from pointers to values. The main judgment refers to judgment $\gamma \vdash_m \delta, a \hookrightarrow \delta, v$, also shown in the figure. It defines the evaluation of atomic expressions a to values v (in a single step).

A stack κ is defined by the grammar at the top of Figure 7; it is either the empty stack $\top$ or a frame $\langle \texttt{let}\ x = \square\ \texttt{in}\ e \mid m, \gamma \rangle :: \kappa$, which is used to model evaluation inside let-bindings in a small-step setting. Rule ST-LetPush evaluates $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ by pushing a frame (which includes the skeleton of the `let`, the current mode m, and the environment γ) and then evaluating e_1. By rule ST-LetPop, when the current expression a is atomic, we single-step a to a final value v, and then pop the top frame and evaluate its contents e_2 in the frame's mode m and in its captured environment γ' extended with variable x mapped to v.

Variables, Binding, and Closures. Rule ST-Int evaluates cleartext constant i to value $i@m$; the location is based on the mode m. Rule ST-Var retrieves a variable's value from the environment and locates it to the current mode m via $\gamma(x)\swarrow_m$.

The rules satisfy the invariant that if $\gamma \vdash_m \delta, a \hookrightarrow \delta, v$ then v is compatible with m. By this we mean v is accessible to some parties $p \subseteq m$, which includes $\bigstar$ when $p = \varnothing$. More formally, this idea is captured by $v\swarrow_m = v$, that is to say, v is already located at m, and re-locating it there does nothing. (Note that $\_\swarrow_m$ is idempotent.) The invariant also applies to $\varsigma \longrightarrow \varsigma'$.

By narrowing the location when a variable is accessed, we don't need to explicitly (re)locate a closure's environment γ. This means that rule Rule ST-Fun constructs a (recursive) closure in the standard way. Rule ST-App requires the closure to be available at m, the current mode, and sets of evaluation of its body in the standard way.

The flip side of the invariant above is that to destruct a value located at m requires running in mode m, i.e., all parties must be present. This invariant helps ensure these parties, when running in a distributed setting with their own store, environment, etc. will agree on the result.

Par mode. The current mode m may change due to $\texttt{par}[p]\ e$. A `par` evaluates e in mode $m \cap p$; i.e., only those parties in p *also* present in m will run e. When $m \cap p$ is non-empty, rule ST-Par directs e to evaluate in the refined mode. If $m \cap p$ is empty, then per rule ST-ParEmpty, e is skipped and $\bigstar$ is returned. (Since $\bigstar$ is not an expression—it is a value—we return a fresh variable and the environment with that variable mapped to $\bigstar$.) Note that because the stack tracks each frame's mode, when the current expression completes the surrounding mode will be restored when the stack frame is popped (if there is one).

Here is an example of how par mode and variable access interact.

$$\begin{aligned}\texttt{par}[A,B]\ &\texttt{let}\ x = \texttt{par}[A]\ 1\ \texttt{in}\\ &\texttt{let}\ y = \texttt{par}[B]\ x\ \texttt{in}\\ &\texttt{let}\ z = \texttt{par}[C]\ 2\ \texttt{in}\ x\end{aligned}$$

The outer $\texttt{par}[A,B]$ evaluates its body in mode $\{A,B\}$, per rule ST-Par. Next, according to rules ST-LetPush, ST-Par, and ST-Int we evaluate 1 in mode $m = \{A,B\} \cap \{A\} = \{A\}$; we bind the result $1@\{A\}$ to x in γ per rule ST-LetPop. Next, according to rules ST-LetPush, ST-Par and ST-Var we evaluate x in mode $m = \{B\}$. Per the premise of ST-Var, we retrieve value $1@\{A\}$ for x, and then $\swarrow_{\{B\}}(1@\{A\})$ yields $\bigstar$ as the result, which is bound to y in γ per rule ST-LetPop. This result makes sense: Party B reads variable x whose contents are only accessible to A, so all it can do is return the opaque value. Finally, $\texttt{par}[C]\ 2$ evaluates to $\bigstar$ according to rule T-ParEmpty, since $m = \{A,B\} \cap \{C\} = \varnothing$. This $\bigstar$ result is bound to z per rule ST-LetPop, and the final result x, evaluated in mode $m = \{A,B\}$ is $\swarrow_{\{A,B\}}(1@\{A\}) = 1@\{A\}$ per rule ST-Var.

Sums, Pairs, Recursive Types. Sums, pairs, and recursive types are essentially standard, modulo the consideration of their values' locations. Rule ST-Inj introduces a sum, and rule ST-Case eliminates it, binding the contents to x in the appropriate branch $i \in \{1,2\}$, in the usual way. Rule ST-Pair introduces a pair, and rule ST-Proj eliminates it. Rule ST-Fold introduces a recursive (folded) value, and rule ST-Unfold eliminates it.

Rule ST-Star is somewhat surprising: It permits destructing ★ as if it were a pair or recursive value, yielding ★. Here is an example to show why this rule is needed.[2]

$$\begin{array}{l}\mathsf{par}[A,B]\ \mathtt{let}\ x = \mathsf{par}[A]\ \langle 1,2\rangle\ \mathtt{in}\\ \qquad\qquad \mathtt{let}\ y = \mathsf{par}[B]\ x\ \mathtt{in}\\ \qquad\qquad \mathtt{let}\ z = \mathsf{par}[C]\ \langle 3,4\rangle\ \mathtt{in}\ \pi_1\ z\end{array}$$

This example is very similar to the earlier one except that instead of binding an integer on the first and third lines, we bind a pair. Consider the $\mathsf{par}[B]\ x$ on the second line—rule ST-Var looks up x, which is bound to $\langle 1@\{A\}, 2@\{A\}\rangle$, and then locates it to $\{B\}$, yielding $\langle ★, ★\rangle$. Subsequent projection (not shown) from y would thus yield ★. On the third line, we evaluate $\mathsf{par}[C]\ \langle 3,4\rangle$ to ★ according to rule T-ParEmpty, and bind it to z. Finally, we evaluate $\pi_1\ z$ according to rule ST-Star. Without it, the program would be stuck. But stuckness seems hard to justify when projecting from y would work fine; y's contents are also inaccessible.

Rule ST-Star reflects part of what we mean when we say that pairs and recursive values are not located—even if their actual contents are missing, the code can nevertheless manipulate the ★ placeholder as if it were a pair or recursive value (producing another ★ value). On the other hand, elimination constructs for located values (e.g., binary operations, function application, dereference) will fail when applied to ★.

References. Rule ST-Ref creates a fresh reference in the usual way, returning a located pointer. ST-Deref takes a reference located in the current mode m and returns the pointed-to contents, also made compatible with m. This rule works for any ref-mode ω. On the other hand, ST-Assign only works for $\mathsf{RW}\#p$ references where $p = m$, the current mode. As usual, it updates the store with the new value (and returns it).

The following example illustrates why we need ref-modes ω.

$$\begin{array}{l}\mathsf{par}[A,B]\ \mathtt{let}\ x = \mathtt{ref}\ 0\ \mathtt{in}\\ \qquad\qquad \mathtt{let}\ \_ = (\mathsf{par}[A]\ x \coloneqq 1)\ \mathtt{in}\\ \qquad\qquad \mathtt{let}\ y = !x\ \mathtt{in}\ \ldots\end{array}$$

This program fails at the assignment. The variable x initially contains a reference $\ell^{\mathsf{RW}\#\{A,B\}}$, but then it is accessed by only A in the assignment on the subsequent line. By ST-Assign, the mode of the reference $\{A, B\}$ must match the current mode, but it does not, since that mode is $\{A\}$. To see the reason, imagine we ran this program at each of A and B separately in the style of the distributed semantics (per Figure 1), where each of A and B have their own store. On A we would do the assignment, but on B it would be skipped. As such, on A the value of y would be 1 but on B it would be 0. If the … part of the program were to branch on y and then in one branch do some MPC constructs but not in the other, then the two parties would fall out of sync.

I/O. Rules ST-Read and ST-Write handle I/O. They work when the mode is a singleton party—we need to know which party is reading from/writing and we locate the result at that party.

Multiparty computation. The remaining rules cover constructs for MPC, as well as local, integer-based computations. Rule ST-Binop handles binary integer operations. The protocol ψ for both integer arguments must be the same, and it must be compatible with the current mode m, per the judgment $m \vdash \psi$. This judgment holds when ψ is either $\cdot$ or $\mathsf{enc}\#m$; i.e., both integers are cleartext, or else are secrets shared among those in the current mode m. The result depends on the particular operation $\odot$, as determined by the semantic function $[\![\_]\!] \in \mathrm{binop} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ which we leave

[2]This example is not syntactically correct per the λ-Symphony grammar, which is restricted to ANF—we write $\langle 1, 2\rangle$ rather than $\mathtt{let}\ x = 1\ \mathtt{in}\ \mathtt{let}\ y = 2\ \mathtt{in}\ \langle x, y\rangle$ to avoid clutter

opaque. Rule ST-Mux is similar in structure to ST-Binop; it handles a multiplexor according to the semantic function cond, where $\text{cond}(i_1, i_2, i_3) \triangleq i_2$ if $i_1 \neq 0$, and $\text{cond}(i_1, i_2, i_3) \triangleq i_3$ if $i_1 = 0$.

Parties p can secret-share a private value using $\texttt{share}[p \to q]$ handled by rule ST-Share. The variable x must contain a cleartext integer located at (least) the sharing parties p, a singleton set. Also present must be those parties q to whom the shares are sent; only $p \cup q$ should be present, and no others. The resulting value is located at q, and has protocol $\texttt{enc\#}q$. Integers can be (locally) converted to shares by $\texttt{embed}[p]$, handled by rule ST-Embed. Note that if $p \not\subseteq m$ then the result $i^{\texttt{enc\#}p}@m$ will be ineligible for use later by rules such as ST-Binop which require p and m to be the same. On the other hand, not prescribing this connection between p and m will prove useful in the distributed semantics.

A secret share is revealed to parties p as cleartext using $\texttt{reveal}[p]$, handled by rule ST-Reveal. All parties q among which the value is shared must be present, as must p. See Section 2.2 (e.g., Figures 3 and 4) for examples involving these constructs.

5 TYPE SYSTEM

The typing judgment for λ-Symphony is $\Gamma \vdash_m e : \tau$; its rules are given in Figure 8. The judgment states that under context Γ, expression e has type τ when evaluating in mode m. We prove that typing is sound with respect to the single-threaded semantics.

5.1 Rules

Looking at the type rules as a whole, there are a few trends. The introduction and elimination rules for non-located types, i.e., T-Pair and T-Proj for pairs $\tau_1 \times \tau_2$ and T-Fold and T-Unfold for recursive values $\mu\alpha.\tau$, are standard. For located types σ, the introduction rules yield types $\sigma@m$, i.e., the location is the given evaluation mode; e.g., see T-Int, T-Ref, or T-Inj. The elimination rules require the given types σ to be *compatible* with mode m. Very often, this requires the types to be literally $\sigma@m$, e.g., see T-Case or T-App. But there is more to it: $\Gamma \vdash_m e : \tau$ implies *type compatibility* $\vdash_m \tau$.

Compatibility. Type compatibility is defined in Figure 9 as $M \vdash_m \tau$ where M is a map from type variables α to modes m; when referenced from the rules in Figure 8, M is empty. For located types $\sigma@m'$, judgment $M \vdash_m \sigma@m'$ holds, per rule WF-Loc, if m' is a subset of the parties m; i.e., those parties which might use σ later are currently present. Moreover, $M \vdash_{m'} \sigma$ must hold.

$M \vdash_m \sigma$ is essentially a homomorphism over σ except when considering function types; for these, the WF-Fun rule additionally requires the arrow annotation m matches the current mode; this is because the closure may reference variables accessible to parties in m. There are no constraints on other σ types because additional checks appear in the type and operational rules for elimination forms to ensure that values are well formed.

Returning to $M \vdash_m \tau$, rule WF-Prod applies to its components as expected, while rules WF-Var and WF-Rec work together to ensure that a recursive type's self reference is compatible in a mode with at least as many parties that it is, itself.

Basic Constructs, Subtyping. Returning to the type system in Figure 8 we consider some rules for the standard language constructs. Rule T-Var ensures that the variable x is given a type compatible with the current mode m: the rule applies subtyping $\tau <: \tau'$ on the looked-up type τ to yield a type τ' such that $\vdash_m \tau'$. This combination is the static analog to $\downarrow_m$ in the semantics rule ST-Var.

Subtyping is defined in the middle of Figure 9. The rules are structurally standard, but enforce the added invariant if $\tau <: \tau'$ then $M \vdash_m \tau$ implies $M \vdash_m \tau'$. As such, per rule Sub-Refl, types $\texttt{int}^{\psi}$ and $\texttt{ref}^{\text{RW\#}m}\ \tau$ only subtype to themselves; per rule Sub-Fun the mode m annotating the function arrow is invariant; and per rule Sub-Loc types σ located at m can be typed as if located at $m' \subseteq m$.

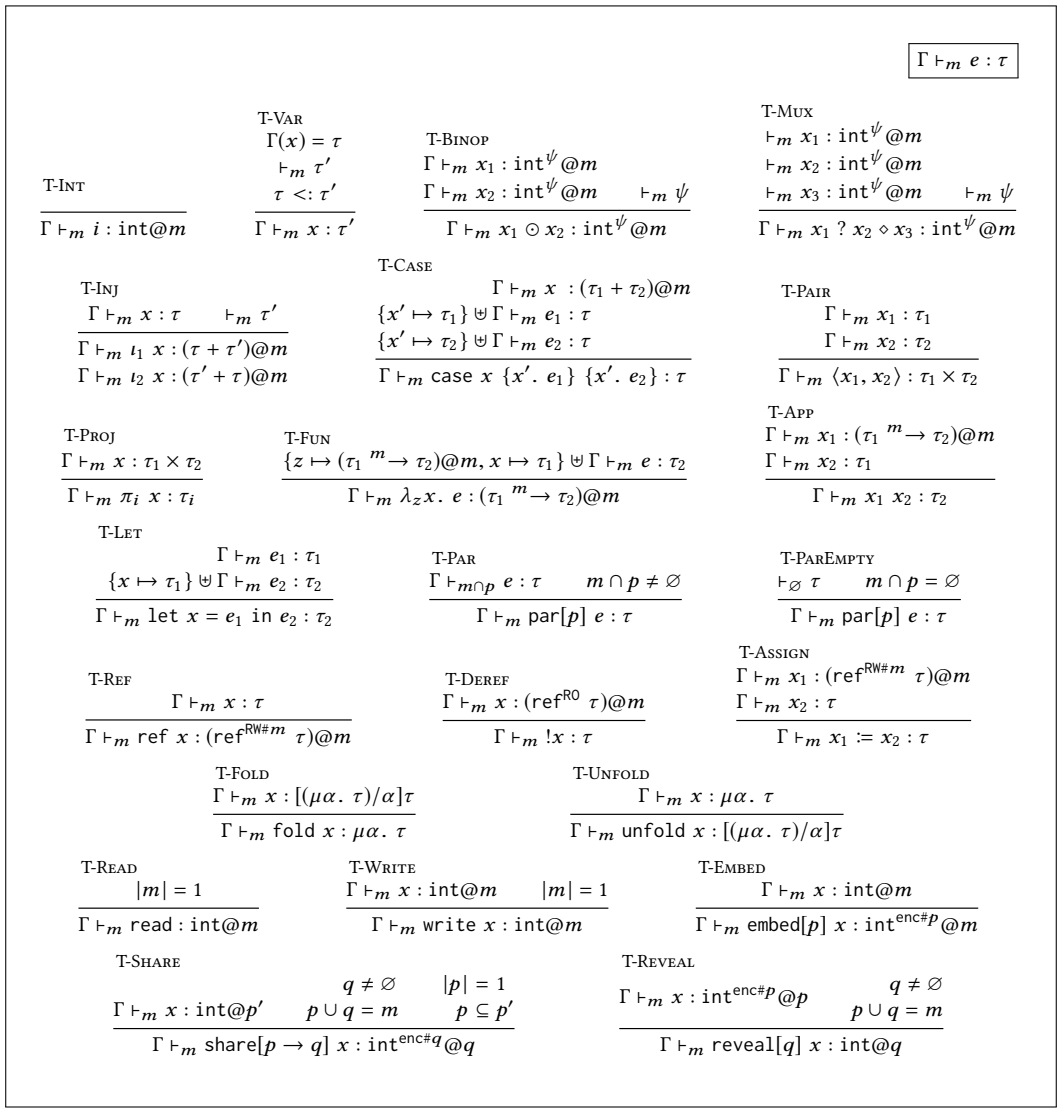

Fig. 8. λ-Symphony Type System

Aside for the elements that ensure the compatibility invariant discussed above, the rules for remaining standard language constructs (sums, functions, recursive values, etc.) are indeed standard.

Par blocks and MPC. Rule T-Par types the expression e in par mode involving parties p. At runtime, only $m \cap p$ parties will actually execute e, and so e is checked in this mode. Rule T-ParEmpty handles the case that $m \cap p = \varnothing$ which means the expression e is not actually executed. As such, it will return $\bigstar$, which we can type as having any type τ that is compatible with mode $\varnothing$.

Rule T-Share types secret sharing. The p argument indicates the party doing the sharing, necessarily a singleton. The (non-empty) q argument indicates the parties to which to share x. This value of x must be accessible to p (i.e., $p \subseteq p'$), and p and the parties q to which it is sharing must

$$M \in \text{mode-scope} \triangleq \text{tvar} \rightharpoonup \text{party-set}$$

$\boxed{\vdash_m \psi}$ $\boxed{M \vdash_m \sigma}$

$$\textsc{WF-Clear}\ \frac{}{\vdash_m \cdot} \qquad \textsc{WF-Enc}\ \frac{}{\vdash_m \mathsf{enc}\#m} \qquad \textsc{WF-Int}\ \frac{}{M \vdash_m \mathsf{int}^{\psi}}$$

$$\textsc{WF-Sum}\ \frac{M \vdash_m \tau_1 \quad M \vdash_m \tau_2}{M \vdash_m \tau_1 + \tau_2} \qquad \textsc{WF-Fun}\ \frac{M \vdash_m \tau_1 \quad M \vdash_m \tau_2}{M \vdash_m \tau_1 \overset{m}{\rightarrow} \tau_2} \qquad \textsc{WF-Ref}\ \frac{M \vdash_m \tau}{M \vdash_m \mathsf{ref}^{\omega}\ \tau}$$

$\boxed{M \vdash_m \tau}$

$$\textsc{WF-Loc}\ \frac{M \vdash_{m'} \sigma \qquad m \supseteq m'}{M \vdash_m \sigma@m'} \qquad \textsc{WF-Var}\ \frac{M(\alpha) \subseteq m}{M \vdash_m \alpha} \qquad \textsc{WF-Prod}\ \frac{M \vdash_m \tau_1 \qquad M \vdash_m \tau_2}{M \vdash_m \tau_1 \times \tau_2}$$

$$\textsc{WF-Rec}\ \frac{\{\alpha \mapsto m'\} \uplus M \vdash_{m'} \tau \qquad m \supseteq m'}{M \vdash_m \mu\alpha.\ \tau}$$

$\boxed{\sigma <: \sigma}$

$$\textsc{Sub-Refl}\ \frac{\sigma = \mathsf{int}^{\psi} \ \vee\ \sigma = \mathsf{ref}^{\mathsf{RW}\#m}\ \tau}{\sigma <: \sigma} \qquad \textsc{Sub-Sum}\ \frac{\tau_1 <: \tau_1' \qquad \tau_2 <: \tau_2'}{\tau_1 + \tau_2 <: \tau_1' + \tau_2'}$$

$$\textsc{Sub-Fun}\ \frac{\tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2'}{\tau_1 \overset{m}{\rightarrow} \tau_2 <: \tau_1' \overset{m}{\rightarrow} \tau_2'} \qquad \textsc{Sub-RefRO}\ \frac{\tau <: \tau'}{\mathsf{ref}^{\omega}\tau <: \mathsf{ref}^{\mathsf{RO}}\tau'}$$

