Modal Crash Types for WAR-Aware Intermittent Computing

MYRA DOTZEL, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
FARZANEH DERAKHSHAN, Illinois Institute of Technology, Chicago, Illinois, USA
MILIJANA SURBATOVICH, University of Maryland, College Park, Maryland, USA
LIMIN JIA, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA

Programs are executed intermittently on devices that experience arbitrary power failures such as Energy Harvesting Devices (EHDs). To ensure progress, intermittent systems need runtime support to checkpoint state and re-execute after power failure by restoring the last saved state. Such re-execution should be *correct*, i.e., simulated by a continuously powered execution. We study the logical underpinning of intermittent computing and model checkpoint, crash, restore, and re-execution operations as computation on crash types. We draw inspiration from adjoint logic and define crash types by introducing two adjoint modality operators to model persistent and transient memory values of partial (re-)executions and the transitions between them caused by checkpoints and restoration. Our formalism is general enough to accommodate a variety of checkpointing policies. We define a crash type system for a core calculus. To prove the correctness of intermittent systems, we define a novel logical relation for crash types.

CCS Concepts: • **Theory of Computation** → **Semantics and reasoning**; • **Software and its engineering** → *Domain specific languages*; • **Computer systems organization** → Embedded software;

Additional Key Words and Phrases: intermittent computing, modal crash type, logical relation

ACM Reference format:
Myra Dotzel, Farzaneh Derakhshan, Milijana Surbatovich, and Limin Jia. 2025. Modal Crash Types for WAR-Aware Intermittent Computing. *ACM Trans. Program. Lang. Syst.* 47, 2, Article 5 (April 2025), 62 pages.
https://doi.org/10.1145/3716311

1 Introduction

Programs execute intermittently on batteryless **Energy Harvesting Devices (EHDs)**, which are powered solely by energy harvested from the environment (e.g., via solar panel) that is stored in a re-chargeable energy buffer. When the energy buffer is full, the device powers on and begins

This work was generously funded in part through the Office of Naval Research Grant N000142412297 and National Science Foundation Graduate Research Fellowship Program Grants DGE1745016 and DGE2140739. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring organizations.
Authors' Contact Information: Myra Dotzel (corresponding author), Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; e-mail: mdotzel@andrew.cmu.edu; Farzaneh Derakhshan, Illinois Institute of Technology, Chicago, Illinois, USA; e-mail: fderakhshan@iit.edu; Milijana Surbatovich, University of Maryland, College Park, Maryland, USA; e-mail: milijana@umd.edu; Limin Jia, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; e-mail: liminjia@andrew.cmu.edu.

ACM 1558-4593/2025/4-ART5
https://doi.org/10.1145/3716311

program execution, consuming the available energy. When drained of all energy, the device powers off, causing the execution to fail at an arbitrary point in code. The device can recharge and attempt re-execution. EHDs enable new applications in environments where battery maintenance may be costly or infeasible and have wide application in domains such as wildlife monitoring [39], small satellites [31, 40], or smart civil infrastructure [1].

These applications may require running large pieces of code, consuming the energy available in the energy buffer and causing a *power failure*. When power fails, volatile memory (e.g., the program counter) is erased while nonvolatile memory persists. To facilitate forward progress despite these power failures, *intermittent system* support is needed to save state before a power failure and restore the saved state once energy is replenished. We refer to programs running on these devices as *intermittent computations*. This process of checkpointing state, handling power failures, and restoring state repeats until the computation is complete.

Two of the main mechanisms for handling power failures are *atomic execution* [32, 34, 54, 62] and **Just-in-Time (JIT)** checkpointing [4, 5]. Atomic execution saves the program state in a checkpoint upon entering an atomic region and if power fails before the atomic region execution completes, execution resumes from the previously saved checkpoint. Alternatively, JIT checkpointing saves program state immediately before a power failure, and upon reboot, execution resumes from the saved state. Due to code re-execution, atomic regions are prone to *memory consistency bugs* caused by reading results [32] or stale sensor data [57] from past executions of a program. The former manifests in **Write-after-Read (WAR)** patterns where improper checkpointing of variables could cause an execution to compute on the old values of past executions resulting in memory state that could not be achieved by a *continuously powered* program execution. Programs that rely on JIT checkpointing do not re-execute code, so they are free from the above mentioned memory consistency bugs.

Given the inaccessible environments to which many EHDs are deployed, it is crucial that intermittent systems run code *correctly* despite frequent power failures and partial (re-)executions. Recent work [9, 17, 58, 59] provides a formal framework with correctness criteria for reasoning about intermittent executions, where correctness is defined as follows: *Any correct intermittent execution can be simulated by a continuously powered execution* [59]. This criterion ensures that an intermittent execution of a program can generate a result that matches the result of its continuously powered execution regardless of the number of power failures and partial re-executions encountered.

Our work provides the first logical interpretation of the key operations of intermittent execution: *crash*, *restore*, and *re-execute*. To accomplish this goal, we introduce *crash types* which capture the key memory pattern of intermittent computing: Some computations persist across power failures while others do not. In particular, nonvolatile memory state persists across power failures and reboots, while volatile memory does not; results from completed (or checkpointed) computations should persist across power failures, while partially computed results should not. The former we call *stable* values and computations and the latter *unstable* values and computations. The key insight is that the interactions between these stable and unstable components bear close resemblance to shifts $\uparrow, \downarrow$ in adjoint logic [8, 53] for switching between different modalities. Computation of a stable value can only rely on locations that store stable values, while computation on unstable values can rely on both stable and unstable values. In particular, the checkpoint and restore operations correspond to these shifts and are internalized in the definition of crash types; checkpointing moves data from volatile memory (unstable) into nonvolatile memory (stable), and restore performs the inverse operation.

We define a core calculus for intermittent computing and develop a type system for crash types by using the two adjoint modality operators. The crash type of an intermittent computation is: $\mathtt{C}_{\mathtt{unit}} = \downarrow(\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}}) \vee \downarrow\uparrow\mathtt{unit}$ which says that the computation will either encounter a

power failure (the left disjunct), or succeed in producing a stable value and commit its values to nonvolatile memory (the right disjunct). In the former case, the computation is suspended until energy arrives, after which it will attempt re-execution. This recursive definition captures the multiple re-executions of a computation under multiple power failures.

To prove correctness of intermittent executions, we define a logical relation over crash types that relates a continuously powered execution to an intermittent execution. We prove that well-typed programs are self-related, or *semantically well-typed*. We further prove that the intermittent executions of semantically well-typed programs are *idempotent*, meaning that they compute the same results as continuously powered executions.

This article makes the following contributions:

- The first logical interpretation of key operations of intermittent execution.
- Novel crash types to specify how stable and unstable portions of the system and computation interact.
- A core calculus for crash types with progress and preservation to ensure correctness of an execution over a single power cycle.
- A novel logical relation to prove the correctness of an execution over potentially many power cycles.

Outline. The article progresses as follows:

- Section 2 reviews the basics of intermittent computing and a certain class of memory consistency bug caused by re-execution. Through an example, we study a commonly used checkpointing policy that enables correct program re-execution.
- Section 3 discusses crash types and their connection to adjoint logic [8, 53] which inspires the formation of our calculus.
- Sections 4 and 5 present a novel calculus for atomic regions, starting with syntax and operational semantics (Section 4) to a formal type system (Section 5).
- Section 6 defines a semantic typing based on a logical relation to relate intermittent executions with continuously powered executions.
- Section 7 extends the system to handle JIT regions. Due to their re-execution behavior, these regions are not prone to the class of memory consistency bugs that informs the focus of this article. We start with an example and then follow with formal rules. We discuss how to compose these regions that resume execution from the line of failure with regions that re-execute.
- Section 8 gives the precise theorem statements including progress and preservation to ensure correctness over *a single power cycle*, the fundamental theorem of logical relation, which shows that statically well-typed programs are semantically well-typed and adequacy, which shows that semantically well-typed programs are idempotent. We sketch the proofs of the fundamental theorem of logical relation and adequacy which, together, show that *any correct intermittent execution can be simulated by a continuously powered execution.*
- Section 9 shows how to generalize the semantic typing to accommodate custom policies beyond the two investigated in this article.
- Section 10 discusses related work, Section 11 provides further discussion of our system and directions for future work, and Section 12 concludes the article.

Changes w.r.t. the Conference Version. This article extends the conference version [19] with new formalisms to accommodate a more flexible checkpointing policy and includes full proofs. Whereas the original system [19] checkpointed *all* written variables, here we show that it is enough to checkpoint only variables that exhibit WAR patterns in regions that re-execute, a

common optimization of real intermittent systems [32]. The resulting system subsumes the original work [19]. Changes by section are as follows.

Section 2.2 is new. It discusses potential memory consistency bugs caused by WAR patterns in atomic regions. As the new checkpointing policy for atomic regions is the main novelty of this system, it has been restructured from the conference version [19] to first develop the system around atomic regions and then discuss JIT regions later on, as these are not prone to WAR bugs.

Section 4 extends the original calculus [19] to accommodate the new checkpointing policy. We introduce additional qualifiers (Figure 4) for more fine-grained tracking of write accesses and define a memory access transition function that ensures proper memory accesses with these new qualifiers.

Section 5 augments the typing rules for commands with a new post-context that tracks changes in write accesses for variables.

Section 6 provides updated definitions of policies `PwOff`, `Commit`, and `Restore` that reflect the new checkpointing policy. The logical relation is extended with extra conditions to help us show that the final memories of the intermittent and continuously powered executions are the same.

Section 7 focuses on developing the system for JIT regions.

In Section 8, we provide additional explanation of the key theorems. The full proofs are given in Appendices B–D which elaborate on the proof sketches provided in the conference version [19]. In Figure 27, we sketch the updated proof of logical relation which follows a similar structure to our original proof but with additional obligations. We provide more thorough explanation with an improved illustration (Figure 28) that highlights the steps taken during a crash, where the inductive hypothesis is applied, and the base case.

Section 9 includes an updated explanation discussing how our type system accepts or rejects certain branching programs depending on variable qualifiers.

Section 10 is extended with related work on persistent memory and broader discussion on crash consistency.

We added Section 11 to discuss interesting system features and topics of future work.

2 Background

We first briefly review intermittent computing on EHDs (Section 2.1). Next we explain WAR dependencies which are the main cause of potential memory inconsistencies in program regions that re-execute, namely atomic regions (Section 2.2). Then we discuss how intermittent systems save and recover state in the presence of a crash in the context of atomic regions (Section 2.3).

2.1 Intermittent Computing on EHDs

EHDs rely on intermittent system support to save necessary state and reboot in the event of a power failure at checkpoints within the program. The placement and behavior of these checkpoints depends on the *intermittent execution model* under which a program runs. There are two prevailing intermittent execution models: JIT checkpointing [4, 5] and atomic execution [32, 34, 54, 62]. Under a JIT model, the intermittent system saves state immediately before a power failure, transferring necessary volatile state, such as the program counter, into nonvolatile memory. Upon reboot, program execution continues from the same point. Under an atomic execution model, the program is segmented into smaller pieces of code called *atomic regions* by a series of checkpoints (either user-defined or determined automatically [16]). If there is a power failure before an atomic region finishes executing, the system will reboot to the beginning of the atomic region, re-executing that atomic region until it succeeds without power failure. Modern intermittent systems rely on a combined "JIT + Atomics" so that the system switches to JIT checkpointing when not executing a defined atomic region [28, 35, 58].

```
1   Ckpt[a1; read-only:{x,z}; MFstWt:{u}, ckpted:{y}](
2       y := y+z;
3       let w=x-y in
4           if w>0 then
5               u := tt
6           else
7               u := ff);
8   let w = not u in
9       if w then
10          x := x+y;
11          w := ff
12      else
13          skip;
14  skip
```

Fig. 1. An example program with an atomic region and a JIT region.

The re-execution behavior of atomic regions complicates the reasoning about their correctness with respect to memory consistency. Next, we discuss how re-execution could cause memory consistency bugs if checkpointing is not handled properly.

2.2 WAR Dependencies

Memory consistency bugs may occur when intermittently executing an atomic region with WAR dependencies [32, 62]. Such code patterns emerge when a nonvolatile memory location is first read from and then written to. In the first partial execution, the program will execute as intended, reading some initial value and then writing a new value to that memory location. After a power failure and reboot, the execution will resume from the last checkpointed program point; the first read of the memory location will return the value written *by the previous execution*, which may be different from what the value would be if the program executes from the initial state, causing *incorrect* program behavior.

To illustrate a WAR memory consistency bug, we consider an execution of the simple program in Figure 1. It consists of two code blocks: the atomic region `a1` declared with the `Ckpt` construct (Lines 1–7 on the left of Figure 1) and a regular code block executed in JIT mode (Lines 8–14 on the right). For now, we will focus our discussion on the atomic region, for this is where a WAR bug may surface.

The program has four variables stored in nonvolatile memory: x, y, and z of type `int`, and u of type `bool`. For now, ignore the classification of these variables on Line 1. As shown in Figure 2(a), a continuously powered execution of the atomic region with initial state $x = 2, y = 0, z = 1, u = \mathsf{ff}$ ends in $x = 2, y = 1, z = 1, u = \mathsf{tt}$.

Incorrect Atomic Region Execution. Consider the example intermittent execution of the same program, shown in Figure 2(b). Suppose that no variables are checkpointed and that power fails immediately after the execution of Line 5. Once the device recharges, the program execution resumes from the start of the atomic region. If the system does not restore y's original value, this re-run computes with the last saved state $x = 2, y = 1, z = 1, u = \mathsf{tt}$, thus taking the wrong branch and obtaining an incorrect result: $x = 2, y = 2, z = 1, u = \mathsf{ff}$.

The intermittent execution in Figure 2 (b) suffers from insufficient variable checkpointing which results in memory inconsistency. To ensure memory consistency, the intermittent system needs to restore the values of certain nonvolatile locations upon reboot.

Proper Checkpointing. As EHDs are highly resource-constrained, the system should save state judiciously; checkpointing all of nonvolatile memory is expensive and unnecessary. For example, variables in an atomic region that are read-only (i.e., never updated) do not change value and need not be checkpointed. Many intermittent systems follow this design of checkpointing all variables that are not read-only [15, 19, 24, 28, 36, 54, 64]. Yet, prior work has shown that checkpointing only the variables that are read from and then written to (called *WAR variables*) is enough to ensure correct intermittent execution of atomic regions without inputs [59].

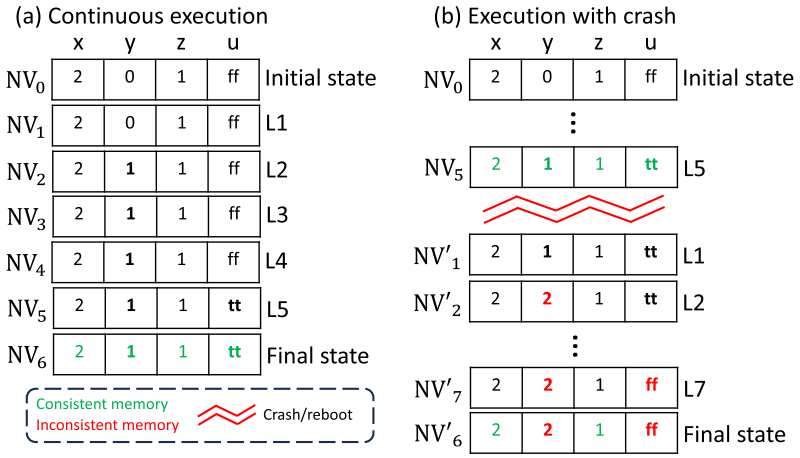

Fig. 2. An example execution of a1 with WAR dependencies *with no variables checkpointed*. NV denotes the nonvolatile memory. The subscript i indicates the memory after executing Line i.

2.3 Correct Atomic Region Execution

In this section, we show that checkpointing only the WAR variables is enough to ensure correct execution of the atomic region in Figure 1. On Line 2 of Figure 1, the variable y is read from and written to making it a WAR variable, and hence it is checkpointed, meaning that its initial checkpointed value persists in nonvolatile memory. However, u does not need to be checkpointed because it is always first written to before it is read, so the effects of previous partial executions on u would have been overwritten by the time it is read. Following prior work [59], we call variables like u *must-first-write* variables.

Each atomic region declares variables as read-only (read-only), must-first-write (MFstWt), or checkpointed (ckpted), as seen on Line 1 of Figure 3. These annotations help our type system check for memory consistency bugs caused by WAR patterns.

Given such a system, Figure 3 shows an execution of the atomic region in Figure 1. For now, ignore the last three columns about typing. To save and restore state, the system follows redo-log semantics. It records updates to checkpointed variables in a special volatile memory. This memory clears if power fails, throwing out partial updates. Upon reaching the next atomic or JIT region, the system commits the updates by copying them back to main memory.

Row (0) shows initial nonvolatile locations, their values, and the mapping between variables and memory locations; locations ℓ_1, ℓ_2, ℓ_3, and ℓ_4 in the nonvolatile memory correspond to variables x, y, z, and u, respectively. When starting to execute the atomic region (Row (1)), the system checkpoints all WAR variables due to the checkpointing policy for atomic regions. In this example, it takes a snapshot of ℓ_2 and stores it in volatile memory. We mark the original nonvolatile location as checkpointed with the superscript ck, i.e., $\ell_2^{\mathtt{ck}}$. Checkpointed location $\ell_2^{\mathtt{ck}}$ remains untouched for the remainder of the atomic region execution. Every access to variable y will instead be associated with its volatile copy ℓ_2, e.g., the assignment in Line 2 is applied to the volatile log of Row (2). When the program executes Line 3, the system adds a new volatile memory location ℓ_5 for variable w on Line 3, corresponding to a volatile execution stack (Row (3)). On Line 5, the program stores the assigned value tt to ℓ_4, the memory location associated with u.

When power fails after Line 5, all volatile memory clears (Row (5)), throwing out the log. The system shuts down until more energy is harvested, at which point the system regenerates the volatile copy ℓ_2 (Row (6)) and resumes execution from Line 2.

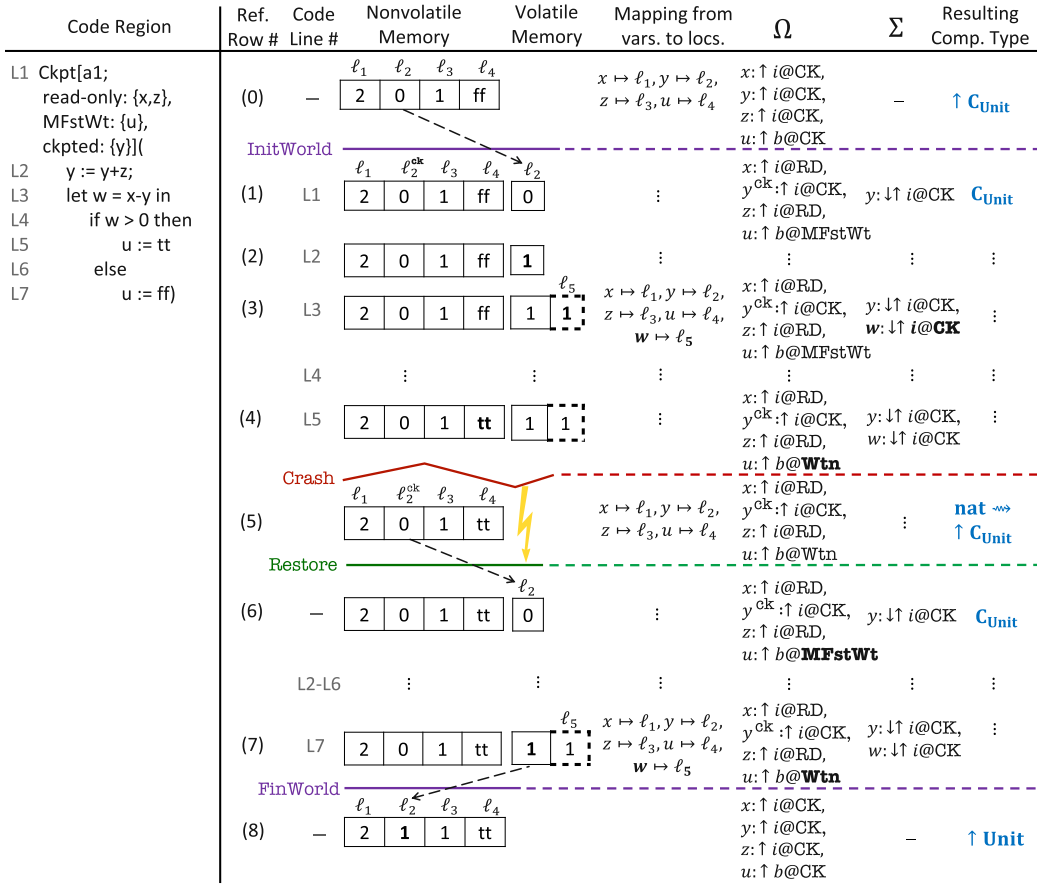

Fig. 3. Intermittent execution of an atomic region. We write i for `int` and b for `bool`.

Rows (6–8) show successful execution of the atomic region. Upon re-execution, program execution proceeds as before, using the checkpointed value stored in ℓ_2^{ck} and values stored in ℓ_1, ℓ_3, and ℓ_4. The successful execution concludes at Row (8) where the value of ℓ_2 in volatile memory is committed to its location ℓ_2^{ck} in nonvolatile memory.

3 Key Ideas of Crash Types

We present the intuition behind the stable and unstable memory types (Section 3.1). Then, we introduce crash types which internalize checkpointing, power failure/crash, restoration, re-execution, and finalization of atomic regions (Section 3.2). Lastly, we discuss the independence principle applied to intermittent computing (Section 3.3).

