The Function Passing Model: Types, Proofs, and Semantics

Philipp Haller, Normen Müller, Heather Miller

May 2016

1 Overview

We formalize our programming model in the context of a typed lambda calculus with records. Figure 1 shows the abstract syntax of our core language. Besides standard terms, the language includes terms related to (a) spores, (b) silos, and (c) futures. The `spore` term creates a new spore. It contains a list of variable definitions, the spore header, and a closure which may only refer to its parameter and variables in the spore header. The `spawn` term creates a new host capable of hosting silos. The `populate` term initializes a new silo on a given host with a given data value. The `map`, `flatMap`, and `persist` terms create lineages of silo transformations represented as silo references. The `send` term forces the materialization of the silo corresponding to its argument silo reference; `send` returns a future which is asynchronously completed with the silo's value. The `await` term waits for the completion of its argument future and returns the future's value. Locations ι are used to refer to futures and hosts, both of which can be created dynamically using the above terms.

Values in our language are as expected: besides abstractions and record values they include spore values, locations, and silo references. Locations and silo references are not part of the "surface syntax" of our language; they are only introduced by evaluation (see Section 1.1). Silo reference values are values of a simple datatype with constructors *Mat, Mapped, FMapped*, and *Persist*. The constructors include all information required for *materializing* a silo with the result of applying the described transformations. Therefore, a silo reference value is also called the *lineage* of its corresponding silo. We defer a detailed explanation of the transformations described by a lineage to the following Section 1.1.

In addition to standard function and record types, the language has types for spores, hosts, silo references, and futures. A spore type $T \Rightarrow T' \; \{ \; \texttt{type} \; \mathcal{C} = \overline{T} \; \}$ includes the types $\overline{T}$ of the variables declared in the header of the spore.

1.1 Operational Semantics

In the following we present a small-step operational semantics of the introduced core language. The semantics is clearly stratified into a deterministic layer and a non-deterministic (concurrent) layer. Importantly, this means our programming model can benefit from existing reasoning techniques for sequential programs. Program transformations that are correct for sequential programs are also cor-

$$
\begin{array}{lll}
t ::= & & \textit{terms:} \\
& x & \text{variable} \\
& \mid (x : T) \Rightarrow t & \text{abstraction} \\
& \mid t\ t & \text{application} \\
& \mid \{\overline{l = t}\} & \text{record construction} \\
& \mid t.l & \text{selection} \\
& \mid \texttt{spore}\ \{\ \overline{x : T = t}\ ; (x : T) \Rightarrow t\ \} & \text{spore} \\
& \mid \texttt{spawn}(t) & \text{spawn host} \\
& \mid \texttt{populate}(t, t) & \text{populate silo} \\
& \mid \texttt{map}(t, t) & \text{map} \\
& \mid \texttt{flatMap}(t, t) & \text{flatMap} \\
& \mid \texttt{persist}(t) & \text{persist} \\
& \mid \texttt{send}(t) & \text{send} \\
& \mid \texttt{await}(t) & \text{await future} \\
& \mid \iota & \text{location} \\
& \mid r & \text{silo reference} \\
\\
v ::= & & \textit{values:} \\
& (x : T) \Rightarrow t & \text{abstraction value} \\
& \mid \{\overline{l = v}\} & \text{record value} \\
& \mid p & \text{spore value} \\
& \mid \iota & \text{location} \\
& \mid r & \text{silo reference} \\
\\
\multicolumn{3}{l}{p ::= \texttt{spore}\ \{\ \overline{x : T = v}\ ; (x : T) \Rightarrow t\ \}} \\
\\
r ::= & & \textit{silo reference values:} \\
& \mathrm{Mat}(\omega) & \text{materialized} \\
& \mid \mathrm{Mapped}(\omega, r, p) & \text{lineage with } \texttt{map} \\
& \mid \mathrm{FMapped}(\omega, r, p) & \text{lineage with } \texttt{flatMap} \\
& \mid \mathrm{Persist}(\omega, r, v) & \text{lineage with } \texttt{persist} \\
\\
\multicolumn{2}{l}{\omega ::= (h, i) \quad \text{where } i \in \mathbb{N}} & \text{decentralized identifier} \\
\\
T ::= & & \textit{types:} \\
& T \Rightarrow T & \text{abstraction type} \\
& \mid \{\overline{l : T}\} & \text{record type} \\
& \mid T \Rightarrow T\ \{\ \texttt{type}\ \mathcal{C} = \overline{T}\ \} & \text{spore type} \\
& \mid \texttt{Host} & \text{host type} \\
& \mid \texttt{SiloRef}[T] & \text{silo reference type} \\
& \mid \texttt{Future}[T] & \text{future type}
\end{array}
$$

Figure 1: Abstract syntax of core language.

rect for distributed programs. Our programming model shares this property with some existing approaches [?].

