A Formal Model for Portable, Heterogeneous Accelerator Programming

Zachary J. Sullivan and Samuel D. Pollard

Sandia National Laboratories, Livermore CA, USA
{spolla,zsulliv}@sandia.gov

Abstract. Programming on modern computer architectures requires logic to utilize both multi-threaded CPUs and accelerators such as GPUs. This can be fraught with errors relating to transmitting and accessing memory not available to all compute resources. Moreover, once the programmer writes correct code for one system, it is often slow or incorrect when run on a different architecture. A bottom-up approach to solving this problem is reified in the C++ library Kokkos. We approach the problem top-down, distilling and generalizing concepts found therein. We design a small language, called H-IMP—which builds on an earlier model of Kokkos called MiniKokkos—with a type system that includes notions of device memory, accelerators, and safe memory access. We show that a well-typed program is safe, which in this context means that there are no heterogeneous memory errors. Our type system enables us to define a precise notion of a *portable* program as a program with free variables representing where data is stored and kernels are executed. Finally, we prove a *portability theorem* for heterogeneous programs: that the program can run safely when instantiated on a specific set of architectures.

Keywords: programming languages; high-performance computing; heterogeneous computing; portability

1 Introduction

Modern compute nodes are structured with a host CPU together with other kinds of accelerators, typically GPUs. While writing any programs that exploit this hardware is already a difficult task, writing programs that are also meant to be portable is compounded by the wide variety of GPU and on-CPU accelerator systems. To ease programming with such machines and to abstract over the different hardware architectures, there exist many libraries and languages which offer a programming model to handle multiple parallel architectures abstractly [2,3,4,5,6,8,10,13,15,9]. In Kokkos [6,13]—a library designed specifically for portability—accelerators are abstracted into a notion of *execution spaces* that we can run kernels on; an obvious example would be a GPU, but another example is an OpenMP kernel running on the CPU. To abstract different kinds of memory accessible to different execution spaces, Kokkos has *memory spaces*; for example, a GPU will have its on-chip memory. Importantly, these have different

performance and accessibility properties depending on which execution space is being used. The host code, and only the host code, can allocate objects which exist in these memory spaces; Kokkos calls these objects *views*.

While Kokkos provides an abstraction that enables portability, using the C++ library alone does not give us the extra reasoning to know whether our code is indeed portable. Consider the following program:

```
Kokkos::View<int *, Kokkos::HostSpace> view ("V", 32);
Kokkos::parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    view(index) = index;
});
```

Implicit in this code is where the parallel for-loop is executed. Behind the scenes, this location is chosen by a configured default; this is how Kokkos code can be instantiated for different systems. If this execution space is configured to be some on-CPU space like OpenMP, then this code will run without issue. However, there exist instantiations of this default that will produce problems; for instance, using a CUDA execution space will result in a memory error when the code attempts to write to `view`. Thus, the code is only portable to a specific subset of systems. Kokkos allows us to avoid declaring the memory space explicitly and it will choose the memory space so that it matches the execution space, but then a portable program must include copying between host and this memory space.

To describe a portable, heterogeneous program as a formal property, we develop a small, formal language including these features alone. Our language includes a type system that takes from two lines of work: region-based memory management and security type systems. First, our system can be seen as a modification of the region calculus [12,1] wherein locations are added to variables to automatically handle allocation and deallocation of objects. Our type system also adds locations, *i.e.* memory spaces, to where Kokkos views are stored. Second, we take inspiration from languages with features for information-flow security [11] wherein code is tagged with either low or high security to restrict permissions. In Kokkos, we think of code being tagged with an execution space that restricts its permission to access certain memory spaces and operations. Portability can then be defined by polymorphism over spaces in a manner which respects these permissions.

Previous work [7] on modeling Kokkos as a small programming language, called MiniKokkos addressed the problem of deadlocks. Our language H-IMP, simplifies their execution model to focus on heterogeneous memory and device permissions. Our contributions include the following:

- A core language (H-IMP) for heterogeneous hardware (Section 2).
- An operational semantics that captures notions of different kernel-executing machines within a global execution of a program (Section 3).
- A type system that provides static checks on the spaces for both computations and memory (Section 4). Our appendix contains the detailed proof

$$
\begin{array}{rrcl}
\chi \in & \textit{Execution Space} & ::= & \texttt{Host} \mid \texttt{Serial} \mid \texttt{Threads} \mid \texttt{OpenMP} \mid \texttt{Cuda} \mid \ldots \\
\mu \in & \textit{Memory Space} & ::= & \texttt{Host} \mid \texttt{CudaUVM} \mid \texttt{Cuda} \mid \ldots \\
E \in & \textit{Expression} & ::= & x \mid x(E) \mid c \mid E_0 \; op_i \; E_1 \\
S \in & \textit{Statement} & ::= & C; S \mid \texttt{ret} \mid \texttt{decl}\ x := E; S \mid \texttt{decl}\ x\ \texttt{in}\ \mu; S \\
C \in & \textit{Command} & ::= & \texttt{set}\ x := E \mid \texttt{set}\ x(E_0) := E_1 \\
 & & & \mid \texttt{fence}(\chi) \mid \texttt{deep\_copy}(E_0, E_1) \mid \texttt{kernel}(\chi, \lambda x_0, \ldots, x_n.\, S)
\end{array}
$$

Fig. 1: H-IMP Syntax

that well-typed programs are free of heterogeneous memory errors by means of a realizability model of the type system over its operational semantics.
– An extension to H-IMP to include variables, like *default*, for execution and memory spaces. Thereby, we can give a *concise, formal* definition of what it means for a program to be portable to other architectures (Section 5). Moreover, our space variables allow us to write programs for portable, multiple-accelerator nodes that are currently not expressible in the Kokkos library.

2 A Syntax for Computing with Accelerators

Figure 1 presents the syntax of H-IMP. The language is a heterogeneous modification of the language IMP, a common model for imperative languages [16]; similarly, we construct programs from statements which consist of commands and expressions. Whereas commands are used to modify program state imperatively, expressions compute pure values from the program state. To model the heterogeneity in a similar manner to Kokkos, H-IMP has execution and memory spaces. Execution spaces, denoted χ, are more general than mere devices; *e.g.* OpenMP is an execution space but may run on CPUs or accelerators. Similarly, several different kinds of memory spaces, denoted μ, can exist on the same device; each with different characteristics. For instance, some memory spaces, like `CudaUVM`, are accessible from multiple execution spaces.

Though we specify a number of execution and memory spaces in Figure 1, these are not intended to be fixed sets, which is why they are written with ellipses. In later sections, we will see how one can expand and contract these sets as well as describe their accessibility properties to influence the strength our portability theorem for a specific program.

The imperative features of H-IMP are for mutating variables and views as well as launching kernels. There are two kinds of commands for declaring local variables: the first declares a local variable for an expression and the second declares a view in memory space μ while binding a pointer to it locally. Here, we require that a view declaration include an explicit memory space where its data is allocated; this is a necessary intermediate step to describing portable programs with *default* spaces in Section 5. Similar to the two kinds of variable access, we have two different notions of mutating variables: those for views and

$$
\begin{array}{rrll}
\mathit{GState} \in & \mathit{Global\ State} & ::= & \langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel S \rangle\!\rangle \\
\mathit{LState} \in & \mathit{Local\ State} & ::= & \langle\!\langle \mathbb{M} \mid\!\models L \parallel S \rangle\!\rangle \\
\mathit{Conf} \in & \mathit{Expr.\ Conf.} & ::= & \langle\!\langle \mathbb{M} \mid\!\models L \parallel E \rangle\!\rangle \\
\mathbb{M} \in & \mathit{Mach.\ Mem.\ Spaces} & = & \mathit{Mem.\ Space} \rightharpoonup \mathit{Pointer} \rightharpoonup \mathbb{N} \to B_i \\
L \in & \mathit{Local\ Mem.} & = & \mathit{Variable} \rightharpoonup \mathit{Mach.\ Value} \\
V, W \in & \mathit{Mach.\ Value} & ::= & (\mu, \pi) \mid c \\
\mathbb{S} \in & \mathit{Ex.\ Space\ Queues} & = & \mathrm{FIFO}^{\mathit{Ex.\ Space}}\ (\mathit{Local\ Mem.} \times \mathit{Statement})
\end{array}
$$

Fig. 2: Operational Semantics Syntax

non-views. Other commands available are those for synchronizing the host with a particular execution space, copying between memory spaces, and launching new kernels on a particular execution space. The notion of a kernel in H-IMP is the more general than those found in Kokkos; that is, our kernels consist only of a particular execution space in which they execute and a set of variables they copy from the host into the runtime environment (which may itself be the host).

Excluded from this study are loops and conditionals because we focus on the features related to portability of heterogeneous systems, *i.e.* those which launch and control kernels of different accelerators while communicating through shared variables. We include two key features that enable communication and synchronization: deep copy and fence, respectively. Deep copy enables the movement of data between views, while a fence blocks until completion of all asynchronous operations. We do include constants c from a set of base types B_i and operations over them $E_0\ op_i\ E_1$ for use in our examples. Indexing into views is done with natural numbers, which are an example of these constants for the base type $\mathbb{N}$.

3 Operational Semantics

The goal of operational semantics is to model the concurrent execution of kernels from different accelerators alongside a collection of memory spaces within an abstract machine. The syntax for it is found in Figure 2. It contains three different kinds of program state for which we define three different notions of evaluation. All states contain a local environment L that contain local variables. The largest state, *i.e.* global state, has access to all of the memory spaces available, written $\mathbb{M}$, as well as the queues of work for the execution spaces available, written $\mathbb{S}$. Local states are for kernel execution and consist of local memory, one statement for execution, and a restricted set of available memory spaces. Local states cannot access any work queues for execution spaces. Expression evaluation configurations contain the same information.

The available memory spaces ($\mathbb{M}$) are a partial map from memory spaces to pointers to indices to values of base types (such as integers). To access a specific index n of a view π in a memory space μ, we write $\mathbb{M}(\mu)(\pi)(n)$; if we just wanted the particular view, then we would write $\mathbb{M}(\mu)(\pi)$; and so on. For simplicity, we

$$\boxed{\langle\langle \mathbb{M} \mid\models L \parallel E \rangle\rangle \Downarrow V}$$

$$\frac{x \in \mathrm{Dom}(L)}{\langle\langle \mathbb{M} \mid\models L \parallel x \rangle\rangle \Downarrow L(x)}\ \mathit{EVar} \qquad \frac{}{\langle\langle \mathbb{M} \mid\models L \parallel c \rangle\rangle \Downarrow c}\ \mathit{EConst}$$

$$\frac{\langle\langle \mathbb{M} \mid\models L \parallel E \rangle\rangle \Downarrow n \quad L(x) = (\mu, \pi) \quad \pi \in \mathrm{Dom}(\mathbb{M}(\mu))}{\langle\langle \mathbb{M} \mid\models L \parallel x(E) \rangle\rangle \Downarrow \mathbb{M}(\mu)(\pi)(n)}\ \mathit{EViewDeref}$$

$$\frac{\langle\langle \mathbb{M} \mid\models L \parallel E_0 \rangle\rangle \Downarrow c_0 \quad \langle\langle \mathbb{M} \mid\models L \parallel E_1 \rangle\rangle \Downarrow c_1}{\langle\langle \mathbb{M} \mid\models L \parallel E_0\ op\ E_1 \rangle\rangle \Downarrow c_0\ op\ c_1}\ \mathit{EOp}$$

Fig. 3: Expression Evaluations

assume that if a view is defined then any index into it is defined. This syntax follows similarly for local memory, but there fewer levels of indirection; that is, we need only write $L(x)$. Additionally, whereas views can only contain values of base types c, local memory can contain both values of base types and pointers to views in memory (μ, π). We use the syntax $L[x \mapsto V]$ to denote either replacing the current mapping of x in L or to insert a new mapping for x when it does not yet exist in L. Likewise, we can update our available memory spaces; $\mathbb{M}[\mu, \pi, n \mapsto c]$ updates (or inserts) c at the nth index of the view at location π in memory space μ. We use Dom (short for domain) to ensure variables exist in their respective environments (local or memory spaces). Finally, we use $\mathbb{M}|_{P(\mu)}$ to denote the restriction of the memory spaces to those in the set of memory spaces μ that satisfy the proposition P.

The execution space queues, denoted $\mathbb{S}$, contained within the global state is an execution-space-indexed first-in-first-out queue. All of the operations on this object include a specific memory space. $\mathbb{S}.\mathrm{empty}(\chi)$ is a proposition that is true if the work queue for execution space χ is empty. $\mathbb{S}.\mathrm{pusht}(\chi, (L, S))$ publishes a new task to the end of χ's work queue. $\mathbb{S}.\mathrm{head}(\chi)$ merely looks at the front of the queue; whereas $\mathbb{S}.\mathrm{poph}(\chi)$ removes the front of the queue. Finally, we have $\mathbb{S}.\mathrm{replaceh}(\chi, (L, S))$ which updates the head of the queue to a new work state.

