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 ℓ 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 $T \Rightarrow T'$ { type $C = T$ } includes the types $T$ 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-
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-

Figure 1: Abstract syntax of core language.
evaluation contexts:

- hole
- application (fun)
- application (arg)
- record
- selection
- spore
- spawn
- populate (host)
- populate (spore)
- map (ref)
- map (fun)
- flatMap (ref)
- flatMap (fun)
- persist
- send
- await

Figure 2: Evaluation context.

tation 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 $E[t] \Downarrow E[t′]$ with stores $\text{store}$ and $\text{store}'$. Stores are required for the dynamic allocation of futures and hosts. A store $\text{store}$ is a partial function mapping locations $\text{location}$ to values $\text{value}$. The annotation with host $h$ is used for creating decentralized identifiers $\text{id} = (h; i)$ 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 $\text{await}(i)$ to $v$ if future $i$ is already completed with $v$ in $\mu$.

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 $(E[t], \text{mu}, Q, S)^h \rightarrow (E[t′], \text{mu}', Q′, S′)^h$ 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 $\text{Req}_v(h, r, \omega)$ requests the value of silo $r$ to be sent to host $h$ for materialization of identifier $\omega$. A message of the form $\text{Res}_v(\omega, v, \text{P})$ represents the corresponding response, containing the identifier $\omega$ to be materialized, value $v$, and persist set $\text{P}$ (the set of hosts which have persisted the
\[
R\text{-AppAbs}
\]
\[
E[((x : T) \Rightarrow t) v'] \mid \mu \rightarrow^h E[[x \mapsto v']t] \mid \mu
\]

\[
R\text{-ProjRcd}
\]
\[
E[[t_i = v_i^{1..n}] \cdot i] \mid \mu \rightarrow^h E[v_i] \mid \mu
\]

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

\[
R\text{-Await}
\]
\[
\mu(i) = \text{Some}(v)
\]
\[
E[\text{await}(i) \mid \mu \rightarrow^h E[v] \mid \mu
\]

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

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

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

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

Figure 3: Sequential reduction.

\[
Q ::= \epsilon \mid \text{message queue values:}
\]
\[
\mid \text{empty queue}
\]
\[
\mid \text{Req}(h, r, \omega) ::= Q \quad \text{request (silo)}
\]
\[
\mid \text{Res}(\omega, r, P) ::= Q \quad \text{response (silo)}
\]
\[
\mid \text{Req}(h, \omega) ::= Q \quad \text{request (future)}
\]

Figure 4: Message queues.
Deﬁnition 1.1 (Host). The host of a silo reference.

\[ host(r) := \begin{cases} 
    h & \text{if } r = \text{Mat}(h, i) \\
    host'(r') & \text{if } r = \text{Mapped}(\omega, r', \_) \\
    host'(r') & \text{if } r = \text{Map}(\omega, r', \_) \\
    host'(r') & \text{if } r = \text{Persist}(\omega, r', \_) 
\end{cases} \]

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

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

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

\[ parent(r) := \begin{cases} 
    \text{None} & \text{if } r = \text{Mat}(\_) \\
    \text{Some}(r') & \text{if } r = \text{Mapped}(\omega, r', \_) \\
    \text{Some}(r') & \text{if } r = \text{Map}(\omega, r', \_) \\
    \text{Some}(r') & \text{if } r = \text{Persist}(\omega, r', \_) 
\end{cases} \]

Deﬁnition 1.4 (Consume silo). Consume silo \( \omega \) 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 \( R-\text{SEQ} \) reduces host \( (E[t], \mu, Q, S)^h \) in case \( E[t] \) reduces in \( \mu \). Rule \( R-\text{SEND1LOCAL} \) reduces \( \text{send}(r) \) to a completed future \( i \) if the corresponding silo is already materialized in silo store \( S \). Rule \( R-\text{SEND2LOCAL} \) 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 \( \text{Req}_i(h, r, id(r)) \) requesting the materialization of silo \( i \) \( id(r) \), and a second message requesting the value of silo \( i \) \( id(r) \) for completing future \( i \). Rule \( R-\text{REQF1} \) processes a message \( \text{Req}_i(i, \omega) \) by completing future \( i \) with the value of the materialized silo \( \omega \). Rule \( R-\text{REQF2} \) delays such a request in case silo \( \omega \) is not yet materialized by moving the request to the back of the queue.