The semantics is based on three reduction relations for (a) sequential reduction of terms, (b) deterministic reduction of hosts, and (c) non-deterministic reduction of sets of hosts. The reduction relations use the definition of evalua-

$E ::=$	*evaluation contexts:*
$[\,]$	hole
$\mid E\ t$	application (fun)
$\mid v\ E$	application (arg)
$\mid \{\overline{l = v}; l_i = E; \overline{l' = t}\}$	record
$\mid E.l$	selection
$\mid \texttt{spore}\ \{\ \overline{x : T = v}; x_i : T_i = E; \overline{x' : T = t}; (x : T) \Rightarrow t\ \}$	spore
$\mid \texttt{spawn}(E)$	spawn
$\mid \texttt{populate}(E, t)$	populate (host)
$\mid \texttt{populate}(v, E)$	populate (spore)
$\mid \texttt{map}(E, t)$	map (ref)
$\mid \texttt{map}(v, E)$	map (fun)
$\mid \texttt{flatMap}(E, t)$	flatMap (ref)
$\mid \texttt{flatMap}(v, E)$	flatMap (fun)
$\mid \texttt{persist}(E)$	persist
$\mid \texttt{send}(E)$	send
$\mid \texttt{await}(E)$	await

Figure 2: Evaluation context.

tion contexts shown in Figure 2. Evaluation contexts capture the notion of the "next subterm to be evaluated." Following a standard approach [?], we write $E[t]$ for the term obtained by replacing the hole in evaluation context E with term t.

Figure 3 shows the rules for sequential reduction. The sequential reduction relation has the form $E[t] \mid \mu \rightarrow^h E[t'] \mid \mu'$ with stores μ and μ'. Stores are required for the dynamic allocation of futures and hosts. A store μ is a partial function mapping locations ι to values v. The annotation with host h is used for creating *decentralized identifiers* $\omega = (h, i)$ for silo references. Rules R-AppAbs and R-ProjRcd are completely standard. Analogous to rule R-AppAbs, rule R-AppSpore describes the application of a spore value to an argument value. Rule R-Await reduces $\texttt{await}(\iota)$ to v if future ι is already completed with v in μ.

Rules R-Map, R-FMap, R-Persist and R-Unpersist describe the creation of lineages. Rules R-Map and R-FMap create silo reference values using the constructors *Mapped* and *FMapped*, respectively. The new silo reference has a fresh identifier (h, i) which uniquely identifies the corresponding (logical) silo. In each case, the spore value p is stored in the new silo reference; this enables a materialization of the silo identified by (h, i) using parent silo reference r and spore p. Rules R-Persist and R-Unpersist create silo reference values using the *Persist* constructor. *Persist* contains a function enabling host h to persist $(\cdot \cup \cdot)$ or unpersist $(\cdot \setminus \cdot)$ silo r, respectively.

The deterministic reduction relation has the form $(E[t], \mu, Q, S)^h \longrightarrow (E[t'], \mu', Q', S')^h$ where Q is a *message queue* and S is a *silo store*. Figure 4 shows the definition of message queues. A message queue Q may contain three kinds of messages. A message of the form $\mathrm{Req}_s(h, r, \omega)$ requests the value of silo r to be sent to host h for materialization of identifier ω. A message of the form $\mathrm{Res}_s(\omega, v, P)$ represents the corresponding response, containing the identifier ω to be materialized, value v, and persist set P (the set of hosts which have persisted the

R-AppAbs
$$E[((x : T) \Rightarrow t)\ v'] \mid \mu \to^h E[[x \mapsto v']t] \mid \mu$$

R-ProjRcd
$$E[\{\overline{l_i = v_i}^{i \in 1..n}\}.l_j] \mid \mu \to^h E[v_j] \mid \mu$$

R-AppSpore
$$E[(\texttt{spore}\ \{\ \overline{x : T = v}\ ; (x : T) \Rightarrow t\ \})\ v'] \mid \mu \to^h E[\overline{[x \mapsto v]}[x \mapsto v']t] \mid \mu$$

R-Await
$$\frac{\mu(\iota) = \text{Some}(v)}{E[\texttt{await}(\iota)] \mid \mu \to^h E[v] \mid \mu}$$

R-Map
$$\frac{r' = \text{Mapped}((h,i),r,p) \quad i \text{ fresh}}{E[\texttt{map}(r,p)] \mid \mu \to^h E[r'] \mid \mu'}$$

R-FMap
$$\frac{r' = \text{FMapped}((h,i),r,p) \quad i \text{ fresh}}{E[\texttt{flatMap}(r,p)] \mid \mu \to^h E[r'] \mid \mu'}$$

R-Persist
$$\frac{r' = \text{Persist}((h,i),r,\cdot \cup \cdot) \quad i \text{ fresh}}{E[\texttt{persist}(r)] \mid \mu \to^h E[r'] \mid \mu'}$$

R-Unpersist
$$\frac{r' = \text{Persist}((h,i),r,\cdot \setminus \cdot) \quad i \text{ fresh}}{E[\texttt{unpersist}(r)] \mid \mu \to^h E[r'] \mid \mu'}$$

Figure 3: Sequential reduction.