We first present big-step reduction of expression configurations to machine values in Figure 3. For variables, we merely look it up in the local memory. For accessing views, we first evaluate the index with the current state to get a pointer to a particular view in a memory space, and then we index into $\mathbb{M}$ with it. If the view that we are trying to dereference is not in the local $\mathbb{M}$ then we would not be able to construct an evaluation derivation. In a real program, this would occur if the current execution space does not have access to that memory space, since it would not be included in local instance of $\mathbb{M}$. Such a restriction is upheld when instantiating a kernel by the *GXStep* rule in Figure 5.

Taking steps locally, which includes execution spaces transitioning, is defined by the deterministic relation in Figure 4. Declaring and mutating variables both happen by evaluating the expression and using the result to manipulate the local environment L. Of course, failing to declare a local variable before setting

$$\boxed{\langle\!\langle \mathbb{M} \mid\!\models L \parallel S \rangle\!\rangle \longmapsto \langle\!\langle \mathbb{M}' \mid\!\models L' \parallel S' \rangle\!\rangle}$$

$$\frac{\langle\!\langle \mathbb{M} \mid\!\models L \parallel E \rangle\!\rangle \Downarrow V}{\langle\!\langle \mathbb{M} \mid\!\models L \parallel \texttt{decl}\ x := E; S \rangle\!\rangle \longmapsto \langle\!\langle \mathbb{M} \mid\!\models L[x \mapsto V] \parallel S \rangle\!\rangle}\ \mathit{LDeclVar}$$

$$\frac{x \in \mathrm{Dom}(L) \quad \langle\!\langle \mathbb{M} \mid\!\models L \parallel E \rangle\!\rangle \Downarrow V}{\langle\!\langle \mathbb{M} \mid\!\models L \parallel \texttt{set}\ x := E; S \rangle\!\rangle \longmapsto \langle\!\langle \mathbb{M} \mid\!\models L[x \mapsto V] \parallel S \rangle\!\rangle}\ \mathit{LSetVar}$$

$$\frac{\begin{array}{c} L(x) = (\mu, \pi) \quad \pi \in \mathrm{Dom}(\mathbb{M}(\mu)) \\ \langle\!\langle \mathbb{M} \mid\!\models L \parallel E_0 \rangle\!\rangle \Downarrow n \quad \langle\!\langle \mathbb{M} \mid\!\models L \parallel E_1 \rangle\!\rangle \Downarrow c \end{array}}{\langle\!\langle \mathbb{M} \mid\!\models L \parallel \texttt{set}\ x(E_0) := E_1; S \rangle\!\rangle \longmapsto \langle\!\langle \mathbb{M}[\mu, \pi, n \mapsto c] \mid\!\models L \parallel S \rangle\!\rangle}\ \mathit{LSetView}$$

Fig. 4: Local Transitions

it will result in a local memory error, so no transition is possible. We may also mutate views from execution spaces, which has a similar restriction that the location must be already defined before changing it. Like with the big-step rule for expressions, $\mathbb{M}$ may or may not contain the particular memory spaces and views to complete a transition depending on the instantiation of the kernel. Finally, local transitions operate over statements, but do not have the permission to allocate new views, deep copy, or fence; thus, such statements would be stuck.

Global transitions are described in Figure 5. Intuitively, these transitions represent the host program, which orchestrates all of the memory and execution spaces. The first rule *GHStep* is for when the host takes a step locally in the same manner as an execution space. Unlike other execution spaces, the host can also declare a view, with *GDeclView*, given an unused view location π. We take $\mathbb{M}[\mu, \pi \mapsto \texttt{init}]$ to mean that for any n that $\mathbb{M}(\mu)(\pi)(n)$ is defined. In *GKernel*, the host program publishes a new unit of work to an execution space's work stack; note that it also copies the local variables captured by the λ-expression in the kernel definition, which can get stuck if the variables are undefined. In *GFence*, we see that the program is stuck until the execution space, for which we are waiting, completes its stack of work. Finishing a unit of work in the stack is achieved by the two concurrent steps of *GXPop* and *GXStep*. The first removes the work when `ret` is the waiting statement. The latter selects one of the execution space queues and takes a single step on it. It is in this rule that we restrict the valid memory spaces for each execution space when it takes a step; we use $\mu \rhd \chi$ for the restricted set of memory spaces μ that are accessible from χ. Note that because global transitions are non-deterministic, these steps model the concurrency implicit in the Kokkos machine model.

The following definitions describe how to run programs in our abstract machine; we will later define *safe* executions of the machine and our static analysis will show well-typed H-IMP programs imply safe execution (Theorem 1).

Definition 1 (Initial State). $\mathrm{Initial}(\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel S \rangle\!\rangle)$ *where* $\mathbb{M}$ *contains a set of empty memory spaces and* $\mathbb{S}$ *contains all empty work stacks.*

$$\boxed{\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel S \rangle\!\rangle \longrightarrow \langle\!\langle \mathbb{M}' \parallel \mathbb{S}' \parallel\!\models L' \parallel S' \rangle\!\rangle}$$

$$\frac{C \in \{\texttt{decl}\ x := E, \texttt{set}\ x := E, \texttt{set}\ x(E_0) := E_1\} \quad \langle\!\langle \mathbb{M} \parallel\!\models L \parallel C; S \rangle\!\rangle \longmapsto \langle\!\langle \mathbb{M}' \parallel\!\models L' \parallel S \rangle\!\rangle}{\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel C; S \rangle\!\rangle \longrightarrow \langle\!\langle \mathbb{M}' \parallel \mathbb{S} \parallel\!\models L' \parallel S \rangle\!\rangle}\ GHStep$$

$$\frac{\pi \notin \mathrm{Dom}(\mathbb{M}(\mu))}{\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel \texttt{decl}\ x\ \texttt{in}\ \mu; S \rangle\!\rangle \longrightarrow \langle\!\langle \mathbb{M}[\mu, \pi \mapsto \texttt{init}] \parallel \mathbb{S} \parallel\!\models L[x \mapsto (\mu, \pi)] \parallel S \rangle\!\rangle}\ GDeclView$$

$$\frac{\forall i \in 0, \ldots, n.\, x_i \in \mathrm{Dom}(L) \quad L' = x_0 \mapsto L(x_0), \ldots, x_n \mapsto L(x_n)}{\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel \texttt{kernel}(\chi, \lambda x_0, \ldots, x_n.\, S_0); S_1 \rangle\!\rangle \longrightarrow \langle\!\langle \mathbb{M} \parallel \mathbb{S}.\mathrm{pusht}(\chi, (L', S_0)) \parallel\!\models L \parallel S_1 \rangle\!\rangle}\ GKernel$$

$$\frac{\mathbb{S}.\mathrm{empty}(\chi)}{\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel \texttt{fence}(\chi); S \rangle\!\rangle \longrightarrow \langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel S \rangle\!\rangle}\ GFence$$

$$\frac{\langle\!\langle \mathbb{M} \parallel\!\models L \parallel E_0 \rangle\!\rangle \Downarrow (\mu_0, \pi_0) \quad \langle\!\langle \mathbb{M} \parallel\!\models L \parallel E_1 \rangle\!\rangle \Downarrow (\mu_1, \pi_1)}{\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel \texttt{deep\_copy}(E_0, E_1); S \rangle\!\rangle \longrightarrow \langle\!\langle \mathbb{M}[\mu_1, \pi_1 \mapsto \mathbb{M}(\mu_0)(\pi_0)] \parallel \mathbb{S} \parallel\!\models L \parallel S \rangle\!\rangle}\ GDeepCopy$$

$$\frac{\mathbb{S}.\mathrm{head}(\chi) = (L', \texttt{ret})}{\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel S \rangle\!\rangle \longrightarrow \langle\!\langle \mathbb{M} \parallel \mathbb{S}.\mathrm{poph}(\chi) \parallel\!\models L \parallel S \rangle\!\rangle}\ GXPop$$

$$\frac{\mathbb{S}.\mathrm{head}(\chi) = (L', S') \quad \langle\!\langle \mathbb{M}|_{\mu \triangleright \chi} \parallel\!\models L' \parallel S' \rangle\!\rangle \longmapsto \langle\!\langle \mathbb{M}' \parallel\!\models L'' \parallel S'' \rangle\!\rangle}{\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel S \rangle\!\rangle \longrightarrow \langle\!\langle \mathbb{M}|_{\overline{\mu \triangleright \chi}} \cup \mathbb{M}' \parallel \mathbb{S}.\mathrm{replaceh}(\chi, (L'', S'')) \parallel\!\models L \parallel S \rangle\!\rangle}\ GXStep$$

Fig. 5: Global Transitions

$$\begin{array}{rrcl} \tau \in & \mathit{Type} & ::= & B_i \mid \texttt{view}(\mu, B_i) \\ \Gamma \in & \mathit{Type\ Environment} & ::= & \varepsilon \mid \Gamma, x{:}\tau \end{array}$$

Fig. 6: H-IMP Type Syntax

Definition 2 (Final States).

For local states, $\mathrm{Final}(\langle\!\langle \mathbb{M} \parallel\!\models L \parallel \texttt{ret} \rangle\!\rangle)$.

For global states, $\mathrm{Final}(\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel \texttt{ret} \rangle\!\rangle)$ *where* $\mathbb{S}$ *contains all empty work stacks.*

4 Type System

We present the syntax of the H-IMP type system in Figure 6. Memory spaces are referenced by $\texttt{view}(\mu, B_i)$ types, which are pointers to data structures over the base type B_i and housed within a memory space μ. As a simplification from Kokkos, we consider views to be arrays of type B_i.

To reason statically about memory spaces and execution spaces of H-IMP programs, the judgements of our type system require information about where their computations occur and the information about the memory spaces accessible from each execution space must be supplied. The type system's rules are presented in Figure 7. There are three main judgements that all end in @ χ signifying the execution space wherein the expression, statement, or command

$$\boxed{\Gamma \vdash E : \tau \mathbin{@} \chi}$$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau \mathbin{@} \chi}\ \mathit{TVar} \qquad \frac{c \in B_i}{\Gamma \vdash c : B_i \mathbin{@} \chi}\ \mathit{TConst}$$

$$\frac{x{:}\texttt{view}(\mu, B_i) \in \Gamma \quad \mu \rhd \chi \quad \Gamma \vdash E : \mathbb{N} \mathbin{@} \chi}{\Gamma \vdash x(E) : B_i \mathbin{@} \chi}\ \mathit{TViewDeref}$$

$$\frac{\Gamma \vdash E_0 : \tau_0 \mathbin{@} \chi \quad \Gamma \vdash E_1 : \tau_1 \mathbin{@} \chi \quad op_i : \tau_0 \to \tau_1 \to \tau}{\Gamma \vdash E_0\ op_i\ E_1 : \tau \mathbin{@} \chi}\ \mathit{TOp}$$

$$\boxed{\Gamma \vdash S \mathbin{@} \chi}$$

$$\frac{\Gamma \vdash C \mathbin{@} \chi \quad \Gamma \vdash S \mathbin{@} \chi}{\Gamma \vdash C; S \mathbin{@} \chi}\ \mathit{TCom} \qquad \frac{}{\Gamma \vdash \texttt{ret} \mathbin{@} \chi}\ \mathit{TRet}$$

$$\frac{\Gamma \vdash E : \tau \mathbin{@} \chi \quad \Gamma, x{:}\tau \vdash S \mathbin{@} \chi}{\Gamma \vdash \texttt{decl}\ x := E; S \mathbin{@} \chi}\ \mathit{TDeclVar} \qquad \frac{\Gamma, x{:}\texttt{view}(\mu, B_i) \vdash S \mathbin{@} \texttt{Host}}{\Gamma \vdash \texttt{decl}\ x\ \texttt{in}\ \mu; S \mathbin{@} \texttt{Host}}\ \mathit{TDeclView}$$

$$\boxed{\Gamma \vdash C \mathbin{@} \chi}$$

$$\frac{x{:}\tau \in \Gamma \quad \Gamma \vdash E : \tau \mathbin{@} \chi}{\Gamma \vdash \texttt{set}\ x := E \mathbin{@} \chi}\ \mathit{TSetVar} \qquad \frac{}{\Gamma \vdash \texttt{fence}(\chi) \mathbin{@} \texttt{Host}}\ \mathit{TFence}$$

$$\frac{x{:}\texttt{view}(\mu, B_i) \in \Gamma \quad \mu \rhd \chi \quad \Gamma \vdash E_0 : \mathbb{N} \mathbin{@} \chi \quad \Gamma \vdash E_1 : B_i \mathbin{@} \chi}{\Gamma \vdash \texttt{set}\ x(E_0) := E_1 \mathbin{@} \chi}\ \mathit{TSetView}$$

$$\frac{\Gamma \vdash E_0 : \texttt{view}(\mu_0, B_i) \mathbin{@} \texttt{Host} \quad \Gamma \vdash E_1 : \texttt{view}(\mu_1, B_i) \mathbin{@} \texttt{Host}}{\Gamma \vdash \texttt{deep\_copy}(E_0, E_1) \mathbin{@} \texttt{Host}}\ \mathit{TDeepCopy}$$

$$\frac{\forall i \in 0, \ldots, n.\, x_i{:}\tau_i \in \Gamma \quad \chi \neq \texttt{Host} \quad x_0{:}\tau_0, \ldots, x_n{:}\tau_n \vdash S \mathbin{@} \chi}{\Gamma \vdash \texttt{kernel}(\chi, \lambda x_0, \ldots, x_n.\, S) \mathbin{@} \texttt{Host}}\ \mathit{TKernel}$$

Fig. 7: H-IMP Typing Rules

is to take place. For instance, $\Gamma \vdash E : \tau \mathbin{@} \chi$ states that with the local type environment Γ the expression E computes a value of type τ in the execution space χ. Certain commands are only available to the host execution space, the orchestrater of H-IMP programs. Specifically, the host is the only execution space that may declare views, fence execution spaces, deep copy views, and launch kernels. However, we cannot launch kernels for the host; one would instead need to use the `Serial` execution space.