$\boxed{\tau <: \tau}$

$$\textsc{Sub-Loc}\ \frac{\sigma <: \sigma' \qquad m \supseteq m'}{\sigma@m <: \sigma'@m'} \qquad \textsc{Sub-Var}\ \frac{}{\alpha <: \alpha} \qquad \textsc{Sub-Pair}\ \frac{\tau_1 <: \tau_1' \qquad \tau_2 <: \tau_2'}{\tau_1 \times \tau_2 <: \tau_1' \times \tau_2'}$$

$$\textsc{Sub-Rec}\ \frac{\tau <: \tau'}{\mu\alpha.\ \tau <: \mu\alpha.\ \tau'}$$

$\boxed{\Sigma \vdash u : \sigma}$

$$\textsc{TV-Int}\ \frac{}{\Sigma \vdash i^{\psi} : \mathsf{int}^{\psi}} \qquad \textsc{TV-Inj}\ \frac{\Sigma \vdash v : \tau \quad \mathrm{FV}(\tau')=\varnothing}{\Sigma \vdash \iota_1\ v : \tau + \tau' \quad \Sigma \vdash \iota_2\ v : \tau' + \tau}$$

$$\textsc{TV-Fun}\ \frac{\Sigma, \Gamma \vdash \gamma \qquad \{z \mapsto (\tau_1 \overset{m}{\rightarrow} \tau_2)@m, x \mapsto \tau_1\} \uplus \Gamma \vdash_m e : \tau_2}{\Sigma \vdash \langle \lambda_z x.\ e, \gamma \rangle : \tau_1 \overset{m}{\rightarrow} \tau_2} \qquad \textsc{TV-Ref}\ \frac{}{\Sigma \vdash \ell^{\omega} : \mathsf{ref}^{\omega}\ \Sigma(\ell)}$$

$\boxed{\Sigma \vdash v : \tau}$

$$\textsc{TV-Loc}\ \frac{\Sigma \vdash u : \sigma}{\Sigma \vdash u@m : \sigma@m} \qquad \textsc{TV-Fold}\ \frac{\Sigma \vdash v : [(\mu\alpha.\ \tau)/\alpha]\tau}{\Sigma \vdash \mathsf{fold}\ v : \mu\alpha.\ \tau} \qquad \textsc{TV-Pair}\ \frac{\Sigma \vdash v_1 : \tau_1 \qquad \Sigma \vdash v_2 : \tau_2}{\Sigma \vdash \langle v_1, v_2 \rangle : \tau_1 \times \tau_2}$$

$$\textsc{TV-Star}\ \frac{\vdash_{\varnothing} \tau}{\Sigma \vdash \bigstar : \tau}$$

$$\Sigma, \Gamma \vdash \gamma \overset{\triangle}{\iff} \forall x \in \mathrm{dom}(\Gamma).\ \Sigma \vdash \gamma(x) : \Gamma(x) \qquad \boxed{\Sigma, \Gamma \vdash \gamma}$$

$$\Sigma \vdash \delta \overset{\triangle}{\iff} \forall \ell \in \mathrm{dom}(\Sigma).\ \Sigma \vdash \delta(\ell) : \Sigma(\ell) \qquad \boxed{\Sigma \vdash \delta}$$

$\boxed{\Sigma \vdash \kappa : \tau \rhd \tau}$

$$\textsc{T-Top}\ \frac{}{\Sigma \vdash \top : \tau \rhd \tau} \qquad \textsc{T-Frame}\ \frac{\{x \mapsto \tau_1\} \uplus \Gamma \vdash_m e : \tau_2 \qquad \Sigma, \Gamma \vdash \gamma \qquad \Sigma \vdash \kappa : \tau_2 \rhd \tau_3}{\Sigma \vdash (\langle \mathsf{let}\ x = \square\ \mathsf{in}\ e \mid m, \gamma \rangle :: \kappa) : \tau_1 \rhd \tau_3}$$

$\boxed{\Sigma, \Gamma \vdash \varsigma : \tau}$

$$\textsc{T-Config}\ \frac{\Sigma, \Gamma \vdash \gamma \qquad \Sigma \vdash \delta \qquad \Sigma \vdash \kappa : \tau' \rhd \tau \qquad \Gamma \vdash_m e : \tau' \qquad m \neq \varnothing}{\Sigma, \Gamma \vdash m, \gamma, \delta, \kappa, e : \tau}$$

Fig. 9. λ-Symphony Type System Auxiliary Definitions

be present ($p \cup q = m$). Finally, x must be a (normal, non-share) integer; the result is a share at q, among parties q. Rule T-Embed types conversion of an integer known to parties m to a share among parties p.[3] Rule T-Reveal types share elimination: a share among parties q is converted to

[3] As with the operational semantics, we will need $p \subseteq m$ if the share is to be computed on later, due the premise $\vdash_m \psi$ in rules like T-Binop and T-Mux.

an integer, but then sent parties p (which must not be empty); the current mode includes those in p, and those q who hold the shares

5.2 Soundness

λ-Symphony's single-threaded semantics enjoys the standard type soundness property: If a program is type correct, then it will either diverge or else run to completion without errors in the single-threaded semantics. We prove this property using the standard progress and preservation lemmas.

Value typing and well-formedness. Towards proving progress and preservation, we extend the typing judgment to values, and define well-formedness judgments for the store, environment, and stack. These are all shown in the lower portion of Figure 9. Each assumes a *store typing* Σ, which maps pointers ℓ to types τ. Judgments $\Sigma \vdash u : \sigma$ and $\Sigma \vdash v : \tau$ type values; judgment $\Sigma, \Gamma \vdash \gamma$ types environments; judgment $\Sigma \vdash \delta$ types stores; and judgment $\Sigma \vdash \kappa : \tau_1 \rhd \tau_2$ types stacks. This latter judgment says that assuming the stack's topmost frame's hole is filled with something of type τ_1, the evaluation of the stack will produce a final value of type τ_2. Configurations—consisting of a mode, environment, store, stack and expression—are typed with the judgment $\Sigma, \Gamma \vdash \varsigma : \tau$.

The type rules for values are mostly standard, and we highlight only interesting or potentially surprising definitions. Rule TV-Int requires that integer values carry the same encryption annotation as their types. Rule TV-Inj requires the injected value is well-typed, and the requirement FV(τ') indicates the type for the other side of the sum must be closed (i.e., all type variables are well-scoped inside a binder, which occurs in recursive types.) Located values are well-typed when their location matches the location of the type. The most interesting case is TV-Star, which says the ★ value can be given any type so long as the type is valid in the empty mode $\varnothing$ (designated by $\vdash_\varnothing \tau$).

Environments γ are well-formed when all values in the environment are well-typed pointwise as determined by Γ, and likewise for stores δ by Σ. Rule T-Top indicates the "top of the stack" when given a value of type τ, will yield a final value of type τ. The T-Frame rule says that an intermediate stack frame, when given a value of type τ_1, will then run the expression e (with x bound to the value of type τ_1 in its stack-closure mode m and environment γ), and then call evaluate the rest of the stack with the resulting value τ_2 (the type the rest of the stack is expecting as input), and finally producing a τ_3 value—what the rest of the stack promised to finally produce. Configurations ς are then well-typed when the environment, store, stack and expression are all well-typed, and when the configuration mode m is nonempty. The requirement that m be nonempty is necessary to prove type safety, and reasonable because any par block with an empty set of parties is promptly skipped by the semantics, so m will never be empty for any reachable configuration.

Note the configuration's expression e must be well-typed in the mode m, but environment, store and value type judgments do not contain mode annotations m. In the semantics, values are restricted to the current mode m during variable access, but within the environment and store they are unrestricted. Stack frames retain the mode and environment of execution when stack frames are pushed, and restore these modes and environments when stack frames are popped.

Theorems. We prove standard progress/preservation for configurations.

Definition 5.1 (Terminal ST Configurations). A configuration $m, \gamma, \delta, \kappa, e$ is *terminal* when $\kappa = \top$, $e = a$ for some atom a, and $\gamma \vdash_m \delta, a \hookrightarrow \delta', v$ for some δ' and v.

Theorem 5.2 (ST Progress). *If* $\Gamma, \Sigma \vdash \varsigma : \tau$*, then either* ς *is terminal, or there exists* ς' *s.t.* $\varsigma \longrightarrow \varsigma'$.

Theorem 5.3 (ST Preservation). *If* $\Gamma, \Sigma \vdash \varsigma : \tau$ *and* $\varsigma \longrightarrow \varsigma'$ *then* $\Gamma', \Sigma' \vdash \varsigma' : \tau$ *for some* Γ' *and* Σ'.

The proof of Theorem 5.2 is by case analysis on the expression e inside the configuration ς, and Theorem 5.3 is by case analysis on the step relation $\varsigma \longrightarrow \varsigma'$. We provide detailed proofs for each theorem in the Appendix, and although somewhat tedious, they are straightforward.

6 DISTRIBUTED SEMANTICS

This section presents λ-Symphony's distributed semantics as an extension of its single-threaded semantics. We prove that the latter *forward-simulates* the former, and that the distributed semantics enjoys *confluent* execution. Together, these two properties jointly imply *soundness* of the single-threaded semantics w.r.t. the distributed semantics.

6.1 Overview

Recall the execution of the 2-party Millionaires example in Figure 1, which illustrates how the parties A and B execute the same program together. Some parts they do on their own: The part $\texttt{let}\ y = \texttt{par}[A]\ \texttt{read}\ \texttt{in}\ \ldots$ happens only on A and is skipped by B, and the part $\texttt{par}[B]\ \texttt{write}\ z$ happens only on B and is skipped by A. Some parts they must do together: A makes a share and B receives it by doing $\texttt{let}\ sx = \texttt{share}[A \rightarrow A, B]\ x\ \texttt{in}\ \ldots$, and both parties communicate to compare two shares when doing $\texttt{let}\ r = sx < sy\ \texttt{in}\ \ldots$

The goal of the distributed semantics is to formalize how this works. The key idea is that each party P will execute its P-local parts of the program using the single-threaded semantics in mode $m = \{P\}$. As such, when A reaches expressions like $\texttt{par}[B]\ \texttt{write}\ z$ in Millionaires, intersecting current mode $\{A\}$ with the $[B]$ on the par yields $\varnothing$ so rule ST-ParEmpty will evaluate the expression to $\bigstar$; i.e., the write will be skipped. On the other hand, expressions like $\texttt{let}\ r = sx < sy\ \texttt{in}\ \ldots$ cannot be executed at only A or B alone, so the semantics allows the parties to synchronize such that the expression is evaluated in mode $m = \{A, B\}$. Solo and synchronized execution are made possible by a *slicing* function that permits selectively joining ("reverse slicing") multiple parties' individual configurations into a single one suitable for single-threaded evaluation, and then splitting ("slicing") the single-threaded configuration that results. Slicing is quite general so we employ a well-formedness condition to ensure that distributed executions are modeled as expected.

We begin by defining the structure of distributed program configurations and how they can be sliced from a single-threaded one (Section 6.2); then we present the distributed semantics (Section 6.3); finally we prove our metatheoretic results (Section 6.4).

6.2 Distributed Configurations

The definition of distributed configurations C is given in Figure 10. A distributed configuration consists of a a finite map from the set of concurrently executing parties to their local states, expressed as 4-tuples consisting of **(1)** a distributed environment $\dot{\gamma}$, **(2)** a distributed store $\dot{\delta}$, **(3)** a distributed stack $\dot{\kappa}$, and **(4)** an expression e. Distributed environments, stores, and stacks are the same as their single-threaded counterparts with two differences. First, instead of containing values v, they contain *distributed values* $\dot{v}$. Distributed values differ from normal values v only in that they are all non-located—there is no mode annotation $@m$ on any of them. Second, stack frames no longer store a saved mode m.

We can relate a single-threaded configuration ς to a distributed one by *slicing* it, written $\varsigma\lightning$. As shown in the middle of the figure, each party A in the mode m of ς is mapped to a 4-tuple that represents its local state; this local state consists of sliced versions of the environment γ, store δ, and stack κ of ς that are specific to A. Slicing is essentially a homomorphism where located values $u@p$ are sliced either to (the sliced version of) u if $A \in p$ or $\bigstar$ otherwise; this is because if $A \in p$ then A can access u, but if $A \notin p$ then it cannot. Note that shares i^{ψ} are left alone (only their location annotation is affected): if ψ is $\texttt{enc\#}m$ we do not modify m, which is important for keeping

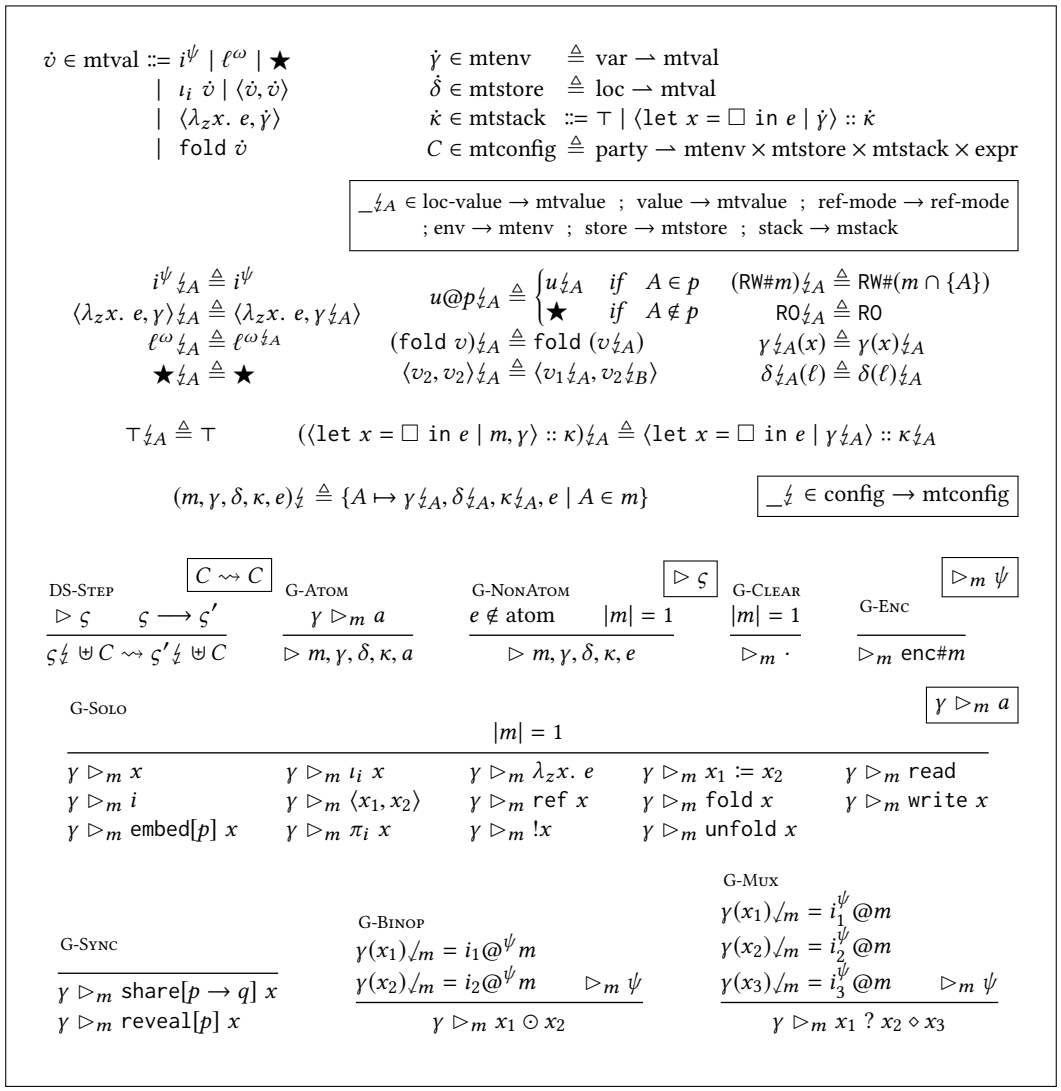

Fig. 10. Meta-theoretic sets and functions that are used to define the distributed semantics.

track which parties must synchronize to compute on the share. On the other hand, we *do* slice m in the $\texttt{RW\#}m$ annotation on references. This is needed to allow assignment to occur at A without synchronization. Note that slicing a configuration ς ignores inactive parties $B \notin m$; the distributed semantics keeps track of these parties' individual states in another manner, as we will see.

6.3 Distributed Operational Semantics

The distributed semantics $C \rightsquigarrow C'$ is defined in the second half of Figure 10. The semantics is defined by a single rule DS-Step which constructs the distributed semantics from steps in the single-threaded global semantics. DS-Step specifies that if configuration ς steps via the *single-threaded* semantics to ς', then the disjoint union of **(1)** the slicing of ς and **(2)** any distributed configuration

C steps in the *distributed* semantics to the disjoint union of **(1)** the slicing of ς' and **(2)** C. The DS-Step additionally requires that $\triangleright\ \varsigma$, discussed shortly.

To evaluate a λ-Symphony program e among parties m, we construct its initial ς with the program e, mode m, and an empty store, store, and stack. Then we slice ς to produce C_0. At this point, C_0 starts evaluating according to DS-Step. Essentially, the rule non-deterministically decomposes C_0 into two parts. The C part represents all of the parties that are inactive for this step. The rest is "reverse silced" to ST configuration ς which then takes a step according to the ST semantics. The result is then forward sliced and recombined with C.

Example: Millionaires. Consider the code from Figure 1 once again. Our initial configuration C_0 is

$$A \mapsto (\varnothing,\ \varnothing,\ \top,\ e),\ B \mapsto (\varnothing,\ \varnothing,\ \top,\ e)$$

where $\varnothing$ is the empty function (used for the empty environment and store), $\top$ is the empty stack, and e is the source program from the figure, i.e., $\texttt{par}[A, B]\ \texttt{let}\ x = \texttt{par}[A]\ \texttt{read in}\ \ldots$ To apply rule DS-Step, we decompose C_0 into $\varsigma\lightning$ and C. There are three ways we could do this. First, we could pick $\varsigma\lightning = (A \mapsto (\varnothing,\ \varnothing,\ \top,\ e))$ and $C = (B \mapsto (\varnothing,\ \varnothing,\ \top,\ e))$, since $\varsigma\lightning \uplus C = C_0$. Given $\varsigma\lightning$, we can "run slicing backward" to yield $\varsigma = (\{A\}, \varnothing,\ \varnothing,\ \top,\ e)$. That is, we have a normal single-threaded configuration with mode $m = \{A\}$. Using the ST semantics, this configuration steps according to rule ST-Par, yielding configuration $\varsigma' = (\{A\}, \varnothing,\ \varnothing,\ \top,\ e')$ since $\{A\} \cap \{A, B\} = \{A\}$, and where e' is $\texttt{let}\ x = \texttt{par}[A]\ \texttt{read in}\ \ldots$. Finally, we compute $\varsigma'\lightning$ and recombine with C, yielding

$$A \mapsto (\varnothing,\ \varnothing,\ \top,\ e'),\ B \mapsto (\varnothing,\ \varnothing,\ \top,\ e)$$

Notice that A's code is now e' while B's is still e.

The second way to split C_0 would be to swap the roles of A's and B's configurations—we use $\varsigma\lightning$ for B's part instead of A's. This would produce the symmetric result: B will have taken a step rather than A. The third and final way would be to choose *both* A's and B's parts, so that we end up with single threaded configuration $\varsigma = (\{A, B\}, \varnothing,\ \varnothing,\ \top,\ e)$. This will have the effect of running the outermost $\texttt{par}[A, B]\ \ldots$ for *both parties at once*. But this is weird: in what sense is this a *distributed* semantics if we can step separate parties in one global step?

Well formed slices. We solve this problem by ruling out non-sensical slices which do not really model distributed computation. The judgment $\triangleright\ \varsigma$ limits the form that ς can take, based on ς's expression component. The rule G-Atom references judgment $\gamma \triangleright_m a$ which requires that for all a that should happen at a single party, m must be a singleton (per rule G-Solo). Rule G-NonAtom places a similar constraint on non-atomic expressions e, which are always local.

For those a that require parties to synchronize, rules G-Sync, G-Binop, and G-Mux constrain on m only according $\triangleright_m \psi$, when computing on shares. The effect is that m may contain multiple parties. Returning to the Millionaires example, $\triangleright\ \varsigma$ will rule out the slice in which we have $m = \{A, B\}$ because rule G-NonAtom requires $|m| = 1$. On the other hand, continuing the example we will reach the point that both parties reach the code $e'' = \texttt{share}[A \to A, B]\ x$. Here, G-Sync will allow $m = \{A, B\}$, which will be needed to step this expression according to ST-Share, with will create shares of x and share them between A and B.