$Q ::=$		*message queue values:*
	ϵ	empty queue
	$\mid \text{Req}_s(h, r, \omega) :: Q$	request (silo)
	$\mid \text{Res}_s(\omega, v, P) :: Q$	response (silo)
	$\mid \text{Req}_\iota(\iota, \omega) :: Q$	request (future)

Figure 4: Message queues.

silo identified by ω). A message of the form $\mathrm{Req}_\iota(\iota, \omega)$ requests future ι to be completed with the value of silo ω. A *silo store* S is a partial function mapping identifiers ω to values of the form $(\mathrm{Val}(v), P)$ or $(\mathrm{Fwd}(r), P)$ where P is a set of hosts which have persisted the silo (the persist set). The former represents a materialized silo with value v. The latter represents a *proxy* forwarding to the silo specified by lineage r.

The deterministic reduction rules use helper functions *host*, *id*, *parent*, and *consume*, which are defined as follows:

Definition 1.1 (Host)**.** *The host of a silo reference.*

$$host(r) := \begin{cases} h & \text{if } r = Mat((h,i)) \\ host(r') & \text{if } r = Mapped(\_, r', \_) \\ host(r') & \text{if } r = FMapped(\_, r', \_) \\ host(r') & \text{if } r = Persist(\_, r', \_) \end{cases}$$

Definition 1.2 (Silo reference identifier)**.** *The identifier of a silo reference.*

$$id(r) := \begin{cases} \omega & \text{if } r = Mat(\omega) \\ \omega & \text{if } r = Mapped(\omega, r', \_) \\ \omega & \text{if } r = FMapped(\omega, r', \_) \\ \omega & \text{if } r = Persist(\omega, r', \_) \end{cases}$$

Definition 1.3 (Silo reference parent)**.** *The parent of a silo reference.*

$$parent(r) := \begin{cases} None & \text{if } r = Mat(\_) \\ Some(r') & \text{if } r = Mapped(\_, r', \_) \\ Some(r') & \text{if } r = FMapped(\_, r', \_) \\ Some(r') & \text{if } r = Persist(\_, r', \_) \end{cases}$$

Definition 1.4 (Consume silo)**.** *Consume silo ω with persist set P in silo store S.*

$$consume(\omega, P, S) := \begin{cases} S - \omega & \text{if } P = \emptyset \\ S & \text{otherwise} \end{cases}$$

We discuss the deterministic reduction rules in two steps. First, we discuss the rules shown in Figure 5. Rule R-Seq reduces host $(E[t], \mu, Q, S)^h$ in case $E[t]$ reduces in μ. Rule R-Send1Local reduces `send`(r) to a completed future ι if the corresponding silo is already materialized in silo store S. Rule R-Send2Local covers the case where the requested silo is not yet materialized. In this case, two request messages are added to the queue: a first message $\mathrm{Req}_s(h, r, id(r))$ requesting the materialization of silo $id(r)$, and a second message requesting the value of silo $id(r)$ for completing future ι. Rule R-ReqF1 processes a message $\mathrm{Req}_\iota(\iota, \omega)$ by completing future ι with the value of the materialized silo ω. Rule R-ReqF2 delays such a request in case silo ω is not yet materialized by moving the request to the back of the queue.

Figure 6 shows the remaining deterministic reduction rules. Rule R-Res processes a message $\mathrm{Res}_s(\omega, v, P)$ by materializing silo ω with value v, yielding silo store S'. Rules R-Req1Local and R-Req2Local process a message $\mathrm{Req}_s(h, r, \omega)$ where silo store S forwards $id(r)$ to another silo $id(r')$. Rules R-ReqMapLocal and R-ReqFMapLocal evaluate a silo reference containing *Mapped* or *FMapped*, respectively, in case the parent silo reference is materialized. In both cases, spore value p, stored in r, is applied to the value of the parent silo. In case of R-ReqMapLocal, the silo store is updated with the materialization result v'. In case of R-ReqFMapLocal, the silo store

R-SEQ

$$\frac{E[t] \mid \mu \rightarrow^h E[t'] \mid \mu'}{(E[t], \mu, Q, S)^h \longrightarrow (E[t'], \mu', Q, S)^h}$$

R-SEND1LOCAL

$$\frac{\mathit{host}(r) = h \quad S(\mathit{id}(r)) = (\mathrm{Val}(v), P) \qquad \iota \text{ fresh} \quad \mu' = [\iota \mapsto \mathrm{Some}(v)]\mu}{(E[\mathtt{send}(r)], \mu, Q, S)^h \longrightarrow (E[\iota], \mu', Q, S)^h}$$