Note that the typing environment Γ will *only* contain variables local to that execution space. During computation this is thread-local memory; see that the *TKernel* rule specifies explicitly the variables that will be copied to its local memory.

Indexed by some sets of memory and execution spaces, our typing system depends on a relation $\mu \rhd \chi$ on $\mathit{Mem}.\ \mathit{Space} \times \mathit{Ex}.\ \mathit{Space}$, which occurred in the

operational semantics. For the set of execution and memory spaces we gave in Figure 1, this relation is defined as the following:

$$(\triangleright) = \{(\texttt{Host}, \texttt{Host}), (\texttt{Host}, \texttt{Serial}), (\texttt{Host}, \texttt{Threads}), (\texttt{Host}, \texttt{OpenMP}), (\texttt{Cuda}, \texttt{Cuda})\} \\ \cup \{(\texttt{CudaUVM}, \chi) \mid \chi \in \text{Ex. Space}\}$$

In this relation, `CudaUVM` can safely be accessed by any execution space χ; of course, this may not be true if we wanted to consider GPUs from another vendor. The rules *TViewDeref* and *TSetView* check that every view referenced is accessible to the current execution space.

4.1 Safety

We must define a notion safety for each class of computable syntax. Expression configurations are the simplest: they are safe if they evaluate to a machine value. Both global and local machine states are safe if they take any number of steps to either a final state or they can continue to step; *i.e.* they cannot reach a stuck state.

Definition 3 (Safe Configurations and States).
For an expression configuration, Safe(*Conf*) *if and only if* $Conf \Downarrow V$.
For an execution-space state, Safe(*XState*) *if and only if* $XState \longmapsto^* XState'$ *implies* Final($XState'$) *or* $XState' \longmapsto XState''$.
For a host state, Safe(*GState*) *if and only if* $GState \longrightarrow^* GState'$ *implies* Final($GState'$) *or* $GState' \longrightarrow GState''$.

Though this looks like an overly simple notion of safety, it implies that we are always accessing an accessible view from the current execution space, that the global state is only manipulated directly by host execution space, and that variables are initialized before they are mutated. Moreover, it even captures safety in the notion of concurrency employed by the global transitions; because for every way that we take a step—there are multiple—we must step to a good final state or keep stepping.

Theorem 1 (Type Safety). *If* $\vdash S \ @ \ \texttt{Host}$, *then* Safe(Init($S$)).

5 Portable Programs

Kokkos programs, and templated C++ programs more generally, require abstracted template variables to be instantiated with concrete types and functions before a complete binary can be run. The programming language that we just presented can be seen as a program where all of the decisions about a node's architecture have been decided. However, this is not how the Kokkos C++ library is intended to be used. We would like H-IMP instead to specify one program that works for many different architectures. To accomplish this, Kokkos programs do not need to specify explicitly the memory spaces wherein views are located, or

the execution spaces whereat kernels are executed. Thus, we may see a source program (in H-IMP) like the following:

$$\begin{array}{l}\texttt{decl}\ x\ \texttt{in}\ \mathit{defaultMem};\\ \texttt{kernel}(\mathit{defaultEx}, \lambda x.\, x(0) := 2; \texttt{ret});\\ \ldots\end{array}$$

Before running this program with our machine machine, we must decide how these default spaces are instantiated. If a program is portable, then it should be the case that *any* instantiation of the default spaces produces a safe program.

Definition 4 (Portable Program). *A program S is* portable *if and only if* $\text{Safe}(\text{Init}(S[\sigma]))$ *for a given set of execution and memory spaces, their accessibility relation, and any instantiation σ of its free execution and memory variables.*

To describe this "templated" H-IMP, we must add memory and execution space variables to programs, denoted with underlines:

$$\begin{array}{rrl}\chi \in & \mathit{Ex.\ Space} & ::= x \mid \underline{\chi}\\ \underline{\chi} \in & \mathit{Inst.\ Ex.\ Space} & ::= \texttt{Host} \mid \texttt{Threads} \mid \texttt{OpenMP} \mid \texttt{Cuda} \mid \ldots\\ \mu \in & \mathit{Mem.\ Space} & ::= x \mid \underline{\mu}\\ \underline{\mu} \in & \mathit{Inst.\ Mem.\ Space} & ::= \texttt{Host} \mid \texttt{CudaUVM} \mid \texttt{Cuda} \mid \ldots\\ \Delta \in & \mathit{Space\ Env.} & ::= \varepsilon \mid \Delta, \texttt{ex}\ x \mid \Delta, \texttt{mem}\ x\end{array}$$

In real Kokkos programs, the *default* memory and execution space variables exist implicitly, but only as special space variables. Here, we have the option of multiple default spaces; consider for instance, a program with *defaultEx1* and *defaultEx2*, which could be instantiated on several kinds of two GPU systems.

To statically reason about space variables, we extend our typing judgments with Δ containing the free memory and execution space variables. For example, our expression judgments would have the form $\Gamma \vdash_\Delta E : \tau \ @\ \chi$. The accessibility relation now only refers to fully instantiated memory spaces $\underline{\mu} \trianglerighteq \underline{\chi}$, and we have a new generalized relation with the judgement $\Delta \vdash \mu \rhd \chi$ that we use in the updated rules from Figure 7.[1]

$$\frac{\Delta \vdash \mu :: \text{MS} \quad \Delta \vdash \chi :: \text{ES} \quad \forall \sigma \in \mathit{Inst}(\Delta), \mu[\sigma] \trianglerighteq \chi[\sigma]}{\Delta \vdash \mu \rhd \chi}$$

The judgements $\Delta \vdash \mu :: \text{MS}$ and $\Delta \vdash \chi :: \text{ES}$ are added to check that a space is either a variable in Δ or an instance. This extended type system is strong enough to give us portability.

Theorem 2 (Typing Ensures Portability). *If $\Gamma \vdash_\Delta S\ @\ \texttt{Host}$, then S is portable.*

[1] The full type system with the new and rewritten rules is found in the appendix.

Note well that constructing portable programs is very limited. We cannot prove the generalized accessibility rule unless we show that for any combination memory and execution space that they are accessible. For example, we cannot prove `mem` *default* $\vdash$ *default* $\rhd$ `Cuda`, because there exists a memory space inaccessible to `Cuda` execution spaces: `Host`. Indeed, given the set of execution and memory spaces of Figure 1 and the relation specified in Section 4 there exist *no* portable programs that make use of views with *default* memory and execution spaces. Thus, specifying a portable program necessarily includes giving a restricted set of execution and memory spaces for which it is portable.

6 Conclusion

We have developed a language H-IMP as a distillation of the features for portable heterogeneity present in Kokkos wherein we can launch kernels for different accelerators. An important notion is that of the permissions for each execution space, which controls the different types of memory it can access and the operations that it may perform. Over this language, we defined a type system that allows us to guarantee that well-typed programs do not misuse heterogeneous memory. Finally, we defined a notion of portable programs for this language and noted that there are no meaningful portable programs without specifying the restricted set of architectures to which a program is portable.

As future work, we plan to enhance the language with the typeclass mechanism [14] found in Haskell. This will allow us to describe portable programs as those that can be run on *any* architecture that satisfy some constraint, thereby avoiding the proviso that we specify a specific set of spaces. For instance, kernel code could have the type $\Gamma \vdash_\Delta S \mathbin{@} \forall\chi.\, \texttt{Host} \rhd \chi \Rightarrow \chi$ meaning that it can run in any execution space that can access host memory. In addition, we imagine an extension of the types to include more detailed information about the architecture including types representing the parallelism hierarchy of certain execution spaces and the interaction of kernels with different memory models. Currently, we are developing a tool for real Kokkos programs that uses this reasoning to identify the sets of architectures for which a program is portable.

Acknowledgment Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Additionally, we thank Jackson Mayo, Keita Teranishi, Christian Trott, Vivek Kale, Shyamali Mukherjee, Richard Rutledge, John Bender, and our anonymous reviewer for comments on draft versions of this paper.

References

1. Ahmed, A., Jia, L., Walker, D.: Reasoning about hierarchical storage. In: 18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings. pp. 33–44 (2003). https://doi.org/10.1109/LICS.2003.1210043

2. Beckingsale, D.A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A.J., Pearce, O., Robinson, P., Ryujin, B.S., Scogland, T.R.W.: RAJA: Portable performance for large-scale scientific applications. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). pp. 71–81 (2019). https://doi.org/10.1109/P3HPC49587.2019.00012
3. Callahan, D., Chamberlain, B.L., Zima, H.P.: The cascade high productivity language. In: 9th International Workshop on High-Level Programming Models and Supportive Environments (HIPS 2004), 26 April 2004, Santa Fe, NM, USA. pp. 52–60. IEEE Computer Society (2004). https://doi.org/10.1109/HIPS.2004.1299190
4. Charles, P., Grothoff, C., Saraswat, V.A., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Johnson, R.E., Gabriel, R.P. (eds.) Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA. pp. 519–538. ACM (2005). https://doi.org/10.1145/1103845.1094852
5. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE computational science and engineering **5**(1), 46–55 (1998). https://doi.org/10.1109/99.660313
6. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing **74**(12), 3202 – 3216 (2014). https://doi.org/10.1016/j.jpdc.2014.07.003
7. Jin, F., Jacobson, J., Pollard, S.D., Sarkar, V.: Minikokkos: A calculus of portable parallelism. In: Laguna, I., Rubio-González, C. (eds.) Sixth IEEE/ACM International Workshop on Software Correctness for HPC Applications, Correctness@SC 2022, Dallas, TX, USA, November 13-18, 2022. pp. 37–44. IEEE (2022). https://doi.org/10.1109/Correctness56720.2022.00010
8. Khronos SYCL Working Group: Sycl 2020 specification (revision 8) (2020), https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf
9. Lee, J.K., Palsberg, J.: Featherweight x10: a core calculus for async-finish parallelism. SIGPLAN Not. **45**(5), 25–36 (jan 2010). https://doi.org/10.1145/1837853.1693459, https://doi.org/10.1145/1837853.1693459
10. Rossbach, C.J., Yu, Y., Currey, J., Martin, J., Fetterly, D.: Dandelion: a compiler and runtime for heterogeneous systems. In: Kaminsky, M., Dahlin, M. (eds.) ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013. pp. 49–68. ACM (2013). https://doi.org/10.1145/2517349.252271
11. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas of Communication **21**(1), 5–19 (2003). https://doi.org/10.1109/JSAC.2002.806121
12. Tofte, M., Talpin, J.: Region-based memory management. Information and Computation **132**(2), 109–176 (1997). https://doi.org/10.1006/inco.1996.2613
13. Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D.S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., Wilke, J.: Kokkos 3: Programming model extensions for the exascale era. IEEE Transactions on Parallel and Distributed Systems **33**(4), 805–817 (2022). https://doi.org/10.1109/TPDS.2021.3097283

14. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 60–76. POPL '89 (1989). https://doi.org/10.1145/75277.7528
15. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC—first experiences with real-world applications. In: Euro-Par 2012 Parallel Processing: 18th International Conference, Rhodes Island, Greece, August 27-31. Proceedings. pp. 859–870. Springer (2012). https://doi.org/10.1007/978-3-642-32820-6_85
16. Winskel, G.: The formal semantics of programming languages - an introduction. Foundation of computing series, MIT Press (1993)

A Soundess of H-IMP Types

As part of the simplification of Kokkos for this paper, we have the following assumptions.