A reverse slice to a non-singleton m, as needed for `share`, works only when the parties' configurations have the same expression e and otherwise compatible stores, stacks, etc.—these would only differ by one set of parties having $\bigstar$ and the others agreeing on the (distributed) value. Consider for e'' what the environments look like for each of A and B (their stack and store are the same). A's environment $\dot{\gamma}_A$ maps x to some integer, 5 say, and maps y to $\bigstar$. The environment $\dot{\gamma}_B$ for B maps x to $\bigstar$ and y to some integer, say 6. To step e'' via DS-Step, we need to reverse slice to some ς whose $m = \{A, B\}$, since both parties must be present to apply ST-Share. The reverse slice will work at this point to produce a γ in ς that maps x to $5@\{A\}$ and y to $6@\{B\}$. This is because $5@\{A\}\lightning_A$

produces 5 and $5@\{A\}\lightning_B$ produces ★, the two things that x maps to in the two environments $\dot{\gamma}_A$ and $\dot{\gamma}_B$. It's a similar story for y. In general, if both A and B had the same value u for some variable z, then reverse slicing would produce $u@\{A, B\}$. If they each had a different (non-★) value, no reverse slice would be possible. This argument applies equally to stores δ.

Forcing the parties to reach equivalent configurations at program points involving MPC models them reaching the exact point at which they need to communicate. But rather than formalize that communication directly, we simply "group them together" to take a step according to the single-threaded semantics. It should be obvious that in an actual MPC implementation, true communication can safely take place.

Final notes. Recall the case of slicing writeable references in Figure 10, which reduces the number of writers, i.e., $\texttt{RW\#}(m \cap \{A\})$. Normally, operational rule ST-Assign requires that all writers on a reference be present in the current mode, but we want to allow those writers to proceed independently in the distributed semantics. The slicing rule makes that possible. The metatheory (next) ensures that this presents no risk. Indeed, we could perform the same trick to allow binary operations on shares to proceed independently, by similarly slicing ψ as $\texttt{enc\#}(m \cap \{A\})$. This is a bit more delicate because sometimes all parties *need* to be present, e.g., to share initial inputs and reveal the final result. As such, we would need to make slicing more context dependent, which could be a little messy, so we don't bother. Finally, note that ST-Embed is such that there is no forced connection between p in $\texttt{embed}[p]$ and the mode m, so it can already proceed within a single party.

6.4 Relating the Single-threaded and Distributed Semantics

This section sketches our proof of *single-threaded soundness*, which states the sense in which we can interpret a λ-Symphony program in terms of its single-threaded semantics, even though in reality it will execute in a distributed fashion. More detailed proofs are provided in the Appendix.

We might hope to prove *bisimulation* for the two semantics. Sadly, backward simulation (DS to ST) does not hold. Consider this program:

$$\texttt{par}[A, B]\ \texttt{let}\ x = \texttt{par}[A]\ \langle \textit{infinite loop} \rangle\ \texttt{in}\ 1$$

In the distributed semantics, party B's execution of the program can reach the end—which returns 1—while A loops. But such a distributed configuration cannot be sliced to from an ST execution: there, the program will get stuck in the loop and never reach the end.

As such, we prove what we call *ST Soundness*, which says that if the ST semantics terminates properly then all possible distributed executions will too, at the sliced-to ST terminal configuration:

Theorem 6.1 (ST Soundness). *If $\varsigma \longrightarrow^* \varsigma'$, ς' is terminal, $\varsigma\lightning \rightsquigarrow^* C$ and $C \not\rightsquigarrow$, then $C = \varsigma'\lightning$.*

(See Section 5.2 for the definition of *terminal* configurations.) The proof follows from two subsidiary lemmas. The first is called *weak forward simulation*. The simulation is "weak" in the sense that it assumes terminal final states.

Lemma 6.2 (ST Weak Forward Simulation). *If $\varsigma \longrightarrow^* \varsigma'$ and ς' is terminal, then $\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$ and $\varsigma'\lightning \not\rightsquigarrow$.*

Proof. By induction on the multistep judgment $\varsigma \longrightarrow^* \varsigma$. There are two base cases: 0 steps and 1 step. The inductive case is by case analysis on the first single step of the trace. □

The other half of the proof aims to show that for any given distributed execution that ends in a terminal state, all other possible executions will terminate as well, at the same state. This result follows primarily from a single-step confluence property, stated thus:

Lemma 6.3 (DS Single-step Confluence). *If $C \rightsquigarrow C_1$, $C \rightsquigarrow C_2$ and $C_1 \neq C_2$, then there exists C_3 s.t. $C_1 \rightsquigarrow C_3$ and $C_2 \rightsquigarrow C_3$.*

Proof. Follows from the single-threaded semantics being deterministic no matter how they are sliced, and distributed configurations associating a unique configuration state with each party. However party A steps on the left and $B \neq A$ steps on the right, each side can employ the other side's rule to meet at a common state. Critically, a party A can appear in only one reverse slice, e.g., due to slicing well-formedness it is not possible to reverse slice to a configuration that can step both with $m = \{A\}$ and $m = \{A, B\}$ (say). □

Single-step confluence implies *multi-step confluence* (a classic result for transition systems). With this, we can trivially show that the final distributed configurations states are unique.

Corollary 6.4 (DS End-state Determinism). *If* $C \rightsquigarrow^* C_1$ *and* $C \rightsquigarrow^* C_2$, $C_1 \not\rightsquigarrow$ *and* $C_2 \not\rightsquigarrow$ *then* $C_1 = C_2$.

Summing up the proof of ST soundness: By weak forward simulation we establish $\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$ and $\varsigma'\lightning \not\rightsquigarrow$, and by end-state determinism we have $C = \varsigma'\lightning$.

7 IMPLEMENTATION AND EXPERIENCE

This section discusses an interpreter we implemented for an extension of λ-Symphony, as well as some of the MPC programs we wrote using it. The interpreter, and programs, are available in our supplementary material.

7.1 Interpreter

We implemented an interpreter for a feature-rich extension of λ-Symphony in ~4,000 lines of Haskell, not including a custom standard library designed for rapid language prototyping. At present, we lack a static type checker and a distributed implementation employing cryptography. Instead, the interpreter is based on the single-threaded semantics presented in Section 4. As such, it usefully gets stuck when a program tries to take a step that would be non-sensical in a distributed setting, as prescribed by the semantics in Section 4. For example, rule SS-Assign prevents modifying a shared reference at one party but not the others. The interpreter serves as a sanity-check that λ-Symphony can express useful programs by rejecting programs where the single-threaded semantics gets stuck.

The interpreter adds convenience features on top of λ-Symphony, including standard conditionals; general data types and pattern matching (generalizing λ-Symphony's sum types); arrays (essentially references to lists); families of distinct share types for different protocols (generalizing λ-Symphony's single share type); and non-local control flow (e.g., early return). Two more interesting added features are *wire bundles* and *first-class parties* (i.e., party sets as data).

Implicit Wire Bundles. In λ-Symphony, normal variables visible to multiple parties always have the same value. Different values require different variables, e.g., as with x and y in Millionaires (Figure 1). However, it is often useful to think of a particular variable as representing a certain *role* in a computation, where each party has their own value for that variable, but they all do the same thing with it. In our implementation we implement a third useful modality for types, called *bundles* and annotated `bundle#`m, which can be used in this way. Operations are then implicitly "vectorized" across each party's local value. For example, if $x : \texttt{int}^{\texttt{bundle\#}\{A,B\}}@\{A, B\}$ is in the environment, then A and B each have their own, local version for the value x, and $x + x$ is a local operation that A and B compute on their own (in the distributed setting), resulting in a value also of type $\texttt{int}^{\texttt{bundle\#}\{A,B\}}@\{A, B\}$.

First-class Parties. In λ-Symphony, parties (and party sets) are static and fixed, and syntax which requires party annotations require they be written directly. In our implementation, we allow party sets to be constructed dynamically and bound to variables. This feature, along with wire bundles,

is useful for writing generic libraries. Instead of fixing the parties in a par block inside the library, the party set is passed by argument. Operations in the par block are made generic by simply iterating over the set as a normal data structure. Using this feature we have developed a ~1,000 line standard library, and algorithms generic in the particular parties (e.g., a generic version of GCD from Figure 4).

Adding these features to λ-Symphony will be interesting future work. Notably, the type system will need to employ parametric polymorphism over parties, and dependent types (with inclusion constraints) for party sets. We say more in the Appendix.

7.2 Experience

```
-- Insert an element x into a
-- sorted list xs so that the
-- output list is also sorted.
def insert : int{yao:A,B}
          -> list int{yao:A,B}
          -> list int{yao:A,B}
def insert x xs = case xs
 { [] -> [x]
 ; (x'::xs') ->
    let (lo, hi) = mux if x < x'
                   then (x, x')
                   else (x', x)
    in  lo :: insert lt hi xs' }

-- Sorts a list of elements.
def sort : list int{yao:A,B}
        -> list int{yao:A,B}
def sort = foldr insert []
```

We have implemented roughly two thousand lines of MPC programs that run under our interpreter. The code to the left illustrates insertion sort over a list of shares among two parties A and B. Integer types `int{yao:A,B}` are shorthand for $\texttt{int}^{\texttt{enc\#}\{A,B\}}@\{A,B\}$ in the core language. The key takeaway is that despite the fact that the program involves a joint, iterative computation between two parties, it looks very much like normal code.

In addition to insertion sort we also implemented a number of other benchmarks including secure median (using a mixed mode trick discussed in Rastogi et al. [2014]), secure quicksort, secure merge sort, secure Karmarkar's algorithm, and secure database functionality including secure database statistic calculation. Brief descriptions of these examples are given in the Appendix, and source code which successfully runs through our interpreter is provided in the non-anonymized supplemental material.

8 RELATED WORK

This section reviews past work on MPC languages and frameworks, as well as on languages/formalisms for distributed, parallel, and concurrent programming.

8.1 Wysteria

λ-Symphony is most similar to Wysteria, a language that also takes a principled approach to MPC language design [Rastogi et al. 2014, 2019]. Wysteria's key contribution is its single threaded interpretation, an idea that λ-Symphony also leverages.

There are three key differences between Wysteria's design and λ-Symphony which, in our view, makes λ-Symphony a superior platform for MPC. First, Wysteria requires input-providing parties to be computation parties; but, as discussed in the context of Millionaires in Figure 3, this approach does not scale. λ-Symphony permits many parties to delegate computation to a chosen few.

Second, Wysteria does not support first-class secret shares. Specifically, in Wysteria such shares are created and disposed of transparently inside of a *sec block*. These sec blocks are intended to emulate a *trusted third party*. While a trusted third party is a useful abstraction for reasoning about security, it can be quite cumbersome for expert-level programming. In particular, sec blocks only model MPC circuits, and thus cannot allow arbitrary recursion or tight interaction between cryptographic and local computation. These features are critical for writing algorithms such as GCD (Figure 4), insertion sort (Section 7.2), and others naturally. Wysteria versions of these algorithms

would be more obfuscated, require much duplicated code, and would hard-code input sizes. λ-Symphony's share-centric view of MPC allows the programmer to freely mix secure computation with cleartext computation. Arbitrary recursive algorithms come for free and are easy to understand.

Third, and partially as a result of the second difference, λ-Symphony's formalization is a concise extension of a standard language: Its single-threaded semantics and type rules are essentially standard, with straightforward additions to indicate "who" (locations @*m* on types and values), "where" (`par` blocks, solely), and "how" MPCs are to take place (shares, slicing of the single-threaded semantics). Wysteria's formalization is more complex—it has several notions of where a value might be located, and its distributed semantics is forced to explicitly model a trusted third party. Subjectively, we believe that λ-Symphony's design is more elegant and easier to understand, and thus easier to extend. That said, Wysteria also includes abstractions for wire bundles and party sets that are very useful for writing more generic computations. We believe we can extend λ-Symphony to include these features (they are in our interpreter already, per Section 7.1) without compromising its concise formal setup.

8.2 Other MPC Frameworks

Most prior MPC programming frameworks can be classified as either library-based, or language-based [Hastings et al. 2019]. We consider both.

Library solutions. Several works provide a library of MPC tools in an existing language. The EMP Toolkit [Wang et al. 2016] provides MPC primitives via `C++` custom types and overloaded operators. Sharemind's SecreC [Randmets 2017] similarly embeds secure operations into a flavor of `C++`. MyPC [Schoenmakers 2019] provides an MPC library implemented in Python.

Library solutions are useful for experts. Cryptographers can experiment with and easily extend these libraries in order to work on new MPC directions. Unfortunately, such tools are less useful for non-experts because the host language does not provide a distributed semantics. Thus, while it is possible use MPC, it becomes the programmer's responsibility to manage distributed computation herself. In practice, this means writing several programs that work together to compute the distributed application (recall Figure 2). This style of programming leads to confusing bugs when the different parts of the application become 'out-of-sync'. That is, when the different parties disagree about program values that the programmer intended for them to agree on.

λ-Symphony provides an interface to MPC through similar means to libraries: we provide simple datatypes whose operations 'build a circuit' which is then dispatched by an MPC backend. This design is influenced by libraries, especially the EMP Toolkit [Wang et al. 2016]. λ-Symphony extends this bare-bones functionality with a simple yet powerful distributed semantics. λ-Symphony's single-threaded interpretation ensures that desynchronization problems are impossible.

Language Solutions. Another direction, which we adopt, is to build MPC in at the language level, either by extending/utilizing an existing language or by building a tailored language.

A number of works compile high-level programs into circuits. CMBC-GC [Franz et al. 2014] compiles a subset of `C` into circuits. Similarly, Frigate [Mood et al. 2016] compiles a `C`-like language to circuits. While these tools are useful for MPC, they are not standalone solutions; neither tool provides a runtime. Instead, the compiled circuits must be used with external MPC backends.

Other works provide language front ends that are designed to feel familiar to most programmers. Obliv-C [Zahur and Evans 2015] embeds MPC programs in `C` programs by implementing a GCC wrapper. SCALE-MAMBA [Aly et al. 2019] is a runtime and simple language (heavily influenced by Python) that focuses on providing a front-end to sophisticated cryptographic protocols. OblivVM [Liu et al. 2015] is a Java-like language for MPC programming. These works focus on porting MPC concepts to languages that feel familiar to programmers. In contrast, λ-Symphony's

focus is sound design. We built λ-Symphony to develop language features, like the single-threaded interpretation and located types, that make reasoning about MPC easier for the programmer.

8.3 Distributed, Parallel, and Concurrent Computing

λ-Symphony's focus is MPC, not general distributed programming. This said, MPC *implies* distributed programming and associated problems, like synchronization between coordinating parties. Thus, works on parallel and concurrent computing are also relevant.

The π-calculus is a foundational calculus for describing concurrent computations [Milner 1999]. Orc is a language that emerged from the π-calculus and can orchestrate complex, distributed computations [Kitchin et al. 2009]. Compared to π-calculi, λ-Symphony is simpler. To achieve λ-Symphony's simplicity, we take advantage of the relatively simple structure of MPC computations, where the parties work together to achieve the same output. This simplicity is made manifest through our single-threaded interpretation.

Session types formalize the rules of message-passing programs [Coppo et al. 2016; Honda et al. 2008; Yoshida et al. 2010]. Such types enforce what messages pass over communication channels and when. Another direction uses modal types to ensure that distributed resources are not inappropriately used [Murphy VII et al. 2007]. Compared to session and modal types, λ-Symphony's type system is relatively simple and focuses on the location of information.

A large body of work has developed sophisticated techniques for writing parallel and concurrent programs [Bocchino Jr et al. 2009; Fluet et al. 2007; Frigo et al. 1998; Kuper and Newton 2013; Reppy 1991; Yelick et al. 1998, etc.]. Such works focus on aspects such as taking advantage of highly parallel machines to compute efficiently. Additionally, many works encourage a distributed message passing style of programming that helps programmers deal with specific distributed programming problems, like managing network partitions and crashing systems [Armstrong et al. 1993; Epstein et al. 2011; Liskov 1988]. The MPC setting, which divides work among hosts for security and not for efficiency or redundancy, warrants a different approach.

9 CONCLUSION

This paper has presented λ-Symphony, an expressive yet concise domain-specific language for expressing MPCs. λ-Symphony extends the simply-typed lambda calculus with shares, par blocks, and located types to express the "how," "where," and "who" aspects of MPC programming. We showed how this concise treatment provides expressive power despite its simplicity. We formalized these language extensions with a single-threaded and distributed semantics, and we proved the correspondence between the two. We also proved type soundness for our type system with located types. Finally, we discussed our interpreter, showing that λ-Symphony is a suitable basis for languages that can handle real-world problems.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Abdelrahaman Aly, Marcel Keller, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. 2019. SCALE-MAMBA. https://homes.esat.kuleuven.be/ nsmart/SCALE/.

Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *ACM CCS 2016: 23rd Conference on Computer and Communications*

Security, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, Vienna, Austria, 805–817. https://doi.org/10.1145/2976749.2978331

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. 1993. Concurrent programming in ERLANG. (1993).

Franz Baader and Tobias Nipkow. 1999. *Term rewriting and all that.* Cambridge university press.

Robert L Bocchino Jr, Vikram S Adve, Sarita V Adve, and Marc Snir. 2009. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*. 4–4.

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global progress for dynamically interleaved multiparty sessions. *Math. Struct. Comput. Sci.* 26, 2 (2016), 238–302.

Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. 2011. Towards Haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*. 118–129.

Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. 2007. Manticore: A heterogeneous parallel language. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*. 37–44.

Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. 2014. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. In *Compiler Construction*, Albert Cohen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 244–249.

Matteo Frigo, Charles E Leiserson, and Keith H Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 212–223.

Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *19th Annual ACM Symposium on Theory of Computing*, Alfred Aho (Ed.). ACM Press, New York City, NY, USA, 218–229. https://doi.org/10.1145/28395.28420

Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1220–1237. https://doi.org/10.1109/SP.2019.00028

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 273–284.

David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. 2009. The Orc programming language. In *Formal techniques for Distributed Systems*. Springer, 1–25.

Lindsey Kuper and Ryan R Newton. 2013. LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*. 71–84.

Barbara Liskov. 1988. Distributed programming in Argus. *Commun. ACM* 31, 3 (1988), 300–312.

Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 359–376. https://doi.org/10.1109/SP.2015.29

Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay - Secure Two-Party Computation System. In *USENIX Security 2004: 13th USENIX Security Symposium*, Matt Blaze (Ed.). USENIX Association, San Diego, CA, USA, 287–302.

Robin Milner. 1999. *Communicating and mobile systems: the pi calculus.* Cambridge university press.

Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. 2016. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 112–127.

Tom Murphy VII, Karl Crary, and Robert Harper. 2007. Type-safe distributed programming with ML5. In *International Symposium on Trustworthy Global Computing*. Springer, 108–123.

Jaak Randmets. 2017. Programming Languages for Secure Multi-party Computation Application Development.

Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*. http://www.cs.umd.edu/~mwh/papers/wysteria.pdf

Aseem Rastogi, Nikhil Swamy, and Michael Hicks. 2019. Wys*: A DSL for Verified Secure Multi-party Computations. In *Proceedings of the Symposium on Principles of Security and Trust (POST)*. http://www.cs.umd.edu/~mwh/papers/wysstar.pdf

John H Reppy. 1991. CML: A higher concurrent language. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 293–305.

Berry Schoenmakers. 2019. MPyC: Secure multiparty computation in Python. Github. https://github.com/lschoe/mpyc

Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *23rd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Chicago, Illinois, 160–164. https://doi.org/10.1109/SFCS.1982.38

Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, et al. 1998. Titanium: a high-performance Java dialect. *Concurrency and Computation: Practice and Experience* 10, 11-13 (1998), 825–836.

Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. 2010. Parameterised multiparty session types. In *International Conference on Foundations of Software Science and Computational Structures*. Springer, 128–145.

Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. Cryptology ePrint Archive, Report 2018/706. https://eprint.iacr.org/2015/1153.