Figure 6 shows the remaining deterministic reduction rules. Rule \( R-\text{RES} \) processes a message \( \text{Res}_i(\omega, v, P) \) by materializing silo \( \omega \) with value \( v \), yielding silo store \( S' \). Rules \( R-\text{REQ1LOCAL} \) and \( R-\text{REQ2LOCAL} \) process a message \( \text{Req}_i(h, r, \omega) \) where silo store \( S \) forwards \( id(r) \) to another silo \( id'(r') \). Rules \( R-\text{REQMAPLOCAL} \) and \( R-\text{REQMAPLOCAL} \) evaluate a silo reference containing \( \text{Map} \) or \( \text{Map} \), 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 \( R-\text{REQMAPLOCAL} \), the silo store is updated with the materialization result \( v' \). In case of \( R-\text{REQMAPLOCAL} \), the silo store
is updated with a forwarding reference to \(r''\), the result of the spore application. Finally, the parent silo \(id(r')\) is consumed (removed from silo store \(S''\)) in case the persist set \(P\) is empty, which means that \(id(r')\) was not persisted. Rule R-ReqPersistLocal materializes silo \(\mu'\) under a persist set \(P'\) which is obtained by modifying the persist set \(P\) of parent silo \(id(r')\) according to the operator \(\star\) stored in \(r\). Rule R-ReqParentLocal enqueues a materialization request \(\text{Req}(h, r, id(r))\) in case the parent \(id(r')\) of a requested silo \(id(r)\) is not materialized yet.

Figure 7 shows the non-deterministic reduction rules. The non-deterministic reduction relation has the form \(H \rightarrow H'\) where \(H\) and \(H'\) are sets of hosts of the form \((t; ; Q; S)\). 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 \(v\). 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 \(\nu\) 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 \(T\). \(\Gamma\) is a standard type environment; \(\Sigma\) is a standard store typing; \(\Delta\) is a silo store typing which is new. \(\Delta\) maps identifiers \(\omega\) to types,
\[\text{R-ReqPersistLocal} \]
\[Q = \text{Req}_s(h', r, \omega) :: Q' \quad S' = [\omega \mapsto (\text{Val}(v), P)]S \]
\[(E[\text{await}(i)], \mu, Q, S)^h \rightarrow (E[\text{await}(i)], \mu, Q', S')^h\]

\[\text{R-ReqLocal} \]
\[Q = \text{Req}_s(h, r, \omega) :: Q' \quad S(id(r)) = (\text{Fwd}(r'), P) \quad S(id(r')) = (\text{Val}(v), P') \]
\[(E[\text{await}(i)], \mu, Q, S)^h \rightarrow (E[\text{await}(i)], \mu, Q', S')^h\]

\[\text{R-ReqLocal} \]
\[Q = \text{Req}_s(h, r, \omega) :: Q' \quad S(id(r)) = (\text{Fwd}(r'), P) \quad id(r') \notin \text{dom}(S) \]
\[(E[\text{await}(i)], \mu, Q, S)^h \rightarrow (E[\text{await}(i)], \mu, Q', S')^h\]

\[\text{R-ReqMapLocal} \]
\[Q = \text{Req}_s(h', r, \omega) :: Q' \quad r = \text{Mapped}(\omega', r', p) \quad S(id(r')) = (\text{Val}(v), P) \]
\[v' = p(v) \quad S' = [\omega' \mapsto (\text{Val}(v'), \emptyset)]S \quad S'' = \text{consume}(id(r'), P, S') \]
\[(E[\text{await}(i)], \mu, Q, S)^h \rightarrow (E[\text{await}(i)], \mu, Q', S')^h\]

\[\text{R-ReqPersistLocal} \]
\[Q = \text{Req}_s(h', r, \omega) :: Q' \quad r = \text{Persist}(\omega', r', *) \quad \omega' = (h'', i) \quad S(id(r')) = (\text{Val}(v), P) \]
\[P' = P * [h'] \quad S' = [\omega' \mapsto (\text{Val}(v), P')]S \quad S'' = \text{consume}(id(r'), P, S') \]
\[(E[\text{await}(i)], \mu, Q, S)^h \rightarrow (E[\text{await}(i)], \mu, Q', S')^h\]

\[\text{R-ReqPersistLocal} \]
\[Q = \text{Req}_s(h', r, \omega) :: Q' \quad \text{Some}(r') = \text{parent}(r) \quad id(r') \notin \text{dom}(S) \]
\[(E[\text{await}(i)], \mu, Q, S)^h \rightarrow (E[\text{await}(i)], \mu, Q', \text{Req}_s(h, r', id(r'))) \cdot \text{Req}_s(h', r, \omega), S)^h\]