R-SEND2LOCAL

$$\frac{\mathit{host}(r) = h \quad \mathit{id}(r) \notin \mathit{dom}(S) \qquad \iota \text{ fresh} \quad \mu' = [\iota \mapsto \mathrm{None}]\mu}{(E[\mathtt{send}(r)], \mu, Q, S)^h \longrightarrow (E[\iota], \mu', Q \cdot \mathrm{Req}_s(h, r, \mathit{id}(r)) \cdot \mathrm{Req}_\iota(\iota, \mathit{id}(r)), S)^h}$$

R-REQF1

$$\frac{\begin{array}{c} Q = \mathrm{Req}_\iota(\iota, \omega) :: Q' \quad S(\omega) = (\mathrm{Val}(v), P) \\ S' = \mathrm{consume}(\omega, P, S) \quad \mu' = [\iota \mapsto \mathrm{Some}(v)]\mu \end{array}}{(E[\mathtt{await}(\iota')], \mu, Q, S)^h \longrightarrow (E[\mathtt{await}(\iota')], \mu', Q', S')^h}$$

R-REQF2

$$\frac{Q = \mathrm{Req}_\iota(\iota, \omega) :: Q' \quad \omega \notin \mathit{dom}(S)}{(E[\mathtt{await}(\iota')], \mu, Q, S)^h \longrightarrow (E[\mathtt{await}(\iota')], \mu, Q' \cdot \mathrm{Req}_\iota(\iota, \omega), S)^h}$$

Figure 5: Deterministic reduction (future).

is updated with a forwarding reference to r'', the result of the spore application. Finally, the parent silo $\mathit{id}(r')$ is consumed (removed from silo store S'') in case the persist set P is empty, which means that $\mathit{id}(r')$ was not persisted. Rule R-REQPERSISTLOCAL materializes silo ω' under a persist set P' which is obtained by modifying the persist set P of parent silo $\mathit{id}(r')$ according to the operator $\star$ stored in r. Rule R-REQPARENTLOCAL enqueues a materialization request $\mathrm{Req}_s(h, r', \mathit{id}(r'))$ in case the parent $\mathit{id}(r')$ of a requested silo $\mathit{id}(r)$ is not materialized yet.

Figure 7 shows the non-deterministic reduction rules. The non-deterministic reduction relation has the form $H \twoheadrightarrow H'$ where H and H' are sets of hosts of the form $(t, \mu, Q, S)^h$. Rule R-SCHEDULE reduces a host chosen non-deterministically from the set of hosts. Rule R-SPAWN creates a new host whose initial term is given by the application of the provided spore to the unit value $\{\}$. The new host has an empty store, an empty queue, and an empty silo store. Rule R-POPULATE materializes a silo with a fresh identifier ω on host h' using value v. Rules R-REQ1-3 and R-SEND are analogous to the corresponding deterministic reduction rules. The main difference is that messages are exchanged between different hosts in the case of non-deterministic reduction.

1.2 Type Assignment

Type assignment is based on a judgment of the form $\Gamma; \Sigma; \Delta \vdash t : T$ which assigns term t type T. Γ is a standard type environment; Σ is a standard store typing; Δ is a *silo store typing* which is new. Δ maps identifiers ω to types, R-Res

$$\frac{Q = \mathrm{Res}_s(\omega, v, P) :: Q' \qquad S' = [\omega \mapsto (\mathrm{Val}(v), P)]S}{(E[\texttt{await}(\iota)], \mu, Q, S)^h \longrightarrow (E[\texttt{await}(\iota)], \mu, Q', S')^h}$$

R-Req1Local

$$\frac{Q = \mathrm{Req}_s(h, r, \omega) :: Q' \quad S(id(r)) = (\mathrm{Fwd}(r'), P) \quad S(id(r')) = (\mathrm{Val}(v), P')}{(E[\texttt{await}(\iota)], \mu, Q, S)^h \longrightarrow (E[\texttt{await}(\iota)], \mu, Q' \cdot \mathrm{Res}_s(\omega, v, P), S)^h}$$

R-Req2Local

$$\frac{Q = \mathrm{Req}_s(h, r, \omega) :: Q' \quad S(id(r)) = (\mathrm{Fwd}(r'), P) \quad id(r') \notin dom(S)}{(E[\texttt{await}(\iota)], \mu, Q, S)^h \longrightarrow (E[\texttt{await}(\iota)], \mu, Q' \cdot \mathrm{Req}_s(h, r', \omega), S)^h}$$

R-ReqMapLocal

$$\frac{\begin{array}{c} Q = \mathrm{Req}_s(h', r, \omega) :: Q' \quad r = \mathrm{Mapped}(\omega', r', p) \quad S(id(r')) = (\mathrm{Val}(v), P) \\ v' = p(v) \quad S' = [\omega' \mapsto (\mathrm{Val}(v'), \emptyset)]S \quad S'' = \mathrm{consume}(id(r'), P, S') \end{array}}{(E[\texttt{await}(\iota)], \mu, Q, S)^h \longrightarrow (E[\texttt{await}(\iota)], \mu, Q' \cdot \mathrm{Req}_s(h', r, \omega), S'')^h}$$