3.1 Modal Store Types

An unstable value is an intermediate result of a partial execution towards a stable value. Unstable values are lost upon power failure, while stable values persist. If the result of a partial execution is committed to a nonvolatile location, it will become stable and thus persist. To reflect the behavior of a memory location in its type, we introduce two adjoint modalities $\uparrow_u^s$ (read as "up shift from unstable to stable") and $\downarrow_u^s$ (read as "down shift from stable to unstable"), where $\uparrow_u^s \tau^u$ indicates that

the location stores a stable value of type τ and $\downarrow_u^s \tau^s$ indicates that the location stores an intermediate result of an execution toward a value of type τ.

To fully capture the access patterns of memory locations by an intermittent execution, we also annotate the type of a memory location with an access qualifier, `RD`, `CK`, or `MFstWt`, representing whether the location is read-only, checkpointed, or must-first-write (must be first written on an execution), respectively. Variables marked as `MFstWt` or `RD` do not need to be checkpointed. To capture whether a `MFstWt` variable has been written to on the current execution, we introduce the qualifier `Wtn`. By the end of an atomic region execution, all must-first-write variables must be written. That is, all `MFstWt` qualifiers should become `Wtn`.

In our example in Figure 3, the read-only variables x and z are stored in nonvolatile memory, so they have types $x :\uparrow_u^s i@\texttt{RD}$ and $z :\uparrow_u^s i@\texttt{RD}$. The checkpointed variable y has type $y^{\texttt{ck}} :\uparrow_u^s i@\texttt{CK}$ in nonvolatile memory, while y's volatile copy has type $y :\downarrow_u^s\uparrow_u^s i@\texttt{CK}$. The variable w in volatile memory also has unstable type $w :\downarrow_u^s\uparrow_u^s i@\texttt{CK}$. The must-first-write variable u is also stored in nonvolatile memory, so it has type $u :\uparrow_u^s i@\texttt{MFstWt}$. When it is written to on Line 5, its type qualifier changes from `MFstWt` to `Wtn` (Row (4)). If the atomic region has not finished executing before the crash, the restore system reverts all `Wtn` qualifiers back to `MFstWt` and the system executes the atomic region from the beginning (Row (6)). We use the context Ω to type nonvolatile memory and the context Σ to type volatile memory, as shown on the right side of Figure 3.

3.2 Crash Types

To capture the effects of intermittent execution in the type of expressions and commands, we introduce *crash types*, as the notion of stable and unstable values alone is insufficient. One might expect the expression $x - y$ to have the type $\downarrow_u^s\uparrow_u^s\texttt{int}$ as it is a (partial) execution ($\downarrow_u^s$) towards computing a stable ($\uparrow_u^s$) integer value. However, this type does not account for steps due to power failure: the crash itself, waiting for the device to charge, restoration, and re-execution. To reflect these runtime system steps at the type level, we assign the expression a type in the form of a disjunction $\boxed{?} \vee \downarrow_u^s\uparrow_u^s\texttt{int}$, where $\boxed{?}$ is a type for computations that handle power failures. This type means that the expression either fails its execution, or succeeds and evaluates to `int`. Next, we fill in $\boxed{?}$ for commands and expressions. $\boxed{?}$ is a recursive type because it handles re-execution.

Commands. The crash type for commands is: $\texttt{C}_{\texttt{unit}} = \downarrow_u^s(\texttt{nat} \rightsquigarrow \uparrow_u^s \texttt{C}_{\texttt{unit}}) \vee \downarrow_u^s\uparrow_u^s\texttt{unit}$. The right disjunct states that if no power failure occurs while executing a command, then it computes a stable value of type `unit` which is committed to nonvolatile memory. The left disjunct states that on power failure, the computation continues as a function; after a (logical) energy input is received from the environment, the atomic region re-executes from the beginning, which is of stable type ($\uparrow_u^s\texttt{C}_{\texttt{unit}}$).

Expressions. The crash type for expressions[1] is: $\texttt{C}_A^{\texttt{atom}} = \downarrow_u^s(\texttt{nat} \rightsquigarrow \uparrow_u^s \texttt{C}_{\texttt{unit}}) \vee \downarrow_u^s\uparrow_u^s A$. As before, the left disjunct represents power failure: If a logical energy input is received, the atomic region will re-execute. Because atomic regions re-execute from the command enclosed in the region, we use command type $\texttt{C}_{\texttt{unit}}$. The right disjunct types the stable value computed for the expression as a basic type A (`int` or `bool`).

To type a program, we develop a type system for crash types. In the next section, we show the structure of these typing rules and explain their relation to adjoint logic.

[1]For now, we can ignore the annotation atom in $\texttt{C}_A^{\texttt{atom}}$.

3.3 Independence Principle for Typing Intermittent Execution

We design our typing rules to follow the rules for $\downarrow^s_u$ and $\uparrow^s_u$ modalities in adjoint logic [8, 53]. To illustrate their connection, below we show skeletons of our typing rules with terms left as blanks (_) to be filled in later.

We introduce two judgment categories. The first category (J_s) is for deriving stable types and corresponds to the judgments of the form $\Omega \vdash \_ : \tau^s$, meaning that the rules can rely only on stable locations to evaluate computation on a stable type. The second category (J_u) is for deriving unstable types and corresponds to the judgments of form $\Omega; \Sigma \vdash \_ : \tau^u$, meaning that the rules can rely on both stable (Ω) and unstable (Σ) locations to evaluate computation on an unstable type.

The adjoint modalities allow going back and forth between judgments J_s and J_u, mirroring checkpointing and restoration operations. The following four rule skeletons show this back-and-forth behavior in our type system.

$$\frac{\Omega; \cdot \vdash \_ : \tau^u}{\Omega \vdash \_ : \uparrow^s_u \tau^u}\ \uparrow R \qquad \frac{\Omega, \_ : \uparrow^s_u A^u; \Sigma, \_ : \downarrow^s_u \uparrow^s_u A^u \vdash \_ : \tau^u}{\Omega, \_ : \uparrow^s_u A^u; \Sigma \vdash \_ : \tau^u}\ \uparrow L^*$$

$$\frac{\Omega \vdash \_ : \tau^s}{\Omega; \Sigma \vdash \_ : \downarrow^s_u \tau^s}\ \downarrow R \qquad \frac{\Omega, \_ : \uparrow^s_u A^u; \Sigma \vdash \_ : \tau^u}{\Omega; \Sigma, \_ : \downarrow^s_u \uparrow^s_u A^u \vdash \_ : \tau^u}\ \downarrow L$$

Our typing rules are based on sequent calculus rules for adjoint logic [8, 48, 53]. Like sequent calculus style rules, we read them bottom-up and match each execution step of a command with the reading of a corresponding rule. Next, we will explain the switching between stable and unstable modes of the above sequent calculus typing rules using the execution steps in Figure 3.

Shifts (Figure 3). A combination of the rules $\uparrow R$ and $\uparrow L^*$ corresponds to creating a volatile log from the nonvolatile locations when starting the atomic region, i.e., the step from Row (0) to Row (1). The Row (0) type $\uparrow^s_u \mathtt{C}_{\mathtt{unit}}$ and typing context Ω correspond to the conclusion of a $\uparrow R$ rule: $\Omega \vdash \_ : \uparrow^s_u \mathtt{C}_{\mathtt{unit}}$. An application of $\uparrow R$ from bottom to top drops the $\uparrow^s_u$ modality from the type of the program and starts an empty typing context for the volatile memory region: $\Omega; \cdot \vdash \_ : \mathtt{C}_{\mathtt{unit}}$. Next, one application of $\uparrow L^*$ copies the variable y of type $\uparrow^s_u \mathtt{int}$ to the volatile memory typing context Σ with the type $\downarrow^s_u \uparrow^s_u \mathtt{int}$. The same combination corresponds to creating a volatile log from a nonvolatile location when restarting the atomic region: the step from Row (5) to Row (6), where the variable y is copied to the volatile memory.

The $\downarrow R$ rule corresponds to a power failure, which erases the volatile memory typing context Σ. From Row (4) to Row (5) in Figure 3, the system loses the volatile locations of y and w and closes off the volatile context. Row (4) corresponds to the conclusion of the rule, and Row (5) corresponds to its premise. The type of the command in Row (4) changes from $\mathtt{C}_{\mathtt{unit}}$ to $\downarrow^s_u(\mathtt{nat} \rightsquigarrow \uparrow^s_u \mathtt{C}_{\mathtt{unit}})$ (by a logical $\vee$-R rule as a crash is detected), and then to the type $(\mathtt{nat} \rightsquigarrow \uparrow^s_u \mathtt{C}_{\mathtt{unit}})$ in Row (5).

Finally, a $\downarrow L$ rule combined with a standard weakening rule and a $\downarrow R$ rule corresponds to the final commit of the volatile context: stepping from Row (7) to Row (8). There, the nonvolatile context drops the location of y with type $\uparrow^s_u \mathtt{int}$ by a weakening rule since y in the nonvolatile context maps to a location with an outdated value. Next, the up-to-date value stored in the volatile location of y is committed to the nonvolatile location of y and thus y in the Σ context is committed to the Ω context by a $\downarrow L$ rule. Then, a $\downarrow R$ rule drops the remaining volatile context, which contains w of type $\downarrow^s_u \uparrow^s_u \mathtt{int}$. The type of the command in Row (8) becomes $\uparrow^s_u \mathtt{unit}$, signifying that the system detects a successful execution. For the rest of the article, we denote $\downarrow^s_u$ and $\uparrow^s_u$ as simply $\downarrow$ and $\uparrow$.

Commands, expressions, and memories

values	v	::=	$\mathsf{n} \mid \mathsf{tt} \mid \mathsf{ff} \mid x$
exprs	e	::=	$v \mid e \odot e$
cmds	c	::=	$\mathsf{skip} \mid \mathsf{let}\, x = e\, \mathsf{in}\, c \mid \mathsf{if}\, e\, \mathsf{then}\, c\, \mathsf{else}\, c \mid x ::= e \mid c; c \mid c;_W c$
progs	p	::=	$\mathsf{skip} \mid \mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c); p$
access qualifier	q	::=	$\mathtt{CK} \mid \mathtt{RD} \mid \mathtt{MFstWt} \mid \mathtt{Wtn}$
nonvolatile mem	NV	::=	$\cdot \mid \ell \,@\, q \hookrightarrow v, \mathsf{NV} \mid \ell_{\mathsf{ck}} \,@\, \mathsf{CK} \hookrightarrow v, \mathsf{NV}$
volatile mem	V	::=	$\cdot \mid \ell \,@\, \mathsf{CK} \hookrightarrow v, \mathsf{V}$
var loc map	γ	::=	$\cdot \mid \gamma, x \mapsto \ell$

Instructions, statements, and configurations

continuations	κ	::=	$c \mid e$
statements	s	::=	$\kappa \mid i \mid p$
crash instrs	i	::=	$\downarrow\varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa) \mid \varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa) \mid \uparrow \kappa$
energy level	g	::=	$\cdot \mid n$
charge stream	χ	::=	$n :: \chi$
open config	K_o	::=	$(\gamma \mid \mathsf{Md} \mid g \mid \mathsf{NV} \mid \mathsf{V} \mid s) \mid (\gamma \mid \mathsf{Md} \mid g \mid \mathsf{NV} \mid s)$
closed config	K_c	::=	$[\chi \rhd \varepsilon] \otimes K_o$
exec. mode	Md	::=	$\mathsf{aID}(c)$

Fig. 4. Summary of syntax.

4 A Basic Calculus for Intermittent Execution

In this section, we introduce the syntax and operational semantics of our basic calculus for intermittent computing, with a focus on atomic region execution.

4.1 Syntax

The syntactic constructs in our language is summarized in Figure 4. Values include constants and variables. Expressions include values and binary operations of expressions. Commands include skip, mutable let bindings, if branching, assignments, and sequencing. A program, for now, is a sequence of atomic regions. Atomic regions are denoted $\mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c)$ with a unique identifier aID, read-only variables ρ, must-first-write variables μ, checkpointed variables ω, and an enclosed command c.

Nonvolatile memory (NV) and volatile memory (V) map locations ℓ to values v, and each location is annotated with its access mode q (RD, CK, MFstWt, or Wtn). The nonvolatile memory location $\ell_{\mathtt{ck}}$ is the checkpointed copy of location ℓ in volatile memory. The context γ maps variable names to memory locations. Access mode qualifiers in V and NV update throughout execution according to certain constraints (to be discussed in Section 4.2).

The command $c_1;_W c_2$ is a runtime instruction used for evaluating c_1 under the execution context W. The execution context W (written concretely as $\gamma \mid \mathsf{V}$) consists of volatile memory (V) state of in-scope variables and a mapping from these variables to their memory locations (γ).

To model energy harvesting from the environment, we assume a unique external energy channel, ε, from which the system receives energy. Three crash instructions control the system in the event of a power failure. For these, we use shifts $\downarrow$ to denote power failure and $\uparrow$ to denote re-execution. The instruction $\downarrow\varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa)$ models the system that faces a power failure, where κ is the interrupted command or expression, and $b > 0$ is a guard to ensure that the bound incoming energy variable b is positive. The instruction $\varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa)$ models the system awaiting an energy input to be bound to b. The instruction $\uparrow\kappa$ models the system ready to restore memory and re-execute.

We write K_o to denote an *open* system configuration, consisting of the mapping γ, the mode of execution Md (for now, just $\mathsf{aID}(c)$ for atomic regions), energy available for this execution g,

$$\frac{n > 0}{Val(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid \mathtt{skip})}\ (\text{V-SKIP}) \qquad \frac{n > 0}{Val(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid \mathsf{n})}\ (\text{V-NAT}) \qquad \frac{n > 0}{Val(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid \mathtt{tt})}\ (\text{V-BOOL-T})$$

$$\frac{n > 0}{Val(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid \mathtt{ff})}\ (\text{V-BOOL-F}) \qquad \frac{}{Val(\gamma \mid \mathtt{Md} \mid 0 \mid \mathsf{NV} \mid \mathsf{V} \mid \kappa)}\ (\text{V-CRASH})$$

$$\frac{}{Val(\gamma \mid \mathtt{Md} \mid \cdot \mid \mathsf{NV} \mid \mathsf{V} \mid \downarrow\varepsilon \,\#\, \mathtt{in}(b > 0, \uparrow\kappa))}\ (\text{V-}\downarrow) \qquad \frac{}{Val(\gamma \mid \mathtt{Md} \mid \cdot \mid \mathsf{NV} \mid \varepsilon \,\#\, \mathtt{in}(b > 0, \uparrow\kappa))}\ (\text{V-\# in})$$

$$\frac{n > 0}{Val(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \uparrow\kappa)}\ (\text{V-}\uparrow) \qquad \frac{n > 0}{Val([\chi \rhd \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid \mathtt{skip})}\ (\text{V-P-DONE})$$

$$\frac{n > 0}{Val([\chi \rhd \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid \mathtt{skip})}\ (\text{V-C-DONE})$$

Fig. 5. Value configurations.

$$\delta(\mathtt{CK}, Rd) = \delta(\mathtt{CK}, Wt) = \mathtt{CK} \qquad \delta(\mathtt{MFstWt}, Wt) = \mathtt{Wtn} \quad \delta(\mathtt{MFstWt}, Rd) = UN$$
$$\delta(\mathtt{RD}, Rd) = \mathtt{RD} \quad \delta(\mathtt{RD}, Wt) = UN \qquad \delta(\mathtt{Wtn}, Wt) = \delta(\mathtt{Wtn}, Rd) = \mathtt{Wtn}$$

Fig. 6. Definition of the memory access transition function δ.

memories, and the statement s to be executed. The energy level $(\cdot)$ models the state right after power failure. We *close* an open configuration with $[\chi \rhd \varepsilon]$; we connect it via an external energy channel ε to an infinite charging stream χ of natural numbers, which models available energy the configuration harvests from the environment at each power failure for re-execution.

4.2 Operational Semantics

The top-level operational semantic rule for evaluating a sequence of atomic regions is of the form $K_c \Rightarrow_p K_c'$; the operational semantic rules for stepping commands and handling power failures inside an atomic region are of the form $K_c \Rightarrow K_c'$ and rules for evaluating commands and expressions under an open configuration where no power failure occurs are of the form $K_o \rightarrow K_o'$. We first define several auxiliary definitions before explaining each set of the operational semantic rules.

Value Configurations. We call a configuration that cannot take a step a *value configuration* (*value* for short), summarized in Figure 5. An open configuration is a value if the statement s under evaluation is a constant or skip (the first four rules), the configuration of s has depleted all energy for this execution (rule V-CRASH), or s is a crash instruction (rules V-$\downarrow$,V-# in, and V-$\uparrow$). The latter two cases are values because they cannot take a step without interacting with the environment or perform operations on the volatile and nonvolatile memory specific to handling power failures. A closed configuration is a value only if the statement s is skip with some energy left ($n > 0$) (rule V-P-DONE for programs and rule V-C-DONE for commands).

Memory Access Transition Function. To ensure appropriate memory accesses, we define a transition function δ on the access qualifiers based on whether a step writes to (Wt) or reads from (Rd) a memory location. This function is defined in Figure 6. Here, UN represents an undefined or prohibited action which is caused by writing to a read-only location or reading from a must-first-write location that has not yet been written. Writing (Wt) to a must-first-write location causes the qualifier MFstWt to change to Wtn. Any subsequent write or read accesses of such locations are allowed and the type qualifier remains Wtn. Similarly, write or read accesses of checkpointed locations and read accesses of read-only locations are allowed and do not change the qualifier.

$$\frac{\begin{array}{c} n > 0 \quad \mathsf{InitWorld}_d(\mathsf{NV};\rho;\mu;\omega;\gamma) = \mathsf{NV}_0 \mid \mathsf{V}_0 \\ [\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{aID}(c_0) \mid n \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0 \Rightarrow^* [\varepsilon : l'] \otimes \gamma' \mid \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid \mathsf{skip} \\ n' > 0 \quad \mathsf{NV}_1 = \mathsf{FinWorld}_d(\mathsf{NV}';\mathsf{V}') \end{array}}{[\chi \rhd \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid \mathsf{Ckpt}[\mathsf{aID},\rho,\mu,\omega](c_0);p \Rightarrow_p [\varepsilon : l'] \otimes \gamma \mid n' \mid \mathsf{NV}_1 \mid p} \text{ (D-P-Ckpt)}$$

Fig. 7. Operational semantic rule for atomic regions.

$$\frac{\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c \rightarrow \gamma \mid \mathsf{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'}{[\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c \Rightarrow [\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'} \text{ (D-step)}$$

$$\frac{}{[\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{Md} \mid 0 \mid \mathsf{NV} \mid \mathsf{V} \mid c \Rightarrow [\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{Md} \mid \cdot \mid \mathsf{NV} \mid \mathsf{V} \mid \downarrow \varepsilon \,\#\, \mathsf{in}(b > 0; \uparrow c)} \text{ (D-Crash)}$$

$$\frac{\mathsf{Md} = \mathsf{aID}(c_0) \quad \gamma' \subseteq \gamma \quad range(\gamma') = dom(\mathsf{NV})}{[\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{Md} \mid \cdot \mid \mathsf{NV} \mid \mathsf{V} \mid \downarrow \varepsilon \,\#\, \mathsf{in}(b > 0; \uparrow \kappa) \quad \Rightarrow [\chi \rhd \varepsilon] \otimes \gamma' \mid \mathsf{Md} \mid \cdot \mid \mathsf{NV} \mid \varepsilon \,\#\, \mathsf{in}(b > 0; \uparrow \kappa)} \text{ (D-S-aID)}$$

$$\frac{}{[n :: \chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{Md} \mid \cdot \mid \mathsf{NV} \mid \varepsilon \,\#\, \mathsf{in}(b > 0; \uparrow \kappa) \Rightarrow [\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \uparrow \kappa} \text{ (D-charge)}$$

$$\frac{\mathsf{NV} = \mathsf{NV}'_{\mathtt{RD},\mathtt{MFstWt}}, \mathsf{NV}''_{\mathtt{ck}}, \mathsf{NV}'''_{\mathtt{Wtn}}}{\begin{array}{l} [\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{aID}(c_0) \mid n \mid \mathsf{NV} \mid \uparrow \kappa \\ \Rightarrow \; [\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{aID}(c_0) \mid n \mid \mathsf{NV}'_{\mathtt{RD},\mathtt{MFstWt}}, \mathsf{NV}''_{\mathtt{ck}}, \mathsf{NV}'''_{\mathtt{MFstWt}} \mid \mathsf{NV}'' \mid c_0 \end{array}} \text{ (D-restore-aID)}$$

Fig. 8. Operational semantics for commands and crash instructions under a closed configuration.

Top-Level Program Execution (Closed Config). The D-P-Ckpt rule in Figure 7 executes the next atomic region in a program. The nonvolatile (NV_0) and volatile (V_0) locations are initialized using the $\mathsf{InitWorld}_d$ function, which takes as inputs: the current NV, declared read-only, must-first-write, and checkpointed variables ρ, μ, and ω, and their mapping to locations γ. The $\mathsf{InitWorld}_d$ function (a) changes the qualifier of locations in NV that are declared as read-only in ρ from `CK` to `RD`, (b) changes the qualifier of locations in NV that are declared as must-first-write in μ from `CK` to `MFstWt`, (c) checks that the rest of the locations in NV are declared as checkpointed in ω and maintains the qualifier `CK` for these locations, (d) creates V_0 by copying the locations of NV that have qualifier `CK`, and (e) marks the original version of the locations ℓ in NV that still have qualifier `CK` as checkpointed ($\ell_{\mathtt{ck}}$) to indicate that they now store checkpointed values. This part corresponds to the step from Row (0) to Row (1) in Figure 3. The next premise evaluates the closed configuration of c_0 until completion, using the rules in Figure 8. This execution may undergo several power failures and corresponds to the steps from Row (1) to Row (7) in Figure 3. Finally, the $\mathsf{FinWorld}_d$ function closes off atomic regions, finalizing the volatile and nonvolatile locations. $\mathsf{FinWorld}_d$ (a) copies the values of volatile locations in V′ that have a checkpointed version into NV′ to commit the changes to these variables to nonvolatile memory, (b) removes the subscript `ck` from the locations in NV′ and thus converts $\ell_{\mathtt{ck}}$ to ℓ to indicate that they now store up-to-date values and not checkpointed values, and (c) replaces the `RD`, `MFstWt`, and `Wtn` qualifiers of the locations in NV′ with `CK` to reset the access mode since they are out of the atomic region. This corresponds to the step from Row (7) to Row (8) in Figure 3.