A APPENDIX

A.1 Adding Polymorphic and Dependent Types

λ-Symphony's type system (Section 5) is monomorphic. Making it polymorphic in the *mode* of a computation would be useful. To see why, consider the following example (in relaxed syntax):

```
par[A,B]
  let f = lam x. par[A] (x,1) in
  let z = par[A,B] 1 in
  let z' = par[A] 1 in
  let y = f z in
  let y' = f z' in ...
```

We could give `f` the following monomorphic type so that both calls to it succeed: $\texttt{int}^{\{A\}\ \{A,B\}}\!\!\longrightarrow \texttt{int}^{\{A\}} \times \texttt{int}^{\{A\}}$. Then the first call would use subtyping on the argument, since $\texttt{int}^{\{A,B\}} <: \texttt{int}^{\{A\}}$. However, consider what would happen if we added the line

```
let z = par[A] f 1 in ...
```

This does not seem any different than the call assigning to `y'` and yet the type system will reject it—we cannot call `f` in a mode other than $\{A, B\}$. While for this function calling it in mode $\{A\}$ would be safe, if the function's body involved a `share` operation including both A and B (e.g., on its argument x) it would have the same type but would not be safe to call it with just A.

We could imagine fixing this by trying to distinguish a lower bound versus an upper bound on which parties should be present. For example, if we had the function:

```
let f = lam _.
  let y = par[A,B] share[A->A,B] 0 in
  par[C] 0
in ...
```

Function `f` must be called in a mode $m \supseteq \{A, B\}$ or else the `par` (and `share`) call will fail. But the return type will differ depending on the actual choice of m. For example, if we call `f` in mode $\{A, B\}$ or $\{A, B, D\}$, the return type should be `int@{}`. If we call `f` in mode $\{A, B, C\}$ or $\{A, B, C, D\}$, the return type should be `int@{C}`.

This line of reasoning leads naturally to the idea that mode indicators m on types can be *polymorphic* (universally quantified), along with constraints on their possible solutions. The above example could be given the type

$$\forall \beta, \gamma.[\beta \supseteq \{A, B\}, \gamma = \beta \cap \{C\}] \Rightarrow \texttt{int@}\{\}\ ^{\beta}\!\!\longrightarrow \texttt{int@}\gamma$$

Such a type conveys the minimum constraints on the caller's mode while at the same time affords precise mode annotations on types affected by the particular choice of mode.

To develop such an approach, we will need to have a logic for deciding set constraints, and allow symbolic party set expressions in place of party literals in types. For example, instead of requiring $\tau = \texttt{int}^{\texttt{enc\#}m}$ when typechecking binary operations in mode m in rule T-Binop, we instead require $\tau = \texttt{int}^{\texttt{enc\#}m'}$ and $\Delta \models m = m'$ in the theory of (simple) sets, and where Δ is the current assumption on symbolic party variables.

As discussed in Section 7.1, we also want to support first-class party sets, which will necessitate dependent types. These should be a natural addition to the above approach. In particular, instead of only reasoning about modes as literals or *type* variables, we can also consider them as *term* variables, following the lead of Wysteria [Rastogi et al. 2014]. To decide the set constraints that result, we can appeal to an SMT solver (Wysteria uses Z3).

The development of this type system is ongoing. We do not foresee problems with these additions, but of course there may be twists and turns as we get further into it. The simplicity of λ-Symphony's semantics and type system give us confidence that the extension will go through without much trouble.

A.2 Example Programs

To demonstrate the expressiveness of our implemented language, which significantly extends the core features in λ-Symphony, we implement a number of non-trivial MPC programs which run successfully through our interpreter, and for which resources required to execute in a realistic MPC implementation can be precisely estimated based on trace events emitted during interpretation. We present multiple example here, all of which are executable using our implemented interpreter, and available as source files in the supplemental material to this paper. We show the first examples, insertion sort as a full code listing with an in-depth description. The rest of the examples are discussed briefly, and available as full source code which successfully runs through our interpreter in the non-anonymzied supplemental material accompanying this paper.

```
-- Insert an element x into a
-- sorted list xs so that the
-- output list is also sorted.
def insert : int{yao:A,B}
          -> list int{yao:A,B}
          -> list int{yao:A,B}
def insert x xs = case xs
 { [] -> [x]
 ; (x'::xs') ->
    let (lo, hi) = mux if x < x'
                   then (x, x')
                   else (x', x)
    in  lo :: insert lt hi xs' }

-- Sorts a list of elements.
def sort : list int{yao:A,B}
        -> list int{yao:A,B}
def sort = foldr insert []
```

Insertion Sort. The core insertion sort algorithm implemented in our interpreted language is shown to the left. In our interpreted language, integer types `int{yao:A,B}` are shorthand for $\texttt{int}^{\texttt{enc\#}\{A,B\}}@\{A, B\}$ in the core language—that is, the encryption annotation and the location annotation are always the same, and written only once, and the underlying particular MPC protocol is mentioned directly (in this case `yao`) instead of an abstract MCP encryption protocol `enc`. A beneficial aspect of the language is that the implementation of `insert` is nearly identical to what a function programmer would write in a standard functional language, except for the use of `mux if` instead of `if`, and especially considering it has a direct interpretation as a distributed program which makes use MPC. The same is true for `sort`: it's immediately recognizable as a standard functional program. We require the programmer to write `mux if` instead of `if` because when branching on secure values, both branches are executed, after which the resulting values from both branches are multiplexed based on the guard—just like the ternary conditional $x\ ?\ x \diamond x$ in λ-Symphony. The `mux if` construct is slightly more powerful that ternary conditionals in λ-Symphony in that compound types (products and encrypted sums) are supported as the return type for the branches—so long as the base types support encrypted multiplexing—whereas ternary conditionals in λ-Symphony only support base types.

Although `sort` looks "just" like a standard functional program, in a distributed MPC setting we have the additional task of setting up and secret sharing the list of encrypted values which are passed to `sort` as its argument, and "revealing" the output of `sort` to some set of designated parties. We show this code next, which utilizes helper functions with definitions shown in the appendix.

```
def main : unit -> list int{C}
def main () =
  par {A,B,C,D,E}
    -- Read in inputs as individually secret lists of secret data
    let inputLists : (list int){bundle:A,B}
    let inputLists = solo {A,B} read (list int) from "input.data"
```

```
-- Share the elements of each list and reveal the list structures of each party's list
let sharedLists : list (list int{yao:C,D})
let sharedLists = fold-f [] (fun P x i -> (preprocessLists P {C,D} x) :: i) inputLists

-- Combine all lists into one list of all data
let sharedCombinedList : list int{yao:C,D}
let sharedCombinedList = concat sharedLists

-- Sort the list
let sharedCombinedSortedList : list int{yao:C,D}
let sharedCombinedSortedList = sort sharedCombinedList

-- Reveal each element in the list
let revealedCombinedSortedList : list int{E}
let revealedCombinedSortedList = map (fun x -> reveal{E} x) sharedCombinedSortedList

in revealedCombinedSortedList
```

In this driver code, input is first read from "input.data", locally to each party `A` and `B` (the "input parties") separately, but in parallel, and bound to variable `inputLists`. The type of this result is a "bundle", `(list int){bundle:A,B}`, which is a list of integers known to `A`, and a separate list of integers known to `B`. Next, we "fold" over the bundle (so execute the folding function two times, once for `A`, and once for `B`) and create encrypted values of the contents, shared between `C` and `D`, the "compute parties" for the MPC, and bound to varaible `sharedLists`. Next, these lists are concatenated to a single list containing all values from `A` and `B`, as encrypted values shared between `C` and `D`. Next, this list of encrypted values is sorted by calling the `sort` MPC algorithm shown previously, and bound to variable `sharedCombinedSortedList`. Finally, we map over the resulting list and reveal each sorted value to `E`, the "output party", bound to variable `revealCombinedSortedList`.

Database Statistics. In this example, the players together maintain a database of records relating to athletes at public universities. The stored information about the players is private, but prospective athletes can make queries that return aggregates of schools' records, assuming that a sufficiently large number of records exist to protect the privacy of the athletes. Additionally, new records can be inserted to the database. Database insertions and queries look indistinguishable to the database hosts: they know that a query occurs, but cannot determine whether a new record was added or if a query was issued. This benchmark extensively uses first class shares, in particular to recursively walk the database to perform updates and accumulate statistics. The full solution is written in 312 lines of code in `examples/db-stats.psl`.

Aggregate Targeted Query. In this example, a list of secret integers from multiple parties is first secret shared and combined into a single list. Parties may submit up to some max number of integers. Additionally, a subset of parties may submit a secret list of queries and targets, where a query is one of the following: max, min, median, or number of unique values. A target is the party designated to receive the answer to a corresponding query. Each query is run in a multi-party computation between all parties, and parties designated as targets to certain queries receive the results of those queries but do not learn who targeted them nor how many times they were targeted, only that they were targeted by at least one party. This example showcases the use and juggling of complex data structures within an mpc computation such as lists filled with sum and product types.

Sorting. We implement a standard, functional, deduplicated merge sort MPC algorithm in 40 LoC in `examples/msort-dedup.psl` in the provided supplemental material. We also implement an in-place quicksort MPC algorithm in 140 LoC in `examples/quick-sort.psl`.

Karmarkar. Karmarkar's algorithm solves linear programming problems, using matrices and linear algebra. A natural context in which linear programming problems would need to be executed securely is the following: a set of parties each holds a linear constraint, which is private, potentially because it models some private information about the party's conditions of satisfaction for some outcome. An additional party holds a private objective function that models their desires for an outcome, and an additional output party is designated to receive the output. The parties that contribute constraints and the objective function agree to perform the computation itself, but when the computation completes, the designated output party should learn the solution of the linear programming problem defined by the various parties' constraints and objective function without learning any additional information about the constraints or function itself. No party that contributes a constraint or the objective function should learn any information about the other parties' constraints or function.

Karmarkar's algorithm is an iterative, interior point algorithm that maintains a point in the solution space of a given linear programming problem; in each iteration, it updates the maintained point by performing matrix and vector arithmetic on matrices defined by the point and the input constraints and objective function. In our implementation of Karmarkar's algorithm for the security setting described above (contained in `examples/karmarkar.psl`), we extended the core functionality so that each party initially reads its inputs from local storage, shares the inputs with other parties who will perform the computation, cooperates to perform the computation over the shares, and then distributes its share of the result to the output party.

Karmarkar illustrates several of our language's more powerful features for programming MPC. Computations are performed over complex data structures that contain shares, and computation is performed using complex control flow, namely loops and multiplexes on private data. Although arithmetic operations on shares are performed throughout the algorithm, the security-specific code is relegated to relatively small setup and "tear-down" functions, executed at the beginning and end of computation. Arithmetic operations themseles are syntactically identical to core functionality, using a simple semantics for overloading operators that is not presented in the paper.

Mixed-mode Median. In this example, two parties both hold a private list and wish to securely compute the median element of the concatenated list. As discussed in [Rastogi et al. 2014], secure median can be efficiently computed via mixed-mode computation. In particular, it turns out that revealing intermediate comparisons between elements of the players respective sorted lists is secure. Thus, we avoid the expensive process of computing median completely under encryption by performing most of the work in cleartext. The benchmark demonstrates λ-Symphony's increased expressiveness as compared to Wysteria: the Wysteria authors wrote the mixed-mode median benchmark for lists of fixed constant length because second class shares meant that a recursive solution was impractical. In λ-Symphony, the recursive solution is relatively trivial. The program is an efficient, divide-and-conquer mixed-mode algorithm that computes the median of two lists of arbitrary (with known upper bound) length. We implement this algorithm in 73 LoC in `examples/two-party-median.psl`

B SINGLE-THREADED SEMANTICS: PROOFS OF META-THEORETIC PROPERTIES

Lemma B.1 (Lemma B1). *If* $\Gamma(x) = \tau$ *and* $\Sigma, \Gamma \vdash \gamma$ *then* $\gamma(x)$ *is defined.*

Proof.

By $\Gamma(x) = \tau$, $x \in dom(\Gamma)$.
By $\Sigma, \Gamma \vdash \gamma$, $\Sigma \vdash \gamma(x) : \Gamma(x)$ so $\gamma(x)$ is defined.

□

Lemma B.2 (Lemma B2). *If* $\Sigma \vdash v : \tau$, $\tau <: \tau'$, *and* $\vdash_m \tau'$, *then* $\Sigma \vdash v\downarrow_m : \tau'$.

Proof. Proof by induction on derivation of $\tau <: \tau'$. □

Lemma B.3 (Lemma B3). *If* $\Gamma \vdash_m x : \tau$ *and* $\Sigma, \Gamma \vdash \gamma$, *then* $\Sigma \vdash \gamma(x)\downarrow_m : \tau$.

Proof. By inversion on $\Gamma \vdash_m x : \tau$:

$$\boxed{\dfrac{\Gamma(x) = \tau' \quad \vdash_m \tau \quad \tau' <: \tau}{\Gamma \vdash_m x : \tau}\ \text{T-Var}}$$

By $\Sigma, \Gamma \vdash \gamma$ and $\Gamma(x) = \tau'$ we know $\Sigma \vdash \gamma(x) : \tau'$.
By lemma B2, $\Sigma \vdash \gamma(x) : \tau'$, $\tau' <: \tau$, and $\vdash_m \tau$, $\Sigma \vdash \gamma(x)\downarrow_m : \tau$. □

Lemma B.4 (Lemma B4). *If* $\vdash_m (\mathtt{ref}^{\omega}\ \tau)@m$, $\Sigma \vdash \ell^{\omega}@m : (\mathtt{ref}^{\omega}\ \tau)@m$ *and* $\Sigma \vdash \delta$, *then* $\Sigma \vdash \delta(\ell)\downarrow_m : \tau$.

Proof. By inversion on $\Sigma \vdash \ell^{\omega}@m : (\mathtt{ref}^{\omega}\ \tau)@m$ we know $\tau = \Sigma(\ell)$
By $\Sigma \vdash \delta$ and $\tau = \Sigma(\ell)$ we know $\Sigma \vdash \delta(\ell) : \tau$.
By inversion on $\vdash_m (\mathtt{ref}^{\omega}\ \tau)@m$ we know $\vdash_m \tau$.
By Lemma B2, $\Sigma \vdash \delta(\ell) : \tau$, $\tau <: \tau$ and $\vdash_m \tau$ we know $\Sigma \vdash \delta(\ell)\downarrow_m : \tau$

□

Lemma B.5 (Lemma B5). *If* $\vdash_m \tau'$ *and* $\{\alpha \mapsto m\} \vdash_m \tau$ *then* $\vdash_m [\tau'/\alpha]\tau$.

Proof. By induction on $\{\alpha \mapsto m\} \vdash_m \tau$. □

Lemma B.6 (Lemma B6). *If* $\vdash_m \tau$ *then* $\vdash_{m \cup p} \tau$.

Proof. By induction on the derivation. □

Lemma B.7 (Progress - Atomic Expressions).

If : $\Gamma \vdash_m a : \tau$	(atomic expression a is well-typed)
And : $\Sigma, \Gamma \vdash \gamma$	(environment γ is well-typed)
And : $\Sigma \vdash \delta$	(store δ is well-typed)
And : $m \neq \varnothing$	(party set m is non-empty)
Then : $\gamma \vdash_m \delta, a \hookrightarrow \delta', v$	(the configuration takes a step)
For some : δ' *and* v	(updated store δ' and result value v)

Proof.
By case analysis on e.

- **Case:** $a = i$
  Let $\delta' = \delta$ and $v = i@m$.
  Need to show: $\gamma \vdash_m \delta, i \hookrightarrow \delta, i@m$.
  Via S-Int.
- **Case:** $a = x$

By inversion on the typing derivation $\Gamma \vdash_m a : \tau$:

T-Var
$\Gamma(x) = \tau'$ *(H1)*
$\tau' <: \tau$ *(H2)*
$\vdash_m \tau$ *(H3)*
$\Gamma \vdash_m x : \tau$

From Lemma B1, H1, and $\Sigma, \Gamma \vdash \gamma$ we know that $\gamma(x)$ is defined.
Let $\delta' = \delta$ and $v = \gamma(x)\downarrow_m$.
Need to show: $\gamma \vdash_m \delta, x \hookrightarrow \delta, \gamma(x)\downarrow_m$.
Via ST-Var

- **Case:** $a = x_1 \odot x_2$

By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

T-Binop
$\Gamma \vdash_m x_1 : \mathtt{int}^{\psi}@m$ *(H1)*
$\Gamma \vdash_m x_2 : \mathtt{int}^{\psi}@m$ *(H2)* $\quad \vdash_m \psi$ *(H3)*
$\Gamma \vdash_m x_1 \odot x_2 : \mathtt{int}^{\psi}@m$

We learn: $\tau = \mathtt{int}^{\psi}@m$.
By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x_1)\downarrow_m : \mathtt{int}^{\psi}@m$ so $\gamma(x_1)\downarrow_m = i_1^{\psi}@m$ for some i_1, and likewise for $\gamma(x_2)\downarrow_m = i_2^{\psi}@m$ for some i_2.
Let $\delta' = \delta$ and $v = [\![\odot]\!](i_1, i_2)@m$
Need to show: $\gamma \vdash_m \delta, x_1 \odot x_2 \hookrightarrow \delta, [\![\odot]\!](i_1, i_2)@m$.
Via ST-Binop with hypotheses demonstrated above and H3.

- **Case:** $a = x_1 \ ? \ x_2 \diamond x_3$

(Analogous to previous case…)
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

T-Mux
$\vdash_m x_1 : \mathtt{int}^{\psi}@m$ *(H1)*
$\vdash_m x_2 : \mathtt{int}^{\psi}@m$ *(H2)*
$\vdash_m x_3 : \mathtt{int}^{\psi}@m$ *(H3)* $\quad \vdash_m \psi$ *(H4)*
$\Gamma \vdash_m x_1 \ ? \ x_2 \diamond x_3 : \mathtt{int}^{\psi}@m$

We learn: $\tau = \mathtt{int}^{\psi}@m$.
By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x_1)\downarrow_m : \mathtt{int}^{\psi}@m$ so $\gamma(x_1)\downarrow_m = i_1^{\psi}@m$ for some i_1, and likewise for $\gamma(x_2)\downarrow_m = i_2^{\psi}@m$ for some i_2 and for $\gamma(x_3)\downarrow_m = i_3^{\psi}@m$ for some i_3.
Let $\delta' = \delta$ and $v = \mathrm{cond}(i_1, i_2, i_3)^{\psi}@m$.
Need to show: $\gamma \vdash_m \delta, x_1 \ ? \ x_2 \diamond x_3 \hookrightarrow \delta, \mathrm{cond}(i_1, i_2, i_3)^{\psi}@m$.
Via ST-Mux with hypotheses demonstrated above and H4.

- **Case:** $a = \iota_i \ x$

We just show the case for $\iota_i = \iota_1$; the case for ι_2 is analogous.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

T-Inj
$\Gamma \vdash_m x : \tau_1$ *(H1)*
$\vdash_m \tau_2$ *(H2)*
$\Gamma \vdash_m \iota_1 \ x : (\tau_1 + \tau_2)@m$

We learn: $\tau = (\tau_1 + \tau_2)@m$.

From Lemma B1, H1, and $\Sigma, \Gamma \vdash \gamma$ we know that $\gamma(x)$ is defined.
Let $\delta' = \delta$ and $v = \iota_1\ \gamma(x)\downarrow_m$
Need to show: $\gamma \vdash_m \delta, \iota_1\ x \hookrightarrow \delta, (\iota_1\ \gamma(x)\downarrow_m)@m$.
Via ST-Inj.

- **Case:** $a = \langle x_1, x_2 \rangle$
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\frac{\Gamma \vdash_m x_1 : \tau_1 \quad \textit{(H1)} \qquad \Gamma \vdash_m x_2 : \tau_2 \quad \textit{(H2)}}{\Gamma \vdash_m \langle x_1, x_2 \rangle : \tau_1 \times \tau_2}\ \textsc{T-Pair}$$

We learn: $\tau = \tau_1 \times \tau_2$.
From H1 and $\Sigma, \Gamma \vdash \gamma$ we know $\gamma(x_1)$ is defined, and likewise for H2 and $\gamma(x_2)$.
Let $\delta' = \delta$ and $v = \langle \gamma(x_1)\downarrow_m, \gamma(x_2)\downarrow_m \rangle$.
Need to show: $\gamma \vdash_m \delta, \langle x_1, x_2 \rangle \hookrightarrow \delta, \langle \gamma(x_1)\downarrow_m, \gamma(x_2)\downarrow_m \rangle$.
By ST-Pair with hypotheses shown above.