Assumption 1 (Memory Space Assumptions)

1. *For any* μ, $\mathbb{M}(\mu)$ *is defined.*
2. *For any* μ,π,n, *if* $\mathbb{M}(\mu)(\pi)$ *is defined, then* $\mathbb{M}(\mu)(\pi)(n)$ *is defined.*
3. *For any* μ, *there exists* $\pi \notin \mathrm{Dom}(\mathbb{M}(\mu))$.

The first assumption is justified because the set of memory spaces is fixed at the start of runtime and there is no observation a program can make that peeks at which memory spaces are defined. The second assumption is related to our simplification of the data within a view; so if a view indeed is accessible, then for this paper we do not care about the index that is accessed.

Proposition 1. $(\longmapsto)$ *is deterministic.*

Proposition 2. *If* $\mathrm{Final}(\mathit{LState})$, *then there exists no transition* $\mathit{LState} \longrightarrow \mathit{LState}'$. *Similarly for final GState's.*

A.1 A Model over the Operational Semantics

We prove that our well-typed programs will capture our notion of safety with a realizability model constructed over the operational semantics; the logical predicates or realizers for which presented in Figure 8. There are predicates that define a well-behaved object for machine values, type environments, machine memory spaces, expressions, kernels, stacks, and global evaluations; essentially one for each part of the operational semantics. We do not need step-indexing because we do not end up in any circular dependency since our semantic memory spaces do not contain anything other than base types.

Within these relations, the environments, both global and local, L and $\mathbb{M}$ are instantiated at different times. The local can be instantiated right away because it is modified directly by code, whereas the global store of views may change (in a type safe-manner) while an execution space is still working. Because a particular $\mathbb{M}$ may change without the code S in a kernel or global state, we instead deal with it indirectly through Ψ, which forms a Kripke world that the predicates are part of. Ψ is a map from memory spaces to pointers to base sets. This Kripke world allows are predicates to be closed over the extension and mutation of views that will happen during concurrent execution.

Definition 5 (Memory Space Typing Poset). $(\Psi, \sqsubseteq, \varepsilon)$ *is the the poset defined by* $\Psi \subseteq \Psi'$ *and* ε *is used to denote* $\{\}$.

Definition 6 (Kripke Property). *A predicate* $\mathcal{A}$ *has the Kripke property iff* $(\Psi', \ldots) \in \mathcal{A}$ *for all* $(\Psi, \ldots) \in \mathcal{A}$ *when* $\Psi \sqsubseteq \Psi'$.

$$\Psi \in \mathit{Semantic\ Mem.\ Spaces} = \mathit{Memory\ Space} \to \mathit{Pointer} \rightharpoonup B_i$$

$$\begin{aligned}
\mathcal{V}[\![B_i]\!] &= \{(\Psi, c) \mid c \in B_i\} \\
\mathcal{V}[\![\texttt{view}(\mu, B_i)]\!] &= \{(\Psi, (\mu, \pi)) \mid \Psi(\mu)(\pi) = B_i\} \\
\mathcal{L}[\![\Gamma]\!] &= \{(\Psi, L) \mid \forall x \in \mathrm{Dom}(\Gamma).\, (\Psi, L(x)) \in \mathcal{V}[\![\Gamma(x)]\!]\} \\
\mathcal{M} &= \{(\Psi, \mathbb{M}, \chi) \mid \forall \mu, \pi, n.\, \mu \rhd \chi \wedge \pi \in \mathrm{Dom}(\Psi(\mu)) \Longrightarrow \\
&\qquad \mathbb{M}(\mu)(\pi)(n) \in \mathcal{V}[\![\Psi(\mu)(\pi)]\!]\} \\
\mathcal{E}[\![\tau]\!] &= \{(\Psi, L, E, \chi) \mid \forall \mathbb{M}.\, (\Psi, \mathbb{M}, \chi) \in \mathcal{M} \Longrightarrow \\
&\qquad \langle\!\langle \mathbb{M} \mid\!\models L \parallel E \rangle\!\rangle \Downarrow V \wedge (\Psi, V) \in \mathcal{V}[\![\tau]\!]\} \\
\mathcal{X} &= \{(\Psi, L, S, \chi) \mid \forall \mathbb{M}.\, (\Psi, \mathbb{M}, \chi) \in \mathcal{M} \Longrightarrow \\
&\qquad \mathrm{Final}(\langle\!\langle \mathbb{M} \mid\!\models L \parallel S \rangle\!\rangle) \\
&\qquad \vee \\
&\qquad \exists! \mathit{XState}.\, \langle\!\langle \mathbb{M} \mid\!\models L \parallel S \rangle\!\rangle \longmapsto \mathit{XState} \wedge \\
&\qquad\quad \exists \mathbb{M}', L', S'.\, \mathit{XState} = \langle\!\langle \mathbb{M}' \mid\!\models L' \parallel S' \rangle\!\rangle \wedge \\
&\qquad\qquad (\Psi, \mathbb{M}', \chi) \in \mathcal{M} \wedge (\Psi, L', S', \chi) \in \mathcal{X}\} \\
\mathcal{S} &= \{(\Psi, \mathbb{S}) \mid \forall L, S, \chi.\, (L, S) \in \mathbb{S}(\chi) \Longrightarrow (\Psi, L, S, \chi) \in \mathcal{X}\} \\
\mathcal{G} &= \{(\Psi, L, S) \mid \forall \mathbb{M}, \mathbb{S}.\, (\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M} \wedge (\Psi, \mathbb{S}) \in \mathcal{S} \Longrightarrow \\
&\qquad \mathrm{Final}(\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel S \rangle\!\rangle) \\
&\qquad \vee \\
&\qquad \left(\begin{array}{l}
\forall \mathit{GState}.\, \langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel S \rangle\!\rangle \longrightarrow \mathit{GState} \Longrightarrow \\
\quad \exists \Psi', \mathbb{M}', \mathbb{S}', L', S'.\, \mathit{GState} = \langle\!\langle \mathbb{M}' \parallel \mathbb{S}' \mid\!\models L' \parallel S' \rangle\!\rangle \wedge \\
\quad\quad \Psi \sqsupseteq \Psi' \wedge (\Psi', \mathbb{M}', \texttt{Host}) \in \mathcal{M} \wedge (\Psi', \mathbb{S}') \in \mathcal{S} \wedge \\
\quad\quad ((L, S) \neq (L', S') \Longrightarrow (\Psi', L', S') \in \mathcal{G}) \\
\wedge \\
\exists \mathit{HState}.\, \langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel S \rangle\!\rangle \longrightarrow \mathit{HState}
\end{array} \right) \\
&\}
\end{aligned}$$

Fig. 8: Logical Predicates for H-IMP.

Proposition 3 (Predicates have Kripke Property). *$\mathcal{V}$, $\mathcal{E}$, $\mathcal{L}$, $\mathcal{X}$, $\mathcal{S}$, and $\mathcal{G}$ have the Kripke property.*

View realizers are only defined with respect to the global environment of views. The notion of a well-behavied view type realizer only checks that the pointer is indeed in that global environment. This means that an execution space can pass around and alias an unaccessible view. It is the notions of expression and statement realizers that guarantee that we use the views in such a way that we do not get stuck.

Realizers for expressions which compute a type $\mathcal{E}[\![\tau]\!]$, kernels $\mathcal{X}$, and global computations $\mathcal{G}$, all contain some world Ψ and local environment. In every case, the world needs to be instantiated with a well-behaved machine memory space $\mathbb{M}$ before we can make a statement about the operational semantics state. Expression realizers are well-behavied when they compute a well-behaved value. Statements are well-behaved when they are final or take a unique step into a well-behaved state while preserving the behavior of the machine memory spaces. The global realizer is the most complex because it has to deal with concurrency. Like a kernel, a state can be well-behavied if it is final. We must also make sure that all of the possible states that it can step to are well-behavied; that is, there exist no bad paths. Finally, we must show that the set of possible future states is non-empty: that we can keep stepping.

Assumption 2 (Value Safe Operations) *For any operation $op : \tau_0 \to \tau_1 \to \tau$, we know $(\Psi, V_0) \in \mathcal{V}[\![\tau_0]\!]$ and $(\Psi, V_1) \in \mathcal{V}[\![\tau_1]\!]$ implies $(\Psi, V_0\ op_i\ V_1) \in \mathcal{V}[\![\tau]\!]$.*

Assumption 3 (Safe View Initialization) *From $\mathbb{M}[\mu, \pi \mapsto \texttt{init}]$, we know that $(\Psi, \mathbb{M}[\mu, \pi, n \mapsto c]) \in \mathcal{V}[\![\tau]\!]$ for some $c : \tau$.*

Definition 7 (Semantic Typing Judgement).
For all χ:

- $\Gamma \vDash E : \tau \ @\ \chi$ *iff* $\forall \Psi, L.\, (\Psi, L) \in \mathcal{L}[\![\Gamma]\!] \implies (\Psi, L, E, \chi) \in \mathcal{E}[\![\tau]\!]$.

For $\chi \neq$ `Host`:

- $\Gamma \vDash S \ @\ \chi$ *iff* $\forall \Psi, L.\, (\Psi, L) \in \mathcal{L}[\![\Gamma]\!] \implies (\Psi, L, S, \chi) \in \mathcal{X}$.
- $\Gamma \vDash C \ @\ \chi$ *iff* $\forall S, \Psi, L.\, (\Gamma \vDash S \ @\ \chi) \wedge (\Psi, L) \in \mathcal{L}[\![\Gamma]\!] \implies (\Psi, L, C; S, \chi) \in \mathcal{X}$.

For $\chi =$ `Host`:

- $\Gamma \vDash S \ @\ \texttt{Host}$ *iff* $\forall \Psi, L.\, (\Psi, L) \in \mathcal{L}[\![\Gamma]\!] \implies (\Psi, L, S) \in \mathcal{G}$.
- $\Gamma \vDash C \ @\ \texttt{Host}$ *iff* $\forall S, \Psi, L.\, (\Gamma \vDash S \ @\ \texttt{Host}) \wedge (\Psi, L) \in \mathcal{L}[\![\Gamma]\!] \implies (\Psi, L, C; S) \in \mathcal{G}$.

We prove that our typing system creates a semantics typing system by showing each rule is sound; these we refer to as compatibility lemmas. Because we have two different notions of semantic correctness for the host versus an execution space, we will need two compatibility lemmas for where their typing rules overlap; that is, *TRet*, *TCom*, *TDeclVar*, *TSetVar*, and *TSetView*.

Proposition 4 (Safe Restriction of Machine Memory Spaces). *For any* χ, $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$ *implies* $(\Psi, \mathbb{M}|_{\mu \triangleright \chi}, \chi) \in \mathcal{M}$.

Lemma 1 (Safe Kernel Step). *For any* Ψ, $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$, $(\Psi, \mathbb{S}) \in \mathcal{S}$ *where* $\mathbb{S}$ *is non-empty, we know the following:*

- *For any* $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel S \rangle\!\rangle \longrightarrow \langle\!\langle \mathbb{M}' \parallel \mathbb{S}' \parallel\!\models L \parallel S \rangle\!\rangle$, *there exists* Ψ' *such that* $\Psi \sqsubseteq \Psi'$, $(\Psi', \mathbb{M}', \texttt{Host}) \in \mathcal{M}$, $(\Psi', \mathbb{S}') \in \mathcal{S}$.
- *There is at least one GState such that* $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel S \rangle\!\rangle \longrightarrow \mathit{GState}$.

Proof.

First, consider an arbitrary step that we may take $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel S \rangle\!\rangle \longmapsto \langle\!\langle \mathbb{M}' \parallel \mathbb{S}' \parallel\!\models L \parallel S \rangle\!\rangle$. By inspection of the transition rules, there are two rules that could apply:

Case *GXPop*:

$$\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel S \rangle\!\rangle \longmapsto \langle\!\langle \mathbb{M} \parallel \mathbb{S}.\mathrm{pop}(\chi) \parallel\!\models L \parallel S \rangle\!\rangle$$

where $\mathbb{S}.\mathrm{head}(\chi) = (L', \texttt{ret})$ for some χ and L'. Here we use the world Ψ for which $\Psi \sqsubseteq \Psi$ by reflexivity. $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$ follows by assumption. $(\Psi, \mathbb{S}.\mathrm{pop}(\chi)) \in \mathcal{S}$ follows by definition and the assumption $(\Psi, \mathbb{S}) \in \mathcal{S}$.