Command Execution (Closed Config). We summarize rules for a closed configuration in Figure 8. Rule D-step steps the closed command configuration using the Figure 9 rules. When the energy for this execution is depleted (i.e., $n = 0$), the D-Crash rule applies, stepping the system to the crash instruction $\downarrow \varepsilon \,\#\, \mathsf{in}(b > 0; \uparrow \kappa)$. Upon a crash, D-S-aID applies and drops all volatile memory locations. The D-charge rule steps the execution when a natural number $n > 0$ is received from

$$\frac{n > 0 \quad \gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e \longrightarrow \gamma \mid \mathsf{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid e'}{\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid \mathsf{let}\, x = e \,\mathsf{in}\, c \longrightarrow \gamma \mid \mathsf{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid \mathsf{let}\, x = e' \,\mathsf{in}\, c} \text{ (D-Let-step)}$$

$$\frac{Val(\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e_1) \quad \gamma' = \gamma, [x \mapsto \ell] \quad \ell \ \mathit{fresh} \quad n = n' + 1}{\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid \mathsf{let}\, x = e_1 \,\mathsf{in}\, c \longrightarrow \gamma' \mid \mathsf{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V}, \ell@\mathsf{CK} \hookrightarrow e_1 \mid c} \text{ (D-Let-v)}$$

$$\frac{n > 0 \quad \gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e \longrightarrow \gamma \mid \mathsf{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid e'}{\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid p := e \longrightarrow \gamma \mid \mathsf{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid p := e'} \text{ (D-Assign-step)}$$

$$\frac{Val(\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e) \quad \gamma = \gamma', [x \to \ell] \quad \mathsf{V} = \mathsf{V}', \ell@q \hookrightarrow v' \quad n = n' + 1}{\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid x := e \longrightarrow \gamma \mid \mathsf{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V}', \ell@q \hookrightarrow e \mid \mathsf{skip}} \text{ (D-Assign-V)}$$

$$\frac{Val(\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e) \quad \gamma = \gamma', [x \to \ell] \quad \mathsf{NV} = \mathsf{NV}', \ell@q \hookrightarrow v' \quad q' = \delta(q, Wt) \neq \mathsf{UN} \quad n = n' + 1}{\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid x := e \longrightarrow \gamma \mid \mathsf{Md} \mid n' \mid \mathsf{NV}', \ell@q' \hookrightarrow e \mid \mathsf{V} \mid \mathsf{skip}} \text{ (D-Assign-NV)}$$

$$\frac{n > 0 \quad \gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e \longrightarrow \gamma \mid \mathsf{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid e'}{\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid \mathsf{if}\, e \,\mathsf{then}\, c_1 \,\mathsf{else}\, c_2 \longrightarrow \gamma \mid \mathsf{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid \mathsf{if}\, e' \,\mathsf{then}\, c_1 \,\mathsf{else}\, c_2} \text{ (D-If)}$$

$$\frac{n = n' + 1 \quad Val(\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid \mathsf{tt})}{\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid \mathsf{if}\, \mathsf{tt} \,\mathsf{then}\, c_1 \,\mathsf{else}\, c_2 \longrightarrow \gamma \mid \mathsf{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid c_1} \text{ (D-If-tt)}$$

$$\frac{n = n' + 1 \quad Val(\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid \mathsf{ff})}{\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid \mathsf{if}\, \mathsf{tt} \,\mathsf{then}\, c_1 \,\mathsf{else}\, c_2 \longrightarrow \gamma \mid \mathsf{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid c_2} \text{ (D-If-ff)}$$

$$\frac{}{\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c_1; c_2 \longrightarrow \gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c_{1;\gamma \mid \mathsf{V}}\, c_2} \text{ (D-seq)}$$

$$\frac{\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c_1 \longrightarrow \gamma' \mid \mathsf{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c_1'}{\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c_1;_W c_2 \longrightarrow \gamma' \mid \mathsf{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c_1';_W c_2} \text{ (D-seq-step)}$$

$$\frac{n = n' + 1 \quad W = \gamma' \mid \mathsf{V}' \quad \mathsf{V}'' = \mathsf{V} \upharpoonright \mathrm{dom}(\mathsf{V}')}{\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid \mathsf{skip};_W c_2 \longrightarrow \gamma' \mid \mathsf{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V}'' \mid c_2} \text{ (D-seq-v)}$$

Fig. 9. Operational semantics for command under an open configuration.

the energy channel. The number n represents the energy available for the re-execution. Finally, the program is restored via D-RESTORE-AID which copies checkpointed locations into volatile memory to prepare for re-execution. D-RESTORE-AID maintains the checkpointed locations $\mathsf{NV}''_{\mathsf{CK}}$ and starts re-executing the original command c_0 in the atomic region. Upon restore, all `Wtn` qualifiers are reverted back to `MFstWt`, expressed as $\mathsf{NV}'''_{\mathtt{Wtn}}$ in the premise changing to $\mathsf{NV}'''_{\mathtt{MFstWt}}$ in the resulting configuration.

Command/Expression Execution (Open Config). The rules for executing commands and expressions in an open configuration are standard. We present them in Figures 9 and 10. Each step decrements the energy level by one. The rules ensure that checkpointed location $\ell_{\mathtt{ck}}$ in NV is not read by the program, as it could store outdated data, and is not written to, as this would tamper with the checkpointed value.

The rule D-NV-READ looks up the value v stored in a location ℓ corresponding to variable x. The premise $q' = \delta(q, \mathtt{RD})$ computes the resulting qualifier q' from the original qualifier q and updates the access qualifier in nonvolatile memory accordingly. The rule D-ASSIGN-NV applies when a nonvolatile memory location ℓ is written to via assignment statements. If the computed qualifier q'

$$\frac{\gamma = \gamma', [x \mapsto \ell] \quad \mathsf{V} = \ell@q \hookrightarrow v, \mathsf{V}' \quad n = n' + 1}{\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid x \to \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid v} \text{ (D-V-READ)}$$

$$\frac{\gamma = \gamma', [x \mapsto \ell] \quad q' = \delta(q, \mathtt{RD}) \quad \mathsf{NV} = \ell@q \hookrightarrow v, \mathsf{NV}' \quad \mathsf{NV}'' = \ell@q' \hookrightarrow v, \mathsf{NV}' \quad n = n' + 1}{\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid x \to \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV}'' \mid \mathsf{V} \mid v} \text{ (D-NV-READ)}$$

$$\frac{n > 0 \quad \gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e_1 \to \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid e_1'}{\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e_1 \odot e_2 \to \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid e_1' \odot e_2} \text{ (D-BINARY-1)}$$

$$\frac{n > 0 \quad \mathit{Val}(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e_1) \quad \gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e_2 \to \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid e_2'}{\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e_1 \odot e_2 \to \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid e_1 \odot e_2'} \text{ (D-BINARY-2)}$$

$$\frac{n = n' + 1 \quad \mathit{Val}(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e_1) \quad \mathit{Val}(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e_2) \quad v = e_1 \odot e_2}{\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e_1 \odot e_2 \to \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid v} \text{ (D-BINARY-V)}$$

Fig. 10. Operational semantics for expression under an open configuration.

is defined, the system replaces the value v' already in the location ℓ with e and updates the access qualifier on ℓ from q to q'.

The sequencing rules D-SEQ, D-SEQ-STEP, and D-SEQ-V use a runtime construct W that handles the scoping of volatile locations. The rule D-SEQ steps $c_1; c_2$ to the scoped command $c_1;_{\gamma|\mathsf{V}} c_2$ which initializes the world with a mapping γ and volatile memory state V. To ensure proper scoping, the idea is to remember the original volatile memory V before evaluating the first command c_1 of a sequence $c_1; c_2$, and to revert the volatile state back to V when the execution of the first command completes successfully. This removes any volatile locations allocated with let during the execution of command c_1. The rule D-SEQ-STEP preserves this world while evaluating the first command c_1. To simplify the proofs, we assume that D-SEQ-STEP always applies to the leftmost sequence, meaning that c_1 is not of the form $c';_W c''$. Finally, when c_1 completely executes to skip, the D-SEQ-V steps to c_2 and only keeps those volatile locations that are declared in the original V', and their corresponding mapping γ'. The rule D-LET-V does not remove let-allocated location bindings as this is handled by other parts of the system. If there is a command that follows, this scoping is handled by the rules D-SEQ, D-SEQ-STEP, and D-SEQ-V as described above. If not, these locations will be dropped at the end of the atomic block by the FinWorld function in the D-SEQ-STEP rule as they are freshly introduced and do not have a checkpointed counterpart in nonvolatile memory.

Finally, we define the well-formedness conditions for configurations to constrain proper scoping of memory locations in the presence of sequencing as follows.

Definition 4.1 (Well-Formedness for Configurations). We say that a configuration $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c$ is well-formed iff when $c = c_1;_{W_1} \cdots ;_{W_{n-1}} c_n;_{W_n} c_{n+1}$ where $n > 1$, all of the following hold:

- $dom(\mathsf{NV}_{\mathtt{ck}}) \subseteq dom(\mathsf{V}_j) \subseteq dom(\mathsf{V}_i) \subseteq dom(\mathsf{V})$
- $\gamma_j \subseteq \gamma_i \subseteq \gamma$

where $1 \leq i < j \leq n$ and $W_k = \gamma_k \mid \mathsf{V}_k$ for all $k \in [1, n]$.

Definition 4.1 constrains configurations whose command is of the form $c_1;_{W_1} \cdots ;_{W_{n-1}} c_n;_{W_n} c_{n+1}$, where W_i records volatile memory V_i and mappings γ_i that are in scope for c_i. During execution, volatile memory V grows monotonically except when stepping with D-SEQ-V which discards out-of-scope variables, and the mapping γ grows and shrinks accordingly. For this reason, the locations of V_i (which was stashed in W_i prior to stepping to this configuration) should always be a subset

store types	A	::=	$\texttt{int} \mid \texttt{bool}$
basic types	T	::=	$\texttt{unit} \mid A$
unstable types	τ^u	::=	$T \mid \downarrow \tau^s \mid \tau^u \vee \tau^u \mid \texttt{C}_T^{\texttt{Md}}$
stable types	τ^s	::=	$\texttt{nat} \rightsquigarrow \tau^s \mid \uparrow \tau^u$
type variables	$\texttt{C}_T^{\texttt{Md}}$	::=	$\texttt{C}_{\texttt{unit}} \mid \texttt{C}_A^{\texttt{Md}}$
volatile store typing context	Σ	::=	$\cdot \mid x : \downarrow\uparrow A@\texttt{CK}, \Sigma$
nonvolatile store typing context	Ω	::=	$\cdot \mid x_{\texttt{ck}} : \uparrow A@\texttt{CK}, \Omega \mid x : \uparrow A@q, \Omega$

Fig. 11. Types and typing contexts.

of the locations of V which contain in scope memory locations for the whole command, and hence $dom(\mathsf{V}_i) \subseteq dom(\mathsf{V})$. Similarly, $dom(\mathsf{V}_j) \subseteq dom(\mathsf{V}_i)$ for $i < j$ because D-SEQ-STEP always applies to the leftmost sequence in a command and first executes the left part of the sequence. As a result, the locations in V_i include newly allocated locations while executing the left part of the sequence and the domain of V_j is a subset of the domain of V_i. The checkpointed locations in nonvolatile memory $\mathsf{NV}_{\texttt{ck}}$ always have corresponding locations in volatile memory. The condition $dom(\mathsf{NV}_{\texttt{ck}}) \subseteq dom(\mathsf{V}_j)$ asserts that the checkpointed memory locations are never scoped out of volatile memory. Lastly, the condition $\gamma_j \subseteq \gamma_i \subseteq \gamma$ follows because the mappings γ, γ_i, and γ_j grow and shrink with their respective memories V, V_i, and V_j. This well-formedness definition is used to establish type preservation in Section 8.

5 Static Typing

In this section, we present the type system of our calculus. We begin with types and typing contexts, and build up to typing judgments and static typing rules.

Types and Typing Contexts. The types are summarized in Figure 11. The modalities $\uparrow$ and $\downarrow$ stratify types into stable (τ^s) and unstable (τ^u) layers and provide a mechanism for shifting between the two ($\uparrow\tau^u$ and $\downarrow\tau^s$ layers). Stable types (τ^s) are defined as $\uparrow\tau^u$ and $\texttt{nat} \rightsquigarrow \tau^s$. The latter represents the suspended computation waiting for energy from the environment. We consider $\texttt{nat} \rightsquigarrow \tau^s$ as a stable type because its result is not susceptible to power failure. Unstable types (τ^u) are defined as $\downarrow\tau^s$, type variables $\texttt{C}_T^{\texttt{Md}}$, basic types T, and $\tau^u \vee \tau^u$. The basic types `int` and `bool` are considered unstable because the values, without being explicitly stored in nonvolatile memory, are transient. A type variable $\texttt{C}_T^{\texttt{Md}}$ denotes a type in the set $\{\texttt{C}_{\texttt{unit}}, \texttt{C}_A^{\texttt{Md}}\}$ and implements the recursive nature of crash types. For now, we consider `Md` to be atomic mode (atom). We repeat the crash types below which were introduced in Section 3.

$$\textit{command crash type} \quad \texttt{C}_{\texttt{unit}} = \downarrow(\texttt{nat} \rightsquigarrow \uparrow \texttt{C}_{\texttt{unit}}) \vee \downarrow\uparrow\texttt{unit}$$
$$\textit{expression crash type} \quad \texttt{C}_A^{\texttt{Md}} = \downarrow(\texttt{nat} \rightsquigarrow \uparrow \texttt{C}_{\texttt{unit}}) \vee \downarrow\uparrow A$$

We include the connectives $\vee$ and $\rightsquigarrow$ solely for the purpose of defining crash types; they are not used elsewhere. Defining crash types using these connectives will allow us to define the logical relation in Section 6 based on the intended meaning of its index type. Some well-formed types, e.g., $\texttt{nat} \rightsquigarrow \texttt{nat} \rightsquigarrow \uparrow\texttt{unit}$, are not accepted by our type system. No well-typed configuration is of these types, so these types have no inhabitants.

A nonvolatile store typing context Ω assigns stable types (of form $\uparrow A$) to variables with locations in nonvolatile memory. A volatile store typing context Σ assigns unstable types (of form $\downarrow\uparrow A$) to variables that have locations in volatile memory. We write $x_{\texttt{ck}}$ to refer to a variable with checkpointed location and that has an active volatile log in Σ.

Typing Judgments. Table 1 summarizes all the typing judgments.

Table 1. Typing Judgment Summary

Command	(J_u)	$\texttt{Md} \mid b\,\mathcal{R}\,0 : \texttt{nat} \mid \Omega; \Sigma \vdash_{\texttt{Sig}} c :: \texttt{C}_{\texttt{unit}} \dashv \Omega'$	c could crash
	(J_u)	$\texttt{Md} \mid b : \texttt{nat} \mid \Omega; \Sigma \vdash_{\texttt{Sig}} \texttt{skip} :: \downarrow\uparrow\texttt{unit} \dashv \Omega'$	c will not crash
	(J_s)	$\texttt{Md} \mid b : \texttt{nat} \mid \Omega \vdash_{\texttt{Sig}} \texttt{skip} :: \uparrow\texttt{unit} \dashv \Omega'$	After commit
Expression	(J_u)	$\texttt{Md} \mid b\,\mathcal{R}\,0 : \texttt{nat} \mid \Omega; \Sigma \vdash_{\texttt{RD;Sig}} e :: \texttt{C}_A^{\texttt{Md}}$	e read, could crash
	(J_u)	$\texttt{Md} \mid b : \texttt{nat} \mid \Omega; \Sigma \vdash_{\texttt{RD;Sig}} v :: \downarrow\uparrow A$	e read no crash
	(J_s)	$\texttt{Md} \mid b : \texttt{nat} \mid \Omega \vdash_{\texttt{RD}} v :: \uparrow A$	e read, commit
	(J_u)	$\texttt{Md} \mid b : \texttt{nat} \mid \Omega; \Sigma \vdash_{\texttt{WT}} x :: \downarrow\uparrow A$	Write on x, no crash
	(J_s)	$\texttt{Md} \mid b : \texttt{nat} \mid \Omega \vdash_{\texttt{WT}} x :: \uparrow A$	Write on x, commit
Program	(J_s)	$\texttt{Md} \mid b : \texttt{nat} \mid \Omega \vdash p :: \uparrow\texttt{C}_{\texttt{unit}}$	Before execution
Crash	(J_u)	$\texttt{Md} \mid b = 0 : \texttt{nat} \mid \Omega; \Sigma \vdash_{\texttt{Sig}} \kappa :: \texttt{C}_T^{\texttt{Md}}$	About to crash
	(J_u)	$\texttt{Md} \mid \cdot \mid \Omega; \Sigma \vdash_{\texttt{Sig}} \downarrow\varepsilon \,\#\, \texttt{in}(b > 0, \uparrow\kappa) :: \downarrow(\texttt{nat} \rightsquigarrow \uparrow \texttt{C}_T^{\texttt{Md}})$	Crash state
	(J_s)	$\texttt{Md} \mid \cdot \mid \Omega \vdash_{\texttt{Sig}} \varepsilon \,\#\, \texttt{in}(b > 0, \uparrow\kappa) :: \texttt{nat} \rightsquigarrow \uparrow \texttt{C}_T^{\texttt{Md}}$	Waiting for energy
	(J_s)	$\texttt{Md} \mid b > 0 : \texttt{nat} \mid \Omega \vdash_{\texttt{Sig}} \uparrow\kappa :: \uparrow\texttt{C}_T^{\texttt{Md}}$	Before re-execution

Judgments are also stratified into two varieties, those that derive a stable type (J_s) and those that derive an unstable type (J_u). Unstable judgments J_u have both nonvolatile and volatile store typing contexts which type the state during an unstable computation. Alternatively, J_s judgments include only the nonvolatile typing context because these type stable command state. The structure of our unstable and stable typing judgments help guarantee that the independence principle for intermittent execution is upheld: Values in volatile memory should not influence those in the nonvolatile memory unless a stable state is reached.

The unstable and stable typing judgments are parameterized over the execution mode Md of the expression or command to be typed. The judgment also tracks a variable b corresponding to the current energy level of this execution. b ranges over natural numbers (nat) and is constrained by a relation $\mathcal{R} \in \{\geq, >\}$ or is set to zero. When the constraint is $b \geq 0$, b is effectively unconstrained. The constraint on b determines whether or not a command can evaluate a value without power failure.

The command judgments use a signature context Sig, which stores the typing judgments for the original command of the Ckpt block such that when typing commands restored from a power failure, the signature is used to check that the restored command typing matches the one stored in the signature without needing to derive it again. This is similar to typing recursive functions, and the signature makes the typing derivations finitary and inductive. There are three judgments for command typing. The first judgment is used when the command has not yet successfully finished executing; its next step, depending on its constraint $\mathcal{R}$, may or may not crash. The second judgment types commands with type $\downarrow\uparrow\texttt{unit}$. In this judgment, b no longer needs to be constrained because this judgment only types commands that have succeeded completing the execution. This judgment invokes the third judgment and uses it as a sub-derivation-tree to type the configuration after the volatile log is committed. The post-context Ω' is the nonvolatile context Ω with updated access qualifiers after execution of the command c.

The expressions that are being written to are only of the simple form x. As no execution is required to evaluate x, we consider the operation atomic so its judgment is crash-free, and hence no constraint is required on b. Unlike commands, expression typing judgments do not carry post-contexts because writes, and hence qualifier updates, occur at the command level and not at the expression level. The labels RD and WT indicate read and write accesses in a program. The static type checking compares these labels to variable access qualifiers in the typing contexts to ensure proper memory accesses of variables.

$$\frac{\Omega_0 \mid \Sigma_0 = \mathsf{InitWorld}_t(\Omega;\rho;\mu;\omega) \quad \mathtt{Sig} = \{\mathsf{aID}(c_0) \mid b \geq 0 : \mathsf{nat} \mid \Omega_0; \Sigma_0 \vdash c_0 : \mathsf{C}_{\mathtt{unit}}\} \quad \mathsf{aID}(c_0) \mid b \geq 0 : \mathsf{nat} \mid \Omega_0;\ \Sigma_0 \vdash_{\mathtt{Sig}} c_0 : \mathsf{C}_{\mathtt{unit}} \dashv \Omega' \quad \Omega' \restriction \{\mathtt{MFstWt}\} = \emptyset \quad b : \mathsf{nat} \mid \Omega \vdash p : \uparrow \mathsf{C}_{\mathtt{unit}}}{b : \mathsf{nat} \mid \Omega \vdash \mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c_0); p : \uparrow \mathsf{C}_{\mathtt{unit}}} \text{ (T-P-Ckpt)}$$

Fig. 12. Program typing.

For program typing, we only have one judgment that refers to the type of the program before the execution of its next block starts.