- **Case:** $a = \pi_i\ x$
We just show the case for $\pi_i = \pi_1$; the case for π_2 is analogous.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\frac{\Gamma \vdash_m x : \tau_1 \times \tau_2 \textit{(H1)}}{\Gamma \vdash_m \pi_1\ x : \tau_1}\ \textsc{T-Proj}$$

We learn that $\tau = \tau_1$.
By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : \tau_1 \times \tau_2$ so either $\gamma(x)\downarrow_m = \langle v_1, v_2 \rangle$ for some v_1 and v_2 or $\gamma(x)\downarrow_m = \bigstar$.
  - **Case:** $\gamma(x)\downarrow_m = \langle v_1, v_2 \rangle$
    Let $\delta' = \delta$ and $v = v_1$
    Need to show: $\gamma \vdash_m \delta, \pi_1\ x \hookrightarrow \delta, v_1$.
    By ST-Proj with hypotheses shown above.
  - **Case:** $\gamma(x)\downarrow_m = \bigstar$
    Let $\delta' = \delta$ and $v = \bigstar$
    Need to show: $\gamma \vdash_m \delta, \pi_1\ x \hookrightarrow \delta, \bigstar$.
    By ST-Star with hypotheses shown above.

- **Case:** $a = \lambda_z x.\ e$
Let $\delta' = \delta$ and $v = \langle \lambda_z x.\ e, \gamma \rangle @m$
Need to show: $\gamma \vdash_m \delta, \lambda_z x.\ e \hookrightarrow \delta, \langle \lambda_z x.\ e, \gamma \rangle @m$.
By ST-Fun with hypotheses shown above.

- **Case:** $a = \texttt{ref}\ x$
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\frac{\Gamma \vdash_m x : \tau' \quad \textit{(H1)}}{\Gamma \vdash_m \texttt{ref}\ x : (\texttt{ref}^{\mathsf{RW\#}m}\ \tau')@m}\ \textsc{T-Ref}$$

We learn: $\tau = (\texttt{ref}^{\mathsf{RW\#}m}\ \tau')@m$.
From Lemma B1, H1, and $\Sigma, \Gamma \vdash \gamma$ we know that $\gamma(x)$ is defined.
Let $\delta' = \{\ell \mapsto \gamma(x)\downarrow_m\} \uplus \delta$ and $v = \ell^{\mathsf{RW\#}m}@m$ for some $\ell \notin \mathrm{dom}(\delta)$.
Need to show: $\gamma \vdash_m \delta, \texttt{ref}\ x \hookrightarrow \{\ell \mapsto \gamma(x)\downarrow_m\} \uplus \delta, \ell^{\mathsf{RW\#}m}@m$.
Via ST-Ref.

- **Case:** $a = !x$
  By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

  T-Deref
  $$\frac{\Gamma \vdash_m x : (\texttt{ref}^{\texttt{RO}}\ \tau)@m \quad \textit{(H1)}}{\Gamma \vdash_m\ !x : \tau}$$

  By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : (\texttt{ref}^{\texttt{RO}}\ \tau)@m$ so $\gamma(x)\downarrow_m = \ell^{\texttt{RO}}@m$ for some ℓ in the domain of δ.
  Let $\delta' = \delta$ and $v = \delta(\ell)\downarrow_m$
  Need to show: $\gamma \vdash_m \delta, !x \hookrightarrow \delta, \delta(\ell)\downarrow_m$.
  By ST-Deref with hypotheses shown above.
- **Case:** $a = x_1 := x_2$
  By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

  T-Assign
  $$\frac{\Gamma \vdash_m x_1 : (\texttt{ref}^{\texttt{RW\#}m}\ \tau)@m \quad \textit{(H1)} \qquad \Gamma \vdash_m x_2 : \tau \quad \textit{(H2)}}{\Gamma \vdash_m x_1 := x_2 : \tau}$$

  By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x_1)\downarrow_m : (\texttt{ref}^{\texttt{RW\#}m}\ \tau)@m$ so $\gamma(x_1)\downarrow_m = \ell^{\texttt{RW\#}m}@m$ for some ℓ.
  By Lemma B3 and H2, we learn that $\Sigma \vdash \gamma(x_2)\downarrow_m : \tau$ so $\gamma(x_2)\downarrow_m = v$ for some $\Sigma \vdash v : \tau$.
  Let $\delta' = \delta[\ell \mapsto v]$ and $v = \gamma(x_2)\downarrow_m$.
  Need to show: $\gamma \vdash_m \delta, x_1 := x_2 \hookrightarrow \delta[\ell \mapsto v], v$.
  By ST-Assign with hypotheses shown above.
- **Case:** $a = \texttt{fold}\ x$
  By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

  T-Fold
  $$\frac{\Gamma \vdash_m x : [(\mu\alpha.\ \tau_0)/\alpha]\tau_0 \textit{(H1)}}{\Gamma \vdash_m \texttt{fold}\ x : \mu\alpha.\ \tau_0}$$

  We learn that $\tau = \mu\alpha.\ \tau_0$
  By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : [(\mu\alpha.\ \tau_0)/\alpha]\tau_0$ so $\gamma(x)\downarrow_m = v$ for some $\Sigma \vdash v : [(\mu\alpha.\ \tau_0)/\alpha]\tau_0$.
  let $\delta' = \delta$
  let $v = \gamma(x)\downarrow_m$
  Need to show: $\gamma \vdash_m \delta, \texttt{fold}\ x \hookrightarrow \delta, \texttt{fold}\ \gamma(x)\downarrow_m$.
  By ST-Fold with hypotheses shown above.
- **Case:** $a = \texttt{unfold}\ x$
  By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

  T-Unfold
  $$\frac{\Gamma \vdash_m x : \mu\alpha.\ \tau_0 \textit{(H1)}}{\Gamma \vdash_m \texttt{unfold}\ x : [(\mu\alpha.\ \tau_0)/\alpha]\tau_0}$$

  We learn that $\tau = [(\mu\alpha.\ \tau_0)/\alpha]\tau_0$.
  By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : \mu\alpha.\ \tau_0$ so either $\gamma(x)\downarrow_m = \texttt{fold}\ v$ for some v or $\gamma(x)\downarrow_m = \bigstar$.
  - **Case:** $\gamma(x)\downarrow_m = \texttt{fold}\ v$
    let $\delta' = \delta$
    let $v = v$
    Need to show: $\gamma \vdash_m \delta, \texttt{unfold}\ x \hookrightarrow \delta, v$.

By ST-Unfold with hypotheses shown above.

- **Case:** $\gamma(x)\downarrow_m = \bigstar$

  let $\delta' = \delta$

  let $v = v$

  Need to show: $\gamma \vdash_m \delta, \texttt{unfold}\ x \hookrightarrow \delta, \bigstar$.

  By ST-Star with hypotheses shown above.

- **Case:** $a = \texttt{read}$

  By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

  $$\frac{\text{T-Read}\quad |m| = 1 \textit{(H1)}}{\Gamma \vdash_m \texttt{read} : \texttt{int@}m}$$

  let $\delta' = \delta$

  let $v = i@m$ for some i.

  Need to show: $\gamma \vdash_m \delta, \texttt{read} \hookrightarrow \delta, i@m$.

  By ST-Read with hypotheses shown above.

- **Case:** $a = \texttt{write}$

  By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

  $$\frac{\text{T-Write}\quad \Gamma \vdash_m x : \texttt{int@}m \textit{(H1)} \qquad |m| = 1 \textit{(H2)}}{\Gamma \vdash_m \texttt{write}\ x : \texttt{int@}m}$$

  We learn that $\tau = \texttt{int@}m$.

  By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : \texttt{int@}m$ so $\gamma(x)\downarrow_m = i@m$ for some i.

  let $\delta' = \delta$

  let $v = i@m$

  Need to show: $\gamma \vdash_m \delta, \texttt{write}\ x \hookrightarrow \delta, i@m$.

  By ST-Write with hypotheses shown above.

- **Case:** $a = \texttt{embed}\ x$

  By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

  $$\frac{\text{T-Embed}\quad \Gamma \vdash_m x : \texttt{int@}m \textit{(H1)}}{\Gamma \vdash_m \texttt{embed}[p]\ x : \texttt{int}^{\texttt{enc\#}p}@m}$$

  We learn that $\tau = \texttt{int}^{\texttt{enc\#}p}@m$.

  By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : \texttt{int@}m$ so $\gamma(x)\downarrow_m = i@m$ for some i.

  let $\delta' = \delta$

  let $v = i^{\texttt{enc\#}m}@m$

  Need to show: $\gamma \vdash_m \delta, \texttt{write}\ x \hookrightarrow \delta, i^{\texttt{enc\#}p}@m$.

  By ST-Embed with hypotheses shown above.

- **Case:** $a = \texttt{share}[p \to q]\ x$

  By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

  $$\frac{\text{T-Share}\quad \Gamma \vdash_m x : \texttt{int@}p' \textit{(H1)} \qquad |p| = 1 \textit{(H2)} \qquad q \neq \varnothing \textit{(H3)} \qquad p \subseteq p' \textit{(H4)} \qquad p \cup q = m \textit{(H5)}}{\Gamma \vdash_m \texttt{share}[p \to q]\ x : \texttt{int}^{\texttt{enc\#}q}@q}$$

  We learn that $\tau = \texttt{int}^{\texttt{enc\#}q}@q$.

  By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : \texttt{int@}p'$ so $\gamma(x)\downarrow_m = i@p'$ for some i.

let $\delta' = \delta$
let $v = i^{\mathsf{enc\#}q}@q$.
Need to show: $\gamma \vdash_m \delta, \mathtt{share}[p \to q]\ x \hookrightarrow \delta, i^{\mathsf{enc\#}q}@q$.
By ST-Share with hypotheses shown above.

- **Case:** $a = \mathtt{reveal}[q]\ x$
  By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\frac{\Gamma \vdash_m x : \mathtt{int}^{\mathsf{enc\#}p}@p \text{(H1)} \qquad q \neq \varnothing \text{(H2)} \qquad p \cup q = m \text{(H3)}}{\Gamma \vdash_m \mathtt{reveal}[q]\ x : \mathtt{int}@q}\ \text{T-Reveal}$$

  We learn that $\tau = \mathtt{int}@q$
  By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : \mathtt{int}^{\mathsf{enc\#}p}@p$ so $\gamma(x)\downarrow_m = i^{\mathsf{enc\#}p}@p$ for some i.
  let $\delta' = \delta$
  let $v = i@q$
  Need to show: $\gamma \vdash_m \delta, \mathtt{reveal}[p]\ x \hookrightarrow \delta, i@q$.
  By ST-Reveal with hypotheses shown above.

□

Lemma B.8 (Progress - Configurations).

If : $\Gamma \vdash_m a : \tau$
And : $\Sigma, \Gamma \vdash \gamma$
And : $\Sigma \vdash \delta$
And : $m \neq \varnothing$
Then Either : $m, \gamma, \delta, \kappa, e \longrightarrow m', \gamma', \delta', \kappa', e'$
For some : $m', \gamma', \delta', \kappa'$ *and* e'
Or : $e = a$ *and* $\kappa = \top$

Proof.
By case analysis on e.

- **Case:** $e = \mathtt{case}\ x\ \{x.\ e_1\}\ \{x.\ e_2\}$
  By inversion on typing derivation $\Gamma \vdash_m e : \tau$:

$$\frac{\begin{array}{r}\Gamma \vdash_m x\ : (\tau_1 + \tau_2)@m \text{(H1)} \\ \{x' \mapsto \tau_1\} \uplus \Gamma \vdash_m e_1 : \tau \text{(H2)} \\ \{x' \mapsto \tau_2\} \uplus \Gamma \vdash_m e_2 : \tau \text{(H3)}\end{array}}{\Gamma \vdash_m \mathtt{case}\ x\ \{x'.\ e_1\}\ \{x'.\ e_2\} : \tau}\ \text{T-Case}$$

  By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : (\tau_1 + \tau_2)@m$ so $\gamma(x)\downarrow_m = (\iota_1\ v)@m$ for some v.
  let $m' = m$
  let $\gamma' = \{x \mapsto v\} \uplus \gamma$
  let $\delta' = \delta$
  let $\kappa' = \kappa$
  let $e' = e_i$
  Need to show: $m, \gamma, \delta, \kappa, \mathtt{case}\ x\ \{x.\ e_1\}\ \{x.\ e_2\} \longrightarrow m, \{x \mapsto v\} \uplus \gamma, \delta, \kappa, e_i$.
  By ST-Case with hypotheses shown above.
- **Case:** $e = \mathtt{par}[p]\ e$
  - **Case:** $|m \cap p| = 0$
    let $m' = m$
    let $\gamma' = \{x \mapsto \bigstar\} \uplus \gamma$
    let $\delta' = \delta$

let $\kappa' = \kappa$
let $e' = x$
Need to show: $m, \gamma, \delta, \kappa, \texttt{par}[p]\ e \longrightarrow m, \{x \mapsto \bigstar\} \uplus \gamma, \delta, \kappa, x$.
By ST-ParEmpty with hypotheses shown above.

- **Case:** $|m \cap p| \neq 0$
  let $m' = m$
  let $\gamma' = \gamma$
  let $\delta' = \delta$
  let $\kappa' = \kappa$
  let $e' = x$
  Need to show: $m, \gamma, \delta, \kappa, \texttt{par}[p]\ e \longrightarrow m \cap p, \gamma, \delta, \kappa, e$.
  By ST-Par with hypotheses shown above.

- **Case:** $e = x_1\ x_2$
  By inversion on typing derivation $\Gamma \vdash_m e : \tau$:

$\Gamma \vdash_m x_1 : (\tau_1 \xrightarrow{m} \tau_2)@m$ *(H1)*	
$]\Gamma \vdash_m x_2 : \tau_1$ *(H2)*	$\Gamma \vdash_m x_1\ x_2 : \tau_2$

  We learn that $\tau = \tau_2$.
  By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x_1)\swarrow_m : (\tau_1 \xrightarrow{m} \tau_2)@m$ so $\gamma(x)\swarrow_m = \langle \lambda_z x.\ e_0, \gamma_0 \rangle @m$ for some e_0 and γ_0.
  From Lemma B1, H2, and $\Sigma, \Gamma \vdash \gamma$ we know $\gamma(x_2)$ is defined.
  let $m' = m$
  let $\gamma' = \{z \mapsto v_1, x \mapsto v_2\} \uplus \gamma_0$
  let $\delta' = \delta$
  let $\kappa' = \kappa$
  let $e' = e_0$
  Need to show: $m, \gamma, \delta, \kappa, x_1\ x_2 \longrightarrow m, \{z \mapsto v_1, x \mapsto v_2\} \uplus \gamma_0, \delta, \kappa, e_0$.
  By ST-App with hypotheses shown above.

- **Case:** $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$
  let $m' = m$
  let $\gamma' = \gamma$
  let $\delta' = \delta$
  let $\kappa' = (\langle \texttt{let}\ x = \square\ \texttt{in}\ e_2 \mid m, \gamma \rangle :: \kappa)$
  let $e' = e_1$
  Need to show: $m, \gamma, \delta, \kappa, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow m, \gamma, \delta, (\langle \texttt{let}\ x = \square\ \texttt{in}\ e_2 \mid m, \gamma \rangle :: \kappa), e_1$.
  By ST-LetPush.

- **Case:** $e = a$
  - **Case:** $\kappa = \top$
    Need to show that $e = a$ and $\kappa = \top$.
    By assumption.
  - **Case:** $\kappa = (\langle \texttt{let}\ x = \square\ \texttt{in}\ e_0 \mid m', \gamma_0 \rangle :: \kappa_0)$
    By In-Frame Progress and the assumptions above, we know that $\gamma \vdash_m \delta, a \hookrightarrow \delta', v$.
    let $m' = m'$
    let $\gamma' = \{x \mapsto v\} \uplus \gamma_0$
    let $\delta' = \delta'$
    let $\kappa' = \kappa_0$
    let $e' = e_0$
    Need to show: $m, \gamma, \delta, (\langle \texttt{let}\ x = \square\ \texttt{in}\ e_0 \mid m', \gamma_0 \rangle :: \kappa_0), a \longrightarrow m', \{x \mapsto v\} \uplus \gamma_0, \delta', \kappa_0, e_0$.

By ST-LetPop with hypotheses shown above.

□

Lemma B.9 (Preservation - Atomic Expressions).

If : $\Gamma \vdash_m a : \tau$
And : $\Sigma, \Gamma \vdash \gamma$
And : $\Sigma \vdash \delta$
And : $\gamma \vdash_m \delta, a \hookrightarrow \delta', v$
Then : $\Sigma' \vdash v : \tau$
And : $\Sigma' \vdash \delta'$
For some : Σ'

Proof.
By cases analysis on the step derivation $\gamma \vdash_m \delta, a \hookrightarrow \delta', v$.

- **Case:**

$$\text{ST-Var}\ \frac{v = \gamma(x)\downarrow_m}{\gamma \vdash_m \delta, x \hookrightarrow \delta, v}$$

We learn: $a = x$, $v = \gamma(x)\downarrow_m$, and $\delta' = \delta$.
By Lemma B3 and $\Gamma \vdash_m x : \tau$, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : \tau$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash \gamma(x)\downarrow_m : \tau$
(2) $\Sigma \vdash \delta$
(1) By $\Sigma \vdash \gamma(x)\downarrow_m : \tau$.
(2) Immediate by assumption $\Sigma \vdash \delta$.

- **Case:**

$$\text{ST-Int}\ \frac{}{\gamma \vdash_m \delta, i \hookrightarrow \delta, i@m}$$

We learn: $a = i$, $v = i@m$, and $\delta' = \delta$.
By the typing judgement T-Int, we learn: $\tau = \texttt{int}@m$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash i@m : \texttt{int}@m$
(2) $\Sigma \vdash \delta$
(1) By the following typing derivation:

$$\text{TV-Loc}\ \frac{\text{TV-Int}\ \dfrac{}{\Sigma \vdash i^{\psi} : \texttt{int}^{\psi}}}{\Sigma \vdash i^{\psi}@m : \texttt{int}^{\psi}@m}$$

(2) Immediate by assumption $\Sigma \vdash \delta$.

- **Case:**

$$\text{ST-Binop}\ \frac{i_1^{\psi}@m = \gamma(x_1)\downarrow_m (H1) \qquad i_2^{\psi}@m = \gamma(x_2)\downarrow_m (H2) \qquad \vdash_m \psi (H3)}{\gamma \vdash_m \delta, x_1 \odot x_2 \hookrightarrow \delta, [\![\odot]\!](i_1, i_2)^{\psi}@m}$$

We learn: $a = x_1 \odot x_2$, $v = [\![\odot]\!](i_1, i_2)^{\psi}@m$, and $\delta' = \delta$.
By the typing judgement T-Binop, we learn: $\tau = \texttt{int}^{\psi}@m$.

Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash [\![\odot]\!](i_1, i_2)^{\psi}@m : \mathtt{int}^{\psi}@m$
(2) $\Sigma \vdash \delta$
(1) By the following typing derivation:

$$\text{TV-Loc}\ \frac{\text{TV-Int}\ \dfrac{}{\Sigma \vdash i^{\psi} : \mathtt{int}^{\psi}}}{\Sigma \vdash i^{\psi}@m : \mathtt{int}^{\psi}@m}$$

(2) Immediate by assumption $\Sigma \vdash \delta$.

- **Case:**

$$\text{ST-Mux}\ \frac{i_1^{\psi}@m = \gamma(x_1)\swarrow_m (H1) \qquad i_2^{\psi}@m = \gamma(x_2)\swarrow_m (H2) \qquad i_3^{\psi}@m = \gamma(x_3)\swarrow_m (H3) \qquad \vdash_m \psi (H4)}{\gamma \vdash_m \delta, x_1\ ?\ x_2 \diamond x_3 \hookrightarrow \delta, \mathrm{cond}(i_1, i_2, i_3)^{\psi}@m}$$

We learn: $a = x_1\ ?\ x_2 \diamond x_3$, $v' = \mathrm{cond}(i_1, i_2, i_3)^{\psi}@m$, and $\delta' = \delta$.
By the typing judgement T-Mux, we learn: $\tau = \mathtt{int}^{\psi}@m$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash \mathrm{cond}(i_1, i_2, i_3)^{\psi}@m : \mathtt{int}^{\psi}@m$
(2) $\Sigma \vdash \delta$
(1) By the following typing derivation:

$$\text{TV-Loc}\ \frac{\text{TV-Int}\ \dfrac{}{\Sigma \vdash i^{\psi} : \mathtt{int}^{\psi}}}{\Sigma \vdash i^{\psi}@m : \mathtt{int}^{\psi}@m}$$

(2) Immediate by assumption $\Sigma \vdash_m \delta$.