R-ReqFMapLocal

$$\frac{\begin{array}{c} Q = \mathrm{Req}_s(h', r, \omega) :: Q' \quad r = \mathrm{FMapped}(\omega', r', p) \quad S(id(r')) = (\mathrm{Val}(v), P) \\ r'' = p(v) \quad S' = [\omega' \mapsto (\mathrm{Fwd}(r''), \emptyset)]S \quad S'' = \mathrm{consume}(id(r'), P, S') \end{array}}{(E[\texttt{await}(\iota)], \mu, Q, S)^h \longrightarrow (E[\texttt{await}(\iota)], \mu, Q' \cdot \mathrm{Req}_s(h', r'', \omega), S'')^h}$$

R-ReqPersistLocal

$$\frac{\begin{array}{c} Q = \mathrm{Req}_s(h', r, \omega) :: Q' \quad r = \mathrm{Persist}(\omega', r', \star) \quad \omega' = (h'', i) \quad S(id(r')) = (\mathrm{Val}(v), P) \\ P' = P \star \{h''\} \quad S' = [\omega' \mapsto (\mathrm{Val}(v), P')]S \quad S'' = \mathrm{consume}(id(r'), P, S') \end{array}}{(E[\texttt{await}(\iota)], \mu, Q, S)^h \longrightarrow (E[\texttt{await}(\iota)], \mu, Q' \cdot \mathrm{Res}_s(\omega, v, P'), S'')^h}$$

R-ReqParentLocal

$$\frac{Q = \mathrm{Req}_s(h', r, \omega) :: Q' \quad \mathrm{Some}(r') = parent(r) \quad id(r') \notin dom(S)}{(E[\texttt{await}(\iota)], \mu, Q, S)^h \longrightarrow (E[\texttt{await}(\iota)], \mu, Q' \cdot \mathrm{Req}_s(h, r', id(r')) \cdot \mathrm{Req}_s(h', r, \omega), S)^h}$$

Figure 6: Deterministic reduction (silo).

R-Schedule

$$\frac{(t, \mu, Q, S)^h \to (t', \mu', Q', S')^h}{\{(t, \mu, Q, S)^h\} \cup H \twoheadrightarrow \{(t', \mu', Q', S')^h\} \cup H}$$

R-Spawn

$$\frac{h' \text{ fresh} \quad \iota \text{ fresh} \quad \mu' = [\iota \mapsto h']\mu}{\begin{array}{c}\{(E[\texttt{spawn}(\texttt{spore} \ \{\ \overline{x : T = v}\ ; (x : T) \Rightarrow t\ \})], \mu, Q, S)^h\} \cup H \\ \twoheadrightarrow \{(E[\iota], \mu', Q, S)^h, ((\texttt{spore} \ \{\ \overline{x : T = v}\ ; (x : T) \Rightarrow t\ \})\ \{\}, \epsilon, \epsilon, \epsilon)^{h'}\} \cup H\end{array}}$$

R-Populate

$$\frac{\mu(\iota) = h' \quad S'' = [\omega \mapsto (\mathrm{Val}(v), \emptyset)]S' \quad \omega = (h', i) \quad i \text{ fresh}}{\{(E[\texttt{populate}(\iota, v)], \mu, Q, S)^h, (t', \mu', Q', S')^{h'}\} \cup H \twoheadrightarrow \{(E[\mathrm{Mat}(\omega)], \mu, Q, S)^h, (t', \mu', Q', S'')^{h'}\} \cup H}$$

R-Req1

$$\frac{Q = \mathrm{Req}_s(h', r, \omega) :: Q'' \quad S(id(r)) = (\mathrm{Val}(v), P) \quad m = \mathrm{Res}_s(\omega, v, P)}{\{(E[\texttt{await}(\iota)], \mu, Q, S)^h, (t', \mu', Q', S')^{h'}\} \cup H \twoheadrightarrow \{(E[\texttt{await}(\iota)], \mu, Q'', S)^h, (t', \mu', Q' \cdot m, S')^{h'}\} \cup H}$$

R-Req2

$$\frac{\begin{array}{c}Q = \mathrm{Req}_s(h', r, \omega) :: Q'' \quad S(id(r)) = (\mathrm{Fwd}(r'), P) \\ S(id(r')) = (\mathrm{Val}(v), P') \quad m = \mathrm{Res}_s(\omega, v, P)\end{array}}{\{(E[\texttt{await}(\iota)], \mu, Q, S)^h, (t', \mu', Q', S')^{h'}\} \cup H \twoheadrightarrow \{(E[\texttt{await}(\iota)], \mu, Q'', S)^h, (t', \mu', Q' \cdot m, S')^{h'}\} \cup H}$$