Case *GXStep*:

$$\langle\!\langle \mathbb{M} \parallel \mathbb{S} \parallel\!\models L \parallel S \rangle\!\rangle \longrightarrow \langle\!\langle \mathbb{M}|_{\overline{\mu \triangleright \chi}} \cup \mathbb{M}' \parallel \mathbb{S}.\mathrm{replaceh}(\chi, (L'', S'')) \parallel\!\models L \parallel S \rangle\!\rangle$$

By inversion on this transition, it follows that

$$\langle\!\langle \mathbb{M}|_{\mu \triangleright \chi} \parallel\!\models L' \parallel S' \rangle\!\rangle \longmapsto \langle\!\langle \mathbb{M}' \parallel\!\models L'' \parallel S'' \rangle\!\rangle$$

$(\Psi, \mathbb{M}|_{\mu \triangleright \chi}, \chi) \in \mathcal{M}$ follows by Proposition 4. From the property of $(\Psi, \mathbb{S}) \in \mathcal{S}$, we know that $(\Psi, L', S', \chi) \in \mathcal{X}$. This property with $(\Psi, \mathbb{M}|_{\mu \triangleright \chi}, \chi) \in \mathcal{M}$ yields that $(\Psi, \mathbb{M}', \chi) \in \mathcal{M}$ and $(\Psi, L'', S'', \chi) \in \mathcal{X}$. These facts allow us to conclude both $(\Psi, \mathbb{M}|_{\overline{\mu \triangleright \chi}} \cup \mathbb{M}', \texttt{Host}) \in \mathcal{M}$ and $(\Psi, \mathbb{S}.\mathrm{replaceh}(\chi, (L'', S'''))) \in \mathcal{S}$; $\Psi \sqsubseteq \Psi$ holds by reflexivity.

Second, we must show that a step exists. By $(\Psi, \mathbb{S}) \in \mathcal{S}$ and that it is non-empty, we know that there exists some χ and $(\Psi, L, S, \chi) \in \mathcal{X}$ in its work queue in $\mathbb{S}$. From $(\Psi, L, S, \chi) \in \mathcal{X}$ with $(\Psi, \mathbb{M}|_{\mu \triangleright \chi}, \chi) \in \mathcal{M}$, we know that either $\langle\!\langle \mathbb{M}|_{\mu \triangleright \chi} \parallel\!\models L \parallel S \rangle\!\rangle$ is either final or it takes a step. If it is final, then $S = \texttt{ret}$ by definition and we can step globally by *GXPop*. Otherwise, we can take a step by *GXStep*.

A.2 Compatibility Lemmas

Lemma 2 (*TVar*). *If* $x{:}\tau \in \Gamma$, *then* $\Gamma \vDash x : \tau \mathbin{@} \chi$.

Proof. Considering an arbitrary Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must show that $(\Psi, L, x, \chi) \in \mathcal{E}[\![\tau]\!]$. Thus, we further suppose some $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$. By the definition of $\mathcal{L}[\![\Gamma]\!]$, we know $(\Psi, L(x)) \in \mathcal{V}[\![\tau]\!]$ since $x{:}\tau \in \Gamma$. Immediately, this is enough to conclude both $\langle\!\langle \mathbb{M} \parallel\!\models L \parallel x \rangle\!\rangle \Downarrow L(x)$ by *EVar* and $(\Psi, L(x)) \in \mathcal{V}[\![\tau]\!]$.

Lemma 3 (*TConst*)**.** *If* $c \in B_i$*, then* $\Gamma \vDash c : B_i \mathbin{@} \chi$*.*

Proof. Considering an arbitrary Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must show that $(\Psi, L, c, \chi) \in \mathcal{E}[\![B_i]\!]$. Thus, we further suppose some $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$. Immediately, this is enough to conclude both $\langle\!\langle \mathbb{M} \mid\!\models L \parallel c \rangle\!\rangle \Downarrow c$ by *EConst* and $(\Psi, c) \in \mathcal{V}[\![\tau]\!]$ by $c \in B_i$.

Lemma 4 (*TViewDeref*)**.** *If* $x{:}\texttt{view}(\mu, B_i) \in \Gamma$*,* $\mu \rhd \chi$*, and* $\Gamma \vDash E : \mathbb{N} \mathbin{@} \chi$*, then* $\Gamma \vDash x(E) : B_i \mathbin{@} \chi$*.*

Proof. Considering an arbitrary Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must show that $(\Psi, L, x(E), \chi) \in \mathcal{E}[\![B_i]\!]$. Thus, we further suppose some $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$. By the definition of $\mathcal{L}[\![\Gamma]\!]$ with the initial assumption that $x{:}\texttt{view}(\mu, B_i) \in \Gamma$, we know $(\Psi, L(x)) \in \mathcal{V}[\![\texttt{view}(\mu, B_i)]\!]$. The definition of $\mathcal{V}[\![\texttt{view}(\mu, B_i)]\!]$ further yields that $L(x) = (\mu, \pi)$ and $\Psi(\mu)(\pi) = B_i$. By $\Gamma \vDash E : \mathbb{N} \mathbin{@} \chi$ with Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we have $(\Psi, L, E, \chi) \in \mathcal{E}[\![\mathbb{N}]\!]$. And when combined with $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$, we know $\langle\!\langle \mathbb{M} \mid\!\models L \parallel E \rangle\!\rangle \Downarrow n$ and $(\Psi, n) \in \mathcal{V}[\![\mathbb{N}]\!]$. This latter fact gives that $n \in \mathbb{N}$. By the property of $\mathcal{M}$ and the initial assumption $\mu \rhd \chi$, we know $\mathbb{M}(\mu)(\pi)(n) \in B_i$ and further that $(\Psi, \mathbb{M}(\mu)(\pi)(n)) \in \mathcal{V}[\![B_i]\!]$. The *EViewDeref* rule with $\langle\!\langle \mathbb{M} \mid\!\models L \parallel E \rangle\!\rangle \Downarrow n$ with the previous fact is enough to prove $\langle\!\langle \mathbb{M} \mid\!\models L \parallel x(E) \rangle\!\rangle \Downarrow \mathbb{M}(\mu)(\pi)(n)$.

Lemma 5 (*TOp*)**.** *If* $\Gamma \vDash E_0 : \tau_0 \mathbin{@} \chi$*,* $\Gamma \vDash E_1 : \tau_1 \mathbin{@} \chi$*, and* $op_i : \tau_0 \to \tau_1 \to \tau$*, then* $\Gamma \vDash E_0 \ op_i \ E_1 : \tau \mathbin{@} \chi$*.*

Proof. Considering an arbitrary Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must show that $(\Psi, L, E_0 \ op_i \ E_1, \chi) \in \mathcal{E}[\![\tau]\!]$. Thus, further suppose some $(\Psi, \mathbb{M}) \in \mathcal{M}$. By our initial assumption $\Gamma \vDash E_0 : \tau_0 \mathbin{@} \chi$ with Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we know $(\Psi, L, E_0, \chi) \in \mathcal{E}[\![\tau_0]\!]$. From this with $(\Psi, \mathbb{M}) \in \mathcal{M}$, we know $\langle\!\langle \mathbb{M} \mid\!\models L \parallel E_0 \rangle\!\rangle \Downarrow V_0$ and $(\Psi, V_0) \in \mathcal{V}[\![\tau_0]\!]$; we may conclude similar facts about E_1. By our assumptions about value safe operations, $(\Psi, c_0) \in \mathcal{V}[\![\tau_0]\!]$ and $(\Psi, c_1) \in \mathcal{V}[\![\tau_1]\!]$ mean that $(\Psi, c_0 \ op_i \ c_1) \in \mathcal{V}[\![\tau]\!]$. And finally, $\langle\!\langle \mathbb{M} \mid\!\models L \parallel E_0 \ op_i \ E_1 \rangle\!\rangle \Downarrow c_0 \ op_i \ c_1$ by *EOp*.

Lemma 6 (Non-Host *TCom*)**.** *If* $\chi \neq \texttt{Host}$*,* $\Gamma \vDash C \mathbin{@} \chi$*, and* $\Gamma \vDash S \mathbin{@} \chi$*, then* $\Gamma \vDash C; S \mathbin{@} \chi$*.*

Proof. Considering an arbitrary Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must show that $(\Psi, L, C; S, \chi) \in \mathcal{X}$. This follows from the property of the first initial assumption $\Gamma \vDash C \mathbin{@} \chi$ with S, Ψ, L, the second initial assumption $\Gamma \vDash S \mathbin{@} \chi$, and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$.

Lemma 7 (Host *TCom*)**.** *If* $\Gamma \vDash C \mathbin{@} \texttt{Host}$*, and* $\Gamma \vDash S \mathbin{@} \texttt{Host}$*, then* $\Gamma \vDash C; S \mathbin{@} \texttt{Host}$*.*

Proof. Considering an arbitrary Ψ, Σ, and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must show that $(\Psi, L, C; S) \in \mathcal{G}$. This follows from the property of the first initial assumption $\Gamma \vDash C \mathbin{@} \texttt{Host}$ with S, Ψ, L, the second initial assumption $\Gamma \vDash S \mathbin{@} \texttt{Host}$, and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$.

Lemma 8 (Non-Host *TRet*). *If* $\chi \neq \texttt{Host}$*, then* $\Gamma \vDash \texttt{ret} \ @ \ \chi$.

Proof. Considering an arbitrary Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must show that $(\Psi, L, \texttt{ret}, \chi) \in \mathcal{X}$. Thus, we further suppose some $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$. We know $\langle\!\langle \mathbb{M} \mid\!\models L \parallel \texttt{ret} \rangle\!\rangle$ is a final state by definition.

Lemma 9 (Host *TRet*). $\Gamma \vDash \texttt{ret} \ @ \ \texttt{Host}$.

Proof. Considering an arbitrary Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must show that $(\Psi, L, \texttt{ret}) \in \mathcal{H}$. Thus, we further supposing some $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$ and $(\Psi, \mathbb{S}) \in \mathcal{S}$. In the case that $\mathbb{S}$ is empty, $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel S \rangle\!\rangle$ is a final state and we are done. Otherwise, the proof follows by the Lemma 1.

Lemma 10 (Non-Host *TDeclVar*). *If* $\Gamma \vDash E : \tau \ @ \ \chi$ *and* $\Gamma, x{:}\tau \vDash S \ @ \ \chi$*, then* $\Gamma \vDash \texttt{decl} \ x := E; S \ @ \ \chi$.

Proof. Considering an arbitrary Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must show that $(\Psi, \Gamma, \texttt{decl} \ x := E; S, \chi) \in \mathcal{X}$. That is, considering $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$, we must show that there is a unique *XState* such that $\langle\!\langle \mathbb{M} \mid\!\models L \parallel \texttt{decl} \ x := E; S \rangle\!\rangle \longmapsto \mathit{XState}$ where *XState* is well-behaved. Uniqueness is guaranteed by the determinism of $(\longmapsto)$. We do not prove the first disjunct because the original state is not final. By $\Gamma \vDash E : \tau \ @ \ \chi$ with $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we know $(\Psi, L, E, \chi) \in \mathcal{E}[\![\tau]\!]$. This with $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$ yields that $\langle\!\langle \mathbb{M} \mid\!\models L \parallel E \rangle\!\rangle \Downarrow V$ and $(\Psi, V) \in \mathcal{V}[\![\tau]\!]$. Thus, the transition *LDeclVar* applies, thereby making *XState* be $\langle\!\langle \mathbb{M} \mid\!\models L[x \mapsto V] \parallel S \rangle\!\rangle$. We have left to show that this state is well-behavied with respect to the world Ψ. $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$ holds because we have already assumed it. By the definition of $\mathcal{L}[\![\Gamma, x{:}\tau]\!]$ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we can say about the extended local environment that $(\Psi, L[x \mapsto V]) \in \mathcal{L}[\![\Gamma, x{:}\tau]\!]$. And from $\Gamma, x{:}\tau \vDash S \ @ \ \chi$ with Ψ and $(\Psi, L[x \mapsto V]) \in \mathcal{L}[\![\Gamma, x{:}\tau]\!]$, we know that $(\Psi, L[x \mapsto V], S, \chi) \in \mathcal{X}$.

Lemma 11 (Host *TDeclVar*). *If* $\Gamma \vDash E : \tau \ @ \ \texttt{Host}$ *and* $\Gamma, x{:}\tau \vDash S \ @ \ \texttt{Host}$*, then* $\Gamma \vDash \texttt{decl} \ x := E; S \ @ \ \texttt{Host}$.