- **Case:**

$$\text{ST-Inj}\ \frac{v = \gamma(x)\swarrow_m}{\gamma \vdash_m \delta, \iota_1\ x \hookrightarrow \delta, (\iota_1\ v)@m}$$

We learn: $a = (\iota_1\ x)$, $v = (\iota_i\ \gamma(x)\swarrow_m)@m$, and $\delta' = \delta$.
We just show the case for $\iota_i = \iota_1$; the case for ι_2 is analogous.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\text{T-Inj}\ \frac{\Gamma \vdash_m x : \tau_0 (H1) \qquad \vdash_m \tau' (H2)}{\Gamma \vdash_m \iota_1\ x : (\tau_0 + \tau')@m}$$

We learn: $\tau = (\tau_0 + \tau')@m$.
By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\swarrow_m : \tau_0$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash (\iota_i\ \gamma(x)\swarrow_m)@m : (\tau_0 + \tau')@m$
(2) $\Sigma \vdash \delta$
(1) By the following typing derivation:

$$\textsc{TV-Loc}\ \frac{\textsc{TV-Inj}\ \dfrac{\dfrac{}{\Sigma \vdash \gamma(x)\swarrow_m : \tau_0}\qquad \dfrac{}{\vdash \tau'}\ \textsc{H2 and Lemma B6}}{\Sigma \vdash (\iota_1\ \gamma(x)\swarrow_m) : (\tau_0 + \tau')}}{\Sigma \vdash (\iota_1\ \gamma(x)\swarrow_m)@m : (\tau_0 + \tau')@m}$$

(2) Immediate by assumption $\Sigma \vdash_m \delta$.

- **Case:**

$$\textsc{ST-Pair}\ \frac{v_1 = \gamma(x_1)\swarrow_m \qquad v_2 = \gamma(x_2)\swarrow_m}{\gamma \vdash_m \delta, \langle x_1, x_2\rangle \hookrightarrow \delta, \langle v_1, v_2\rangle}$$

We learn: $e = \langle x_1, x_2\rangle$, $v = \langle \gamma(x_1)\swarrow_m, \gamma(x_2)\swarrow_m\rangle@m$, and $\delta' = \delta$.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\textsc{T-Pair}\ \frac{\Gamma \vdash_m x_1 : \tau_1 \textit{(H1)} \quad \Gamma \vdash_m x_2 : \tau_2 \textit{(H2)}}{\Gamma \vdash_m \langle x_1, x_2\rangle : \tau_1 \times \tau_2}$$

We learn: $\tau = \tau_1 \times \tau_2$
By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x_1)\swarrow_m : \tau_1$, and likewise for H2 and $\Sigma \vdash \gamma(x_2)\swarrow_m : \tau_2$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash \langle \gamma(x_1)\swarrow_m, \gamma(x_2)\swarrow_m\rangle : (\tau_1 \times \tau_2)$
(2) $\Sigma \vdash \delta$
(1) By the following typing derivation:

$$\textsc{TV-Pair}\ \frac{\dfrac{}{\Sigma \vdash \gamma(x_1)\swarrow_m : \tau_1}\qquad \dfrac{}{\Sigma \vdash \gamma(x_2)\swarrow_m : \tau_2}}{\Sigma \vdash \langle \gamma(x_2)\swarrow_m, \gamma(x_2)\swarrow_m\rangle : (\tau_1 \times \tau_2)}$$

(2) Immediate by assumption $\Sigma \vdash_m \delta$.

- **Case:**

$$\textsc{ST-Proj}\ \frac{\langle v_1, v_2\rangle = \gamma(x)\swarrow_m}{\gamma \vdash_m \delta, \pi_1\ x \hookrightarrow \delta, v_1}$$

We learn: $a = \pi_1\ x$, $v = v_1$, and $\delta' = \delta$.
We just show the case for $\pi_i = \pi_1$; the case for π_2 is analogous.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\textsc{T-Proj}\ \frac{\Gamma \vdash_m x : \tau_1 \times \tau_2 \textit{(H1)}}{\Gamma \vdash_m \pi_1\ x : \tau_1}$$

We learn: $\tau = \tau_1$.
By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x_1)\swarrow_m : \tau_1 \times \tau_2$, so $\Sigma \vdash \langle v_1, v_2\rangle : \tau_1 \times \tau_2$.
We learn the types of v_1 and v_2 by inversion on the typing derivation:

$$\textsc{TV-Pair}\ \frac{\Sigma \vdash v_1 : \tau_1 \quad \textit{(H2)} \qquad \Sigma \vdash v_2 : \tau_2 \quad \textit{(H3)}}{\Sigma \vdash \langle v_1, v_2\rangle : \tau_1 \times \tau_2}$$

Let $\Sigma' = \Sigma$.
Need to show:

(1) $\Sigma \vdash v_1 : \tau_1$
(2) $\Sigma \vdash \delta$
(1) By H2.
(2) Immediate by assumption $\Sigma \vdash_m \delta$.

- **Case:**

$$\textsc{ST-Fun}\ \frac{}{\gamma \vdash_m \delta, \lambda_z x.\ e_0 \hookrightarrow \delta, \langle \lambda_z x.\ e_0, \gamma \rangle @m}$$

We learn: $a = \lambda_z x.\ e$, $v = \langle \lambda_z x.\ e, \gamma \rangle @m$, and $\delta' = \delta$.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\textsc{T-Fun}\ \frac{\{z \mapsto (\tau_1\ {}^{m}\!\!\rightarrow \tau_2)@m, x \mapsto \tau_1\} \uplus \Gamma \vdash_m e_0 : \tau_2 \textit{(H1)}}{\Gamma \vdash_m \lambda_z x.\ e_0 : (\tau_1\ {}^{m}\!\!\rightarrow \tau_2)@m}$$

We learn: $\tau = (\tau_1\ {}^{m}\!\!\rightarrow \tau_2)@m$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash \langle \lambda_z x.\ e, \gamma \rangle @m : (\tau_1\ {}^{m}\!\!\rightarrow \tau_2)@m$
(2) $\Sigma \vdash \delta$
(1) By the following typing derivation:

$$\textsc{TV-Loc}\ \frac{\textsc{TV-Fun}\ \dfrac{\Sigma, \Gamma \vdash \gamma \qquad \textsc{H1}\ \dfrac{}{\{z \mapsto (\tau_1\ {}^{m}\!\!\rightarrow \tau_2)@m, x \mapsto \tau_1\} \uplus \Gamma \vdash_m e_0 : \tau_2}}{\Sigma \vdash \langle \lambda_z x.\ e_0, \gamma \rangle : (\tau_1\ {}^{m}\!\!\rightarrow \tau_2)}}{\Sigma \vdash \langle \lambda_z x.\ e_0, \gamma \rangle @m : (\tau_1\ {}^{m}\!\!\rightarrow \tau_2)@m}$$

(2) Immediate by assumption $\Sigma \vdash_m \delta$.

- **Case:**

$$\textsc{ST-Ref}\ \frac{v_0 = \gamma(x)\downarrow_m}{\gamma \vdash_m \delta, \texttt{ref}\ x \hookrightarrow \{\ell \mapsto v_0\} \uplus \delta, \ell^{\mathsf{RW\#}m}@m}$$

We learn: $a = \texttt{ref}\ x$, $v = \ell^{\mathsf{RW\#}m}@m$, and $\delta' = \{\ell \mapsto \gamma(x)\downarrow_m\} \uplus \delta$.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\textsc{T-Ref}\ \frac{\Gamma \vdash_m x : \tau_0 \textit{(H1)}}{\Gamma \vdash_m \texttt{ref}\ x : (\texttt{ref}^{\mathsf{RW\#}m}\ \tau_0)@m}$$

We learn: $\tau = (\texttt{ref}^{\mathsf{RW\#}m}\ \tau_0)@m$.
By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x_1)\downarrow_m : \tau_0$.
Let $\Sigma' = \{\ell \mapsto \tau_0\} \uplus \Sigma$.
Need to show:
(1) $\{\ell \mapsto \tau_0\} \uplus \Sigma \vdash \ell^{\mathsf{RW\#}m}@m : (\texttt{ref}^{\mathsf{RW\#}m}\ \tau_0)@m$
(2) $\{\ell \mapsto \tau_0\} \uplus \Sigma \vdash \{\ell \mapsto \gamma(x)\downarrow_m\} \uplus \delta$
(1) By the following typing derivation:

$$\textsc{TV-Loc}\ \frac{\textsc{TV-Ref}\ \dfrac{}{\{\ell \mapsto \tau_0\} \uplus \Sigma \vdash \ell^{\mathsf{RW\#}m} : \texttt{ref}^{\mathsf{RW\#}m}\ (\{\ell \mapsto \tau_0\} \uplus \Sigma)(\ell)}}{\{\ell \mapsto \tau_0\} \uplus \Sigma \vdash \ell^{\mathsf{RW\#}m}@m : (\texttt{ref}^{\mathsf{RW\#}m}\ (\{\ell \mapsto \tau_0\} \uplus \Sigma)(\ell)@m}$$

And $(\{\ell \mapsto \tau_0\} \uplus \Sigma)(\ell) = \tau_0$.
(2) By environment extension and $\Sigma \vdash \gamma(x)\downarrow_m : \tau_0$.

- **Case:**

$$\textsc{ST-Deref}\ \frac{\ell^{\omega}@m = \gamma(x)\downarrow_m}{\gamma \vdash_m \delta, !x \hookrightarrow \delta, \delta(\ell)\downarrow_m}$$

We learn: $a = !x$, $v = \delta(\ell)\downarrow_m$, and $\delta' = \delta$.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\textsc{T-Deref}\ \frac{\Gamma \vdash_m x : (\texttt{ref}^{\texttt{RO}}\ \tau)@m \textit{(H1)}}{\Gamma \vdash_m !x : \tau}$$

By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : (\texttt{ref}^{\texttt{RO}}\ \tau)@p$, so $\Sigma \vdash \ell^{\omega}@m : (\texttt{ref}^{\texttt{RO}}\ \tau)@m$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash \delta(\ell)\downarrow_m : \tau$
(2) $\Sigma \vdash \delta$
(1) By Lemma B4, $\vdash_m (\texttt{ref}^{RO}\ \tau)@m$, $\Sigma \vdash \ell^{\omega}@m : (\texttt{ref}^{\texttt{RO}}\ \tau)@m$, $\Sigma \vdash \delta$.
(2) Immediate by assumption $\Sigma \vdash_m \delta$.

- **Case:**

$$\textsc{ST-Assign}\ \frac{\ell^{\texttt{RW\#}m}@m = \gamma(x_1)\downarrow_m \qquad v = \gamma(x_2)\downarrow_m}{\gamma \vdash_m \delta, x_1 := x_2 \hookrightarrow \delta[\ell \mapsto v], v}$$

We learn: $a = x_1 := x_2$, $v = \gamma(x_2)\downarrow_m$, and $\delta' = \delta[\ell \mapsto v]$.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\textsc{T-Assign}\ \frac{\Gamma \vdash_m x_1 : (\texttt{ref}^{\texttt{RW\#}m}\ \tau)@m \textit{(H1)} \qquad \Gamma \vdash_m x_2 : \tau \textit{(H2)}}{\Gamma \vdash_m x_1 := x_2 : \tau}$$

By $\Sigma, \Gamma \vdash \gamma$ and Value Location WF, we learn that $\Sigma \vdash \gamma(x_2)\downarrow_m : \tau$.
By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x_1)\downarrow_m : (\texttt{ref}^{\texttt{RW}}\ \tau)@p$, so $\Sigma \vdash \ell^{\omega}@m : (\texttt{ref}^{\texttt{RW}}\ \tau)@m$.
By Lemma B4, $\vdash_m (\texttt{ref}^{RW}\ \tau)@m$, $\Sigma \vdash \ell^{\omega}@m : (\texttt{ref}^{\texttt{RW}}\ \tau)@m$, $\Sigma \vdash \delta$, we know that $\Sigma \vdash \delta(\ell)\downarrow_m : \tau$.
By Lemma B3 and H2, we learn that $\Sigma \vdash \gamma(x_2)\downarrow_m : \tau$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash \gamma(x_2)\downarrow_m : \tau$
(2) $\Sigma \vdash \delta[\ell \mapsto \gamma(x_2)\downarrow_m]$
(1) Immediate from the statements above.
(2) By $\Sigma \vdash \gamma(x_2)\downarrow_m : \tau$ and $\Sigma(\ell) = \tau$.

- **Case:**

$$\textsc{ST-Fold}\ \frac{v = \gamma(x)\downarrow_m}{\gamma \vdash_m \delta, \texttt{fold}\ x \hookrightarrow \delta, \texttt{fold}\ v}$$

We learn: $a = \texttt{fold}\ x$, $v = \texttt{fold}\ \gamma(x)\downarrow_m$, and $\delta' = \delta$.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\textsc{T-Fold}\ \frac{\Gamma \vdash_m x : [(\mu\alpha.\ \tau)/\alpha]\tau \textit{(H1)}}{\Gamma \vdash_m \texttt{fold}\ x : \mu\alpha.\ \tau}$$

We learn: $\tau = \mu\alpha.\ \tau$.
By Lemma B3 and H2, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : [(\mu\alpha.\ \tau)/\alpha]\tau$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash \texttt{fold}\ \gamma(x)\downarrow_m : \mu\alpha.\ \tau$
(2) $\Sigma \vdash \delta$
(1) By the following typing derivation:

$$\textsc{TV-Fold}\ \frac{\overline{\Sigma \vdash \gamma(x)\downarrow_m : [(\mu\alpha.\ \tau)/\alpha]\tau}}{\Sigma \vdash \texttt{fold}\ \gamma(x)\downarrow_m : \mu\alpha.\ \tau}$$

(2) Immediate by assumption $\Sigma \vdash \delta$.

- **Case:**

$$\textsc{ST-Unfold}\ \frac{\texttt{fold}\ v = \gamma(x)\downarrow_m}{\gamma \vdash_m \delta, \texttt{unfold}\ x \hookrightarrow \delta, v}$$

We learn: $a = \texttt{unfold}\ x$, $v = v$, and $\delta' = \delta$.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\textsc{T-Unfold}\ \frac{\Gamma \vdash_m x : \mu\alpha.\ \tau_0 \textit{(H1)}}{\Gamma \vdash_m \texttt{unfold}\ x : [(\mu\alpha.\ \tau_0)/\alpha]\tau_0}$$

We learn: $\tau = [(\mu\alpha.\ \tau_0)/\alpha]\tau_0$.
By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : \mu\alpha.\ \tau_0$.
By inversion on the typing derivation:

$$\textsc{TV-Fold}\ \frac{\Sigma \vdash \gamma(x)\downarrow_m : [(\mu\alpha.\ \tau_0)/\alpha]\tau_0 \textit{(H2)}}{\Sigma \vdash \texttt{fold}\ \gamma(x)\downarrow_m : \mu\alpha.\ \tau_0}$$

We learn that $\Sigma \vdash v : [(\mu\alpha.\ \tau_0)/\alpha]\tau_0$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash \gamma(x)\downarrow_m : [(\mu\alpha.\ \tau_0)/\alpha]\tau_0$
(2) $\Sigma \vdash \delta$
(1) By H2.
(2) Immediate by assumption $\Sigma \vdash \delta$.

- **Case:**

$$\textsc{ST-Write}\ \frac{i@m = \gamma(x)\downarrow_m \qquad |m| = 1 \textit{(H1)}}{\gamma \vdash_m \delta, \texttt{write}\ x \hookrightarrow \delta, i@m}$$

We learn: $a = \texttt{write}\ x$, $v = i@m$, and $\delta' = \delta$.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\textsc{T-Write}\ \frac{|m| = 1 \textit{(H2)} \qquad \Gamma \vdash_m x : \texttt{int@}m \textit{(H3)}}{\Gamma \vdash_m \texttt{write}\ x : \texttt{int@}m}$$

We learn: $\tau = \texttt{int@}m$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash i@m : \texttt{int@}m$
(2) $\Sigma \vdash \delta$
(1) By the following typing derivation:

$$\textsc{TV-Loc}\ \dfrac{\textsc{TV-Int}\ \dfrac{}{\Sigma \vdash i : \texttt{int}}}{\Sigma \vdash i@m : \texttt{int@}m}$$

(2) Immediate by assumption $\Sigma \vdash \delta$.

- **Case:**

$$\textsc{ST-Read}\ \dfrac{|m| = 1 \textit{(H1)}}{\gamma \vdash_m \delta, \texttt{read} \hookrightarrow \delta, i@m}$$

We learn: $a = \texttt{read}$, $v = i@m$, and $\delta' = \delta$.
By the typing judgement T-Read, we learn: $\tau = \texttt{int@}m$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash i@m : \texttt{int@}m$
(2) $\Sigma \vdash \delta$
(1) By the following typing derivation:

$$\textsc{TV-Loc}\ \dfrac{\textsc{TV-Int}\ \dfrac{}{\Sigma \vdash i : \texttt{int}}}{\Sigma \vdash i@m : \texttt{int@}m}$$

(2) Immediate by assumption $\Sigma \vdash \delta$.

- **Case:**

$$\textsc{ST-Embed}\ \dfrac{i@m = \gamma(x)\downarrow_m \textit{(H1)}}{\gamma \vdash_m \delta, \texttt{embed}[p]\ x \hookrightarrow \delta, i^{\texttt{enc\#}p}@m}$$

We learn: $a = \texttt{embed}[p]\ x$, $v = i^{\texttt{enc\#}p}@m$, and $\delta' = \delta$.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\textsc{T-Embed}\ \dfrac{\Gamma \vdash_m x : \texttt{int@}m \textit{(H2)}}{\Gamma \vdash_m \texttt{embed}[p]\ x : \texttt{int}^{\texttt{enc\#}p}@m}$$

We learn: $\tau = \texttt{int}^{\texttt{enc\#}p}@m$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash_m i^{\texttt{enc\#}m}@m : \texttt{int}^{\texttt{enc\#}m}@m$
(2) $\Sigma \vdash_m \delta$
(1) By the following typing derivation:

$$\textsc{TV-Loc}\ \dfrac{\textsc{TV-Int}\ \dfrac{}{\Sigma \vdash i^{\texttt{enc\#}p} : \texttt{int}^{\texttt{enc\#}p}}}{\Sigma \vdash i^{\texttt{enc\#}p}@m : \texttt{int}^{\texttt{enc\#}p}@m}$$

(2) Immediate by assumption $\Sigma \vdash_m \delta$.

- **Case:**

$$\textsc{ST-Share}\ \frac{i@p' = \gamma(x)\downarrow_m (H1) \quad |p| = 1 (H2) \quad |q| \neq 0 (H3) \quad p \subseteq p' (H4) \quad p \cup q = m (H5)}{\gamma \vdash_m \delta, \mathtt{share}[p \to q]\ x \hookrightarrow \delta, i^{\mathtt{enc\#}q}@q}$$

We learn: $a = \mathtt{share}[p \to q]\ x$, $v = i^{\mathtt{enc\#}q}@q$, and $\delta' = \delta$.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\textsc{T-Share}\ \frac{\Gamma \vdash_m x : \mathtt{int}@p' (H6) \quad |p| = 1 (H2) \quad |q| \neq 0 (H3) \quad p \subseteq p' (H4) \quad p \cup q = m (H5)}{\Gamma \vdash_m \mathtt{share}[p \to q]\ x : \mathtt{int}^{\mathtt{enc\#}q}@q}$$

We learn: $\tau = \mathtt{int}^{\mathtt{enc\#}q}@q$
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash i^{\mathtt{enc\#}q}@q : \mathtt{int}^{\mathtt{enc\#}q}@q$
(2) $\Sigma \vdash \delta$
(1) By the following typing derivation:

$$\textsc{TV-Loc}\ \frac{\textsc{TV-Int}\ \dfrac{}{\Sigma \vdash i^{\mathtt{enc\#}q} : \mathtt{int}^{\mathtt{enc\#}q}}}{\Sigma \vdash i^{\mathtt{enc\#}q}@q : \mathtt{int}^{\mathtt{enc\#}q}@q}$$

(2) Immediate by assumption $\Sigma \vdash_m \delta$.