R-Req3

$$\frac{\begin{array}{c}Q = \mathrm{Req}_s(h'', r, \omega) :: Q'' \quad S(id(r)) = (\mathrm{Fwd}(r'), P) \\ id(r') \notin dom(S) \quad h' = host(r') \quad m = \mathrm{Req}_s(h'', r', \omega)\end{array}}{\{(E[\texttt{await}(\iota)], \mu, Q, S)^h, (t', \mu', Q', S')^{h'}\} \cup H \twoheadrightarrow \{(E[\texttt{await}(\iota)], \mu, Q'', S)^h, (t', \mu', Q' \cdot m, S')^{h'}\} \cup H}$$

R-Send

$$\frac{host(r) = h' \quad h' \neq h \quad m = \mathrm{Req}_s(h, r, id(r)) \quad \iota \text{ fresh} \quad \mu'' = [\iota \mapsto \mathrm{None}]\mu}{\{(E[\texttt{send}(r)], \mu, Q, S)^h, (t', \mu', Q', S')^{h'}\} \cup H \twoheadrightarrow \{(E[\iota], \mu'', Q, S)^h, (t', \mu', Q' \cdot m, S')^{h'}\} \cup H}$$

Figure 7: Non-deterministic reduction.

$$\frac{x : T \in \Gamma}{\Gamma; \Sigma; \Delta \vdash x : T}\ \textsc{T-Var} \qquad \frac{\Sigma(\iota) : T}{\Gamma; \Sigma; \Delta \vdash \iota : T}\ \textsc{T-Loc} \qquad \frac{\Gamma, x : T; \Sigma; \Delta \vdash t : T'}{\Gamma; \Sigma; \Delta \vdash ((x : T) \Rightarrow t) : T \Rightarrow T'}\ \textsc{T-Abs}$$

$$\frac{\Gamma; \Sigma; \Delta \vdash t : T \Rightarrow T' \quad \Gamma; \Sigma; \Delta \vdash t' : T}{\Gamma; \Sigma; \Delta \vdash (t\ t') : T'}\ \textsc{T-App} \qquad \frac{\Gamma; \Sigma; \Delta \vdash \overline{t} : \overline{T}}{\Gamma; \Sigma; \Delta \vdash \{\overline{l = t}\} : \{\overline{l : T}\}}\ \textsc{T-Record}$$

$$\frac{\Gamma; \Sigma; \Delta \vdash t : \{\overline{l : T}\}}{\Gamma; \Sigma; \Delta \vdash t.l_i : T_i}\ \textsc{T-Select}$$

$$\frac{\Gamma; \Sigma; \Delta \vdash \overline{t} : \overline{T} \quad \overline{x : T}, x : T; \emptyset; \Delta \vdash t : T' \quad \forall T_i \in \overline{T}.\ \mathit{serializable}(T_i)}{\Gamma; \Sigma; \Delta \vdash (\texttt{spore}\ \{\ \overline{x : T = t}\ ; (x : T) \Rightarrow t\ \}) : T \Rightarrow T'\ \{\ \texttt{type}\ \mathcal{C} = \overline{T}\ \}}\ \textsc{T-Spore}$$

$$\frac{\Gamma; \Sigma; \Delta \vdash t : T \Rightarrow T'\ \{\ \texttt{type}\ \mathcal{C} = \overline{T}\ \} \quad \Gamma; \Sigma; \Delta \vdash t' : T}{\Gamma; \Sigma; \Delta \vdash (t\ t') : T'}\ \textsc{T-AppSpore}$$

$$\frac{\Gamma; \Sigma; \Delta \vdash t : (\{\} \Rightarrow T\ \{\ \texttt{type}\ \mathcal{C} = \overline{T}\ \})}{\Gamma; \Sigma; \Delta \vdash \texttt{spawn}(t) : \texttt{Host}}\ \textsc{T-Spawn}$$

$$\frac{\Gamma; \Sigma; \Delta \vdash t : \texttt{Host} \quad \Gamma; \Sigma; \Delta \vdash t' : T \quad \mathit{serializable}(T)}{\Gamma; \Sigma; \Delta \vdash \texttt{populate}(t, t') : \texttt{SiloRef}[T]}\ \textsc{T-Populate}$$

$$\frac{\Gamma; \Sigma; \Delta \vdash t : \texttt{SiloRef}[T] \quad \Gamma; \Sigma; \Delta \vdash t' : (T \Rightarrow T'\ \{\ \texttt{type}\ \mathcal{C} = \overline{T}\ \})}{\Gamma; \Sigma; \Delta \vdash \texttt{map}(t, t') : \texttt{SiloRef}[T']}\ \textsc{T-Map}$$

$$\frac{\Gamma; \Sigma; \Delta \vdash t : \texttt{SiloRef}[T] \quad \Gamma; \Sigma; \Delta \vdash t' : (T \Rightarrow \texttt{SiloRef}[T']\ \{\ \texttt{type}\ \mathcal{C} = \overline{T}\ \})}{\Gamma; \Sigma; \Delta \vdash \texttt{flatMap}(t, t') : \texttt{SiloRef}[T']}\ \textsc{T-FMap}$$