Proof. Considering an arbitrary Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must show that $(\Psi, L, \texttt{decl} \ x := E; S) \in \mathcal{G}$. That is, considering $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$ and $(\Psi, \mathbb{S}) \in \mathcal{S}$, we must show that $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel \texttt{decl} \ x := E; S \rangle\!\rangle$ is well-behaved. Since it is not a final state, we must prove the second disjunct.

First, we consider any *GState* such that $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel \texttt{decl} \ x := E; S \rangle\!\rangle \longrightarrow \mathit{GState}$. By inspection of the transition rules, there are three possible cases: *GHStep*, *GXPop*, or *GXStep*. The latter two cases are covered by Lemma 1. For the case of *GHStep*, we know additionally that *LDeclVar* was used locally and thus *GState* is $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L[x \mapsto V] \parallel S \rangle\!\rangle$. We have left to show that this state is well-behavied with respect to a future world. The future world is the same world Ψ, which $\Psi \sqsubseteq \Psi$ holds by reflexivity. Left unchanged, the assumptions $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$ and $(\Psi, \mathbb{S}) \in \mathcal{S}$ hold. By $\Gamma \vDash E : \tau \ @ \ \texttt{Host}$ with $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we know $(\Psi, L, E, \texttt{Host}) \in \mathcal{E}[\![\tau]\!]$. This with $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$ yields that $\langle\!\langle \mathbb{M} \mid\!\models L \parallel E \rangle\!\rangle \Downarrow V$ and $(\Psi, V) \in \mathcal{V}[\![\tau]\!]$. By the definition of $\mathcal{L}[\![\Gamma, x{:}\tau]\!]$ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we can say about the extended local environment that

$(\Psi, L[x \mapsto V]) \in \mathcal{L}[\![\Gamma, x{:}\tau]\!]$. And from $\Gamma, x{:}\tau \vDash S$ @ `Host` with Ψ and $(\Psi, L[x \mapsto V]) \in \mathcal{L}[\![\Gamma, x{:}\tau]\!]$, we know that $(\Psi, L[x \mapsto V], S) \in \mathcal{G}$.

Second, we need to show that a step is always possible. This follows from the assumption $\Gamma \vDash E : \tau$ @ `Host`. That is, we know that we can generate a well-behaved local environment and take a step with *LDeclVar* and *GHStep*.

Lemma 12 (*TDeclView***).**
If $\Gamma, x{:}\texttt{view}(\mu, B_i) \vDash S$ @ `Host`*, then* $\Gamma \vDash$ `decl` x `in` $\mu; S$ @ `Host`.

Proof. Considering an arbitrary Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must be able to show that $(\Psi, L, \texttt{decl}\ x\ \texttt{in}\ \mu; S) \in \mathcal{G}$. That is, further considering $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$ and $(\Psi, \mathbb{S}) \in \mathcal{S}$, that $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel \texttt{decl}\ x\ \texttt{in}\ \mu; S \rangle\!\rangle$ is well-behaved. Since it is not a final state, we must prove the second disjunct.

First, we consider any *GState* such that $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel \texttt{decl}\ x\ \texttt{in}\ \mu; S \rangle\!\rangle \longrightarrow$ *GState*. By inspection of the transition rules, there are three possible cases: *GDeclView*, *GXPop*, and *GXStep*. The latter two cases are handled by Lemma 1. For *GDeclView*, *GState* is $\langle\!\langle \mathbb{M}[\mu, \pi \mapsto \texttt{init}] \parallel \mathbb{S} \mid\!\models L[x \mapsto (\mu, \pi)] \parallel S \rangle\!\rangle$ and we must show that this is well-behaved with respect to a future world. That future world is $\Psi[\mu, \pi \mapsto B_i]$. By the definition of $\mathcal{V}[\![\texttt{view}(\mu, B_i)]\!]$, we know that $(\Psi[\mu, \pi \mapsto B_i], (\mu, \pi)) \in \mathcal{V}[\![\texttt{view}(\mu, B_i)]\!]$. Since $\Psi \sqsubseteq \Psi[\mu, \pi \mapsto B_i]$ and type environment realizers are closed under future worlds, we have that $(\Psi[\mu, \pi \mapsto B_i], L) \in \mathcal{L}[\![\Gamma]\!]$. By the definition of $\mathcal{L}[\![\Gamma, x{:}\texttt{view}(\mu, B_i)]\!]$ and $(\Psi[\mu, \pi \mapsto B_i], L) \in \mathcal{L}[\![\Gamma]\!]$, we know that $(\Psi[\mu, \pi \mapsto B_i], L[x \mapsto (\mu, \pi)]) \in \mathcal{L}[\![\Gamma, x{:}\texttt{view}(\mu, B_i)]\!]$. And from the initial assumption $\Gamma, x{:}\texttt{view}(\mu, B_i) \vDash S$ @ `Host` with $\Psi[\mu, \pi \mapsto B_i]$ and $(\Psi[\mu, \pi \mapsto B_i], L[x \mapsto (\mu, \pi)]) \in \mathcal{L}[\![\Gamma, x{:}\texttt{view}(\mu, B_i)]\!]$, we know that $(\Psi[\mu, \pi \mapsto B_i], L[x \mapsto (\mu, \pi)], S) \in \mathcal{G}$. Our safe initialization assumption for views gives that $(\Psi[\mu, \pi \mapsto B_i], \mathbb{M}[\mu, \pi \mapsto \texttt{init}], \texttt{Host}) \in \mathcal{M}$ and $(\Psi[\mu, \pi \mapsto B_i], \mathbb{S}) \in \mathcal{S}$ by the Kripke property of $\mathcal{S}$.

Second, we need to show that there is always a step we can take. Because of our assumption that there is always $\pi \notin \mathrm{Dom}(\mathbb{M}(\mu))$, we can always step with *GDeclView*.

Lemma 13 (Non-Host ***TSetVar*****).**
If $x{:}\tau \in \Gamma$ *and* $\Gamma \vDash E : \tau$ @ χ*, then* $\Gamma \vDash$ `set` $x := E$ @ χ.

Proof. Considering an arbitrary Ψ, $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, and $\Gamma \vDash S$ @ χ, we must show that $(\Psi, L, \texttt{set}\ x := E; S, \chi) \in \mathcal{X}$. Thus, we futher suppose $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$, and we must show that $\langle\!\langle \mathbb{M} \mid\!\models L \parallel \texttt{set}\ x := E; S \rangle\!\rangle$ is well behaved; since it is not a final state that means proving the second disjunct. By $\Gamma \vDash E : \tau$ @ χ with $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we know $(\Psi, L, E, \chi) \in \mathcal{E}[\![\tau]\!]$. This with $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$ yields that $\langle\!\langle \mathbb{M} \mid\!\models L \parallel E \rangle\!\rangle \Downarrow V$ and $(\Psi, V) \in \mathcal{V}[\![\tau]\!]$. By $x{:}\tau \in \Gamma$ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we know that $L(x)$ is defined. Therefore, the *LSetVar* rule applies so that we step to $\langle\!\langle \mathbb{M} \mid\!\models L[x \mapsto V] \parallel S \rangle\!\rangle$. We have left to show that this state is good for the world Ψ. Note that $(\Psi, L[x \mapsto V]) \in \mathcal{L}[\![\Gamma]\!]$ since it merely replaces the old value for x with another well-behaved one: $(\Psi, V) \in \mathcal{V}[\![\tau]\!]$. Thus, using the property of $\Gamma \vDash S$ @ χ with Ψ and $(\Psi, L[x \mapsto V]) \in \mathcal{L}[\![\Gamma]\!]$ gives $(\Psi, L[x \mapsto V], S, \chi) \in \mathcal{X}$. Finally, the property with $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$ by assumption.

Lemma 14 (Host *TSetVar*).
If $x{:}\tau \in \Gamma$ *and* $\Gamma \vDash E : \tau \mathbin{@} \chi$*, then* $\Gamma \vDash \texttt{set}\ x := E \mathbin{@} \chi$.

Proof. Considering an arbitrary Ψ, $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, and $\Gamma \vDash S \mathbin{@} \texttt{Host}$, we must show that $(\Psi, L, \texttt{set}\ x := E; S) \in \mathcal{G}$. Thus, we futher suppose $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$ and $(\Psi, \mathbb{S}) \in \mathcal{S}$, and we must show that $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel \texttt{set}\ x := E; S \rangle\!\rangle$ is well behaved; since it is not a final state that means proving the second disjunct.

First, we consider any *GState* such that $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel \texttt{set}\ x := E; S \rangle\!\rangle \longmapsto$ *GState*. By inspection of the transition rules, there are three cases for which we could have stepped: *GHStep*, *GXPop*, or *GXStep*. The latter two cases are covered by Lemma 1. In the case of *GHStep*, we know that *LSetVar* was used locally and thus *GState* is $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L[x \mapsto V] \parallel S \rangle\!\rangle$ which we must show is well behaved. By $\Gamma \vDash E : \tau \mathbin{@} \texttt{Host}$ with $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we know $(\Psi, L, E, \texttt{Host}) \in \mathcal{E}[\![\tau]\!]$. This with $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$ yields that $\langle\!\langle \mathbb{M} \mid\!\models L \parallel E \rangle\!\rangle \Downarrow V$ and $(\Psi, V) \in \mathcal{V}[\![\tau]\!]$. We have left to show that this state is good for some future world. That future world is Ψ, which is a future world by reflexivity. Note that $(\Psi, L[x \mapsto V]) \in \mathcal{L}[\![\Gamma]\!]$ since it merely replaces the old value for x with another well-behaved one: $(\Psi, V) \in \mathcal{V}[\![\tau]\!]$. Thus, using the property of $\Gamma \vDash S \mathbin{@} \texttt{Host}$ with Ψ and $(\Psi, L[x \mapsto V]) \in \mathcal{L}[\![\Gamma]\!]$ gives $(\Psi, L[x \mapsto V], S) \in \mathcal{G}$. Finally, the properties $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$ and $(\Psi, \mathbb{S}) \in \mathcal{S}$ hold by assumption.

Second, we need to show that there is always a step that we can take. By $x{:}\tau \in \Gamma$ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we know that $L(x)$ is defined. Therefore, the *LSetVar* rule applies so that we step locally and then *GHStep* to step globally.

Lemma 15 (Non-Host *TSetView*). *If* $x{:}\texttt{view}(\mu, B_i) \in \Gamma$*,* $\mu \rhd \chi$*,* $\Gamma \vDash E_0 : \mathbb{N} \mathbin{@} \chi$*, and* $\Gamma \vDash E_1 : B_i \mathbin{@} \chi$*, then* $\Gamma \vDash \texttt{set}\ x(E_0) := E_1 \mathbin{@} \chi$.

Proof. Considering an arbitrary Ψ, $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, and $\Gamma \vDash S \mathbin{@} \chi$, we must show that $(\Psi, \Gamma, \texttt{set}\ x(E_0) := E_1; S, \chi) \in \mathcal{X}$. Thus, we suppose further $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$, and then we have left to show $\langle\!\langle \mathbb{M} \mid\!\models L \parallel \texttt{set}\ x(E_0) := E_1; S \rangle\!\rangle$ is well behaved. Note that it is not a final state so we prove the second disjunct. By the initial assumption $\Gamma \vDash E_0 : \mathbb{N} \mathbin{@} \chi$ with $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we know that $(\Psi, L, E, \chi) \in \mathcal{E}[\![\mathbb{N}]\!]$. From this property $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$, we know that $\langle\!\langle \mathbb{M} \mid\!\models L \parallel E_0 \rangle\!\rangle \Downarrow n$ and $(\Psi, n) \in \mathcal{V}[\![\mathbb{N}]\!]$. We can conclude $\langle\!\langle \mathbb{M} \mid\!\models L \parallel E_1 \rangle\!\rangle \Downarrow c$ and $(\Psi, c) \in \mathcal{V}[\![B_i]\!]$ similarly by the initial assumption $\Gamma \vDash E_1 : B_i \mathbin{@} \chi$. By $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$ and $x{:}\texttt{view}(\mu, B_i) \in \Gamma$, we know that $(\Psi, L(x)) \in \mathcal{V}[\![\texttt{view}(\mu, B_i)]\!]$; thus, $L(x) = (\mu, \pi)$ and $\Psi(\mu)(\pi) = B_i$. Further still, $\Psi(\mu)(\pi) = B_i$ and $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$ with the initial assumption $\mu \rhd \chi$ means that $\mathbb{M}(\mu)(\pi)$ is defined. Therefore, the *LSetView* rule applies so that we can step to the state $\langle\!\langle \mathbb{M}[\mu, \pi, n \mapsto c] \mid\!\models L \parallel S \rangle\!\rangle$, which we have to show is well behaved. Note that $(\Psi, \mathbb{M}[\mu, \pi, n \mapsto c], \chi) \in \mathcal{M}$ since it merely replaces the old value for x with another $(\Psi, c) \in \mathcal{V}[\![B_i]\!]$. Using the property of $\Gamma \vDash S \mathbin{@} \chi$ with Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we know $(\Psi, L, S, \chi) \in \mathcal{X}$.