The rest of the judgments type states after a crash. The first judgment has the constraint $b = 0$, which corresponds to the power failure condition. It invokes the second judgment, which types a state right after a crash. The third judgment types the state awaiting energy to continue re-execution, and the final judgment types the state that is ready for restoration and re-execution.

Program Typing. In Figure 12, the T-P-Ckpt rule types the command c_0 enclosed in an atomic region under the mode $\mathsf{aID}(c_0)$ and then types the rest of the program p.

The first premise sets up the initial typing contexts for nonvolatile and volatile memories. The partial function $\mathsf{InitWorld}_t$ initializes the volatile typing context Σ_0 by creating a log of variables in the nonvolatile typing context Ω that are checkpointed. Ω can be uniquely split into Ω^c, Ω^m, and Ω^r, where Ω^r is the set of all read-only locations in Ω, Ω^m is the set of all must-first-write locations in Ω, and Ω^c is the set of all locations that are checkpointed. To formally define this function, we use the following notations.

We write $\Omega \restriction t$ to denote the subset of Ω whose domain includes all the locations in the set t. We define $\Omega_{\mathtt{ck}}$ to be the same mapping as Ω except that a subscript $\mathtt{ck}$ is added to all the variable in the domain of $\Omega_{\mathtt{ck}}$. This is used to generate typing context for checkpointed locations. We define $\downarrow\Omega$ to be the context resulting from adding a $\downarrow$ in front of each type that variables in the domain of Ω are mapped to. This definition is used to generating a typing context for the volatile log used for the checkpointed locations. They are formally defined below:

$$\Omega_{\mathtt{ck}} = \{x_{\mathtt{ck}} : \uparrow A@q \mid x : \uparrow A@q \in \Omega\} \qquad \downarrow\Omega = \{x : \downarrow\uparrow A@q \mid x : \uparrow A@q \in \Omega\}.$$

Next, we define the $\mathsf{InitWorld}_t$ function as follows.

Definition 5.1. $\Omega_0 \mid \Sigma_0 = \mathsf{InitWorld}_t(\Omega;\rho;\mu;\omega)$ iff

— $\mathsf{dom}(\Omega) = \rho \cup \mu \cup \omega$,
— $\rho \cap \mu = \emptyset$, $\mu \cap \omega = \emptyset$, $\rho \cap \omega = \emptyset$,
— $\Omega_0 = \Omega^r, \Omega^m, \Omega^c_{\mathtt{ck}}$, and
— $\Sigma_0 = \downarrow\Omega^c$,

where $\Omega = \Omega^r, \Omega^m, \Omega^c$ and $\Omega^r = \Omega \restriction \rho$, $\Omega^m = \Omega \restriction \mu$, and $\Omega^c = \Omega \restriction \omega$.

If the sets of read-only, must-first-write, and checkpointed variables, ρ, μ, and ω, do not comprise the domain of Ω or are not pairwise disjoint, then the function $\mathsf{InitWorld}_t$ is not defined and type-checking fails.

After constructing the new typing contexts using InitWorld, the rule T-P-Ckpt uses the signature to remember that the atomic region's original command c_0 is typed under memory contexts $\Omega_0; \Sigma_0$. The region is then typed relative to the signature. The nonvolatile post-context Ω' contains updated qualifiers from executing the command c_0. In particular, the post-context helps us check that all must-first-write variables in an atomic region are written by the end of its execution, as guaranteed by the second to last premise $\Omega' \restriction \{\mathtt{MFstWt}\} = \emptyset$. The last premise checks the remainder of the program p with the original typing context Ω, implicitly resetting the qualifiers to $\mathtt{CK}$ for the next program segment.

$$\frac{}{\mathsf{Md} \mid b : \mathsf{nat} \mid \Omega \vdash_{\mathsf{Sig}} \mathsf{skip} : \uparrow\mathtt{unit} \dashv \Omega} \text{(T-Skip)}$$

$$\frac{\Sigma = \downarrow\Sigma' \quad \Omega = \Omega', \Omega''_{\mathsf{ck}} \quad \mathsf{Md} \mid b : \mathsf{nat} \mid \Omega', \Sigma' \vdash_{\mathsf{Sig}} \mathsf{skip} : \uparrow\mathtt{unit} \dashv \Omega_1}{\mathsf{Md} \mid b : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} \mathsf{skip} : \downarrow\uparrow\mathtt{unit} \dashv \Omega_1 \restriction dom(\Omega)} \text{(T-C-Shift)}$$

$$\frac{\mathsf{Md} \mid b : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} \mathsf{skip} : \downarrow\uparrow\mathtt{unit} \dashv \Omega'}{\mathsf{Md} \mid b > 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} \mathsf{skip} : \tau \vee \downarrow\uparrow\mathtt{unit} \dashv \Omega'} \text{(T-$\vee$-Succ)}$$

$$\frac{\mathsf{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{RD;Sig}} e_1 : \mathsf{C}_A^{\mathsf{Md}} \quad \mathsf{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega; \Sigma, x{:}\downarrow\uparrow A@\mathsf{CK} \vdash_{\mathsf{Sig}} c : \tau \dashv \Omega'}{\mathsf{Md} \mid b > 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} \mathsf{let}\, x = e_1 \,\mathsf{in}\, c : \tau \dashv \Omega'} \text{(T-Let)}$$

$$\frac{\mathsf{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{RD;Sig}} e : \mathsf{C}_A^{\mathsf{Md}} \quad \mathsf{Md} \mid b > 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{WT}} x : \downarrow\uparrow A \dashv \Omega'}{\mathsf{Md} \mid b > 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} x := e : \mathsf{C}_{\mathtt{unit}}^{\mathsf{Md}} \dashv \Omega'} \text{(T-Assign)}$$

$$\frac{\begin{array}{c}\mathsf{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{RD;Sig}} e : \mathsf{C}_{\mathtt{bool}}^{\mathsf{Md}} \\ \mathsf{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} c_1 : \tau \dashv \Omega' \quad \mathsf{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} c_2 : \tau \dashv \Omega'\end{array}}{\mathsf{Md} \mid b > 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} \mathsf{if}\, e \,\mathsf{then}\, c_1 \,\mathsf{else}\, c_2 : \tau \dashv \Omega'} \text{(T-If)}$$

$$\frac{\mathsf{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} c_1 : \mathsf{C}_{\mathtt{unit}}^{\mathsf{Md}} \dashv \Omega' \quad \mathsf{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega'; \Sigma \vdash_{\mathsf{Sig}} c_2 : \tau \dashv \Omega''}{\mathsf{Md} \mid b > 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} c_1; c_2 : \tau \dashv \Omega''} \text{(T-seq)}$$

$$\frac{\begin{array}{c}W = \gamma \mid \mathsf{V} \\ \mathsf{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} c_1 : \mathsf{C}_{\mathtt{unit}}^{\mathsf{Md}} \dashv \Omega' \quad \Sigma' = \mathtt{trim}(\Sigma, \mathsf{V}, \gamma) \quad \mathsf{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega'; \Sigma' \vdash_{\mathsf{Sig}} c_2 : \tau \dashv \Omega''\end{array}}{\mathsf{Md} \mid b > 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} c_1 ;_W c_2 : \tau \dashv \Omega''} \text{(T-seq-d)}$$

$$\frac{\mathsf{Md} \mid b = 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} c : \tau \quad \mathsf{Md} \mid b > 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} c : \tau \dashv \Omega'}{\mathsf{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} c : \tau \dashv \Omega'} \text{(T-enough?)}$$

Fig. 13. Command typing.

Command Typing. The typing rules for commands are presented in Figure 13. The T-Skip rule types the command skip at the stable type $\uparrow\mathtt{unit}$. At this point, the command execution is complete and the initial nonvolatile context Ω should match the final nonvolatile context. Rule T-$\vee$-Succ applies when the command successfully completes its execution and still has at least one unit of energy available ($b > 0$) to conclude the execution by committing. In this case, we close off the constraint on the energy level variable and continue typing the command against the type $\downarrow\uparrow \mathtt{unit}$. Rule T-C-Shift is invoked by T-$\vee$-Succ and updates the memory typing contexts by removing checkpointed locations in Ω as now they are not needed, and making locations in Σ stable as now they are committed. This corresponds to the last step of Figure 3. The post-context $\Omega_1 \restriction dom(\Omega)$ discards any variables from Ω_1 that do not occur in Ω. In other words, this operation updates the variables in the post-context from Ω by considering Ω_1 and discarding variables that appear in the volatile context Σ' that are not checkpointed in Ω ensuring that the domain of the post-context matches Ω.

The rules T-Let and T-Assign are mostly standard except that we consider crashes. For example, in typing the command $x := e$, the first premise of T-Assign considers the type of expression e to be the crash type $\mathsf{C}_A^{\mathsf{Md}}$, since the evaluation of e could crash. The constraint on the energy levels for this premise is $b \geq 0$, as we use one energy unit to deconstruct the assignment command. The second premise types the location x to be of type $\downarrow\uparrow A$ because the assignment only occurs when e completes evaluation.

$$\frac{\Omega, \Sigma' = x{:}\uparrow A@q, \Omega_2' \quad q' = \delta(q, Wt) \neq UN}{\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega, \Sigma' \vdash_{\mathtt{WT}} x : \uparrow A \dashv x{:}\uparrow A@q', \Omega_2'} \text{(T-Loc-Write)}$$

$$\frac{\Sigma =\downarrow \Sigma' \quad \Omega = \Omega', \Omega''_{\mathsf{ck}} \quad \mathtt{Md} \mid b : \mathtt{nat} \mid \Omega', \Sigma' \vdash_{\mathtt{WT}} x : \uparrow A \dashv \Omega_1}{\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{WT}} x : \downarrow\uparrow A \dashv \Omega_1 \restriction dom(\Omega)} \text{(T-W-Shift)} \qquad \frac{\Omega(x) = \uparrow A@q}{\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathtt{RD}} x : \uparrow A} \text{(T-Loc-Read)}$$

$$\frac{}{\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathtt{RD}} \mathtt{tt} :\uparrow \mathtt{bool}} \text{(T-Bool-t)} \qquad \frac{}{\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathtt{RD}} \mathtt{ff} : \uparrow\mathtt{bool}} \text{(T-Bool-f)}$$

$$\frac{}{\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathtt{RD}} \mathtt{n} : \uparrow\mathtt{int}} \text{(T-Int)} \qquad \frac{\Sigma = \downarrow\Sigma' \quad \Omega = \Omega', \Omega''_{\mathsf{ck}} \quad \mathtt{Md} \mid b : \mathtt{nat} \mid \Omega', \Sigma' \vdash_{\mathtt{RD}} v : \uparrow A}{\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} v : \downarrow\uparrow A} \text{(T-R-Shift)}$$

$$\frac{\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} x : \downarrow\uparrow A}{\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} x : \tau_1 \vee \downarrow\uparrow A} \text{(T-V-Succ)}$$

$$\frac{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e_1 : \mathsf{C}_T^{\mathtt{Md}} \quad \mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e_2 : \mathsf{C}_{T'}^{\mathtt{Md}} \quad \odot : \uparrow T \times \uparrow T' \to \uparrow T''}{\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e_1 \odot e_2 : \mathsf{C}_{T''}^{\mathtt{Md}}} \text{(T-Binary)}$$

$$\frac{\mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} e : \tau \quad \mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e : \tau}{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e : \tau} \text{(T-enough?)}$$

Fig. 14. Expression typing.

In the rule T-If, the first premise checks that expression e has type $\mathsf{C}_{\mathtt{bool}}^{\mathtt{Md}}$, and the second and third premises type the commands c_1 and c_2. Both branches should end with memories whose variables have the same qualifiers. Programs that have branching that causes variables to have different qualifiers are rejected by the type system. For example, consider a branching program with a must-first-write variable that is written in one branch but not the other. At the end of the branch, the qualifiers in each post-context are inconsistent. Even if that variable is written immediately after the branch, our type system will reject this program. This is one limitation of our type system.

The rule T-seq types the command $c_1; c_2$ by first typing c_1 and then typing c_2. Starting with the initial nonvolatile context Ω, the typing of c_1 yields the intermediary context Ω' with updated qualifiers by executing c_1. In the second premise, Ω' acts as the initial nonvolatile context for typing c_2 which produces the final context Ω''.

The rule T-seq-d applies to the runtime sequencing command where an extra context $W = \gamma \mid \mathsf{V}$ keeping track of the scoping of volatile locations is used. The premise $\Sigma' = \mathtt{trim}(\Sigma, \mathsf{V}, \gamma)$ trims the volatile typing context to remove let variables scoped within c_1 and match the scoped volatile memory V and mapping γ. The trimmed context Σ' is the original context for typing the second command c_2.

The rule T-enough? cases on $b \geq 0$ and considers two cases: $b = 0$ (the first premise) where the execution crashes and $b > 0$ (the second premise) where there is at least one unit of energy available to decompose a command construct, e.g., T-Let or T-Assign.

Expression Typing. Expression typing rules are very similar to those of the commands and they are shown in Figure 14. The T-Loc-Write and T-Loc-Read rules match the location variable x with an existing variable inside the context. T-Loc-Write updates the qualifier of x to reflect that it has been written according to the semantics of δ, implicitly checking that x is not a read-only variable. Note that in the case where a location in volatile memory is being written, the post-context matches the pre-context because the qualifier of x remains Ck. The T-W-Shift rule is similar to T-C-Shift but considers the variable x at type $\downarrow\uparrow A$ under an unstable typing judgment in the conclusion, and

$$\frac{\mathsf{Md} \mid \cdot \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} \downarrow\varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa') : \downarrow(\mathsf{nat} \rightsquigarrow \uparrow \mathsf{C}^{\mathsf{Md}}_{T'})}{\mathsf{Md} \mid b = 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} \kappa' : \downarrow(\mathsf{nat} \rightsquigarrow \uparrow \mathsf{C}^{\mathsf{Md}}_{T'}) \vee \downarrow\uparrow T} \text{ (T-V-Crash)}$$

$$\frac{\mathsf{aID}(c_0) \mid \cdot \mid \Omega \vdash_{\mathsf{Sig}} \varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa') : (\mathsf{nat} \rightsquigarrow \uparrow \mathsf{C}^{\mathsf{Md}}_{\mathsf{unit}})}{\mathsf{aID}(c_0) \mid \cdot \mid \Omega; \Sigma \vdash_{\mathsf{Sig}} \downarrow\varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa') : \downarrow(\mathsf{nat} \rightsquigarrow \uparrow \mathsf{C}^{\mathsf{Md}}_{\mathsf{unit}})} \text{ (T-aID-stop)}$$

$$\frac{\varepsilon \,\#\, \mathsf{in}() : \mathsf{nat} > 0 \quad \mathsf{Md} \mid b > 0 : \mathsf{nat} \mid \Omega \vdash_{\mathsf{Sig}} \uparrow \kappa' : \uparrow \mathsf{C}^{\mathsf{Md}}_{T}}{\mathsf{Md} \mid \cdot \mid \Omega \vdash_{\mathsf{Sig}} \varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow \kappa') : (\mathsf{nat} \rightsquigarrow \uparrow \mathsf{C}^{\mathsf{Md}}_{T})} \text{ (T-Charge)}$$

$$\frac{\Omega = \Omega', \Omega''_{\mathsf{ck}} \quad \mathsf{aID}(c_0) \mid b \geq 0 : \mathsf{nat} \mid \Omega; \downarrow\Omega'' \vdash c_0 : \mathsf{C}_{\mathsf{unit}} \in \mathsf{Sig}}{\mathsf{aID}(c_0) \mid b > 0 : \mathsf{nat} \mid \Omega \vdash_{\mathsf{Sig}} \uparrow\kappa' :\uparrow \mathsf{C}_{\mathsf{unit}}} \text{ (T-aID-Restore)}$$

Fig. 15. Crash, restore, and checkpoint typing.

at $\uparrow A$ under a stable typing judgment in the premise. The post-context $\Omega_1 \restriction dom(\Omega)$ only keeps variables of Ω_1 that occur in Ω, and discards variables in the post-context of the premise Ω_1 if they correspond to variables in Σ'. In T-W-Shift and T-C-Shift, $\Omega_1 \restriction dom(\Omega)$ ensures that the domain of the pre- and post-contexts of a judgment always match, and they only differ on the qualifiers associated with the variables in their domains.

Statement Typing. The typing rules for crash instructions are presented in Figure 15. A crash is detected by the depleted energy level $b = 0$ in the T-V-crash rule. In the premise, the crash instruction $\downarrow\varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa')$ is typed. The T-aID-stop rule simply drops the volatile locations in Σ. The T-Charge rule inputs a new energy level from the energy channel ε. The first premise shows that the energy channel is needed to provide a natural number greater than zero. Finally, the T-aID-Restore rule prepares for re-execution; volatile memory is restored from the checkpointed locations in Ω. The checkpointed locations persist in Ω as we may need them if there is another power failure. Execution continues with the original command c_0 enclosed in the atomic region. Instead of retyping the restored judgments, we check if there are already typing derivations by matching them up with the saved judgment in the signature.

5.1 Store Typing

We present the well-formedness definitions (Figure 16) to ensure the runtime stores NV and V are well-typed with respect to typing contexts Ω and Σ. The rule V-loc checks a volatile location's type with respect to Σ and γ. The rule NV-loc-aID-1 checks non-checkpointed locations in nonvolatile memory by checking that the value stored in the location ℓ allocated for x has the same type as x. The rule NV-loc-aID-2 applies when x has a checkpointed value stored in NV and an up-to-date value stored in the log in V. In this case, the rule ensures that these two values have the same type. Allowing for volatile memory V and the volatile typing context Σ to be $\cdot$, these rules also define well-formedness of NV with respect to Ω which arises when configurations only contain nonvolatile memory.

6 Logical Relation for Intermittent Execution

To prove the correctness of an intermittent execution, we need to show that there exists a corresponding continuous execution that results in the same final nonvolatile memory after an arbitrary number of crashes (idempotency) [59, 60]. In this section, we define a logical relation for relating intermittent and continuously powered executions of a single atomic region. We use the logical relation to introduce a semantic typing for programs consisting of atomic regions. Later, in Section 8,

$$\boxed{\vdash^{\mathsf{aID}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma}$$

$$\frac{}{\vdash^{\mathsf{aID}}_{\gamma} \cdot \mid \cdot : \cdot \mid \cdot}\ (\textsc{Empty})$$

$$\frac{\vdash^{\mathsf{aID}}_{\gamma'} \mathsf{NV} \mid \mathsf{V}' : \Omega \mid \Sigma \quad \mathsf{V} = \mathsf{V}', \ell \mathbin{@} q \hookrightarrow v \quad q = \mathsf{Ck} \quad \gamma = \gamma', [x \mapsto \ell] \quad \mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathsf{RD}} v : \uparrow A}{\vdash^{\mathsf{aID}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma, (x{:}\downarrow\uparrow A@q)}\ (\textsc{V-Loc})$$

$$\frac{\vdash^{\mathsf{aID}}_{\gamma'} \mathsf{NV}' \mid \mathsf{V} : \Omega \mid \Sigma \quad \mathsf{NV} = \mathsf{NV}', \ell \mathbin{@} q \hookrightarrow v \quad q \neq \mathsf{Ck} \quad \gamma = \gamma', [x \mapsto \ell] \quad \mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathsf{RD}} v : \uparrow A}{\vdash^{\mathsf{aID}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega, (x{:}\uparrow A@q) \mid \Sigma}\ (\textsc{NV-Loc-aID-1})$$

$$\frac{\begin{array}{c}\vdash^{\mathsf{aID}}_{\gamma'} \mathsf{NV}' \mid \mathsf{V}' : \Omega \mid \Sigma \quad \mathsf{NV} = \mathsf{NV}', \ell_{\mathsf{ck}} \mathbin{@} q \hookrightarrow v \\ \mathsf{V} = \mathsf{V}', \ell \mathbin{@} q \hookrightarrow v' \quad q = \mathsf{Ck} \quad \gamma = \gamma', [x \mapsto \ell] \quad \mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathsf{RD}} v : \uparrow A \quad \mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathsf{RD}} v' : \uparrow A\end{array}}{\vdash^{\mathsf{aID}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega, (x_{\mathsf{ck}}{:}\uparrow A@q) \mid \Sigma, (x{:}\downarrow\uparrow A@q)}\ (\textsc{NV-Loc-aID-2})$$

Fig. 16. Well-formedness of $\mathsf{NV} \mid \mathsf{V}$ w.r.t. $\Omega \mid \Sigma$.

we show that statically well-typed programs are semantically well-typed and that our semantic typing ensures idempotency.

6.1 Semantic Typing via a Logical Relation

We define a binary logical relation for crash types that relates an intermittent execution with a continuously powered execution for a single atomic region. Since crash types are recursive, we rely on step-indexing to ensure the well-foundedness of the definitions. The step index is the maximum number of executions that an observer would allow for the intermittent execution. Similar to prior work [2, 22, 61], our definition consists of a term relation $\mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^m$ and a value relation $\mathcal{V}[\![\tau]\!]^m$, which relate an open configuration with at most m attempts for executing the region to an open configuration with continuous power. The value relation requires the intermittently executed configuration to be a value configuration.

The logical relation is constructed in such a way that two related configurations will result in the same nonvolatile memories upon completing evaluation of the command, which captures the idempotency requirements. The term and value logical relations are inductively defined over a lexicographic induction on the index m and the structure of the types. Our logical relation is presented in Figure 17.

Auxiliary Definitions. Before explaining details of our logical relation, we define notations and policies used in its definition.