Figure 6: Deterministic reduction (silo).
\[
\text{R-Schedule} \\
(t, \mu, Q, S)^h \rightarrow (t', \mu', Q', S')^h \\
\{(t, \mu, Q, S)^h\} \cup H \rightarrow \{(t', \mu', Q', S')^h\} \cup H
\]

\[
\text{R-Spawn} \\
h' \text{ fresh} \quad t \text{ fresh} \quad \mu' = [t \mapsto h'] \mu \\
\{(E[\text{spawn}(\text{spore} \{ x : T = \nu ; (x : T) \Rightarrow t \})], \mu, Q, S)^h\} \cup H \\
\rightarrow ((E[i], \mu', Q, S)^h, (\text{spore} \{ x : T = \nu ; (x : T) \Rightarrow t \}) \{\}, e, e)^{h'} \cup H
\]

\[
\text{R-Populate} \\
\mu(i) = h' \quad S'' = [\omega \mapsto (\text{Val}(v), \emptyset)]S' \quad \omega = (h', i) \quad i \text{ fresh} \\
\{(E[\text{populate}(i, v)], \mu, Q, S)^h, (t', \mu', Q', S')^{h'}\} \cup H \rightarrow \{(E[\text{Mat}(\omega)], \mu, Q, S)^h, (t', \mu', Q', S'')^{h'}\} \cup H
\]

\[
\text{R-Req1} \\
Q = \text{Req_x}(h', r, \omega) :: Q'' \quad S(id(r)) = (\text{Val}(v), P) \quad m = \text{Res}_x(\omega, v, P) \\
\{(E[\text{await}(i)], \mu, Q, S)^h, (t', \mu', Q', S')^{h'}\} \cup H \rightarrow \{(E[\text{await}(i)], \mu, Q'', S)^h, (t', \mu', Q' \cdot m, S'')^{h'}\} \cup H
\]

\[
\text{R-Req2} \\
Q = \text{Req_x}(h', r, \omega) :: Q'' \quad S(id(r)) = (\text{Fwd}(r'), P) \quad m = \text{Res}_x(\omega, v, P) \\
\{(E[\text{await}(i)], \mu, Q, S)^h, (t', \mu', Q', S')^{h'}\} \cup H \rightarrow \{(E[\text{await}(i)], \mu, Q'', S)^h, (t', \mu', Q' \cdot m, S'')^{h'}\} \cup H
\]

\[
\text{R-Req3} \\
Q = \text{Req_x}(h''', r, \omega) :: Q'''' \quad S(id(r)) = (\text{Fwd}(r'), P) \\
id(r') \notin \text{dom}(S) \quad h' = \text{host}(r') \quad m = \text{Req_x}(h''', r', \omega) \\
\{(E[\text{await}(i)], \mu, Q, S)^h, (t', \mu', Q', S')^{h'}\} \cup H \rightarrow \{(E[\text{await}(i)], \mu, Q'', S)^h, (t', \mu', Q' \cdot m, S'')^{h'}\} \cup H
\]

\[
\text{R-Send} \\
host(r) = h' \quad h' \neq h \quad m = \text{Req_x}(h, r, id(r)) \quad t \text{ fresh} \quad \mu'' = [t \mapsto \text{None}]\mu \\
\{(E[\text{send}(r)], \mu, Q, S)^h, (t', \mu', Q', S')^{h'}\} \cup H \rightarrow \{(E[i], \mu'', Q, S)^h, (t', \mu', Q' \cdot m, S'')^{h'}\} \cup H
\]