$$\frac{\Gamma; \Sigma; \Delta \vdash t : \texttt{SiloRef}[T]}{\Gamma; \Sigma; \Delta \vdash \texttt{persist}(t) : \texttt{SiloRef}[T]}\ \textsc{T-Persist} \qquad \frac{\Gamma; \Sigma; \Delta \vdash t : \texttt{SiloRef}[T]}{\Gamma; \Sigma; \Delta \vdash \texttt{send}(t) : \texttt{Future}[T]}\ \textsc{T-Send}$$

$$\frac{\Delta(\mathit{id}(r)) = T \quad \Delta \vdash r}{\Gamma; \Sigma; \Delta \vdash r : \texttt{SiloRef}[T]}\ \textsc{T-SiloRef} \qquad \frac{\Gamma; \Sigma; \Delta \vdash t : \texttt{Future}[T]}{\Gamma; \Sigma; \Delta \vdash \texttt{await}(t) : T}\ \textsc{T-Await}$$

Figure 8: Type assignment.

S-Record
$$\frac{\forall T_i \in \overline{T}.\ \mathit{serializable}(T_i)}{\mathit{serializable}(\{\overline{l : T}\})}$$

S-Spore
$$\frac{\forall T_i \in \overline{T}.\ \mathit{serializable}(T_i)}{\mathit{serializable}(T \Rightarrow T'\ \{\ \texttt{type}\ \mathcal{C} = \overline{T}\ \})}$$

S-SiloRef
$$\mathit{serializable}(\texttt{SiloRef}[T])$$

Figure 9: Serializable types.

thereby providing a typing for silo stores S. Figure 8 shows the rules for type assignment. Rules T-Var, T-Loc, T-Abs, T-App, T-Record, and T-Select are unchanged compared to a standard typed lambda calculus with records [?].

Rule T-Spore assigns a type to spore literals. Importantly, the body of the spore's closure, t, must be well-typed in a type environment containing only the closure parameter x and the variables $\overline{x}$ in the spore's header, as well as an empty store typing. Furthermore, the types of captured variables must be serializable. The predicate *serializable* is defined in Figure 9. These constraints ensure that spore values are always independent of the environment and store of the creating host. This independence is expressed by the following theorem:

Theorem 1.1. (Serializable Values) *If* $\Gamma; \Sigma; \Delta \vdash v : T$ *and* $\mathit{serializable}(T)$ *then* $\emptyset; \emptyset; \Delta \vdash v : T$.

Proof. By induction on the derivation of $\Gamma; \Sigma; \Delta \vdash v : T$. See Appendix **??**. ☐

Rule T-AppSpore is analogous to rule T-App. Rule T-Spawn requires argument t to be a spore with domain type unit; the result has type `Host`. Rule T-Populate leverages the *serializable* predicate to ensure the value of the silo to be populated is independent of its source context. Rules T-Map, T-FMap, and T-Persist are straightforward; note that `map` and `flatMap` are polymorphic in the types of the captured variables of their spore argument types. Rules T-Send and T-Await are entirely unsurprising. Rule T-SiloRef is the only rule that uses the silo store typing Δ. Analogous to rule T-Loc, the type of silo $id(r)$ is looked up in Δ. Furthermore, T-SiloRef requires r to be well-formed in Δ, written $\Delta \vdash r$ (see below).

1.3 Well-Formed Configurations

Figure 10 shows the rules for well-formed configurations. These rules are essential for establishing subject reduction (see Section 2). Rules WF-Store1 and WF-Store2 are standard. Rules WF-Ref1-2 require the types given by the silo store typing Δ to be consistent with the corresponding type of spore p. Rule WF-Ref3 requires the type of silo ω to be equal to the type of its parent silo $id(r)$ in silo store typing Δ. Rule WF-Ref4 requires Δ to be defined for the identifier of a materialized silo. Finally, rules WF-Ref1-3 require parent silo references to be well-formed. Rules WF-SiloStore1-3 require a well-formed silo store to be consistent with silo store typing Δ. Rules WF-Q1-4 specify well-formedness of message queues in Δ and Σ. Rules WF-HostConfig, WF-Host1, and WF-Host2 combine the previous rules in the expected way.

WF-Store1
$$\emptyset \vdash \emptyset$$

WF-Store2
$$\frac{\Sigma \vdash \mu}{[\iota \mapsto T]\Sigma \vdash [\iota \mapsto v]\mu}$$

WF-Ref1
$$\frac{\Delta(\omega) = T \quad \Delta(id(r)) = T' \quad \exists \Gamma, \Sigma.\ \Gamma; \Sigma; \Delta \vdash p : T' \Rightarrow T\ \{\ldots\} \quad \Delta \vdash r}{\Delta \vdash \text{Mapped}(\omega, r, p)}$$