We write K_o^{∞} to denote an open configuration with an infinite energy level: They are of the form $(\gamma \mid \mathtt{Md} \mid \infty \mid \mathsf{NV} \mid \mathsf{V} \mid c)$. Such configuration is used to model a continuously powered execution. We write $K_o \rightarrow^*_{irred} K'_o$ to mean that K_o evaluates to an irreducible configuration K'_o which cannot take any more steps. Since our semantics for commands is deterministic, for each configuration K_o there is exactly one such irreducible configuration. An irreducible configuration might not be a value but rather an ill-typed configuration that cannot take any more steps. Syntactic typing, by enabling a proof of progress and preservation, ensures that an irreducible configuration can only be a value configuration as defined in Figure 5. However, our logical relation does not assume syntactic typing of configurations. Instead, by defining a value relation for each type, the logical relation ensures that an irreducible configuration reachable from a semantically well-typed program is indeed a value configuration.

Binary relation for commands

$$\begin{array}{l}\mathsf{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega \mid \Sigma \Vdash c_1 \leq c_2 : \mathsf{C_{unit}} \\ \text{iff } \forall n, m \geq 0.\ \forall \gamma, \mathsf{NV}, \mathsf{V}. s.t.\ \vdash^{\mathsf{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma. \\ \qquad\qquad (\gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c_1, \gamma \mid \mathsf{Md} \mid \infty \mid \mathsf{NV} \mid \mathsf{V} \mid c_2) \in \mathcal{E}[\![\mathsf{C_{unit}}]\!]^m\end{array}$$

Term relation

$$\begin{array}{lcl}\mathcal{E}[\![\mathsf{C_{unit}}]\!]^{m+1} & = & \{(K_{o1}, K^{\infty}_{o2}) \mid \exists. K'_{o1}\ s.t. K_{o1} \rightarrow^{*}_{irred} K'_{o1} \wedge \exists. K^{\infty'}_{o2}\ s.t. K^{\infty}_{o2} \rightarrow^{*} K^{\infty'}_{o2} \wedge \\ & & \qquad (K'_{o1}, K^{\infty'}_{o2}) \in \mathcal{V}[\![\mathsf{C_{unit}}]\!]^{m+1}\} \\ \mathcal{E}[\![\mathsf{C_{unit}}]\!]^{0} & = & \{(K_{o1}, K^{\infty}_{o2})\}\end{array}$$

Value relation

$$\begin{array}{lcl}\mathcal{V}[\![\uparrow\mathsf{unit}]\!] & = & \{(K_{o1}, K^{\infty}_{o2}) \mid K_{o1} = (\gamma \mid \mathsf{Md} \mid n_1 \mid \mathsf{NV} \mid \mathsf{skip}) \wedge K^{\infty}_{o2} = (\gamma \mid \mathsf{Md} \mid \infty \mid \mathsf{NV} \mid \mathsf{skip})\} \\ \mathcal{V}[\![\downarrow\uparrow\mathsf{unit}]\!] & = & \{(K_{o1}, K^{\infty}_{o2}) \mid K_{o1} = (\gamma_1 \mid \mathsf{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid \mathsf{skip}) \wedge \\ & & \qquad K^{\infty}_{o2} = (\gamma_2 \mid \mathsf{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid \mathsf{skip}) \wedge \\ & & \qquad \mathtt{Commit}(\gamma_i, \mathsf{Md}, \mathsf{NV}_i, \mathsf{V}_i) = \gamma'_i \mid \mathsf{NV}'_i \wedge \\ & & \qquad (\gamma'_1 \mid \mathsf{Md} \mid n_1 \mid \mathsf{NV}'_1 \mid \mathsf{skip}, \gamma'_2 \mid \mathsf{Md} \mid \infty \mid \mathsf{NV}'_2 \mid \mathsf{skip}) \in \mathcal{V}[\![\uparrow\mathsf{unit}]\!]\} \\ \mathcal{V}[\![\uparrow\mathsf{C_{unit}}]\!]^m & = & \{(K_{o1}, K^{\infty}_{o2}) \mid K_{o1} = (\gamma_1 \mid \mathsf{Md} \mid n \mid \mathsf{NV}_1 \mid \uparrow\kappa) \wedge \\ & & \qquad K^{\infty}_{o2} = (\gamma_2 \mid \mathsf{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2) \wedge \\ & & \qquad \mathtt{Restore}(\gamma_1, \mathsf{Md}, \mathsf{NV}_1, \kappa) = \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0 \wedge \\ & & \qquad \mathsf{NV}_0 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}_2 \setminus \{\mathtt{MFstWt}\} \wedge \\ & & \qquad (\gamma_1 \mid \mathsf{Md} \mid n \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0, \gamma_2 \mid \mathsf{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2) \in \mathcal{E}[\![\mathsf{C_{unit}}]\!]^m\} \\ \mathcal{V}[\![\mathsf{nat} \rightsquigarrow \uparrow\mathsf{C_{unit}}]\!]^m & = & \{(K_{o1}, K^{\infty}_{o2}) \mid K_{o1} = (\gamma_1 \mid \mathsf{Md} \mid \cdot \mid \mathsf{NV}_1 \mid \varepsilon \# \mathsf{in}(b > 0, \uparrow\kappa)) \wedge \\ & & \qquad \forall n{>}0. (\gamma_1 \mid \mathsf{Md} \mid n \mid \mathsf{NV}_1 \mid \uparrow\kappa, K^{\infty}_{o2}) \in \mathcal{V}[\![\uparrow \mathsf{C_{unit}}]\!]^m\} \\ \mathcal{V}[\![\downarrow(\mathsf{nat} \rightsquigarrow \uparrow\mathsf{C_{unit}})]\!]^m & = & \{(K_{o1}, K^{\infty}_{o2}) \mid K_{o1} = (\gamma_1 \mid \mathsf{Md} \mid \cdot \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid \downarrow\varepsilon \# \mathsf{in}(b > 0, \uparrow\kappa)) \wedge \\ & & \qquad \mathtt{PwOff}(\gamma_1, \mathsf{Md}, \mathsf{NV}_1, \mathsf{V}_1) = \gamma'_1 \mid \mathsf{V}' \wedge \\ & & \qquad (\gamma'_1 \mid \mathsf{Md} \mid \cdot \mid \mathsf{V}'_{\mathsf{ck}}, \mathsf{NV}_1 \mid \varepsilon \# \mathsf{in}(b > 0, \uparrow\kappa), K^{\infty}_{o2}) \in \mathcal{V}[\![\mathsf{nat} \rightsquigarrow \uparrow\mathsf{C_{unit}}]\!]^m\} \\ \mathcal{V}[\![\mathsf{C_{unit}}]\!]^{m+1} & = & \{(K_{o1}, K^{\infty}_{o2}) \mid K_{o1} = (\gamma_1 \mid \mathsf{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1) \wedge \\ & & \qquad \textit{either } n_1 = 0 \wedge (\gamma_1 \mid \mathsf{Md} \mid \cdot \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid \downarrow\varepsilon \# \mathsf{in}(b > 0, \uparrow c_1), K^{\infty}_{o2}) \\ & & \qquad\qquad \in \mathcal{V}[\![\downarrow(\mathsf{nat} \rightsquigarrow \uparrow \mathsf{C_{unit}})]\!]^m, \\ & & \qquad \textit{or} \quad n_1 > 0 \wedge (\gamma_1 \mid \mathsf{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1, K^{\infty}_{o2}) \in \mathcal{V}[\![\downarrow\uparrow \mathsf{unit}]\!]\}\end{array}$$

Fig. 17. Step-indexed logical relation for crash types.

We write $\mathsf{NV} \setminus \{\mathtt{MFstWt}\}$ to denote the resulting memory after removing all locations with the qualifier `MFstWt` from NV.

To account for the power failure, recovery, and finalizing atomic regions' effects on memory, the logical relation relies on `PwOff`, `Restore`, and `Commit` policies, referred to as power failure, restore, and commit policies, respectively. We formally define these policies in Figure 18. The definitions match the memory operations in the dynamic rules that deal with crash, restore, and re-execution (D-S-aID, D-RESTORE-aID, and D-P-CKPT) for atomic regions, though they can be generalized as we will discuss in Sections 7 and 9.

In Figure 18, the commit policy is defined to capture the effects of finalizing the nonvolatile memory after a successful atomic region execution. The commit policy copies the values of the checkpointed volatile memory locations into the nonvolatile memory and changes the qualifiers of all other locations in the nonvolatile memory to `ck`. Formally: $\mathtt{Commit}(\gamma, \mathsf{aID}(c_0), \mathsf{NV}_1, \mathsf{V}_1) = \gamma' \mid \mathsf{NV}'_{1\,\mathsf{ck}}, \mathsf{V}''$, where $\mathsf{NV}_1 = \mathsf{NV}'_{1\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{NV}''_{\mathsf{ck}}$ and $\mathsf{V}_1 = \mathsf{V}'_1, \mathsf{V}''$ and $\mathsf{dom}(\mathsf{V}'') = \mathsf{dom}(\mathsf{NV}'')$. Additionally, we have that the original map γ covers the committed map γ' ($\gamma' \subseteq \gamma$) and that the locations mapped to by γ' comprise the locations of NV_1, i.e., $range(\gamma') = dom(\mathsf{NV}_1)$.

The restore policy describes the effect of setting up the memories for re-executing an atomic region. The restore policy is defined as $\mathtt{Restore}(\gamma, \mathsf{aID}(c_0), \mathsf{NV}_1, \kappa) = \mathsf{NV}'_{\mathtt{RD,MFstWt}}, \mathsf{NV}''_{\mathsf{ck}}, \mathsf{NV}^3_{\mathtt{MFstWt}} \mid$

$$\frac{\mathsf{NV}_2 = \mathsf{NV}'_{1\,\mathtt{ck}}, \mathsf{V}'' \quad \mathsf{V}_1 = \mathsf{V}'_1, \mathsf{V}'' \quad \mathsf{dom}(\mathsf{V}'') = \mathsf{dom}(\mathsf{NV}'') \quad \mathsf{range}(\gamma') = \mathsf{dom}(\mathsf{NV}_1) \quad \gamma' \subseteq \gamma}{\mathtt{Commit}(\gamma, \mathsf{aID}(c_0), \mathsf{NV}_1, \mathsf{V}_1) = \gamma' \mid \mathsf{NV}_2}$$

$$\frac{\mathsf{NV}_1 = \mathsf{NV}'_{\mathtt{RD},\mathtt{MFstWt}}, \mathsf{NV}''_{\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{Wtn}} \quad \mathsf{NV}_2 = \mathsf{NV}'_{\mathtt{RD},\mathtt{MFstWt}}, \mathsf{NV}''_{\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{MFstWt}} \quad \mathsf{V}_2 = \mathsf{NV}''}{\mathtt{Restore}(\gamma, \mathsf{aID}(c_0), \mathsf{NV}_1, \kappa) = \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c}$$

$$\frac{\gamma' \text{ is the largest restriction of } \gamma \text{ with} \mathit{range}(\gamma') = \mathsf{dom}(\mathsf{NV}_1) \quad \mathsf{V}_2 = \emptyset}{\mathtt{PwOff}(\gamma, \mathsf{aID}(c_0), \mathsf{NV}_1, \mathsf{V}_1) = \gamma' \mid \mathsf{V}_2}$$

Fig. 18. `Commit`, `Restore`, and `PwOff` policy definitions for atomic regions.

$\mathsf{NV}'' \mid c$ where $\mathsf{NV}_1 = \mathsf{NV}'_{\mathtt{RD},\mathtt{MFstWt}}, \mathsf{NV}''_{\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{Wtn}}$, marking all locations with the qualifier `Wtn` with `MFstWt`; these locations hold dirty values that will be rewritten on the next execution before they can be read.

Finally, we define a power off policy to state the effect of the system in the event of a power failure. Because atomic regions already checkpoint the necessary locations before they begin executing, the `PwOff` policy here is defined to lose the volatile memory state upon power failure, i.e., $\mathtt{PwOff}(\gamma, \mathsf{aID}(c_0), \mathsf{NV}_1, \mathsf{V}_1) = \gamma' \mid \emptyset$, where γ' is the largest restriction of γ with $\mathit{range}(\gamma') = \mathsf{dom}(\mathsf{NV}_1)$.

Now we are ready to explain our term and value relations.

Term Relation. A pair of open command configurations of type $\mathtt{C}_{\mathtt{unit}}$ are in the term relation of index m if any intermittent execution of the first configuration after m observed executions is indistinguishable from a continuous execution of the second configuration. In the base case where the index is $m = 0$, no execution is observed, so any two configurations are in the term relation. In the inductive case where the index is $m + 1$, the term relation relates two configurations at type $\mathtt{C}_{\mathtt{unit}}$ if the first configuration eventually steps to a value configuration and the second configuration can take zero or more steps such that the pair continues to be in the value relation of $\mathcal{V}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{m+1}$.

Value Relation. The value relation is defined based on the intended meaning of the type, and relates two value configurations that will have the same effect on the stores.

The first two rules omit the index m because the types $\downarrow\uparrow\mathtt{unit}$ and $\uparrow\mathtt{unit}$ are not recursive and thus no index is needed. The value relation at type $\uparrow\mathtt{unit}$ relates two configurations that have finalized their executions and thus requires they have the same nonvolatile memories NV. The value relation at type $\downarrow\uparrow\mathtt{unit}$ relates two configurations that have completed their executions and right before they commit their changes to nonvolatile memory and requires that the commands in both configurations be skip and that after committing changes to their nonvolatile memory, the configurations be related at type $\uparrow\mathtt{unit}$. In other words, this requires these two configurations to have the same effect on nonvolatile memory.

The value relations in the last four rows of Figure 17 are defined based on the type of the *first configuration*, as it goes through power failure, charging, and recovery, characteristic of intermittent execution. However, the second configurations in these relations do not encounter power failures at all and continue to be of type $\mathtt{C}_{\mathtt{unit}}$. Only in the relations defined in the first and second rows of the Figure 17 term relation do the types of both configurations match the indexed type of the relation. Hence, the value relation has varying arity: In the first and second rows of Figure 17, the relation is *binary* while in the rest, the relation degenerates to *unary*, with the second configuration as its Kripke world [27].

Each definition relates a configuration with a crash instruction to a continuously powered execution. It includes conditions on the store that quantify the effects of the partial execution, crash, or recovery, so that these three rules together require that previous partial execution, crashes, and recovery have no impact to the final nonvolatile memory compared to the continuously powered execution and at the same time, allow unharmful effects of the partial execution. Because the structure of the crash type enforces that power failure, charging, and recovery happens in sequence, each definition applies effects of its own crash instruction and then hands off the resulting configuration to the previous definition.

The value relation at type $\uparrow \mathtt{C}_{\mathtt{unit}}$ requires that the intermittent configuration runs the crash instruction $\uparrow\kappa$, which restores a continuation of the execution. The restored configuration continues to be related to the continuously powered configuration in the term interpretation at its original type $\mathtt{C}_{\mathtt{unit}}$. The constraint $\mathsf{NV}_0 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}_2 \setminus \{\mathtt{MFstWt}\}$ requires that the intermittent memory is the same as the continuously powered one, modulo must-first-writes. This constraint is necessary because after a partial execution of an atomic region, memory may be updated in its written variables. This means that the memory of the intermittently powered configuration may differ from the continuously powered configuration in those locations with the qualifier `Wtn`. The restore policy for atomic regions resets the qualifier `Wtn` of these locations to `MFstWt`. Thus, at this point, the value of locations with qualifier `MFstWt` may differ between the intermittently powered and continuously powered configurations. Such differences, however, do not violate the idempotency of the intermittent execution; in the next execution, these variables will not be read from before being rewritten, and if the next execution succeeds, these variables will be rewritten.

The value relation at type $(\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}})$ requires the intermittent configuration to run an instruction for charging the energy. It is defined similarly to a function type in a value relation and requires the configurations to be related at type $(\uparrow \mathtt{C}_{\mathtt{unit}})$ for every energy input level n provided to the first configuration.

The value relation at type $\downarrow(\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}})$ relates two configurations if the first one runs the crash instruction $\downarrow\varepsilon \mathbin{\#} \mathsf{in}(n > 0, \uparrow\kappa)$ which processes the power failure. The power failure policy creates a checkpoint of volatile locations such that the configurations continue to be in the value relation at type $(\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}})$.

Finally, the value relation relates two open command configurations at type $\mathtt{C}_{\mathtt{unit}}$ and index $m + 1$ if either (a) the first configuration has faced a power failure, and the two configurations continue to relate by $\mathcal{V}[\![\downarrow(\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}})]\!]^m$, or (b) the first configuration executed successfully without any power failures, and the two configurations are related by $\mathcal{V}[\![\downarrow\uparrow \mathtt{unit}]\!]$. This definition matches the disjunctive nature of type $\mathtt{C}_{\mathtt{unit}}$, which is recursively defined in the signature as $\downarrow(\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}}) \vee \downarrow\uparrow \mathtt{unit}$. Since we unfold the recursive definition of $\mathtt{C}_{\mathtt{unit}}$, we decrease the index from $m + 1$ to m to ensure the relation's well-foundedness. Note that the value relation is neither defined nor used by other definitions for $\mathtt{C}_{\mathtt{unit}}$ at index 0.

Semantic Typing. The top-level logical relation is written $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega \mid \Sigma \Vdash c_1 \leq c_2 : \mathtt{C}_{\mathtt{unit}}$ stating that each intermittent execution of c_1 yields the same nonvolatile memory as a continuously powered execution of c_2, if it begins execution with well-formed memories w.r.t. Ω and Σ (see Figure 17).

We formalize *semantic typing* as every atomic region of the program being logically related to itself. The rule P-Ckpt-semantic says that a program is semantically well-typed if every atomic region of it is self-related under our logical relation.

$$\frac{\begin{array}{c}\Omega_0 \mid \Sigma_0 = \mathsf{InitWorld}_t(\Omega; \rho; \mu; \omega) \\ \mathsf{aID}(c_0) \mid b \geq 0 : \mathtt{nat} \mid \Omega_0 \mid \Sigma_0 \Vdash c_0 \leq c_0 : \mathtt{C}_{\mathtt{unit}} \quad b : \mathtt{nat} \mid \Omega \Vdash p : \uparrow \mathtt{C}_{\mathtt{unit}}\end{array}}{b : \mathtt{nat} \mid \Omega \Vdash \mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c_0); p : \uparrow \mathtt{C}_{\mathtt{unit}}} \text{ (P-Ckpt-semantic)}$$

Self-relatedness of a block helps us to build a continuously powered execution for each intermittent execution of it as required for idempotency. We give the precise theorem and proof that a program comprised of only self-related blocks is idempotent in Section 8.

The P-Ckpt-semantic rule allows us to reason about programs by considering each code block independently. We can extend this approach to accommodate other code blocks in a program. In the next section, we extend the system to accommodate JIT blocks and explain how to compose them with atomic regions in programs.

7 JIT Region Execution

Up until this point, we have focused our discussion entirely on atomic regions. In this section, we extend the system we have developed thus far to also include JIT regions, written $c; p$, where c is the JIT region and p is the rest of the program. We also add the jit mode to the execution mode, which is used to index our operational semantics and typing rules. Below, we give the full syntax of programs and execution modes extended to accommodate JIT programs:

$$\begin{array}{llll} \textit{progs} & p & ::= & \mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c); p \mid c; p \mid \mathsf{skip} \\ \textit{exec. mode} & \mathtt{Md} & ::= & \mathsf{aID}(c) \mid \mathsf{jit} \end{array}$$

In Section 7.1, we explain the example JIT region from Figure 1. In Section 7.2, we extend the calculus from above with operational semantic rules for executing JIT regions. Sections 7.3 and 7.4 introduce types and typing rules specific to JIT mode. In Section 7.5, we revisit the semantic typing and logical relation considering the JIT mode.

7.1 Example

Unlike atomic regions, JIT regions are not susceptible to WAR bugs because they do not re-execute. Instead, all volatile state is checkpointed in nonvolatile memory just before a power failure so that upon reboot, program execution resumes from the line of failure. Due to this, variable annotations are not needed in the program syntax of JIT regions, for all writeable variables are checkpointed in JIT mode. As a result, all variables in a JIT block are annotated with the qualifier `CK`. Figure 19 shows the details of executing the JIT region from the second half of Figure 1 (repeated in the left column).

Row (0) shows the initial nonvolatile locations, their values, and the mapping from variables to locations. The system starts executing the JIT region by creating an empty context to be populated by volatile locations (Row (1)). The `let` construct in Line 8 allocates a fresh location ℓ_5 in volatile memory and updates the mapping to associate variable w to ℓ_5. On a power failure in JIT mode, the system creates a nonvolatile copy of the volatile location ℓ_5 just before it loses the location (Row (3)). It marks the nonvolatile copy with the superscript `ck`. When resuming program execution, the system restores these copies to volatile memory which it accesses during execution instead of their nonvolatile backups (Row (4)). Execution then continues with the `if` clause on Lines 9–12, finally dropping the volatile location ℓ_5, as it is out of scope (Row (5)).