Lemma 16 (Host *TSetView*). *If* $x{:}\texttt{view}(\mu, B_i) \in \Gamma$*,* $\mu \rhd \texttt{Host}$*,* $\Gamma \vDash E_0 : \mathbb{N} \mathbin{@} \texttt{Host}$*, and* $\Gamma \vDash E_1 : B_i \mathbin{@} \texttt{Host}$*, then* $\Gamma \vDash \texttt{set}\ x(E_0) := E_1 \mathbin{@} \texttt{Host}$.

Proof. Considering an arbitrary Ψ, $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, and $\Gamma \vDash S \mathbin{@} \texttt{Host}$, we must show that $(\Psi, \Gamma, \texttt{set}\ x(E_0) := E_1; S) \in \mathcal{G}$. Thus, we suppose $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$

and $(\Psi, \mathbb{S}) \in \mathcal{S}$, and then we must show $\langle\langle \mathbb{M} \parallel \mathbb{S} \parallel\models L \parallel \texttt{set}\ x(E_0) := E_1; S\rangle\rangle$ is well behaved. Note that it is not a final state so we prove the second disjunct.

First, consider some *GState* where $\langle\langle \mathbb{M} \parallel \mathbb{S} \parallel\models L \parallel \texttt{set}\ x(E_0) := E_1; S\rangle\rangle \longrightarrow$ *GState*. By inspection of the transition rules, there are three possible cases *GHStep*, *GXPop*, and *GXStep*. The latter two cases are covered by Lemma 1. In the case *GHStep*, we know that *LSetView* was used locally so *GState* is $\langle\langle \mathbb{M}[\mu, \pi, n \mapsto c] \parallel \mathbb{S} \parallel\models L \parallel S\rangle\rangle$. This we must show is well behavied. By the initial assumption $\Gamma \vDash E_0 : \mathbb{N}$ @ `Host` with $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we know that $(\Psi, L, E, \texttt{Host}) \in \mathcal{E}[\![\mathbb{N}]\!]$. From this property $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$, we know that $\langle\langle \mathbb{M} \parallel\models L \parallel E_0\rangle\rangle \Downarrow n$ and $(\Psi, n) \in \mathcal{V}[\![\mathbb{N}]\!]$. We can conclude $\langle\langle \mathbb{M} \parallel\models L \parallel E_1\rangle\rangle \Downarrow c$ and $(\Psi, c) \in \mathcal{V}[\![B_i]\!]$ similarly by the initial assumption $\Gamma \vDash E_1 : B_i$ @ `Host`. By $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$ and $x{:}\texttt{view}(\mu, B_i) \in \Gamma$, we know that $(\Psi, L(x)) \in \mathcal{V}[\![\texttt{view}(\mu, B_i)]\!]$; thus, $L(x) = (\mu, \pi)$ and $\Psi(\mu)(\pi) = B_i$. Further still, $\Psi(\mu)(\pi) = B_i$ and $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$ with the initial assumption $\mu \rhd \texttt{Host}$ means that $\mathbb{M}(\mu)(\pi)$ is defined. Note that $(\Psi, \mathbb{M}[\mu, \pi, n \mapsto c], \texttt{Host}) \in \mathcal{M}$ since it merely replaces the old value for x with another $(\Psi, c) \in \mathcal{V}[\![B_i]\!]$. $(\Psi, \mathbb{S}) \in \mathcal{S}$ holds by assumption and $(\Psi, L, S) \in \mathcal{G}$ holds by $\Gamma \vDash S$ @ `Host` with $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$.

Second, we must show that there exists a step that we can take. The reasoning above shows that from the initial assumption that $\langle\langle \mathbb{M} \parallel\models L \parallel E_0\rangle\rangle \Downarrow n$, $\langle\langle \mathbb{M} \parallel\models L \parallel E_1\rangle\rangle \Downarrow c$, and $\mathbb{M}(\mu)(\pi)$ is defined. Thus, we can step locally with *LSetView* and globally with *GHStep*.

Lemma 17 (*TFence*). $\Gamma \vDash \texttt{fence}(\chi)$ @ `Host`.

Proof. Considering some Ψ, $\Gamma \vDash S$ @ `Host`, and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must show that $(\Psi, L, \texttt{fence}(\chi); S) \in \mathcal{G}$. Thus, we further suppose some $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$, $(\Psi, \mathbb{S}) \in \mathcal{S}$. Note that $\langle\langle \mathbb{M} \parallel \mathbb{S} \parallel\models L \parallel \texttt{fence}(\chi); S\rangle\rangle$ is not a final state, so we must prove the second property.

First, let us consider $\langle\langle \mathbb{M} \parallel \mathbb{S} \parallel\models L \parallel \texttt{fence}(\chi); S\rangle\rangle \longrightarrow$ *GState*. There are three possible transitions: *GFence*, *GXPop*, and *GXStep*. Lemma 1 covers the latter two cases. For *GFence*, we know that $\langle\langle \mathbb{M} \parallel \mathbb{S} \parallel\models L \parallel \texttt{fence}(\chi); S\rangle\rangle \longrightarrow \langle\langle \mathbb{M} \parallel \mathbb{S} \parallel\models L \parallel S\rangle\rangle$, which we can conclude by $\Gamma \vDash S$ @ `Host`.

Second, we need to show that there exists a global step. This follows from whether $\mathbb{S}$ contains an empty queue for χ. If so then *GFence* applies; otherwise, either *GXPop* or *GXStep* applies.

Lemma 18 (*TDeepCopy*). *If* $\Gamma \vDash E_0 : \texttt{view}(\mu_0, B_i)$ @ `Host` *and* $\Gamma \vDash E_1 : \texttt{view}(\mu_1, B_i)$ @ `Host`*, then* $\Gamma \vDash \texttt{deep\_copy}(E_0, E_1)$ @ `Host`*.*

Proof. Considering some Ψ, $\Gamma \vDash S$ @ `Host`, and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must show that $(\Psi, L, \texttt{deep\_copy}(E_0, E_1); S) \in \mathcal{G}$. Thus, we further suppose $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$ and $(\Psi, \mathbb{S}) \in \mathcal{S}$. Note that $\langle\langle \mathbb{M} \parallel \mathbb{S} \parallel\models L \parallel \texttt{deep\_copy}(E_0, E_1); S\rangle\rangle$ is not a final state, so we must prove the second disjunct.

First, we consider some $\langle\langle \mathbb{M} \parallel \mathbb{S} \parallel\models L \parallel \texttt{deep\_copy}(E_0, E_1); S\rangle\rangle \longrightarrow$ *GState*, which by inspection of the transition rules could have been *GDeepCopy*, *GXPop*, or *GXStep*. The latter two cases are proved by Lemma 1. In the *GDeepCopy* case, we know that *GState* is $\langle\langle \mathbb{M}[\mu_1, \pi_1 \mapsto \mathbb{M}(\mu_0)(\pi_0)] \parallel \mathbb{S} \parallel\models L \parallel S\rangle\rangle$. We know

$(\Psi, \mathbb{S}) \in \mathcal{S}$ by assumption; however, we have left to show that $(\Psi, \mathbb{M}[\mu_1, \pi_1 \mapsto \mathbb{M}(\mu_0)(\pi_0)], \texttt{Host}) \in \mathcal{M}$ and $(\Psi, L, S) \in \mathcal{G}$ for the world $\Psi \sqsubseteq \Psi$. By the first of the initial assumptions $\Gamma \vDash E_0 : \texttt{view}(\mu_0, B_i)$ @ Host with Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we know $(\Psi, L, E_0, \texttt{Host}) \in \mathcal{E}[\![\texttt{view}(\mu_0, B_i)]\!]$; further using this property with $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}[\![\Psi]\!]$, we know $\langle\!\langle \mathbb{M} \mid\models L \parallel E_0 \rangle\!\rangle \Downarrow (\mu_0, \pi_0)$, $(\Psi, (\mu_0, \pi_0)) \in \mathcal{V}[\![\texttt{view}(\mu_0, B_i)]\!]$, and $\Psi(\mu_0)(\pi_0) = B_i$. We can conclude similar facts from $\Gamma \vDash E_1 : \texttt{view}(\mu_1, B_i)$ @ Host. Since they both have the same type of constants, we can conclude the update is well-behaved $(\Psi, \mathbb{M}[\mu_1, \pi_1 \mapsto \mathbb{M}(\mu_0)(\pi_0)], \chi) \in \mathcal{M}$ for any execution space χ. Finally, $(\Psi, L, S) \in \mathcal{G}$ follows from the assumption $\Gamma \vDash S$ @ Host together with $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$.

Second, we need to know that we may take a step. As the first conjunct showed $\langle\!\langle \mathbb{M} \mid\models L \parallel E_0 \rangle\!\rangle \Downarrow (\mu_0, \pi_0)$ and similarly for the E_1. Thus, the global step *HDeepCopy* always applies.

Lemma 19 (*TKernel*). *If* $\forall i \in 0, \ldots, n.\, x_i{:}\tau_i \in \Gamma$ *and* $x_0{:}\tau_0, \ldots, x_n{:}\tau_n \vDash S$ @ χ, *then* $\Gamma \vDash \texttt{kernel}(\chi, \lambda x_0, \ldots, x_n.\, S)$ @ Host.

Proof. Considering some Ψ, $\Gamma \vDash S'$ @ Host, and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we must show that $(\Psi, L, \texttt{kernel}(\chi, \lambda x_0, \ldots, x_n.\, S); S') \in \mathcal{G}$. Thus, we further consider some well-behaved global memory $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$ and execution space queues $(\Psi, \mathbb{S}) \in \mathcal{S}$. Note that $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\models L \parallel \texttt{kernel}(\chi, \lambda x_0, \ldots, x_n.\, S); S' \rangle\!\rangle$ is not a final state so we must prove the second disjunct.

First, we consider $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\models L \parallel \texttt{kernel}(\chi, \lambda x_0, \ldots, x_n.\, S); S' \rangle\!\rangle \longrightarrow \mathit{GState}$. By inspection of the rule, we know this is by *GKernel*, *GXPop*, or *GXStep*. The latter two cases are proved by Lemma 1. In the *GKernel* case, we know that $\langle\!\langle \mathbb{M} \parallel \mathbb{S}.\text{pusht}(\chi, (L', S)) \mid\models L \parallel S' \rangle\!\rangle$ where L' is $x_0 \mapsto L(x_0), \ldots, x_n \mapsto L(x_n)$; this is defined by the definition of $\mathcal{L}[\![\Gamma]\!]$ and the assumption that $\forall i \in 0, \ldots, n.\, x_i{:}\tau_i \in \Gamma$. We can also conclude that $(\Psi, L') \in \mathcal{L}[\![x_0{:}\tau_0, \ldots, x_n{:}\tau_n]\!]$ by definition and $(\Psi, L(x_i)) \in \mathcal{V}[\![\Gamma]\!]$ for each x_i. By the initial assumption that $x_0{:}\tau_0, \ldots, x_n{:}\tau_n \vDash S$ @ χ with Ψ and $(\Psi, L') \in \mathcal{L}[\![x_0{:}\tau_0, \ldots, x_n{:}\tau_n]\!]$, we know that $(\Psi, L', S, \chi) \in \mathcal{X}$. Thus, we know that $(\Psi, \mathbb{S}.\text{pusht}(\chi, (L', S))) \in \mathcal{S}$ by definition.From the property of the assumption $\Gamma \vDash S'$ @ Host with Ψ and $(\Psi, L) \in \mathcal{L}[\![\Gamma]\!]$, we know that $(\Psi, L, S') \in \mathcal{G}$ thereby concluding this conjunct.

Second, we need to know that we may take a step. Since L' is well-defined by our initial assumptions, *GKernel* is always a valid step here.

A.3 Type Safety

Lemma 20 (Reachability Preserves State Relations).