WF-Ref2
$$\frac{\Delta(\omega) = T \quad \Delta(id(r)) = T' \quad \exists \Gamma, \Sigma.\ \Gamma; \Sigma; \Delta \vdash p : T' \Rightarrow \texttt{SiloRef}[T]\ \{\ldots\} \quad \Delta \vdash r}{\Delta \vdash \text{FMapped}(\omega, r, p)}$$

WF-Ref3
$$\frac{\Delta(\omega) = T \quad \Delta(id(r)) = T \quad \Delta \vdash r}{\Delta \vdash \text{Persist}(\omega, r, \star)}$$

WF-Ref4
$$\frac{\omega \in dom(\Delta)}{\Delta \vdash \text{Mat}(\omega)}$$

WF-SiloStore1
$$\Delta \vdash \emptyset$$

WF-SiloStore2
$$\frac{\Delta(\omega) = T \quad \emptyset; \emptyset; \Delta \vdash v : T \quad \Delta \vdash S}{\Delta \vdash [\omega \mapsto (\text{Val}(v), P)]S}$$

WF-SiloStore3
$$\frac{\Delta(id(r)) = \Delta(\omega) \quad \Delta \vdash r \quad \Delta \vdash S}{\Delta \vdash [\omega \mapsto (\text{Fwd}(r), P)]S}$$

WF-Q1
$$\Delta; \Sigma \vdash \epsilon$$

WF-Q2
$$\frac{\Delta(\omega) = T \quad \Sigma(\iota) = \texttt{Future}[T] \quad \Delta; \Sigma \vdash Q}{\Delta; \Sigma \vdash \text{Req}_\iota(\iota, \omega) :: Q}$$

WF-Q3
$$\frac{\Delta(\omega) = T \quad \emptyset; \emptyset; \Delta \vdash v : T \quad \Delta; \Sigma \vdash Q}{\Delta; \Sigma \vdash \text{Res}_s(\omega, v, P) :: Q}$$

WF-Q4
$$\frac{\Delta(id(r)) = \Delta(\omega) \quad \Delta \vdash r \quad \Delta; \Sigma \vdash Q}{\Delta; \Sigma \vdash \text{Req}_s(h, r, \omega) :: Q}$$

WF-HostConfig
$$\frac{\Sigma \vdash \mu \quad \Delta \vdash S \quad \Delta; \Sigma \vdash Q \quad \Gamma; \Sigma; \Delta \vdash t : T}{\Delta; \Sigma \vdash (t, \mu, Q, S)^h}$$

WF-Host1
$$\Delta \vdash \emptyset$$

WF-Host2
$$\frac{\exists \Sigma.\ \Delta; \Sigma \vdash (t, \mu, Q, S)^h \quad \Delta \vdash H}{\Delta \vdash \{(t, \mu, Q, S)^h\} \cup H}$$

Figure 10: Well-formedness.

2 Subject Reduction

This section establishes a subject reduction theorem for the presented core language. The complete proof is provided in the appendix; here, we restrict ourselves to summarizing the main results.

Lemma 2.1. (Substitution) *If* $\Gamma, x : T'; \Sigma; \Delta \vdash t : T$ *and* $\Gamma; \Sigma; \Delta \vdash v : T'$ *then* $\Gamma; \Sigma; \Delta \vdash [x \mapsto v]t : T$.

Proof. By induction on the derivation of $\Gamma, x : T'; \Sigma; \Delta \vdash t : T$. □

Lemma 2.2. (Queue Concatenation) *If* $\Delta; \Sigma \vdash Q$ *and* $\Delta; \Sigma \vdash Q'$ *then* $\Delta; \Sigma \vdash Q ::: Q'$.

Proof. By induction on the length of Q. See Appendix **??**. □

Theorem 2.1. (Subject Reduction)

1. *If* $\Gamma; \Sigma; \Delta \vdash t : T$, $\Sigma \vdash \mu$, *and* $t \mid \mu \to^h t' \mid \mu'$ *then* $\Gamma; \Sigma'; \Delta' \vdash t' : T$, *and* $\Sigma' \vdash \mu'$ *for some* $\Sigma' \supseteq \Sigma$ *and* $\Delta' \supseteq \Delta$.
2. *If* $\Delta; \Sigma \vdash (t, \mu, Q, S)^h$ *and* $(t, \mu, Q, S)^h \longrightarrow (t', \mu', Q', S')^h$ *then* $\Delta'; \Sigma' \vdash (t', \mu', Q', S')^h$ *for some* $\Delta' \supseteq \Delta$ *and* $\Sigma' \supseteq \Sigma$.
3. *If* $\Delta \vdash H$ *and* $H \twoheadrightarrow H'$ *then* $\Delta' \vdash H'$ *for some* $\Delta' \supseteq \Delta$.

Proof. Part 1: by induction on the derivation of $t \mid \mu \to^h t' \mid \mu'$. Part 2: by induction on the derivation of $(t, \mu, Q, S)^h \longrightarrow (t', \mu', Q', S')^h$. Part 3: by induction on the derivation of $H \twoheadrightarrow H'$. See Appendix **??** for the complete proof. □