Shifts in JIT Mode (Figure 19): Below, we recall the typing rule skeletons from Section 3 and use them to build the typing contexts in Figure 19.

$$\frac{\Omega; \cdot \vdash \_ : \tau^u}{\Omega \vdash \_ : \uparrow^s_u \tau^u}\ \uparrow R \qquad \frac{\Omega, \_ : \uparrow^s_u A^u; \Sigma, \_ : \downarrow^s_u \uparrow^s_u A^u \vdash \_ : \tau^u}{\Omega, \_ : \uparrow^s_u A^u; \Sigma \vdash \_ : \tau^u}\ \uparrow L^*$$

$$\frac{\Omega \vdash \_ : \tau^s}{\Omega; \Sigma \vdash \_ : \downarrow^s_u \tau^s}\ \downarrow R \qquad \frac{\Omega, \_ : \uparrow^s_u A^u; \Sigma \vdash \_ : \tau^u}{\Omega; \Sigma, \_ : \downarrow^s_u \uparrow^s_u A^u \vdash \_ : \tau^u}\ \downarrow L$$

The process of constructing typing contexts for JIT regions is the same as for atomic regions. However, all variables in JIT regions are checkpointed, and thus, the resulting typing contexts

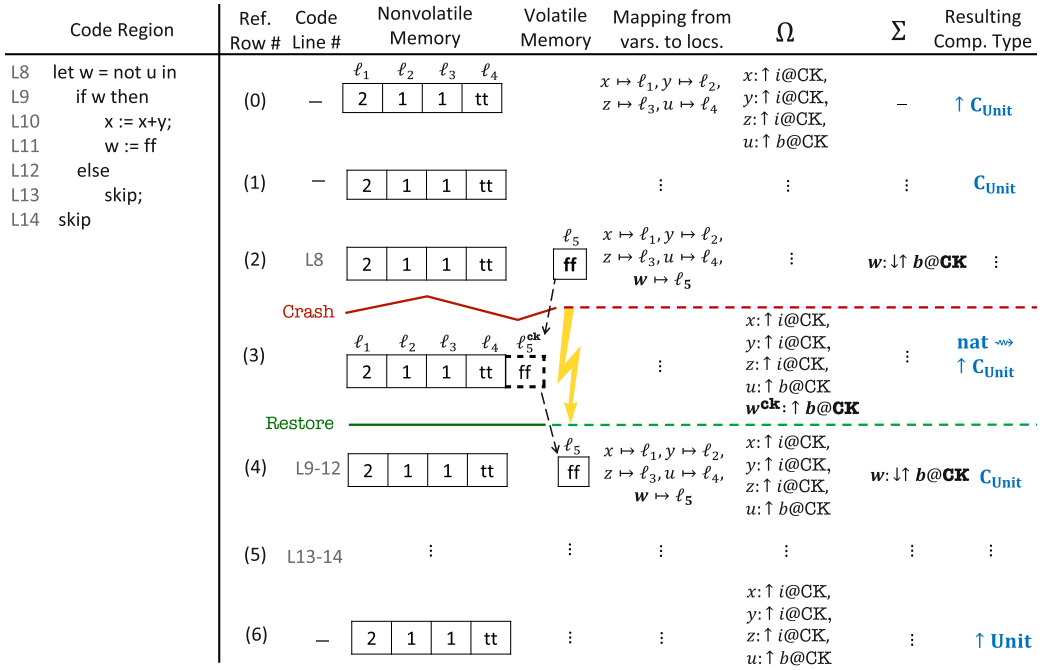

Fig. 19. Intermittent execution of a JIT region. We write i for `int` and b for `bool`.

differ from those for atomic regions even when typing the same commands. In Figure 19, we create an empty volatile context Σ when starting the JIT region (the step from Row (0) to Row (1)) by an application of the $\uparrow R$ rule. A combination of the rules $\downarrow L$ and $\downarrow R$ corresponds to a power failure, i.e., the stepping from Row (2) to Row (3) in Figure 19. An application of the $\downarrow L$ rule copies the variable w of type $\downarrow^s_u \uparrow^s_u b$ in Figure 19 from volatile memory context Σ into nonvolatile memory context Ω. A $\downarrow R$ rule closes off the (empty) nonvolatile memory context. As in atomic mode, a combination of $\uparrow R$ and $\uparrow L^*$ rules corresponds to creating a volatile log from a nonvolatile location when restarting the command after the failure, i.e., the step from Row (3) to Row (4). The $\uparrow R$ rule prepares for re-execution by setting up an empty volatile context into which the $\uparrow L^*$ rule copies variable w from the nonvolatile context. We assume an extra weakening rule to eliminate the remaining variable w in nonvolatile memory context. The dropping of volatile memory context at the end of execution (Row (5)) is not a modal step, but rather follows from a standard rule for the let clause. Lastly, an application of $\downarrow R$ drops the volatile typing context and produces the stable type $\uparrow^s_u \texttt{unit}$, detecting a successful execution.

7.2 Operational Semantics

The operational semantic rules for evaluating JIT regions are of the same forms as those for atomic regions: $K_c \Rightarrow_p K'_c$ (program level), $K_c \Rightarrow K'_c$ (command level), and $K_o \longrightarrow K'_o$ (expression level).

Top-Level Program Execution. The top-level semantic rule for stepping a JIT program is shown in Figure 20.

The D-P-SEQ rule applies when the next code block is a regular command c. The closed configuration of c with an empty initial set of volatile locations is fully evaluated in JIT mode. In Figure 19, we use the rule D-P-SEQ to set up an initially empty volatile context to begin executing the JIT

$$\frac{n > 0 \quad n' > 0 \quad [\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{jit} \mid n \mid \mathsf{NV} \mid \cdot \mid c \Rightarrow^* [\chi' \rhd \varepsilon] \otimes \gamma' \mid \mathsf{jit} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid \mathsf{skip}}{[\chi \rhd \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid c; p \Rightarrow_p [\chi' \rhd \varepsilon] \otimes \gamma \mid n' \mid \mathsf{NV}' \mid p} \text{ (D-P-SEQ)}$$

Fig. 20. Operational semantic rule for JIT regions.

$$\frac{}{[\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{jit} \mid \cdot \mid \mathsf{NV} \mid \mathsf{V} \mid \downarrow\varepsilon \,\#\, \mathsf{in}(b > 0; \uparrow\kappa) \quad \Rightarrow \quad [\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{jit} \mid \mathsf{NV}, \mathsf{V}_{\mathsf{ck}} \mid \varepsilon \,\#\, \mathsf{in}(b > 0; \uparrow\kappa)} \text{ (D-S-JIT)}$$

$$\frac{\mathsf{NV} = \mathsf{NV}', \mathsf{NV}''_{\mathsf{ck}}}{[\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{jit} \mid n \mid \mathsf{NV} \mid \uparrow \kappa \;\Rightarrow\; [\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{jit} \mid n \mid \mathsf{NV}' \mid \mathsf{NV}'' \mid \kappa} \text{ (D-RESTORE-JIT)}$$

Fig. 21. Operational semantics for crash instructions under a closed configuration (JIT mode).

type variables	$\mathtt{C}_T^{\mathtt{Md}}$	::=	$\mathtt{C}_{\mathtt{unit}} \mid \mathtt{C}_A^{\mathtt{atom}} \mid \mathtt{C}_A^{\mathtt{jit}}$
command crash type	$\mathtt{C}_{\mathtt{unit}}$	::=	$\downarrow(\mathsf{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}}) \vee \downarrow\uparrow \mathtt{unit}$
atomic crash type	$\mathtt{C}_A^{\mathtt{atom}}$	::=	$\downarrow(\mathsf{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}}) \vee \downarrow\uparrow A$
jit crash type	$\mathtt{C}_A^{\mathtt{jit}}$	::=	$\downarrow(\mathsf{nat} \rightsquigarrow \uparrow \mathtt{C}_A^{\mathtt{jit}}) \vee \downarrow\uparrow A$

Fig. 22. Crash types for both JIT and atomic mode.

region (the step from Row (0) to Row (1)), and to step the JIT region (from Row (1) to Row (5)) according to the third premise. Then the resulting volatile locations V' scoped in c are dropped because in JIT mode the nonvolatile memory always stores up-to-date values and V' only contains out-of-scope let variables. This corresponds to the step from Row (5) to Row (6) in Figure 19. The constraint $n > 0$ checks that there is energy available for the first code block to take a step, and $n' > 0$ checks that the first code block finishes executing successfully and is not in the midst of a power failure.

Having both rules D-P-Seq and D-P-Ckpt allows running programs with both JIT and atomic regions, and the combination of rules T-P-Seq and T-P-Ckpt allows us to statically check them.

Command Execution (Closed Config). We give the rules for a closed configuration in JIT mode in Figure 21. The rule D-S-Jit corresponds to checkpointing, and stores all volatile memory in nonvolatile locations. The rule D-restore-Jit moves the checkpointed locations into volatile memory, thereby dropping the checkpointed locations from the scope of the nonvolatile memory. D-restore-Jit recovers the interrupted command κ for the program to resume execution.

7.3 Static Typing

We extend the type system to accommodate JIT regions by introducing crash types and typing rules for JIT mode, summarized in Figure 22.

Crash Types. We begin by extending the definition of type variables $\mathtt{C}_T^{\mathtt{Md}}$. Because the mode has been extended to accommodate both atomic regions and JIT regions, we replace the single type variable $\mathtt{C}_A^{\mathtt{Md}}$ with a type variable for each mode: $\mathtt{C}_A^{\mathtt{atom}}$ and $\mathtt{C}_A^{\mathtt{jit}}$. The definition of crash type for commands in JIT mode ($\mathtt{C}_{\mathtt{unit}}$) is the same as for atomic mode, where the right disjunct represents successful execution and the left disjunct represents a power failure. To capture the JIT policy, the type $\mathtt{C}_{\mathtt{unit}}$ in the left disjunct represents the same command that was interrupted by the power failure. The definition of expression crash type for JIT mode ($\mathtt{C}_A^{\mathtt{jit}}$) is similar to the one for atomic mode ($\mathtt{C}_A^{\mathtt{atom}}$), where the right disjunct types a successful execution and the left disjunct represents power failure. However, the type $\uparrow\mathtt{C}_A^{\mathtt{jit}}$ in the left disjunct captures the JIT recovery policy, that an interrupted run of an expression in JIT mode will be restored to the expression itself.

Program Typing. To type JIT programs, we extend our type system with the rule T-P-SEQ. The T-P-SEQ rule types program $c; p$ by first typing c under JIT mode with $b \geq 0$, which allows for the possibility of power failure. The command c is typed under an empty volatile memory typing context, which will be populated when let commands in c allocate new volatile locations. The signature $\texttt{Sig}$ for typing c is also empty and is populated later at individual points of failure (rule T-ENOUGH?). Note that the rule T-P-SEQ does not need additional constraints on the post-context Ω' because the access qualifiers never change when running a command c in JIT mode. The second premise types the rest of the program p under the initial nonvolatile typing context Ω.

$$\frac{\texttt{jit} \mid b \geq 0 : \texttt{nat} \mid \Omega; \cdot \vdash_{\emptyset} c : \mathsf{C}_{\texttt{unit}} \dashv \Omega' \quad b : \texttt{nat} \mid \Omega \vdash p : \uparrow \mathsf{C}_{\texttt{unit}}}{b : \texttt{nat} \mid \Omega \vdash c; p : \uparrow \mathsf{C}_{\texttt{unit}}} \text{ (T-P-SEQ)}$$

Command and Expression Typing. We extend the command and expression typing rules to accommodate JIT mode which just entails updating our T-ENOUGH? rules for commands and expressions (shown in Figure 23).

We update the T-ENOUGH? rule for commands to populate the appropriate signature $\texttt{Sig}''$. The second premise states that the signature remains the same if the mode is atomic, but is populated by $\texttt{Sig}'$ if the mode is JIT. In JIT mode, after a power failure, the command c is restored to itself, and $\texttt{Sig}'$ remembers that the well-typedness of the command (when the energy level is non-negative) has already been checked. The T-ENOUGH? rule for expressions is similar, but is used to remember the well-typedness of an expression in case of a power failure.

Statement Typing. The typing rules for crash instructions specific to JIT mode are shown in Figure 24. The T-JIT-STOP rule brings a checkpointed version of all the volatile variables in Σ inside Ω since they are checkpointed upon power failure. Once an energy input is received from the environment ($b > 0$), the rule T-JIT-RESTORE applies to type the restored command. Here, volatile memory is restored from the checkpointed locations in Ω. Checkpoints are dropped from Ω and execution resumes with the expression or command κ, which is the code running right before the power failure.

7.4 Store Typing

We present the store typing rules for JIT mode in Figure 25: EMPTY for checking empty stores, V-LOC-JIT for checking volatile locations, and NV-LOC-JIT for checking nonvolatile locations. The rule V-LOC-JIT is similar to the volatile store typing rule for atomic mode. The rule checks that the volatile location ℓ has the qualifier $\texttt{CK}$, and that the type of the value v stored in ℓ matches the type of the corresponding variable x in the volatile typing context Σ. The first premise recursively checks the rest of the volatile store. The rule NV-LOC-JIT checks a nonvolatile location ℓ by ensuring that the variable x, for which ℓ is allocated, has the same type in Ω as the value v stored in ℓ and that the qualifier $q = \texttt{CK}$. The rule then recursively checks the rest of the store. The generalized store typing rules for both JIT and atomic modes are presented in Appendix A.

7.5 Logical Relation

Our logical relation from Section 6.1 is designed to reason about JIT regions too, but with different policies. We provide policy definitions for JIT mode in Figure 26. The policies are defined to match the dynamic rules for crash, restore, and re-execution for JIT execution (D-S-JIT, D-RESTORE-JIT, and D-P-SEQ).

The commit policy for JIT mode simply drops all volatile memory and changes the qualifiers of all locations in nonvolatile memory to $\texttt{ck}$. In particular, $\texttt{Commit}(\gamma, \texttt{jit}, \mathsf{NV}_1, \mathsf{V}_1) = \gamma' \mid \mathsf{NV}^1_{\texttt{ck}}, \mathsf{NV}^2_{\texttt{ck}},$

$$\frac{\mathtt{Sig}' = \{\mathtt{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash c : \tau\} \quad \mathtt{Sig}'' = \mathit{if}\ \mathtt{Md} = \mathsf{jit}, \mathit{then}\ \mathtt{Sig}', \mathit{else}\ \mathtt{Sig} \quad \mathtt{Md} \mid b = 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}''} c : \tau \quad \mathtt{Md} \mid b > 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau \dashv \Omega'}{\mathtt{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau \dashv \Omega'}\ \text{(T-ENOUGH?)}$$

$$\frac{\mathtt{Sig}' = \{\mathtt{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD}} e : \tau\} \quad \mathtt{Sig}'' = \mathit{if}\ \mathtt{Md} = \mathsf{jit}, \mathit{then}\ \mathtt{Sig}', \mathit{else}\ \mathtt{Sig} \quad \mathtt{Md} \mid b = 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}''} e : \tau \quad \mathtt{Md} \mid b > 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD};\mathtt{Sig}} e : \tau}{\mathtt{Md} \mid b \geq 0 : \mathsf{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD};\mathtt{Sig}} e : \tau}\ \text{(T-ENOUGH?)}$$

Fig. 23. T-ENOUGH? rules for commands and expressions, extended to JIT mode.

$$\frac{\Sigma = \downarrow\uparrow\Sigma' \quad \mathsf{jit} \mid \cdot \mid \Omega, \uparrow\Sigma'_{\mathtt{ck}} \vdash_{\mathtt{Sig}} \varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa') : (\mathsf{nat} \rightsquigarrow \uparrow \mathtt{C}_T^{\mathtt{Md}})}{\mathsf{jit} \mid \cdot \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \downarrow\varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa') : \downarrow(\mathsf{nat} \rightsquigarrow \uparrow \mathtt{C}_T^{\mathtt{Md}})}\ \text{(T-JIT-STOP)}$$

$$\frac{\Omega = \Omega', \Omega''_{\mathtt{ck}} \quad \mathsf{jit} \mid b \geq 0 : \mathsf{nat} \mid \Omega'; \downarrow\Omega'' \vdash \kappa' : \mathtt{C}_T^{\mathtt{Md}} \in \mathtt{Sig}}{\mathsf{jit} \mid b > 0 : \mathsf{nat} \mid \Omega \vdash_{\mathtt{Sig}} \uparrow\kappa' :\uparrow \mathtt{C}_T^{\mathtt{Md}}}\ \text{(T-JIT-RESTORE)}$$

Fig. 24. Restore and checkpoint typing in JIT mode.

$\boxed{\vdash^{\mathsf{Jit}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma}$

$$\frac{}{\vdash^{\mathsf{Jit}}_{\gamma} \cdot \mid \cdot : \cdot \mid \cdot}\ \text{(EMPTY)}$$

$$\frac{\vdash^{\mathsf{Jit}}_{\gamma'} \mathsf{NV} \mid \mathsf{V}' : \Omega \mid \Sigma \quad \mathsf{V} = \mathsf{V}', \ell\,@\,q \hookrightarrow v \quad q = \mathtt{Ck} \quad \gamma = \gamma', [x \mapsto \ell] \quad \mathtt{Md} \mid b : \mathsf{nat} \mid \Omega \vdash_{\mathtt{RD}} v : \uparrow A}{\vdash^{\mathsf{Jit}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma, (x{:}\downarrow\uparrow A@q)}\ \text{(V-LOC-JIT)}$$

$$\frac{\vdash^{\mathsf{Jit}}_{\gamma'} \mathsf{NV}' \mid \mathsf{V} : \Omega \mid \Sigma \quad \mathsf{NV} = \mathsf{NV}', \ell\,@\,q \hookrightarrow v \quad q = \mathtt{Ck} \quad \gamma = \gamma', [x \mapsto \ell] \quad \mathtt{Md} \mid b : \mathsf{nat} \mid \Omega \vdash_{\mathtt{RD}} v : \uparrow A}{\vdash^{\mathsf{Jit}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega, (x{:}\uparrow A@q) \mid \Sigma}\ \text{(NV-LOC-JIT)}$$

Fig. 25. Well-formedness of $\mathsf{NV} \mid \mathsf{V}$ w.r.t. $\Omega \mid \Sigma$ in JIT mode.

$$\frac{\mathsf{NV}_1 = \mathsf{NV}^1_{\mathtt{RD}}, \mathsf{NV}^2_{\mathtt{ck}} \quad \mathsf{NV}_2 = \mathsf{NV}^1_{\mathtt{ck}}, \mathsf{NV}^2_{\mathtt{ck}} \quad \mathsf{range}(\gamma') = \mathsf{dom}(\mathsf{NV}_1) \quad \gamma' \subseteq \gamma}{\mathtt{Commit}(\gamma, \mathsf{jit}, \mathsf{NV}_1, \mathsf{V}_1) = \gamma' \mid \mathsf{NV}_2}$$

$$\frac{\mathsf{NV}_1 = \mathsf{NV}', \mathsf{NV}''_{\mathtt{ck}} \quad \mathsf{NV}_2 = \mathsf{NV}' \quad \mathsf{V}_2 = \mathsf{NV}''}{\mathtt{Restore}(\gamma, \mathsf{jit}, \mathsf{NV}_1, \kappa) = \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c} \qquad \frac{\mathsf{V}_2 = \mathsf{V}_1 \quad \gamma' = \gamma}{\mathtt{PwOff}(\gamma, \mathsf{jit}, \mathsf{NV}_1, \mathsf{V}_1) = \gamma' \mid \mathsf{V}_2}$$

Fig. 26. `Commit`, `Restore`, and `PwOff` policy definitions for JIT mode.

and $\mathsf{NV}_1 = \mathsf{NV}^1_{\mathtt{RD}}, \mathsf{NV}^2_{\mathtt{ck}}$. Additionally, the original map γ covers the committed map γ' ($\gamma' \subseteq \gamma$), and the locations mapped to by γ' comprise the locations of NV_1 ($\mathit{range}(\gamma') = \mathit{dom}(\mathsf{NV}_1)$).

The restore policy for JIT mode is defined as $\mathtt{Restore}(\gamma, \mathsf{jit}, \mathsf{NV}_1, \kappa) = \mathsf{NV}' \mid \mathsf{NV}'' \mid \kappa$ where $\mathsf{NV}_1 = \mathsf{NV}', \mathsf{NV}''_{\mathtt{ck}}$. We write $\mathsf{NV}_1 = \mathsf{NV}', \mathsf{NV}''_{\mathtt{ck}}$ to state that NV_1 can be uniquely partitioned into all locations that are checkpointed ($\mathsf{NV}''_{\mathtt{ck}}$), which are of the form $\ell_{\mathtt{ck}}$, and regular locations (NV') of the form ℓ. NV'' is the non-checkpointed version of $\mathsf{NV}''_{\mathtt{ck}}$ which could be retrieved by removing the ck subscript from every location in $\mathsf{NV}''_{\mathtt{ck}}$.

Lastly, the power failure policy is defined to checkpoint all volatile locations in JIT mode, i.e., $\mathtt{PwOff}(\gamma, \mathsf{jit}, \mathsf{NV}_1, \mathsf{V}_1) = \gamma \mid \mathsf{V}_1$.

Semantic Typing. Lastly, we extend our semantic typing definition to accommodate JIT regions too. We do so with the rule P-SEQ-SEMANTIC which says that a program of the form $c; p$ is semantically well-typed if the JIT block c is self-related under the logical relation and if the remaining program p is semantically well-typed.

$$\frac{\mathtt{jit} \mid b \geq 0 : \mathtt{nat} \mid \Omega \mid \cdot \Vdash c \leq c : \mathtt{C}_{\mathtt{unit}} \quad b : \mathtt{nat} \mid \Omega \Vdash p : \uparrow \mathtt{C}_{\mathtt{unit}}}{b : \mathtt{nat} \mid \Omega \Vdash c; p : \uparrow \mathtt{C}_{\mathtt{unit}}} \text{ (P-SEQ-SEMANTIC)}$$

For programs composed of a combination of JIT and atomic regions, the semantic typing rules P-SEQ-SEMANTIC and P-CKPT-SEMANTIC indicate that a program is semantically well-typed if every program block (*JIT and atomic region*) of it is self-related under our logical relation.

8 Metatheory

This section establishes the main properties of our system: progress and preservation, adequacy—which states that intermittent executions of semantically well-typed programs are correct—and the fundamental theorem of logical relation which states that statically well-typed programs are semantically well-typed. Combining the latter two, it follows that statically well-typed programs can be correctly executed intermittently. The detailed proofs of these theorems are provided in Appendices B–E.