Figure 7: Non-deterministic reduction.
\[
\begin{align*}
\text{T-Var} & \quad x : T \in \Gamma \\
\Gamma; \Sigma; \Delta \vdash x : T \\
\text{T-Loc} & \quad \Sigma(t) : T \\
\Gamma; \Sigma; \Delta \vdash t : T \\
\text{T-Abs} & \quad \Gamma, x : T; \Sigma; \Delta \vdash t : T' \\
\Gamma; \Sigma; \Delta \vdash ((x : T) \Rightarrow t) : T \Rightarrow T' \\
\text{T-App} & \quad \Gamma; \Sigma; \Delta \vdash t : T \Rightarrow T' \\
\Gamma; \Sigma; \Delta \vdash t' : T \\
\Gamma; \Sigma; \Delta \vdash (t \, t') : T' \\
\text{T-Record} & \quad \Gamma; \Sigma; \Delta \vdash I : T \\
\Gamma; \Sigma; \Delta \vdash \{I = t\} : \{I : T\} \\
\text{T-Select} & \quad \Gamma; \Sigma; \Delta \vdash t : \{I : T\} \\
\Gamma; \Sigma; \Delta \vdash t.l_i : T_i \\
\text{T-Spore} & \quad \Gamma; \Sigma; \Delta \vdash t : T \\
\Gamma; \Sigma; \Delta \vdash t : t ; \emptyset ; \Delta \vdash t' : T' \quad \forall T_i \in T, \text{serializable}(T_i) \\
\Gamma; \Sigma; \Delta \vdash \{\text{spore} \{x : T = t ; (x : T) \Rightarrow t\} : T \Rightarrow T' \{\text{type } C = T\}\} \\
\text{T-AppSpore} & \quad \Gamma; \Sigma; \Delta \vdash t : T \Rightarrow T' \{\text{type } C = T\} \\
\Gamma; \Sigma; \Delta \vdash t' : T \\
\Gamma; \Sigma; \Delta \vdash (t \, t') : T' \\
\text{T-Spawn} & \quad \Gamma; \Sigma; \Delta \vdash t : \{\} \Rightarrow T \{\text{type } C = T\} \\
\Gamma; \Sigma; \Delta \vdash \text{spawn}(t) : \text{Host} \\
\text{T-Populate} & \quad \Gamma; \Sigma; \Delta \vdash t : \text{Host} \\
\Gamma; \Sigma; \Delta \vdash t' : T \quad \text{serializable}(T) \\
\Gamma; \Sigma; \Delta \vdash \text{populate}(t, t') : \text{SiloRef}[T] \\
\Gamma; \Sigma; \Delta \vdash t : \text{SiloRef}[T] \\
\Gamma; \Sigma; \Delta \vdash t' : (T \Rightarrow T' \{\text{type } C = T\}) \\
\Gamma; \Sigma; \Delta \vdash \text{map}(t, t') : \text{SiloRef}[T'] \\
\text{T-FMap} & \quad \Gamma; \Sigma; \Delta \vdash t : \text{SiloRef}[T] \\
\Gamma; \Sigma; \Delta \vdash t' : (T \Rightarrow \text{SiloRef}[T'] \{\text{type } C = T\}) \\
\Gamma; \Sigma; \Delta \vdash \text{flatMap}(t, t') : \text{SiloRef}[T'] \\
\text{T-Persist} & \quad \Gamma; \Sigma; \Delta \vdash t : \text{SiloRef}[T] \\
\Gamma; \Sigma; \Delta \vdash \text{persist}(t) : \text{SiloRef}[T] \\
\Gamma; \Sigma; \Delta \vdash \text{send}(t) : \text{Future}[T] \\
\text{T-SiloRef} & \quad \Delta(\text{id}(r)) = T \\
\Delta \vdash r \\
\Gamma; \Sigma; \Delta \vdash r : \text{SiloRef}[T] \\
\text{T-Send} & \quad \Gamma; \Sigma; \Delta \vdash t : \text{SiloRef}[T] \\
\Gamma; \Sigma; \Delta \vdash \text{send}(t) : \text{Future}[T] \\
\text{T-Await} & \quad \Gamma; \Sigma; \Delta \vdash t : \text{Future}[T] \\
\Gamma; \Sigma; \Delta \vdash \text{await}(t) : T \\
\end{align*}
\]

Figure 8: Type assignment.
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 $x$ 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 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 $\text{host}$. 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 fMap 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 $\Delta$. Analogous to rule $T$-LOC, the type of silo $id(r)$ is looked up in $\Delta$. Furthermore, $T$-SILOREF requires $r$ to be well-formed in $\Delta$, 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 $\Delta$ to be consistent with the corresponding type of spore $p$. Rule WF-REF3 requires the type of silo $\omega$ to be equal to the type of its parent silo $id(r)$ in silo store typing $\Delta$. Rule WF-REF4 requires $\Delta$ 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 $\Delta$. Rules WF-Q1-4 specify well-formedness of message queues in $\Delta$ and $\Sigma$. Rules WF-HOSTCONFIG, WF-HOST1, and WF-HOST2 combine the previous rules in the expected way.
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 $t \mid \mu \overset{h}{\rightarrow} t' \mid \mu'$ 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 $(t, \mu, Q, S)^h \rightarrow (t', \mu', Q', S')^h$ then $\Delta'; \Sigma' \vdash (t', \mu', Q', S')^h$ for some $\Delta' \supseteq \Delta$ and $\Sigma' \supseteq \Sigma$.

3. If $\Delta \vdash H$ and $H \overset{h}{\rightarrow} H'$ then $\Delta' \vdash H'$ for some $\Delta' \supseteq \Delta$.

Proof. Part 1: by induction on the derivation of $t \mid \mu \overset{h}{\rightarrow} t' \mid \mu'$. Part 2: by induction on the derivation of $(t, \mu, Q, S)^h \rightarrow (t', \mu', Q', S')^h$. Part 3: by induction on the derivation of $H \overset{h}{\rightarrow} H'$. See Appendix ?? for the complete proof.