- **Case:**

$$\textsc{ST-Reveal}\ \frac{i^{\mathtt{enc\#}p}@p = \gamma(x)\downarrow_m (H1) \quad p \cup q = m (H2) \quad |q| \neq 0 (H3)}{\gamma \vdash_m \delta, \mathtt{reveal}[q]\ x \hookrightarrow \delta, i@q}$$

We learn: $a = \mathtt{reveal}[p]\ x$, $v = i@p$, and $\delta' = \delta$.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

$$\textsc{T-Reveal}\ \frac{\Gamma \vdash_m x : \mathtt{int}^{\mathtt{enc\#}p}@p (H4) \quad |q| \neq 0 (H3) \quad p \cup q = m (H2)}{\Gamma \vdash_m \mathtt{reveal}[q]\ x : \mathtt{int}@q}$$

We learn: $\tau = \mathtt{int}@q$
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash i@p : \mathtt{int}@q$
(2) $\Sigma \vdash \delta$
(1) By the following typing derivation:

$$\textsc{TV-Loc}\ \frac{\textsc{TV-Int}\ \dfrac{}{\Sigma \vdash i : \mathtt{int}}}{\Sigma \vdash i@q : \mathtt{int}@q}$$

(2) Immediate by assumption $\Sigma \vdash_m \delta$.

- **Case:**

$$\textsc{ST-Star}\ \frac{\bigstar = \gamma(x)\downarrow_m}{\gamma \vdash_m \delta, \pi_1\ x \hookrightarrow \delta, \bigstar}$$

We learn: $a = \pi_i\ x$, $v = \bigstar$, and $\delta' = \delta$.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:

T-Proj
$$\frac{\Gamma \vdash_m x : \tau_1 \times \tau_2 \textit{(H1)}}{\Gamma \vdash_m \pi_i\ x : \tau_i}$$
We learn: $\tau = \tau_i$.
By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : \tau_1 \times \tau_2$, so $\Sigma \vdash \bigstar : \tau_1 \times \tau_2$.
By inversion on the typing derivation:
TV-Star
$$\frac{\vdash_\varnothing : \tau_1 \times \tau_2 \textit{(H2)}}{\Sigma \vdash \bigstar : \tau_1 \times \tau_2}$$
By inversion on H2:
WF-Prod
$$\frac{\vdash_\varnothing \tau_1 \qquad \vdash_\varnothing \tau_2}{\vdash_\varnothing \tau_1 \times \tau_2}$$
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash \bigstar : \tau_i$
(2) $\Sigma \vdash \delta$
(1)
TV-Star
$$\frac{\vdash_\varnothing : \tau_i}{\Sigma \vdash \bigstar : \tau_i}$$
(2) Immediate by assumption $\Sigma \vdash_m \delta$.

- **Case:**

ST-Star
$$\frac{\bigstar = \gamma(x)\downarrow_m}{\gamma \vdash_m \delta, \texttt{unfold}\ x \hookrightarrow \delta, \bigstar}$$
We learn: $a = \texttt{unfold}\ x$, $v = \bigstar$, and $\delta' = \delta$.
By inversion on typing derivation $\Gamma \vdash_m a : \tau$:
T-Unfold
$$\frac{\Gamma \vdash_m x : \mu\alpha.\ \tau_0 \textit{(H1)}}{\Gamma \vdash_m \texttt{unfold}\ x : [(\mu\alpha.\ \tau_0)/\alpha]\tau_0}$$
We learn: $\tau = [(\mu\alpha.\ \tau)/\alpha]\tau$.
By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : \mu\alpha.\ \tau_0$, so $\Sigma \vdash \bigstar : \mu\alpha.\ \tau_0$.
By inversion on the typing derivation:
TV-Star
$$\frac{\vdash_\varnothing : \mu\alpha.\ \tau_0 \textit{(H2)}}{\Sigma \vdash \bigstar : \mu\alpha.\ \tau_0}$$
By inversion on H2:
WF-Rec
$$\frac{\{\alpha \mapsto m'\} \vdash_{m'} \tau_0 \textit{(H3)} \qquad \varnothing \supseteq m' \textit{(H4)}}{\vdash_\varnothing \mu\alpha.\ \tau_0}$$
By Lemma B5, $\vdash_\varnothing \mu\alpha.\ \tau_0$, and $\{\alpha \mapsto m'\} \vdash_{m'} \tau_0$, we know that $\vdash_{m'} [(\mu\alpha.\ \tau_0)/\alpha]\tau_0$.
By H4 we know that $m' = \varnothing$.
Let $\Sigma' = \Sigma$.
Need to show:
(1) $\Sigma \vdash \bigstar : [(\mu\alpha.\ \tau_0)/\alpha]\tau_0$

(2) $\Sigma \vdash \delta$
(1) By inversion on the typing derivation:
(2)
$$\dfrac{\dfrac{}{\vdash_{\varnothing} : [(\mu\alpha.\ \tau_0)/\alpha]\tau_0}\ \textsc{TV-Star}}{\Sigma \vdash \bigstar : [(\mu\alpha.\ \tau_0)/\alpha]\tau_0}$$
(3) Immediate by assumption $\Sigma \vdash_m \delta$.

□

Lemma B.10 (Preservation - Configurations).

If : $\Gamma \vdash_m e : \tau$
And : $\Sigma, \Gamma \vdash \gamma$
And : $\Sigma \vdash \delta$
And : $\Sigma \vdash \kappa : \tau \rhd \tau''$
And : $m \neq \varnothing$
And : $m, \gamma, \delta, \kappa, e \longrightarrow m', \gamma', \delta', \kappa', e'$
Then : $\Gamma' \vdash_{m'} e' : \tau'$
And : $\Sigma', \Gamma' \vdash \gamma'$
And : $\Sigma' \vdash \delta'$
And : $\Sigma' \vdash \kappa' : \tau' \rhd \tau''$
And : $m' \neq \varnothing$
For some : Γ' *and* Σ'

Proof.
By case analysis on the step derivation $m, \gamma, \delta, \kappa, e \longrightarrow m', \gamma', \delta', \kappa', e'$.

- **Case:**

$$\dfrac{(\iota_1\ v)@m = \gamma(x)\downarrow_m \quad \textit{(H1)}}{m, \gamma, \delta, \kappa, \mathsf{case}\ x\ \{x.\ e_1\}\ \{x.\ e_2\} \longrightarrow m, \{x \mapsto v\} \uplus \gamma, \delta, \kappa, e_1}\ \textsc{ST-Case}$$

We learn: $e = \mathsf{case}\ x\ \{x.\ e_1\}\ \{x.\ e_2\}$, $m' = m$, $\gamma' = \{x \mapsto v\} \uplus \gamma$, $\delta' = \delta$, $\kappa' = \kappa$, $e' = e_1$.
By inversion on typing derivation $\Gamma \vdash_m e : \tau$:

$$\dfrac{\begin{array}{ll}\Gamma \vdash_m x\ : (\tau_1 + \tau_2)@m & \textit{(H2)}\\ \{x \mapsto \tau_1\} \uplus \Gamma \vdash_m e_1 : \tau & \textit{(H3)}\\ \{x \mapsto \tau_2\} \uplus \Gamma \vdash_m e_2 : \tau & \textit{(H4)}\end{array}}{\Gamma \vdash_m \mathsf{case}\ x\ \{x.\ e_1\}\ \{x.\ e_2\} : \tau}\ \textsc{T-Case}$$

By Lemma B3 and H2, we learn that $\Sigma \vdash \gamma(x)\downarrow_m : (\tau_1 + \tau_2)@m$ so $\Sigma \vdash (\iota_1\ v)@m : (\tau_1 + \tau_2)@m$.
Let $\Gamma' = \{x \mapsto \tau_i\} \uplus \Gamma$, $\Sigma' = \Sigma$, and $\tau' = \tau_0$.
Need to show:
(1) $\{x \mapsto \tau_i\} \uplus \Gamma \vdash_m e_i : \tau_0$
(2) $\Sigma, \{x \mapsto \tau_i\} \uplus \Gamma \vdash \{x \mapsto v\} \uplus \gamma$
(3) $\Sigma \vdash \delta$
(4) $\Sigma \vdash \kappa : \tau_0 \rhd \tau''$
(5) $m \neq \varnothing$
(1) By H3/H4.
(2) By environment extension and by inversion on the typing derivation:

$$\frac{\Sigma \vdash v : \tau_1 \qquad \vdash \tau_2}{\Sigma \vdash \iota_1\ v : \tau_1 + \tau_2}\ \textsc{TV-Inj}$$

Or similar for $\iota_2\ v$.

(3) Immediate by assumption.
(4) Immediate by $\Sigma \vdash \kappa : \tau \rhd \tau''$ and $\tau = \tau_0$.
(5) Immediate by assumption.

- **Case:**

$$\frac{v_1 = \gamma(x_1)\downarrow_m \qquad v_2 = \gamma(x_2)\downarrow_m \qquad \langle \lambda_z x.\ e_0, \gamma_0 \rangle @ m = v_1}{m, \gamma, \delta, \kappa, x_1\ x_2 \longrightarrow m, \{z \mapsto v_1, x \mapsto v_2\} \uplus \gamma_0, \delta, \kappa, e_0}\ \textsc{ST-App}$$

We learn: $e = x_1\ x_2$, $m' = m$, $\gamma' = \{z \mapsto v_1, x \mapsto v_2\} \uplus \gamma_0$, $\delta' = \delta$, $\kappa' = \kappa$, $e' = e_0$.
By inversion on typing derivation $\Gamma \vdash_m e : \tau$:

$$\frac{\Gamma \vdash_m x_1 : (\tau_1\ {}^m\!\!\longrightarrow \tau_2)@m \textit{(H1)} \qquad \Gamma \vdash_m x_2 : \tau_1 \textit{(H2)}}{\Gamma \vdash_m x_1\ x_2 : \tau_2}\ \textsc{T-App}$$

We learn: $\tau = \tau_2$.
By Lemma B3 and H1, we learn that $\Sigma \vdash \gamma(x_1)\downarrow_m : (\tau_1\ {}^m\!\!\longrightarrow \tau_2)@m$ so $\Sigma \vdash \langle \lambda_z x.\ e_0, \gamma_0 \rangle @ m : (\tau_1\ {}^m\!\!\longrightarrow \tau_2)@m$.
By Lemma B3 and H2, we learn that $\Sigma \vdash \gamma(x_2)\downarrow_m : \tau_1$ so $\Sigma \vdash v_2 : \tau_1$.
By inversion on:

$$\frac{\Sigma, \Gamma \vdash \gamma \textit{(H3)} \qquad \{z \mapsto (\tau_1\ {}^m\!\!\longrightarrow \tau_2)@m, x \mapsto \tau_1\} \uplus \Gamma \vdash_m e_0 : \tau_2 \textit{(H4)}}{\Sigma \vdash \langle \lambda_z x.\ e_0, \gamma_0 \rangle : \tau_1\ {}^m\!\!\longrightarrow \tau_2}\ \textsc{TV-Fun}$$

Let $\Gamma' = \{z \mapsto (\tau_1\ {}^m\!\!\longrightarrow \tau_2)@m, x \mapsto \tau_1\} \uplus \Gamma$, $\Sigma' = \Sigma$, and $\tau' = \tau_2$.
Need to show:
(1) $\{z \mapsto (\tau_1\ {}^m\!\!\longrightarrow \tau_2)@m, x \mapsto \tau_1\} \uplus \Gamma \vdash_m e_0 : \tau_2$
(2) $\Sigma, \{z \mapsto (\tau_1\ {}^m\!\!\longrightarrow \tau_2)@m, x \mapsto \tau_1\} \uplus \Gamma \vdash \{z \mapsto \langle \lambda_z x.\ e_0, \gamma_0 \rangle @ m, x \mapsto v_2\} \uplus \gamma_0$
(3) $\Sigma \vdash \delta$
(4) $\Sigma \vdash \kappa : \tau_2 \rhd \tau''$
(5) $m \neq \varnothing$
(1) By H4.
(2) By environment extension, $\Sigma \vdash \langle \lambda_z x.\ e_0, \gamma_0 \rangle @ m : (\tau_1\ {}^m\!\!\longrightarrow \tau_2)@m$, and $\Sigma \vdash v_2 : \tau_1$.
(3) Immediate by assumption.
(4) Immediate by $\Sigma \vdash \kappa : \tau \rhd \tau''$ and $\tau = \tau_2$.
(5) Immediate by assumption.

- **Case:**

$$\frac{|m \cap p| \neq 0 \textit{(H1)}}{m, \gamma, \delta, \kappa, \mathsf{par}[p]\ e_0 \longrightarrow m \cap p, \gamma, \delta, \kappa, e_0}\ \textsc{ST-Par}$$

We learn: $e = \mathsf{par}[p]\ e$, $m' = m \cap p$, $\gamma' = \gamma$, $\delta' = \delta$, $\kappa' = \kappa$, and $e' = e_0$.
By inversion on typing derivation $\Gamma \vdash_m e : \tau$:

T-Par
$$\frac{\Gamma \vdash_{m \cap p} e : \tau \text{(H2)} \qquad m \cap p \neq \varnothing \text{(H3)}}{\Gamma \vdash_m \mathsf{par}[p]\ e : \tau}$$

Let $\Gamma' = \Gamma$, $\Sigma' = \Sigma$, and $\tau' = \tau$.
Need to show:
(1) $\Gamma \vdash_{m \cap p} e_0 : \tau$
(2) $\Sigma, \Gamma \vdash \gamma$
(3) $\Sigma \vdash \delta$
(4) $\Sigma \vdash \kappa : \tau \rhd \tau''$
(5) $m \cap p \neq \varnothing$
(1) By H2.
(2) Immediate by assumption.
(3) Immediate by assumption.
(4) Immediate by assumption.
(5) By H1.

- **Case:**

ST-ParEmpty
$$\frac{|m \cap p| = 0 \text{(H1)} \qquad \gamma' = \{x \mapsto \bigstar\} \uplus \gamma}{m, \gamma, \delta, \kappa, \mathsf{par}[p]\ e_0 \longrightarrow m, \gamma', \delta, \kappa, x}$$

We learn: $e = \mathsf{par}[p]\ e_0$, $m' = m$, $\gamma' = \{x \mapsto \bigstar\} \uplus \gamma$, $\delta' = \delta$, $\kappa' = \kappa$, and $e' = x$.
By inversion on typing derivation $\Gamma \vdash_m e : \tau$:

T-ParEmpty
$$\frac{|m \cap p| = 0 \text{(H1)} \qquad \vdash_\varnothing \tau \text{(H2)}}{\Gamma \vdash_m \mathsf{par}[p]\ e_0 : \tau}$$

By inversion on the typing derivation $\Sigma \vdash \bigstar : \tau$:

TV-Star
$$\frac{\vdash_\varnothing \tau \text{(H2)}}{\Sigma \vdash \bigstar : \tau}$$

By Lemma B6 and H2, we know that $\vdash_m \tau$.
Let $\Gamma' = \{x \mapsto \tau\} \uplus \Gamma$, $\Sigma' = \Sigma$, and $\tau' = \tau$.
Need to show:
(1) $\{x \mapsto \tau\} \uplus \Gamma \vdash_m x : \tau$
(2) $\Sigma, \{x \mapsto \tau\} \uplus \Gamma \vdash \{x \mapsto \bigstar\} \uplus \gamma$
(3) $\Sigma \vdash \delta$
(4) $\Sigma \vdash \kappa : \tau \rhd \tau''$
(5) $m \neq \varnothing$
(1) By the typing derivation:
T-Var
$$\frac{\{x \mapsto \tau\} \uplus \Gamma(x) = \tau \qquad \tau <: \tau \qquad \vdash_m \tau}{\{x \mapsto \tau\} \uplus \Gamma \vdash_m x : \tau}$$
(2) Immediate by environment extension and $\Sigma \vdash \bigstar : \tau_0$.
(3) Immediate by assumption.
(4) Immediate by assumption.
(5) Immediate by assumption.

- **Case:**

$$\frac{\kappa' = \langle \texttt{let } x = \square \texttt{ in } e_2 \mid m, \gamma \rangle :: \kappa}{m, \gamma, \delta, \kappa, \texttt{let } x = e_1 \texttt{ in } e_2 \longrightarrow m, \gamma, \delta, \kappa', e_1} \text{ ST-LetPush}$$

We learn: $e = \texttt{let } x = e_1 \texttt{ in } e_2$, $m' = m$, $\gamma' = \gamma$, $\delta' = \delta$, $\kappa' = \langle \texttt{let } x = \square \texttt{ in } e_2 \mid m, \gamma \rangle :: \kappa$, $e' = e_1$.

By inversion on typing derivation $\Gamma \vdash_m e : \tau$:

$$\frac{\Gamma \vdash_m e_1 : \tau_1 \textit{(H1)} \quad \{x \mapsto \tau_1\} \uplus \Gamma \vdash_m e_2 : \tau_2 \textit{(H2)}}{\Gamma \vdash_m \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2} \text{ T-Let}$$

We learn: $\tau = \tau_2$.

Let $\Gamma' = \Gamma$, $\Sigma' = \Sigma$, and $\tau' = \tau_1$.

Need to show:

(1) $\Gamma \vdash_m e_1 : \tau_1$

(2) $\Sigma, \Gamma \vdash \gamma$

(3) $\Sigma \vdash \delta$

(4) $\Sigma \vdash (\langle \texttt{let } x = \square \texttt{ in } e_2 \mid m, \gamma \rangle :: \kappa) : \tau_1 \rhd \tau''$

(5) $m \neq \varnothing$

(1) By H1.

(2) Immediate by assumption.

(3) Immediate by assumption.

(4) By the typing derivation:

(5)
$$\frac{\text{H2}\ \overline{\{x \mapsto \tau_1\} \uplus \Gamma \vdash_m e : \tau_2} \qquad \overline{\Sigma, \Gamma \vdash \gamma} \qquad \overline{\Sigma \vdash \kappa : \tau_2 \rhd \tau_3}}{\Sigma \vdash (\langle \texttt{let } x = \square \texttt{ in } e \mid m, \gamma \rangle :: \kappa) : \tau_1 \rhd \tau_3} \text{ T-Frame}$$

(6) Immediate by assumption.

- **Case:**

$$\frac{\gamma \vdash_m \delta, a \hookrightarrow \delta', v \textit{(H1)} \qquad \kappa = \langle \texttt{let } x = \square \texttt{ in } e_0 \mid m', \gamma_0 \rangle :: \kappa'}{m, \gamma, \delta, \kappa, a \longrightarrow m', \{x \mapsto v\} \uplus \gamma_0, \delta', \kappa', e_0} \text{ ST-LetPop}$$

We learn: $e = a$, $m' = m'$, $\gamma' = \{x \mapsto v\} \uplus \gamma_0$, $\delta' = \delta'$, $\kappa' = \kappa_0$, $e' = e_0$.

By In-Frame Preservation using $\Gamma \vdash_m a : \tau$, $\Sigma, \Gamma \vdash \gamma$, $\Sigma \vdash \delta$, and H1, we learn that $\Sigma' \vdash v : \tau$ and $\Sigma' \vdash \delta'$.

By inversion on the typing derivation:

$$\frac{\{x \mapsto \tau\} \uplus \Gamma \vdash_m e_0 : \tau_1 \textit{(H2)} \qquad \Sigma, \Gamma \vdash \gamma \textit{(H3)} \qquad \Sigma \vdash \kappa_0 : \tau_1 \rhd \tau'' \textit{(H4)}}{\Sigma \vdash (\langle \texttt{let } x = \square \texttt{ in } e_0 \mid m', \gamma_0 \rangle :: \kappa_0) : \tau \rhd \tau''} \text{ T-Frame}$$

Let $\Gamma' = \{x \mapsto \tau\} \uplus \Gamma$, $\Sigma' = \Sigma'$, and $\tau' = \tau_1$.

Need to show:

(1) $\{x \mapsto \tau\} \uplus \Gamma \vdash_m e_0 : \tau_1$

(2) $\Sigma', \{x \mapsto \tau\} \uplus \Gamma \vdash \{x \mapsto v\} \uplus \gamma_0$

(3) $\Sigma' \vdash \delta'$

(4) $\Sigma' \vdash \kappa_0 : \tau_1 \rhd \tau''$

(5) $m \neq \varnothing$

(1) By H2.