8.1 Statically Well-Typed Programs Are Type Safe

We prove type safety according to progress and preservation. The progress and preservation theorems assume that memory locations are well-formed, $\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$. The progress theorem (formally defined below) states that if a command c is well-typed, then for all energy levels n and well-formed memories NV and V, either the configuration of c is a value configuration or it can take a step according to the operational semantics.

Theorem 8.1 (Progress for Commands). *If* $\mathtt{Md} \mid b \mathrel{\mathcal{R}} m : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau \dashv \Omega'$, *then* $\forall\, n : \mathtt{nat}$ *with* $n \mathrel{\mathcal{R}} m$ *and* $\forall\, \gamma, \mathsf{NV}, \mathsf{V}$ *with* $\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$, *either*

— $Val(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c)$ *or*
— $\exists(\gamma' \mid \mathtt{Md}' \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c')$ *s.t.* $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c \;\rightarrow\; \gamma' \mid \mathtt{Md}' \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'$.

Here, $m = 0$ and the configuration's concrete energy level n instantiates energy variable b in the typing judgment. The energy level decrements with each command step until the configuration of c is a value where $n > 0$ (indicating a successful execution), or until $n = 0$ (indicating a crash) which our semantics also considers a value. The constraint $n \mathrel{\mathcal{R}} m$ in the theorem ensures that the concrete energy level n preserves the relation of b with m as required by typing. Note that this progress theorem does not ensure that a given program will always make enough progress to complete the execution, because sufficient energy may not be available.

The preservation theorem for commands is formally defined below.

Theorem 8.2 (Preservation for Commands). *If* $\mathtt{Md} \mid b \;\geq\; 0 : nat \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau \dashv \Omega'$ *where*

(1) $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c$ *is well-formed,*
(2) $\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$,
(3) natural number $n \geq 0$, *and*
(4) $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c \;\rightarrow\; \gamma' \mid \mathtt{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'$

then for some Ω_0, Σ', $\mathtt{Md} \mid b \geq 0 : nat \mid \Omega_0; \Sigma' \vdash_{\mathtt{Sig}} c' : \tau \dashv \Omega'$ *where*

(5) $\gamma' \mid \mathtt{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'$ *is well-formed,*
(6) $\vdash^{\mathtt{Md}}_{\gamma'} \mathsf{NV}' \mid \mathsf{V}' : \Omega_0 \mid \Sigma'$,
(7) natural number $n' \geq 0$,
(8) if $\mathtt{Md} = \mathsf{aID}(c_0)$, *then* $\mathsf{NV}' \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\} = \mathsf{NV} \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\}$,
(9) $dom(\mathsf{NV}) = dom(\mathsf{NV}')$, *and*
(10) $\Omega > \Omega_0$.

Theorem 8.2 states that if c is a syntactically well-typed command, whose configuration $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c$ is well-formed (1) for well-formed memories NV and V (2) and non-negative natural number n (3), steps to another configuration $\gamma' \mid \mathtt{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'$ (4), then c' is also syntactically well-typed. Additionally, the stepped configuration is well-formed (5), the memories NV' and V' are well-formed (6), and n' is a nonnegative natural number (7). Since an arbitrary command step could write to memory, we also prove that the initial and resulting memories are the same excluding their must-first-write and written memory locations for the mode $\mathsf{aID}(c_0)$ (8). As another key invariant, we prove that no new nonvolatile memory locations can be introduced with a given step, meaning that they can only be updated (9). Theorem 8.2 obligation (10) enforces that the qualifiers of the variables in contexts Ω_0 and Ω are within one δ transition of each other and is defined below in Definition 8.3.

Definition 8.3. $\Omega > \Omega'$ iff $\forall x{:}\tau@q \in$, we have $x{:}\tau@q' \in \Omega'$ where $q' \neq UN$ and either $q = q'$ or $\delta(q, Wt) = q'$.

8.2 Statically Well-Typed Programs Are Semantically Well-Typed

We prove the soundness of our static typing rules with regard to the semantic typing rules (Theorem 8.4). We sketch the proof below and provide the full proof in Appendix C.

Theorem 8.4 (Fundamental Theorem). *If* $b : \mathtt{nat} \mid \Omega \vdash p : \uparrow \mathsf{C}_{\mathtt{unit}}$, *then* $b : \mathtt{nat} \mid \Omega \Vdash p : \uparrow \mathsf{C}_{\mathtt{unit}}$.

The proof of Theorem 8.4 is by induction on the static typing derivation for p and considers the last step in the derivation. Here, we sketch the proof for the case where D-P-Ckpt is the last step of the derivation. For the complete proof of Theorem 8.4, see the appendix. By inversion on the D-P-Ckpt rule, we get $p = \mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c); p'$. Also, c is well-typed for static contexts Ω and Σ, where $\Omega = \Omega'', \Sigma_{\mathtt{ck}}$.

The goal is to establish that c is related to itself in the term interpretation for arbitrary n, m, γ, NV, and V where $\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega'', \Sigma_{\mathtt{ck}} \mid \Sigma$. This means that we need to prove $(\gamma \mid \mathsf{aID}(c) \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c, \gamma \mid \mathsf{aID}(c) \mid \infty \mid \mathsf{NV} \mid \mathsf{V} \mid c) \in \mathcal{E}[\![\mathsf{C}_{\mathtt{unit}}]\!]^m$ for all indices m given that the condition $\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$ holds. The last condition enforces that the static contexts match the dynamic context. In particular, $\mathsf{NV} = \mathsf{NV}', \mathsf{V}_{\mathtt{ck}}$ and $range(\gamma) = dom(\mathsf{NV})$.

We prove a generalized version of this goal in the main proof of Theorem 8.4. In particular, we prove that a well-typed command c is related to itself for any two nonvolatile memories NV_1 and NV_2 that agree on every value of their locations except those with must-first-write qualifiers. The goal is to show $(\gamma \mid \mathsf{aID}(c) \mid n \mid \mathsf{NV}_1 \mid \mathsf{V} \mid c, \gamma \mid \mathsf{aID}(c) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V} \mid c) \in \mathcal{E}[\![\mathsf{C}_{\mathtt{unit}}]\!]^m$ where (a) $range(\gamma) = dom(\mathsf{NV}_1)$, (b) $\mathsf{NV}_1 \setminus \mathtt{MFstWt} = \mathsf{NV}_2 \setminus \mathtt{MFstWt}$, and (c) $\mathsf{NV}_1 = \mathsf{NV}_1 \setminus \mathtt{Wtn}$. The condition (a) asserts that γ covers the locations in the nonvolatile memory NV_1. The condition (b) enforces that the nonvolatile memories NV_1 and NV_2 be related up to their $\mathtt{MFstWt}$ qualifiers. Lastly, the condition (c) asserts that the intermittent nonvolatile memory NV_1 does not have any $\mathtt{Wtn}$ variables at the beginning of executing an atomic region.

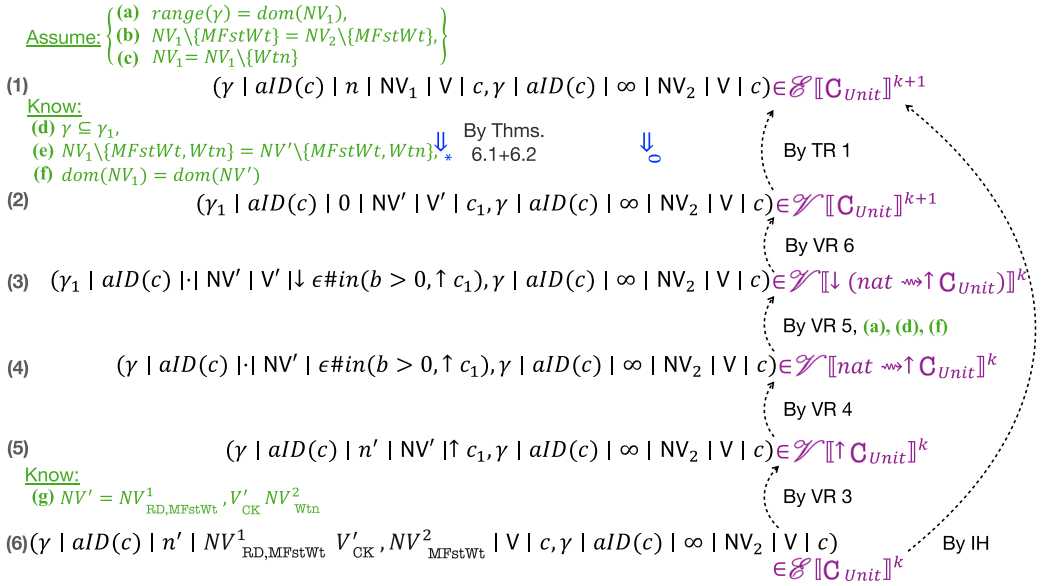

Fig. 27. Proof of the fundamental theorem for D-P-Ckpt (inductive case).

Proving this generalized result is crucial to building the simulation because even though the initial configuration of the continuously powered execution $\gamma \mid \mathsf{aID}(c) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V} \mid c$ matches the initial configuration of the intermittent execution, the intermediate configurations of the continuously powered execution are not necessarily the same as their intermittent counterparts, yet they are still related (for example, point (6) in Figure 27). In particular, the configurations are the same except for their nonvolatile memories which are equivalent up to their `MFstWt` locations, as enforced by condition (b) in Figure 27. We discuss how to establish this relation in the inductive case explanation below.

Command c being self-related is a special case of this generalized goal where the nonvolatile memories are the same ($\mathsf{NV}_1 = \mathsf{NV}_2 = \mathsf{NV}$) and are well-formed with respect to the context initialized by inversion on the D-P-Ckpt rule. Observe that the three conditions required by the inductive hypothesis hold in this special case: (a) $range(\gamma) = dom(\mathsf{NV})$ holds as described above, (b) $\mathsf{NV} \setminus \mathtt{MFstWt} = \mathsf{NV} \setminus \mathtt{MFstWt}$ holds vacuously, and (c) $\mathsf{NV}_1 = \mathsf{NV}_1 \setminus \mathtt{Wtn}$ is true due to the definitions of InitWorld and well-formedness.

We then proceed to prove the generalized result mentioned above by induction on the index m.

Base Case. For $m = 0$, the logical relation in Figure 17 relates any two configurations with the index zero; and thus, the base case immediately follows from the term interpretation at type $\mathtt{C}_{\mathtt{unit}}$.

Inductive Case. We now discuss the inductive case where $m = k + 1$ in Figure 27. The green conditions (d)–(g) are learned throughout the proof, and (a)–(c) are known by assumption. The places where these conditions are used in the proof are marked on the right hand side of the diagram. The dashed lines represent implications. For example, to prove point (1), it suffices to establish point (2) which follows by the term interpretation at type $\mathtt{C}_{\mathtt{unit}}$. To prove point (2), it is enough to prove point (3), and so on. The proof continues unfolding the logical relation in this manner until point (6) is reached. Point (6) states that c is self-related by the term interpretation at type $\mathtt{C}_{\mathtt{unit}}$ but at the smaller index k. By establishing the appropriate conditions, we can apply the

inductive hypothesis to establish the desired result. This is the high-level proof strategy, and we provide more detail in the explanation below.

Our goal is to prove $(\gamma \mid \mathsf{aID}(c) \mid n \mid \mathsf{NV}_1 \mid \mathsf{V} \mid c, \gamma \mid \mathsf{aID}(c) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V} \mid c) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{k+1}$ which corresponds to point (1) in Figure 27 with assumptions (a)–(c). By the progress and preservation theorems (Theorems 8.1 and 8.2), the first configuration can take multiple steps until it becomes a value $\gamma_1 \mid \mathsf{aID}(c) \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c_1$ that continues to be well-typed. If $n' > 0$, the second configuration steps similarly to completion and establishes that the two resulting configurations are in the value relation. This case is omitted from Figure 27. If $n' = 0$, the second configuration does not step and instead reaches point (2) in Figure 27. At point (2), the proof must show that the configurations are in the value interpretation at type $\mathtt{C}_{\mathtt{unit}}$.

The cascade of implications (dashed lines) follows the definition of the value relations at each type (marked in purple). At each step, we invert on the typing rule of the open configuration and show that runtime memories stay well-defined for static contexts.

At point (4), we apply the power failure policy for atomic regions, which drops the volatile memory V' and creates a mapping using the domain of NV'. By the prior conditions established, we know the created mapping is the original mapping γ.

At point (6), we apply the restore policy for atomic regions, which creates a new volatile memory based on NV' and marks all written memory locations in NV' as must-first-write. By the semantics of the restore function, we know the volatile memory created is the original volatile memory V containing only checkpointed locations.

The goal at point (6) is similar to our original goal at point (1) but at the smaller index k. We can apply the inductive hypothesis to establish the goal at point (6). To do so, it is enough to show that conditions (a)–(c) hold at this point. First, (a) holds by the original condition (a) combined with (f) and the definition of NV' given by (g). The condition (b) holds by combining the original condition (b) with (c), (e), and the observation that NV', sans the locations marked `MFstWt` and `Wtn`, is the same as the final intermittent nonvolatile memory, sans the locations marked `MFstWt`. (c) continues to be true for each subsequent execution due to the semantics of the restore function which replaces all `Wtn` qualifiers in nonvolatile memory with `MFstWt`.

8.3 Semantically Well-Typed Programs Are Idempotent

A program is idempotent if every intermittent execution of it can be simulated by a continuous execution. Definition 8.5 defines this property formally.

Definition 8.5 (Idempotency). A triple of a program p, nonvolatile memory NV, and a mapping γ is idempotent, if every intermittent execution of the program can be simulated by a continuous execution of it: For all $n, n', \chi_1, \chi_1', \mathsf{NV}', p'$, if $[\chi_1 \rhd \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid p \Rightarrow_p [\chi_1' \rhd \varepsilon] \otimes \gamma \mid n' \mid \mathsf{NV}' \mid p'$, then $[\chi_2 \rhd \varepsilon] \otimes \gamma \mid \infty \mid \mathsf{NV} \mid p \Rightarrow_p [\chi_2 \rhd \varepsilon] \otimes \gamma \mid \infty \mid \mathsf{NV}' \mid p'$.

When showing that a program p with memory NV and mapping γ is idempotent, we construct a simulation of the continuous execution assuming that it begins with the same program code p and memory NV as the intermittent execution. We use the logical relation from Section 6 to enforce that the stepped configurations are related in the same way; they share the same program code p' and memory NV'. The charge stream χ_2 remains unchanged between the initial and final configurations as the continuous execution does not experience power failures.

Theorem 8.6 states that semantically well-typed programs are idempotent. In the theorem, we use the store typing judgment for the jit mode to ensure the well-formedness of NV with respect to a typing context Ω. This reflects the fact that the programs start executing in jit mode by default.

Theorem 8.6 (Adequacy). *Consider* $b : \mathtt{nat} \mid \Omega \Vdash p : \mathtt{C}_{\mathtt{unit}}$, *a nonvolatile memory* NV, *and a map* γ *such that* $\vdash^{\mathsf{jit}}_{\gamma} \mathsf{NV} \mid \cdot : \Omega \mid \cdot$. *The triple of* p, NV, *and* γ *is idempotent.*

The proof builds the simulation required by idempotency using induction on the number of power failures during each step of a program's intermittent execution. The construction of this simulation (and proof) is illustrated below. See Appendix D for the complete proof of Theorem 8.6.

Executing an Atomic Region. Assume that c_1 is self-related and that the configuration $[\chi_1 \triangleright \varepsilon] \otimes \gamma_1 \mid \mathsf{aID} \mid n \mid \mathsf{NV}_1 \mid \mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c_1); p$ takes a step to $[\chi_k \triangleright \varepsilon] \otimes \gamma \mid \mathsf{aID} \mid n' \mid \mathsf{NV}' \mid p$ via the rule D-P-Ckpt while incurring possibly m-many power failures. We need to show that there exists a continuously powered execution of the same program that steps the configuration $[\chi \triangleright \varepsilon] \otimes \gamma_1 \mid \mathsf{aID} \mid \infty \mid \mathsf{NV}_1 \mid \mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c_1); p$ to $[\chi \triangleright \varepsilon] \otimes \gamma \mid \mathsf{aID} \mid \infty \mid \mathsf{NV}' \mid p$. Our proof constructs such continuously powered execution.

The proof assumes that program $\mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c_1); p$ is semantically well-typed. Applying P-Ckpt-semantic, we learn that the command c_1 is semantically well-typed and self-related at type $\mathtt{C}_{\mathtt{unit}}$. By the definition of logical relation (Figure 17), it follows that the intermittent configuration of c_1 with energy level n is related to the continuously powered configuration of c_1 with energy level ∞ at the type $\mathtt{C}_{\mathtt{unit}}$ for every index, including $m + 1$. We use this relation to build the desired simulation. Towards this goal, we prove a more general result: For any two related configurations, if the first configuration takes zero or more steps, then the second configuration takes zero or more steps such that the two stepped configurations remain related. By definition of our relation, both configurations result in the same final nonvolatile memories.

The proof is by induction on the number of power failures m. Figure 28 also shows the inductive case (Lines (1)–(6)) and the base case (Lines (7)–(9)) for the case of an atomic region. Starting with index $m + 1$ (point (1) in Figure 28), we build a continuously powered execution for c_1 while showing that the intermediate configurations continue to relate to their intermittent counterparts at a smaller index m (point (6) in Figure 28). At that point, we apply the inductive hypothesis to obtain the desired result. Finally, the base case establishes that the memories of the final configurations are the same.

Inductive Case. By definition of the term interpretation at type $\mathtt{C}_{\mathtt{unit}}$, c_1 in the first configuration runs via D-step until the next power failure. Moreover, it follows by the term interpretation at type $\mathtt{C}_{\mathtt{unit}}$ that the second configuration can also be executed such that the resulting configurations remain related (point (2)). The first configuration takes a step from point (2) to point (3) with the rule D-Crash. Since the energy level of the first configuration is 0, it follows by the value interpretation at type $\mathtt{C}_{\mathtt{unit}}$ that these two configurations are related by the value interpretation at type $\downarrow(\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C}_{\mathtt{unit}})$ (point (3)). Using the rule D-S-aID, the first configuration takes a step from point (3) to point (4). The volatile memory V'_1 is dropped, and hence $\mathsf{V}' = \emptyset$. Note that this matches the power off policy for atomic mode. By assumption to D-S-aID, the mapping γ''_1 drops the discarded volatile memory locations from γ'_1 such that $\gamma''_1 \subseteq \gamma'_1$. The two configurations remain related by the value interpretation at type $\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C}_{\mathtt{unit}}$ (point (4)). The first configuration then takes another step to point (5) by D-Charge, inputting the harvested energy n_0, thus updating the energy stream to χ'_1 where $\chi_1 = n_0 :: \chi'_1$. By definition of the value relation, the two configurations remain related at the type $\uparrow\mathtt{C}_{\mathtt{unit}}$ for all n_0 (point (5)).

From point (5) to point (6), the first configuration steps via D-restore-aID. In doing so, it replaces all $\mathtt{Wtn}$ qualifiers in the nonvolatile memory $\mathsf{NV}'_1, \mathsf{V}'$ with $\mathtt{MFstWt}$ in NV_0 and loads the checkpointed data of the nonvolatile memory into the volatile memory V_0. As such, the program counter resets to its original value, and the atomic region re-executes from the beginning such that $c_0 = c_1$. The variable mapping stays the same ($\gamma_0 = \gamma''_1$). Note that this matches the restore policy for atomic mode. The two configurations are then related by the term interpretation at the type $\mathtt{C}_{\mathtt{unit}}$. This is similar to what we had initially but with one fewer power failure remaining. At this point, we apply the inductive hypothesis to build the simulation at the index m.

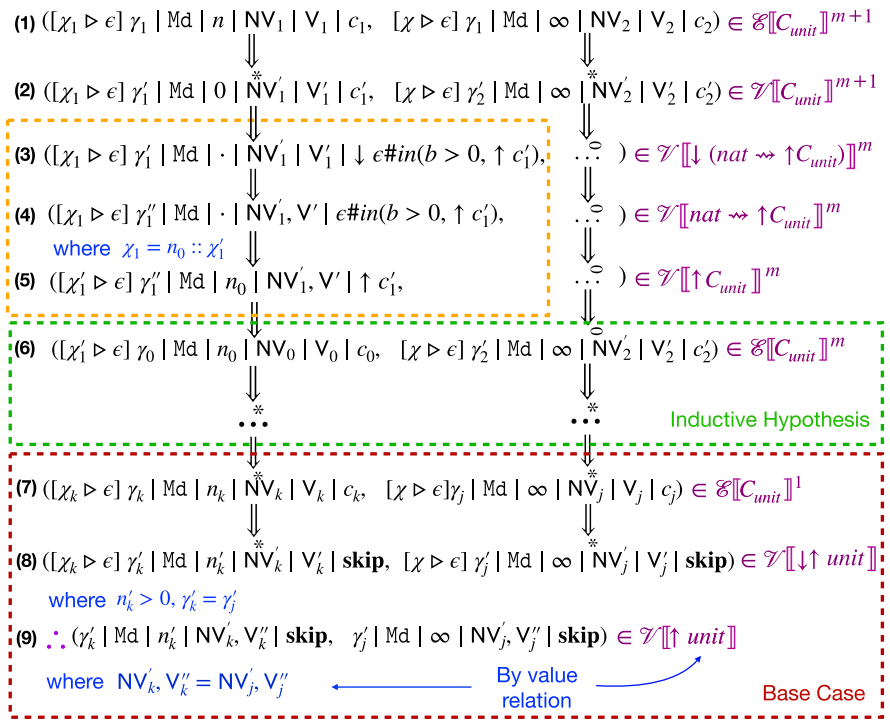

Fig. 28. Illustration of related configurations evaluating to the same store. We include the trace of the intermittent execution (left) and build the corresponding continuous execution (right). The purple text shows the relation between the two configurations at each step according to our logical relation. The orange box highlights the steps taken during a power failure. The green box highlights where the inductive hypothesis is applied. The red box highlights the base case. The blue text gives conditions that are known by assumption or are learned during the proof.