- *If* $(\Psi, \mathbb{M}, \chi) \in \mathcal{M}$, $(\Psi, L, S, \chi) \in \mathcal{X}$, *and* $\langle\!\langle \mathbb{M} \mid\models L \parallel S \rangle\!\rangle \longmapsto^* \mathit{XState}$, *then* $\mathit{XState} = \langle\!\langle \mathbb{M}' \mid\models L' \parallel S' \rangle\!\rangle$ *such that* $(\Psi', \mathbb{M}', \chi) \in \mathcal{M}$, $(\Psi', L', S', \chi) \in \mathcal{X}$, *and* $\Psi \sqsubseteq \Psi'$ *for some* Ψ', $\mathbb{M}'$, L', *and* S'.
- *If* $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$, $(\Psi, \mathbb{S}) \in \mathcal{S}$, $(\Psi, L, S) \in \mathcal{G}$, *and* $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\models L \parallel S \rangle\!\rangle \longrightarrow^* \mathit{GState}$, *then* $\mathit{GState} = \langle\!\langle \mathbb{M}' \parallel \mathbb{S}' \mid\models L' \parallel S' \rangle\!\rangle$ *such that* $(\Psi', \mathbb{M}', \texttt{Host}) \in \mathcal{M}$, $(\Psi', \mathbb{S}') \in \mathcal{S}$, $(\Psi', L', S') \in \mathcal{G}$, *and* $\Psi \sqsubseteq \Psi'$ *for some* Ψ', $\mathbb{M}'$, $\mathbb{S}'$, L', *and* S'.

Proof.

- Follows by induction on the transition sequence:
  - **Case** the transition sequence is empty so $\mathit{XState} = \langle\!\langle \mathbb{M} \mid\!\models L \parallel S \rangle\!\rangle$ and the properties follow from the original assumptions.
  - **Case** the transition sequence is $\langle\!\langle \mathbb{M} \mid\!\models L \parallel S \rangle\!\rangle \longmapsto \mathit{XState}' \longmapsto^* \mathit{XState}$. Since the original state takes a step, it is not final. By $(\Psi, L, S, \chi) \in \mathcal{X}$ with $(\Psi, \mathbb{M}, \chi)$ and that the state is not final, we know $\mathit{XState}' = \langle\!\langle \mathbb{M}' \mid\!\models L' \parallel S' \rangle\!\rangle$ such that $(\Psi', \mathbb{M}', \chi) \in \mathcal{M}$, $(\Psi', L', S', \chi) \in \mathcal{X}$, and $\Psi \sqsubseteq \Psi'$ for some Ψ', $\mathbb{M}'$, L', and S'. The inductive hypothesis with these properties and the smaller transition sequence $\langle\!\langle \mathbb{M}' \mid\!\models L' \parallel S' \rangle\!\rangle \longmapsto^* \mathit{XState}$ concludes the proof.
- Also follows by induction on the transition sequence:
  - **Case** the transition sequence is empty so $\mathit{GState} = \langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel S \rangle\!\rangle$ and the properties follow from the original assumptions.
  - **Case** the transition sequence is that $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel S \rangle\!\rangle \longrightarrow \mathit{GState}' \longrightarrow^* \mathit{GState}$. Since the original state takes a step, it is not final. By $(\Psi, L, S) \in \mathcal{G}$ with the assumed facts that $(\Psi, \mathbb{M}, \texttt{Host}) \in \mathcal{M}$, $(\Psi, \mathbb{S}) \in \mathcal{S}$, and $\langle\!\langle \mathbb{M} \parallel \mathbb{S} \mid\!\models L \parallel S \rangle\!\rangle \longrightarrow \mathit{GState}'$, we know $\mathit{GState}' = \langle\!\langle \mathbb{M}' \parallel \mathbb{S}' \mid\!\models L' \parallel S' \rangle\!\rangle$ such that $(\Psi', \mathbb{M}', \texttt{Host}) \in \mathcal{M}$, $(\Psi', \mathbb{S}') \in \mathcal{S}$, $\Psi \sqsubseteq \Psi'$, and $(L, S) \neq (L', S')$ implies $(\Psi', L', S') \in \mathcal{G}$ for some Ψ', $\mathbb{M}'$, $\mathbb{S}'$, L', and S'. The inductive hypothesis with these properties and the smaller transition sequence $\langle\!\langle \mathbb{M}' \parallel \mathbb{S}' \mid\!\models L' \parallel S' \rangle\!\rangle \longrightarrow^* \mathit{GState}$ concludes the proof. Note that if $(L, S) = (L', S')$, then we know $(\Psi', L, S) \in \mathcal{G}$ by the Kripke property.

Theorem 3 (Type Soundness).

- *If* $\Gamma \vdash E : \tau \mathbin{@} \chi$*, then* $\Gamma \vDash E : \tau \mathbin{@} \chi$.
- *If* $\Gamma \vdash C \mathbin{@} \chi$*, then* $\Gamma \vDash C \mathbin{@} \chi$.
- *If* $\Gamma \vdash S \mathbin{@} \chi$*, then* $\Gamma \vDash S \mathbin{@} \chi$.

Proof. By mutual induction on the typing derivations and the compatibility lemmas for each typing rule.

Theorem 4 (Semantic Judgements imply Safety). *If* $\vDash S \mathbin{@} \texttt{Host}$*, then* Safe(Init($S$)).

Proof. Note that Init(S) is $\langle\!\langle \varepsilon \parallel \varepsilon \mid\!\models \varepsilon \parallel S \rangle\!\rangle$. Supposing $\langle\!\langle \varepsilon \parallel \varepsilon \mid\!\models \varepsilon \parallel S \rangle\!\rangle \longrightarrow^* \mathit{GState}$, we must show that Final(GState) or that we can take another step. From our initial assumption with the empty world ε and an empty type environment realizer $(\varepsilon, \varepsilon) \in \mathcal{L}[\![\varepsilon]\!]$, we know $(\varepsilon, \varepsilon, S) \in \mathcal{G}$. Moreover, we know trivially that the empty memory state is a realizer $(\varepsilon, \varepsilon, \texttt{Host}) \in \mathcal{M}$ and the empty execution space stack is as well $(\varepsilon, \varepsilon) \in \mathcal{S}$. Thus, Lemma 20 allows us to conclude the proof.

$\boxed{\Delta \vdash \chi :: \text{ES}}$

$$\frac{\texttt{ex}\ x \in \Delta}{\Delta \vdash x :: \text{ES}}\ \mathit{EsVar} \qquad \frac{}{\Delta \vdash \underline{\chi} :: \text{ES}}\ \mathit{EsAvail}$$

$\boxed{\Delta \vdash \mu :: \text{MS}}$

$$\frac{\texttt{mem}\ x \in \Delta}{\Delta \vdash x :: \text{MS}}\ \mathit{MsVar} \qquad \frac{}{\Delta \vdash \underline{\mu} :: \text{MS}}\ \mathit{MsAvail}$$

$\boxed{\Delta \vdash \mu \rhd \chi}$

$$\frac{\Delta \vdash \mu :: \text{MS} \quad \Delta \vdash \chi :: \text{ES} \quad \forall \sigma \in \mathit{Inst}(\Delta), \mu[\sigma] \trianglerighteq \chi[\sigma]}{\Delta \vdash \mu \rhd \chi}$$

$\boxed{\Gamma \vdash_\Delta E : \tau\ @\ \chi}$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash_\Delta x : \tau\ @\ \chi}\ \mathit{TVar} \qquad \frac{c \in B_i}{\Gamma \vdash_\Delta c : B_i\ @\ \chi}\ \mathit{TConst}$$

$$\frac{x{:}\texttt{view}(\mu, B_i) \in \Gamma \quad \Delta \vdash \mu \rhd \chi \quad \Gamma \vdash_\Delta E : \mathbb{N}\ @\ \chi}{\Gamma \vdash_\Delta x(E) : B_i\ @\ \chi}\ \mathit{TViewDeref}$$

$$\frac{\Gamma \vdash_\Delta E_0 : \tau_0\ @\ \chi \quad \Gamma \vdash_\Delta E_1 : \tau_1\ @\ \chi \quad op_i : \tau_0 \to \tau_1 \to \tau}{\Gamma \vdash_\Delta E_0\ op_i\ E_1 : \tau\ @\ \chi}\ \mathit{TOp}$$

$\boxed{\Gamma \vdash_\Delta S\ @\ \chi}$

$$\frac{\Gamma \vdash_\Delta C\ @\ \chi \quad \Gamma \vdash_\Delta S\ @\ \chi}{\Gamma \vdash_\Delta C; S\ @\ \chi}\ \mathit{TCom} \qquad \frac{}{\Gamma \vdash_\Delta \texttt{ret}\ @\ \chi}\ \mathit{TRet}$$

$$\frac{\Gamma \vdash_\Delta E : \tau\ @\ \chi \quad \Gamma, x{:}\tau \vdash_\Delta S\ @\ \chi}{\Gamma \vdash_\Delta \texttt{decl}\ x := E; S\ @\ \chi}\ \mathit{TDeclVar}$$

$$\frac{\Delta \vdash \mu :: \text{MS} \quad \Gamma, x{:}\texttt{view}(\mu, B_i) \vdash_\Delta S\ @\ \texttt{Host}}{\Gamma \vdash_\Delta \texttt{decl}\ x\ \texttt{in}\ \mu; S\ @\ \texttt{Host}}\ \mathit{TDeclView}$$

$\boxed{\Gamma \vdash_\Delta C\ @\ \chi}$

$$\frac{x{:}\tau \in \Gamma \quad \Gamma \vdash_\Delta E : \tau\ @\ \chi}{\Gamma \vdash_\Delta \texttt{set}\ x := E\ @\ \chi}\ \mathit{TSetVar}$$

$$\frac{x{:}\texttt{view}(\mu, B_i) \in \Gamma \quad \Delta \vdash \mu \rhd \chi \quad \Gamma \vdash_\Delta E_0 : \mathbb{N}\ @\ \chi \quad \Gamma \vdash_\Delta E_1 : B_i\ @\ \chi}{\Gamma \vdash_\Delta \texttt{set}\ x(E_0) := E_1\ @\ \chi}\ \mathit{TSetView}$$

$$\frac{}{\Gamma \vdash_\Delta \texttt{fence}(\chi)\ @\ \texttt{Host}}\ \mathit{TFence}$$

$$\frac{\Gamma \vdash_\Delta E_0 : \texttt{view}(\mu_0, B_i)\ @\ \texttt{Host} \quad \Gamma \vdash_\Delta E_1 : \texttt{view}(\mu_1, B_i)\ @\ \texttt{Host}}{\Gamma \vdash_\Delta \texttt{deep\_copy}(E_0, E_1)\ @\ \texttt{Host}}\ \mathit{TDeepCopy}$$

$$\frac{\Delta \vdash \chi :: \text{ES} \quad \forall i \in 0, \ldots, n.\, x_i{:}\tau_i \in \Gamma \quad \chi \neq \texttt{Host} \quad x_0{:}\tau_0, \ldots, x_n{:}\tau_n \vdash_\Delta S\ @\ \chi}{\Gamma \vdash_\Delta \texttt{kernel}(\chi, \lambda x_0, \ldots, x_n.\, S)\ @\ \texttt{Host}}\ \mathit{TKernel}$$

Fig. 9: H-IMP with Space Variables Typing Rules

B Portability Theorem

Definition 8 (Instantiation).

$$\mathit{Inst}(\Delta) = \{\sigma : \mathit{Var} \rightharpoonup \mathit{Inst}.\ \mathit{Space} \mid \mathrm{Dom}(\Delta) \subseteq \mathrm{Dom}(\sigma)\}$$

Lemma 21 (Instantiation Preserves Typing).

- $\Gamma \vdash_\Delta E : \tau \mathbin{@} \chi$ *and* $\sigma \in \mathit{Inst}(\Delta)$ *implies* $\Gamma \vdash E[\sigma] : \tau \mathbin{@} \chi[\sigma]$,
- $\Gamma \vdash_\Delta S \mathbin{@} \chi$ *and* $\sigma \in \mathit{Inst}(\Delta)$ *implies* $\Gamma \vdash S[\sigma] \mathbin{@} \chi[\sigma]$, *and*
- $\Gamma \vdash_\Delta S \mathbin{@} \chi$ *and* $\sigma \in \mathit{Inst}(\Delta)$ *implies* $\Gamma \vdash S[\sigma] \mathbin{@} \chi[\sigma]$.

Proof. Follows by mutual induction on the typing derivation for expressions, statements, and commands. The interesting cases are those from *TViewDeref* and *TSetView* where $\Gamma \vdash \mu \rhd \chi$ must be used to show $\mu[\sigma] \rhd \chi[\sigma]$ in the original type system. This is given by inversion on $\Gamma \vdash \mu \rhd \chi$.

Theorem 5 (Typing Ensures Portability). *If* $\Gamma \vdash_\Delta S \mathbin{@} \texttt{Host}$, *then* S *is portable.*

Proof. From Lemma 21, we know that $\Gamma \vdash S[\sigma] \mathbin{@} \texttt{Host}$ for any instantiation σ. Thus, we know the program is portable by the soundness of H-IMP's types.