(2) By environment extension and $\Sigma' \vdash v : \tau$.
(3) From the conclusion of In-Frame Preservation.
(4) By H4.
(5) Immediate by assumption.

□

C DISTRIBUTED SEMANTICS: PROOFS OF META-THEORETIC PROPERTIES

In this section, we prove the key meta-theoretic properties of the distributed semantics, namely forward simulation (Appendix C.1) and confluence (Appendix C.2), along with their corollaries.

C.1 Forward Simulation

The key lemma for proving simulation states that if global single-threaded configuration c steps to c', then the slicing of c steps to c' over multiple steps of the multi-threaded semantics. The basic structure of the proof is, based on the form of step from global configuration c, to construct a sequence of distributed steps that each updates the local configuration of some party in the mode of c. For non-atomic expressions, there is exactly one step for every party in the mode; the most interesting case are global steps that are applications SS-Par: these are simulated by a sequence of steps which may be built from applications of SS-Par themselves *or* SS-Empty. For expressions that evaluate an atom and bind the result, there is a single step, performed by all parties.

Lemma C.1 (Forward Simulation-Step). *If $c \longrightarrow c'$, then $c\lightning \rightsquigarrow^* c'\lightning$.*

Proof. $c \longrightarrow c'$, by assumption. Let $(m, \gamma, \delta, \kappa, e) = c$ and let $(m', \gamma', \delta', \kappa', e') = c'$. Proceed by cases on the form of the evidence of $c \longrightarrow c'$:

SS-Case $c\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow c'\lightning$, where each C_i is $c\lightning|_{[0,i]} \uplus c'\lightning|_{[i+1,|m|]}$ (where $C|I$ denotes distributed configuration C restricted to parties at indices I).
The proof that each C_i steps to C_{i+1} is as follows. Apply DS-Step, with ς as the configuration

$$m_i, \gamma, \delta, \kappa, \text{case } x\{x_1.e_1\}\{x_2.e_2\}$$

ς' as the configuration

$$m_i, \{x \mapsto v\} \uplus \gamma, \delta, \kappa, e_j$$

where $\gamma(x) \not\downarrow_{m_i} = (\iota_j v)@\{m_i\}$ and $C_i|_{[0,i-1],[i+1,|m|]}$ as C. ς is well-limited by G-NonAtom, the fact that e is a case expression (and thus not an atom), and the fact that the mode contains the single party m_i. $\varsigma \longrightarrow \varsigma'$ by SS-Case. C_{i+1} is $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \varsigma'\} \uplus C_i|_{[i+1,|m|]}$.

SS-Par $c\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow c'\lightning$, where each C_i is $c\lightning|_{[0,i]} \uplus c'\lightning|_{[i+1,|m|]}$.
The proof that each C_i steps to C_{i+1} is as follows. If $m_i \in p$, then apply DS-Step, with ς as the configuration

$$m_i, \gamma, \delta, \kappa, \texttt{par}\ p\ e$$

ς' as the configuration

$$m_i, \gamma, \delta, \kappa, e$$

and $C_i|_{[0,i-1],[i+1,|m|]}$ as C. ς is well-limited by G-NonAtom, the fact that $\texttt{par}\ x\ e$ is not an atom, and the fact that the mode contains the single party m_i. $\varsigma \longrightarrow \varsigma'$ by SS-Par, because $m_i \in p$ and thus $\{m_i\} \cap p = \{m_i\} \neq \emptyset$.
If $m_i \notin p$, then apply DS-Step, with ς as the configuration

$$m_i, \gamma, \delta, \kappa, \texttt{par}\ p\ e$$

ς' as the configuration

$$m_i, \{x \mapsto \bigstar\} \uplus \gamma, \delta, \kappa, x$$

and $C_i|_{[0,i-1],[i+1,|m|]}$ as C. ς is well-limited by G-NonAtom, the fact that $\texttt{par}\ x\ e$ is not an atom, and the fact that the mode contains the single party m_i. $\varsigma \to \varsigma'$ by SS-NonPar, because $m_i \notin p$ and thus $\{m_i\} \cap p = \emptyset$.

In both cases, C_{i+1} is $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \varsigma'\} \uplus C_i|_{[i+1,|m|]}$.

SS-ParEmpty $c\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow c'\lightning$, where each C_i is $c\lightning|_{[0,i]} \uplus c'\lightning|_{[i+1,|m|]}$.

The proof that each C_i steps to C_{i+1} is as follows. Apply DS-Step, with ς as the configuration

$$m_i, \gamma, \delta, \kappa, \texttt{par}\ p\ e$$

ς' as the configuration

$$m_i, \{x \mapsto \bigstar\} \uplus \gamma, \delta, \kappa, x$$

and $C_i|_{[0,i-1],[i+1,|m|]}$ as C. ς is well-limited by G-NonAtom, the fact that $\texttt{par}\ x\ e$ is not an atom, and the fact that the mode contains the single party m_i. $\varsigma \to \varsigma'$ by SS-NonPar, because $m \cap p = \emptyset$ by the fact that $c \to c$ is an application of SS-ParEmpty; thus $\{m_i\} \cap p = \emptyset$. C_{i+1} is $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \varsigma'\} \uplus C_i|_{[i+1,|m|]}$.

SS-App $c\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow c'\lightning$, where each C_i is $c\lightning|_{[0,i]} \uplus c'\lightning|_{[i+1,|m|]}$.

The proof that each C_i steps to C_{i+1} is as follows. Let $\langle \lambda_z x.e', \gamma' \rangle @ m = v_1 = \gamma(x_1)$, which holds in the case that $c \to c'$ is an application of SS-App.

Apply DS-Step, with ς as the configuration

$$m_i, \gamma, \delta, \kappa, x_1\ x_2$$

ς' as the configuration

$$m_i, \{z \mapsto v_1, x_2 \mapsto \gamma(x_2)\} \uplus \gamma, \delta, \langle \texttt{let}\ x\ =\ \_\ \texttt{in}\ e_2 \mid \gamma \rangle :: \kappa, e'$$

and $C_i|_{[0,i-1],[i+1,|m|]}$ as C. ς is well-limited by G-NonAtom, the fact that $x_1\ x_2$ is not an atom, and the fact that the mode contains the single party m_i. $\varsigma \to \varsigma'$ by SS-App. C_{i+1} is $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \varsigma'\} \uplus C_i|_{[i+1,|m|]}$.

SS-LetPush $c\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow c'\lightning$, where each C_i is $c\lightning|_{[0,i]} \uplus c'\lightning|_{[i+1,|m|]}$.

The proof that each C_i steps to C_{i+1} is as follows. Apply DS-Step, with ς as the configuration

$$m_i, \gamma, \delta, \kappa, \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2$$

ς' as the configuration

$$m_i, \gamma, \delta, \langle \texttt{let}\ x\ =\ \_\ \texttt{in}\ e_2 \mid \rangle :: \kappa, e_1$$

and $C_i|_{[0,i-1],[i+1,|m|]}$ as C. ς is well-limited by G-NonAtom, the fact that e is a let expression (and thus not an atom), and the fact that the mode contains the single party m_i. $\varsigma \to \varsigma'$ by SS-LetPush. C_{i+1} is $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \varsigma'\} \uplus C_i|_{[i+1,|m|]}$.

SS-LetPop e is some atom a, δ and a step to δ' and some value v under γ in mode m, and $\kappa = \langle \texttt{let}\ x\ =\ \_\ \texttt{in}\ e' \mid m', \gamma'' \rangle :: \kappa'$, and $\gamma' = \{x \mapsto v\} \uplus \gamma''$, by assumption.

Proceed by cases on the fact that δ and a step to δ' and some value v under γ in mode m.

Subcase: solo atom In the case that the evaluation is an application of SS-Int, SS-Var, SS-Fun, SS-Inj, SS-Pair, SS-Proj, SS-Ref, SS-Deref, SS-Assign, SS-Fold, SS-Unfold, SS-Read, SS-Write, SS-Embed, and SS-Star, $c\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow c'\lightning$, where each C_i is $c\lightning|_{[0,i]} \uplus c'\lightning|_{[i+1,|m|]}$.

The proof that each C_i steps to C_{i+1} is as follows. Apply DS-Step, with ς as the configuration

$$m_i, \gamma, \delta, \langle \texttt{let}\ x\ =\ \_\ \texttt{in}\ e' \mid m', \gamma'' \rangle :: \kappa', a$$

ς' as the configuration

$$m_i, \{x \mapsto v\} \uplus \gamma, \delta', \kappa', e'$$

and $C_i|_{[0,i-1],[i+1,|m|]}$ as C. ς is well-limited by G-Atom applied to G-Solo applied to the fact that the mode contains the single party m_i. $\varsigma \to \varsigma'$ by the rule applied to build the single-threaded step under consideration (e.g., SS-Int or SS-Var). C_{i+1} is $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \varsigma'\} \uplus C_i|_{[i+1,|m|]}$.
Subcases: binary operation over clear data The subcase in which evaluation is an application of SS-Binop, a is of the form $x_1 \oplus x_2$, $i_1^{\cdot}@m = \gamma(x_1) \swarrow_m$, and $i_2^{\cdot}@m = \gamma(x_1) \swarrow_m$ (i.e., the computation is an binary operation over clear data) is directly similar to the previous subcase. The only distinction is that the fact that a is well-limited under m is by G-Binop applied to an application of G-Clear.
Subcase: mux on clear data The subcase in which evaluation is an application of SS-Mux, a is of the form $\texttt{mux if } x_1 \texttt{ then } x_2 \texttt{ else } x_3$, $i_1^{\cdot}@m = \gamma(x_1) \swarrow_m$, $i_2^{\cdot}@m = \gamma(x_2) \swarrow_m$, and $i_3^{\cdot}@m = \gamma(x_3) \swarrow_m$ is directly similar to the previous subcases. The only distinction is that the fact that a is well-limited under m is by G-Mux applied to an application of G-Clear.
Subcase: binary operation on encrypted data For the subcase in which evaluation is an application of SS-Binop, a is of the form $x_1 \oplus x_2$, $i_1^{\mathsf{enc}\#m}@m = \gamma(x_1) \swarrow_m$, and $i_2^{\mathsf{enc}\#m}@m = \gamma(x_1) \swarrow_m$ (i.e., the computation is an binary operation over encrypted data), $c\lightning$ steps to $c'\lightning$ by application of DS-Step, with

$$m, \gamma, \delta, \kappa, x_1 \oplus x_2$$

as ς,

$$m, \{x \mapsto v\} \uplus \gamma, \delta, \kappa, x$$

as ς', and $c\lightning|_{\mathsf{parties}\setminus m}$ as C. ς is well-limited by G-Binop applied to an application of G-Enc. ς steps to ς' by SS-Binop.
Subcase: mux on encrypted data The subcase in which evaluation is an application of SS-Mux, a is of the form $\texttt{mux if } x_1 \texttt{ then } x_2 \texttt{ else } x_3$, $i_1^{\mathsf{enc}\#m}@m = \gamma(x_1) \swarrow_m$, $i_2^{\mathsf{enc}\#m}@m = \gamma(x_2) \swarrow_m$, and $i_3^{\mathsf{enc}\#m}@m = \gamma(x_3) \swarrow_m$ is directly similar to the previous subcase. The only distinction from the previous subcase is that the fact that a is well-limited under m is by G-Mux applied to an application of G-Enc.
Subcases: synchronization The subcases in which evaluation is an application of SS-Share or SS-Reveal are directly similar to the previous two subcaes, in that they are simulated by a single step of the distributed semantics. The only distinction from the previous subcases is that the fact that a is well-limited under m is by G-Sync.

□

The proof of weak forward simulation follows directly from Lemma C.1.

Lemma C.2 (ST Weak Forward Simulation). *If $\varsigma \longrightarrow^* \varsigma'$ and ς' is terminal, then $\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$ and $\varsigma'\lightning \not\rightsquigarrow$.*

Proof. The claim holds by induction on the multistep judgment $\varsigma \longrightarrow^* \varsigma$.

Empty If the trace is empty, then ς is ς'. $\varsigma\lightning$ multi-steps to $\varsigma\lightning$ over the empty sequence of steps.
Non-empty If the trace is of the form $\varsigma \to \varsigma'' \to^* \varsigma'$, then $\varsigma\lightning \rightsquigarrow^* \varsigma''$ by Lemma C.1 and $\varsigma''\lightning \rightsquigarrow^* \varsigma'\lightning$ by the inductive hypothesis. $\varsigma\lightning \rightsquigarrow^* \varsigma\lightning$ by the fact that the concatenation of two traces is a trace.

□

C.2 Confluence and End-State Determinism

In order to prove Lemma 6.3, we will first claim and prove a lemma that establishes that distinct sub-configurations that can step within each step of a distributed configuration in fact update the local configurations of disjoint sets of parties.

Lemma C.3. *For all distributed configurations C, C_0, and C_1 and all well-limited non-halting global configurations ς_0 and ς_1 such that*

$$C = \varsigma_0 \lightning \uplus C_0 = \varsigma_1 \lightning \uplus C_1$$

one of the following cases holds:

(1) ς_0 is ς_1 and C_0 is C_1;
(2) the domains of $\varsigma_0 \lightning$ and $\varsigma_1 \lightning$ are disjoint.

Proof. Proceed by cases on whether the domains of $\varsigma_0 \lightning$ and $\varsigma_1 \lightning$ are disjoint. If the domains are disjoint, then the second clause of the claim is satisfied.

Otherwise, there is some party m_i in the domains of both $\varsigma_0 \lightning$ and $\varsigma_1 \lightning$. The domains of $\varsigma_0 \lightning$ and $\varsigma_1 \lightning$ are the same, by cases on the active expression e_i in the configuration located at m_i: if e_i is a non-atom or a variable occurrence, an integer literal, a sum injection, a pair creation, a pair projection, a function creation, a dereference, a reference assignment, a recursive type introduction, a read, a write, or an encryption, then the domains are singletons. Thus the domains are the same, because they are singletons that overlap.

In the case that the expression is a binary operation $x_1 \oplus x_2$, that fact that ς_0 and ς_1 are well-limited is an application of G-Binop. In the subcase that their contexts map x_1 and x_2 to clear data, the domains of $\varsigma_0 \lightning$ and $\varsigma_1 \lightning$ are singletons and thus overlapping (because they overlap). In the subcase that their contexts map x_1 and x_2 to encrypted data, the data must be encrypted for a fixed mode m, by the premise of G-Binop. Thus the domains are the identical mode m.

In the case that the active expression is a multiplex `mux if` x `then` x_1 `else` $e_1 x_2 e_2$, ς_1 and ς_2 are well-limited by application of G-Mux. If the configuration contexts map x, x_1, and x_2 to clear data, then the domains are singletons by the definition of G-Mux; thus the domains are identical, by the fact that they are overlapping singletons. Otherwise, the configuration contexts map x, x_1, and x_2 to data encrypted for fixed parties m; then the domains are the identical set of parties m.

In the case that the expression shares a value from p to q, the steps from ς_1 and ς_2 are applications ST-Share, which has a premise that the mode is $p \cup q$; thus, the domains of ς_1 and ς_2 are the identical set of parties $p \cup q$.

In the case that the expression reveals the value bound to variable x to parties q, the steps from ς_1 and ς_2 are applications of ST-Reveal, which has a premise that the value bound to x is encrypted for parties p, and that the active parties are $p \cup q$; thus, the domains of ς_1 and ς_2 are the identical set of parties $p \cup q$. C_0 and C_1 are thus the same, given they are the restrictions of C to the complements of the domains of ς_1 and ς_2, respectively.

The configuration slicing operation is a bijection: this is proved per component on the slices of arbitrary configurations that slice to the same distributed configuration, performing induction on the values in the ranges of contexts and heaps. Thus ς_0 and ς_1 are identical. □

With Lemma C.3 defined, we can prove the diamond property.

Proof. There are global configuration ς_0 and ς_0' and distributed configuration C_0' such that C is $\varsigma_0 \lightning \uplus C_0'$, ς_0 is well-limited, C_0 is $\varsigma_0' \lightning \uplus C_0'$, and $\varsigma_0 \longrightarrow \varsigma_0'$, by inverting the fact that $C \rightsquigarrow C_0$ is an application of DS-Step. Similarly, there are global configurations ς_1 and ς_1' and distributed configuration C_1 such that C is $\varsigma_1 \lightning \uplus C_1'$, ς_1 is well-limited, C_1 is $\varsigma_1' \lightning \uplus C_1'$, and $\varsigma_1 \longrightarrow \varsigma_1'$, by the fact that $C \rightsquigarrow C_1$ is an application of DS-Step.

Proceed by cases on the result of applying Lemma C.3 to C, ς_0, C_0, ς_1, and C_1:

Identical In this case, let $\varsigma' = \varsigma_0' = \varsigma_1'$. It holds that $\varsigma_0' \lightning = \varsigma_1' \lightning$ and C_0' is C_1'; thus

$$C_0 = \varsigma_0' \lightning \uplus C_0' = \varsigma_1' \lightning \uplus C_1' = C_1$$

Choose C' to be $C_0 = C_1$.

Disjoint In this case, let C_0'' be the restriction of C_0' to local configurations outside of the domain of $\varsigma_1' \lightning$, and let C_1'' be the restriction of C_1' to local configurations outside of the domain $\varsigma_0' \lightning$. Choose C' to be

$$\varsigma_0 \lightning \uplus \varsigma_1 \lightning \uplus C_0'' \uplus C_1''$$

C' is well-defined because the domains of $\varsigma_0 \lightning$ and $\varsigma_1 \lightning$ are disjoint, by assumption of this clause.

To prove that $C_0 \rightsquigarrow C'$, apply DS-Step with ς_1 as the free configuration ς and $\varsigma_0 \lightning \uplus C_0'' \uplus C_1''$ as the constant distributed configuration. Symmetrically, to prove that $C_1 \rightsquigarrow C'$, apply DS-Step with ς_0 as the free configuration ς and $\varsigma_1 \lightning \uplus C_0'' \uplus C_1''$ as the constant distributed configuration.

□

Given that the distributed semantics satisfies the diamond property (Lemma 6.3), confluence (Lemma C.4) is a direct consequence of fundamental properties of general transition and rewrite systems.

Lemma C.4 (DS Multi-step Confluence). *If $C \rightsquigarrow^* C_1$ and $C \rightsquigarrow^* C_2$ then there exists C_3 s.t. $C_1 \rightsquigarrow^* C_3$ and $C_2 \rightsquigarrow^* C_3$.*

Proof. Apply the fact that any binary relation that satisfies the Diamond property satisfies confluence [Baader and Nipkow 1999] to Lemma 6.3. □

An direct corollary of confluence is that all halting states reached from the same state are the same.

Corollary C.5 (DS End-state Determinism). *If $C \rightsquigarrow^* C_1$ and $C \rightsquigarrow^* C_2$, $C_1 \not\rightsquigarrow$ and $C_2 \not\rightsquigarrow$ then $C_1 = C_2$.*

Proof. There is some distributed configuration C' such that $C_1 \rightsquigarrow^* C'$ and $C_2 \rightsquigarrow^* C'$, by applying Lemma C.4 to the fact that $C \rightsquigarrow^* C_1$ and $C \rightsquigarrow^* C_2$. C' is C_1 by the fact that C_1 is halting and thus C_1 multi-steps to C' over the empty sequence of steps; C' is C_2 by a symmetric argument. Thus, C_1 is C_2. □

Finally, for every halting execution of the global semantics, every halting distributed execution that begins from a slice of the initial state ends in a slice of the global final state.

Theorem C.6 (ST Soundness). *If $\varsigma \longrightarrow^* \varsigma'$, ς' is terminal, $\varsigma \lightning \rightsquigarrow^* C$ and $C \not\rightsquigarrow$, then $C = \varsigma' \lightning$.*

Proof. $\varsigma \lightning \rightsquigarrow^* \varsigma' \lightning$ and $\varsigma' \lightning \not\rightsquigarrow$ by application of Lemma C.2. $\varsigma \lightning$ is C by Corollary C.5, applied to the facts that $\varsigma \longrightarrow^* \varsigma'$ (by assumption), $\varsigma' \lightning \not\rightsquigarrow$, $\varsigma \lightning \rightsquigarrow^* C$, and $C \not\rightsquigarrow$ (by assumption). □