Base Case. When the first configuration finally steps to completion by the D-STEP rule (point (8) in Figure 28), it follows by the definition of logical relation that the second configuration also steps to skip thus detecting a successfully completed execution. This completes the simulation of the continuous execution of c_1. Here, we can apply the rule D-P-CKPT on this construction to obtain a continuously powered execution of the program $\mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c_1); p$. The $\mathsf{FinWorld}_d$ function copies the up-to-date values of volatile memory V_k' that have checkpointed locations back into nonvolatile memory NV_k', removes the subscript $\mathtt{ck}$ from these locations, and changes the qualifiers of all locations in nonvolatile memory to $\mathtt{CK}$. The volatile memory is dropped and the mapping is reset to γ. This matches the commit policy defined for atomic mode. By the value interpretation at type $\downarrow\uparrow\mathtt{unit}$, the two configurations remain related at the type $\uparrow\mathtt{unit}$ (Line (9)). Then, by the value interpretation at type $\uparrow\mathtt{unit}$, the final nonvolatile memories are the same ($\mathsf{NV}_k', \mathsf{V}_k'' = \mathsf{NV}_j', \mathsf{V}_j''$) which is the last piece we needed.

For the special case where $c_2 = c_1$, $\mathsf{NV}_2 = \mathsf{NV}_1$, and $\mathsf{V}_2 = \mathsf{V}_1$, we can apply the rule D-P-CKPT to the constructed execution to find a continuously powered execution of $\mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c_1); p$ as desired: $[\chi \triangleright \varepsilon] \otimes \gamma_1 \mid \mathsf{aID} \mid \infty \mid \mathsf{NV}_1 \mid \mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c_1); p \Rightarrow_p [\chi \triangleright \varepsilon] \otimes \gamma \mid \mathsf{aID} \mid \infty \mid \mathsf{NV}' \mid p$.

Executing a JIT Block. Assume that c_1 is self-related, and $[\chi_1 \triangleright \varepsilon] \otimes \gamma_1 \mid \mathsf{jit} \mid n \mid \mathsf{NV}_1 \mid c_1; p$ takes a step to $[\chi_k \triangleright \varepsilon] \otimes \gamma \mid \mathsf{jit} \mid n' \mid \mathsf{NV}' \mid p$ via the D-P-SEQ rule with possibly m-many power failures

along the way. We need to show that there exists a continuously powered execution stepping the configuration $[\chi \triangleright \varepsilon] \otimes \gamma_1 \mid \mathsf{jit} \mid \infty \mid \mathsf{NV}_1 \mid c_1; p$ to $[\chi \triangleright \varepsilon] \otimes \gamma \mid \mathsf{jit} \mid \infty \mid \mathsf{NV}' \mid p$. Our proof constructs such continuously powered execution.

The overall proof structure is the same as the case for atomic regions. Similarly, the key idea is that the power off and restore policies in the logical relation exactly follow how the rules D-S-JIT and D-RESTORE-JIT handle nonvolatile and volatile memories in the operational semantics.

The proof assumes that $c_1; p$ is semantically well-typed, which by P-SEQ-SEMANTIC, yields that c_1 is semantically well-typed and self-related at type $\mathtt{C}_{\mathtt{unit}}$. By definition of logical relation (Figure 17), if c_1 is semantically well-typed, then its intermittent configuration with energy level n is related with the same configuration but with energy level ∞ for every index, including $m + 1$. We use this condition to obtain the desired result. In particular, we prove a more general goal stating that for any two related configurations, if the first configuration takes zero or more steps, then the second configuration can take zero or more steps such that the stepped configurations remain related. Moreover, in both configurations, the final nonvolatile memories at the end of the JIT block execution will store the same values.

The proof is by induction on the number of power failures m until the first configuration succeeds. Figure 28 shows the inductive case (Lines (1)–(6)) and the base case (Lines (7)–(9)). Starting with index $m + 1$ (point (1) in Figure 28), we show that the intermediate configurations continue to relate at a smaller index m. Then, we apply the inductive hypothesis to obtain the desired result. Finally, in the base case, we show that the memories of the two final configurations are the same.

Inductive Case. By definition of the term relation, c_1 in the first configuration is executed via D-STEP until the next power failure occurs, which is detected by a 0 energy level. Moreover, by the term relation at type $\mathtt{C}_{\mathtt{unit}}$, we can execute c_2 in the second configuration to c_2' such that the resulting configurations remain related by the value relation at type $\mathtt{C}_{\mathtt{unit}}$ (point (2) in Figure 28).

The orange box around Lines (3)–(5) highlights the steps taken when handling a power failure. Each step corresponds to one of the key operations crash, power off, restore, and re-execute. Since power failures are incurred by intermittent executions, we step the first configuration according to the corresponding rules in the operational semantics, but do not step the second configuration through this sequence. Here, the first configuration takes a step from point (2) to point (3) using the D-CRASH rule. By the definition of the logical relation, the two configurations continue to be related by the value interpretation at type $\downarrow(\mathtt{nat} \rightsquigarrow\uparrow \mathtt{C}_{\mathtt{unit}})$. Then, the first configuration takes a step from point (3) to point (4) by the D-S-JIT rule. In this case, we know (by the assumptions of the rule) that $\mathsf{V}' = \mathsf{V}_1'$ and $\gamma_1'' = \gamma_1'$. This matches the definition of the power-off policy for JIT blocks (see Section 6.1, Figure 18). By the value interpretation at type $\downarrow(\mathtt{nat} \rightsquigarrow\uparrow \mathtt{C}_{\mathtt{unit}})$, the two configurations remain related by the value relation at type $\mathtt{nat} \rightsquigarrow\uparrow \mathtt{C}_{\mathtt{unit}}$. Next, the first configuration takes a step to point (5) by the rule D-CHARGE which inputs a new energy level (n_0) from the environment and updates the energy stream from χ_1 to χ_1' where $\chi_1 = n_0 :: \chi_1'$. By the definition of the value relation, the two configurations remain related by the value interpretation at type $\uparrow \mathtt{C}_{\mathtt{unit}}$.

Finally, the configuration steps to point (6) by D-RESTORE-JIT which copies all checkpointed locations in the nonvolatile memory into volatile memory and continues by running the interrupted command c_1'. That is, $\mathsf{V}_0 = \mathsf{V}'$ and $c_0 = c_1'$, and the nonvolatile memory and variable mapping remain the same ($\mathsf{NV}_0 = \mathsf{NV}_1'$ and $\gamma_0 = \gamma_1''$). This matches the restore policy defined for JIT regions. Thus, the configurations continue to be related by the *term relation* at type $\mathtt{C}_{\mathtt{unit}}$, similar to what we had earlier at point (1) in Figure 28, but with fewer power failures remaining. At this point, we apply the inductive hypothesis (highlighted by a green box in Figure 28) to build the continuous execution.

Base Case. The red box in Figure 28 highlights the base case. Since the index is 1, there are 0 remaining power failures. So, $n'_k > 0$ and the first configuration steps to completion via the D-STEP rule. By the definition of the value interpretation, we know that the second configuration also steps to skip indicating a complete execution. This completes the construction of the continuous execution for c_1. The volatile memory V'_j is dropped, and the mapping is reset to γ. This matches the commit policy defined for JIT blocks.

Note that the value interpretation at type $\downarrow\uparrow\texttt{unit}$ establishes that the two configurations remain related at the type $\uparrow\texttt{unit}$ (Line (9)). The commit policy drops the volatile memories V'_k and V'_j such that $\mathsf{V}''_k = \mathsf{V}''_j = \emptyset$. Finally, by the value interpretation at type $\uparrow\texttt{unit}$, we get that the final nonvolatile memories are the same ($\mathsf{NV}'_j, \mathsf{V}''_j = \mathsf{NV}'_k, \mathsf{V}''_k$) which completes the proof.

In the special case, where the initial configurations are the same, we have that $c_2 = c_1, \mathsf{NV}_2 = \mathsf{NV}_1$, and $\mathsf{V}_2 = \mathsf{V}_1$. Here, we can apply the D-P-SEQ rule to the construction to obtain a continuous execution for the program $c_1; p$ as desired: $[\chi \triangleright \varepsilon] \otimes \gamma_1 \mid \mathsf{jit} \mid \infty \mid \mathsf{NV}_1 \mid c_1; p \Rightarrow_p [\chi \triangleright \varepsilon] \otimes \gamma \mid \mathsf{jit} \mid \infty \mid \mathsf{NV}' \mid p$.

9 More General Policies

So far, we have only considered programs consisting of atomic and JIT regions. In this section, we show how to accommodate other kinds of intermittent execution models beyond atomic and JIT regions by utilizing our semantic typing to allow custom policies for power failure, restore, and commit. We generalize the grammar of programs as follows:

$$p := \cdot \mid \mathsf{Reg}[\mathsf{aID}, \overrightarrow{arg}](c); p$$

$\overrightarrow{arg}$ refers to the arguments that the programmer decides to pass to the region for initialization. To each region, we assign a unique identifier aID that is associated with the three policies and two functions $\texttt{InitGeneral}_t$ and $\texttt{InitGeneral}_d$ to initialize the static and dynamic memories, respectively. We add the following semantic typing rule for the general regions:

$$\frac{\begin{array}{c} c_0 \mid \Omega_0 \mid \Sigma_0 = \texttt{InitGeneral}_t(\Omega; (\mathsf{aID}; \overrightarrow{arg}); c;) \\ \mathsf{aID}(c_0) \mid b \geq 0 : \texttt{nat} \mid \Omega_0; \Sigma_0 \Vdash c_0 \leq c_0 : \mathsf{C}_{\texttt{unit}} \quad b : \texttt{nat} \mid \Omega \Vdash p : \uparrow\mathsf{C}_{\texttt{unit}} \end{array}}{b : \texttt{nat} \mid \Omega \Vdash \mathsf{Reg}[\mathsf{aID}, \overrightarrow{arg}](c); p : \uparrow\mathsf{C}_{\texttt{unit}}} \text{ (P-REG-SEMANTIC)}$$

For a self-related region to be idempotent, the policies for power failure, restore, and commit in the logical relation must match the policies in the dynamics. To achieve this, we add the dynamic rules for custom regions in Figure 29 using the same custom policies `Commit`, `PwOff`, and `Restore` used in the logical relation. The JIT and atomic region policies and their dynamic rules are instances of these general policies.

With the custom policies, the logical relation accepts more programs than our type system. For example, consider an atomic region with the code "if z then $x := 1$ else skip." This piece of code writes the variable x in one branch but not the other. Our type system only accepts the program if x is checkpointed. Otherwise, if x is declared as a `MFstWt`, the post-states of typing the two branches will not be the same and the type system rejects it. If the qualifier of x is `MFstWt` initially, it will be `Wtn` at the end of the first branch but remain `MFstWt` after the second branch. However, a programmer can customize the checkpointing policy of this block to not checkpoint variable x, and the program remains semantically well-typed and thus idempotent. The intermittent and continuous executions of this block will always take the same branch and result in the same nonvolatile memories at the end of execution because the value of z does not change in different

$$\frac{\mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0 = \mathtt{Restore}(\gamma, \mathsf{Md}, \mathsf{NV}, \kappa)}{[\chi \triangleright \varepsilon] \otimes \gamma \mid \mathsf{Md} \mid n \mid \mathsf{NV} \mid\!\uparrow \kappa \;\Rightarrow\; [\chi \triangleright \varepsilon] \otimes \gamma_0 \mid \mathsf{Md} \mid n \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0} \text{ (D-R-Reg)}$$

$$\frac{\begin{array}{c} n > 0 \quad \mathtt{InitGeneral}_d(\mathsf{NV}; (\mathsf{aID}; \overrightarrow{arg}); c; \gamma) = c_0, \mathsf{NV}_0, \mathsf{V}_0 \\ [\chi \triangleright \varepsilon] \otimes (\mathsf{aID}(c_0); \overrightarrow{arg}) \mid n \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0 \Rightarrow^* [\chi' \triangleright \varepsilon] \otimes (\mathsf{aID}(c_0); \overrightarrow{arg}) \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid \mathsf{skip} \\ n' > 0 \quad \gamma_1 \mid \mathsf{NV}_1 = \mathtt{Commit}(\gamma, (\mathsf{aID}; \overrightarrow{arg}), \mathsf{NV}', \mathsf{V}') \end{array}}{[\chi \triangleright \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid \mathsf{Reg}[(\mathsf{aID}; \overrightarrow{arg})](c); p \;\Rightarrow\; [\chi' \triangleright \varepsilon] \otimes \gamma_1 \mid n' \mid \mathsf{NV}_1 \mid p} \text{ (D-Reg)}$$

$$\frac{\gamma' \mid \mathsf{V}' = \mathtt{PwOff}(\gamma, \mathsf{Md}, \mathsf{NV}, \mathsf{V})}{[\chi \triangleright \varepsilon] \otimes \gamma \mid \mathsf{Md} \mid \cdot \mid \mathsf{NV} \mid \mathsf{V} \mid \downarrow\varepsilon \,\#\, \mathsf{in}(b > 0; \uparrow\kappa) \;\Rightarrow\; [\chi \triangleright \varepsilon] \otimes \gamma' \mid \mathsf{Md} \mid \cdot \mid \mathsf{NV}, \mathsf{V}' \mid \varepsilon \,\#\, \mathsf{in}(b > 0; \uparrow\kappa)} \text{ (D-S-Reg)}$$

Fig. 29. Custom dynamic rules.

re-executions. However, because the intermittent and continuously powered executions of this region always execute the same branch, it is self-related and thus semantically well-typed.

To accept such programs statically, other type systems [60] implement a conservative merge (making their's less conservative than our type system) that reconciles the differences in post-state between the two branches. Implementing this merge in our type system would not change the logical relation. The policy we have investigated in this work, which checkpoints only WAR variables, is implemented by real systems [32, 34, 59].

10 Related Work

Intermittent Computing. Prior work on designing runtime systems for intermittent computing [34, 52, 62] relied on informal notions of correctness. This left systems susceptible to two classes of memory consistency bugs which result from reading computations from partial executions [32] or allowing stale sensor readings from prior executions to persist in nonvolatile memory [57]. Surbatovich et al. [59] provide the first formal framework for reasoning about intermittent execution, give the correctness definition that we use, and identify precise memory invariants needed for an execution to be correct. Our crash types capture the invariant that WAR variables need to be checkpointed; however, we do not reason about the effect of non-deterministic sensor inputs and leave it to future work.

Curricle developed an information flow type system for checking and inferring which variables in an intermittent program are idempotent or non-idempotent, and reasoning about correctness as an analog to non-interference by asserting that non-idempotent data and power failures do not influence idempotent data [60]. Variable types are annotated with (1) idempotence qualifiers that have interior access mode qualifiers which correspond to the qualifiers in our work and (2) taintedness qualifiers for tracking whether a computation is dependent on an input operation. This tracking allows their type system to detect another class of memory consistency bugs caused by access patterns that involve input dependencies. Their work, while also type-based, focuses on reasoning about these access mode qualifiers whereas our work reasons about the key operations of intermittent computation.

Other works that investigate the formal properties of intermittent computing either reason about the effects of intermittent execution on peripheral interactions [9] or enforce timeliness constraints on sensor readings [58], which are orthogonal to ours.

Persistent Memory. Other works study fault tolerance with respect to different memory models. Notably, persistent memory models rely solely on nonvolatile memory to perform a computation [14, 33, 38, 63]. Work in this area builds persistent memory interfaces that constrain write orders for executing concurrent [42, 43] or parallel [11] programs. Our work reasons about sequential program execution on processors with both volatile and nonvolatile memories.

Weak persistency semantics formalizes the behavior of programs running under persistency models for specific processors [49–51]. They prove their systems are correct according to persistent linearizability, which characterizes the correctness of concurrent program execution according to write orders [20, 25]. Our formalism is at a higher level than theirs, not considering specific processors. To our knowledge, our work is the first to characterize the logical interaction between volatile and nonvolatile memories.

Adjoint Logic. Benton et al. [7, 8] provided the first categorical foundation for using adjoint functors to combine linear and nonlinear logics and showed that a well-behaved calculus requires an independence principle: Linear formulae cannot appear in the assumptions of a nonlinear sequent. Follow-up works further generalized the system [29, 30, 53]. There, the relation to Pfenning and Davies's [44] formulation of the lax ◯ modality was noted; ◯ corresponds to UF, where F and U are adjunctions between truth and validity categories. Short of a full Curry-Howard correspondence for our type system and underlying logic, we designed the rules for $\uparrow$ and $\downarrow$ based on the above calculi. Our stable and unstable contexts correspond to the validity and truth contexts, respectively. Thus, we speculate that the combination $\uparrow\downarrow$ in our system corresponds to the lax modality.

Several prior works used type systems with adjoint modalities to model switching between program modes [6, 18, 48], e.g., switching a process's mode between shared and unshared [6], or adding multicasting, replicable services, and cancelation modes to a session-typed message passing system [48]. We are the first to use these modalities to handle unforeseen shut-downs and distinguish between stable and power-failure prone modes.

Logical Relations. Prior work [3, 61] uses step indexing to ensure the well-foundedness of logical relations that handle heaps with cyclic references, dynamic memory allocation, or recursive types. Our crash types model the infinite computation that an atomic region can experience under a non-deterministic number of power failures and re-executions. This recursion necessitates an indexed relation that limits the number of execution attempts a program can make.

Jung and Tiuryn introduced a logical relation for lambda definability that allows varying arities [27]. The idea is to increase the arity when passing to later worlds instead of starting with a large arity. Our logical relation can also be viewed as a relation with different arities; the initial type of the relation is binary, while after a crash the type of the value relation only corresponds to the intermittent configuration. During these value steps, the relation is unary, with the continuous configuration acting as a Kripke world for the intermittent configuration. After restoration, the relation reverts to binary.

Logical relations have been widely used to prove program equivalence, e.g., [2, 3, 10, 22]. At a high level, idempotency is similar to program equivalence, but it handles re-execution and requires us only to prove simulation from an intermittent to continuous run, not vice-versa.

Algebraic Effect Handlers. Algebraic effect handlers [37, 45–47] give a unified theory for computational effects, e.g., exceptions and interactive input/output. A handler accesses the continuation to transform the computation. Following effect handler syntax, we write effectful environmental interactions of our system as $\varepsilon\#\mathsf{in}(b > 0, \uparrow\kappa)$, where b refers to a natural number returned by the environment and $\uparrow\kappa$ is the continuation. Our restore policy resembles a handler, in that it has access to the continuation, but an atomic region may dismiss the continuation, restarting from a saved command.

Crash Consistency. The failure and recovery patterns this article observes for intermittent computing are similar to those of file systems. Verifying the correctness of file system operation under crashes is also an important task [26] of which several works have studied [12, 13, 23, 41, 55, 56].

Notably, Bornholt et al. [12] develop crash-consistency models, similar to memory consistency models, to characterize the failure and recovery behaviors of file systems across crashes. Their framework Ferrite can be used to verify formal specifications of file systems by both exhaustively executing litmus tests against all possible crash behaviors and symbolically executing litmus tests against a specification. Their memory model is similar to ours, considering both volatile and nonvolatile state, and providing rules that define the interaction between these memories in the presence of a crash. Like idempotence for intermittent computing, the notion of correctness for crash-consistency models relates a crashy (intermittent) trace of a program to its canonical (continuously powered) trace. Our work provides formal proofs that automatically verify idempotence once and for all programs in our language.

Crash Hoare Logic (CHL) [13] ensures the correctness of crash and restore operations in a file system. CHL extends Hoare logic with a crash condition and a recovery procedure. The crash condition states what happens to the state on a crash. The recovery procedure runs after the crash and manipulates the state before resuming. The system checks that if the program crashes, the storage system will recover to a state consistent with the specifications. Unlike us, they do not consider idempotency, requiring manual effort to formalize the crash condition and recovery policy. Our syntactic typing fixes the power failure, restore, and commit policies, and our formal results guarantee that following the policies ensures idempotency, the common correctness condition for intermittent execution. We also allow the programmer to formalize bespoke semantically well-typed policies.

11 Discussion and Future Work

In this section, we discuss assumptions and limitations of our current system and highlight potential directions for future work.

Language Features. Our calculus models a simple imperative language, which is based on constructs common to embedded programs. We do not, for example, consider unbounded loops, dynamic memory allocation, and first-class pointers, as they are not commonly used in embedded programs. Bounded loops can be unrolled into commands with if statements, which we do support. We leave extending our work to handle pointers as future work. Our recent work suggests that with a type system like Rust, pointers can be straightforwardly incorporated [60].

Variable Annotations. Our typing rules ensure the correctness of intermittent execution, taking programmer-provided typing annotations as input. A typing error would surface if the program's access patterns of a variable do not agree with its typing annotations. For example, if a variable exhibiting WAR patterns is not annotated as being checkpointed, a type error would occur. Programmers need to change the annotation of this variable for the program to type check. To give another example, T-P-Ckpt checks that all must-first-write variables have been written by the end of an atomic region by checking that no variables in the post-context have type qualifier `MFstWt`. Additionally, the $\mathsf{InitWorld}_t$ function, which sets up the initial typing contexts for an atomic region, checks that the provided read-only, must-first-write, and checkpointed sets are disjoint from each other. Such annotations can be automatically inferred; for instance, our recent work [60] provides inference of qualifiers similar to ours.

System Assumptions. The programs accepted by our type system are free from memory corruption errors because our simple calculus does not expose raw pointers like C. However, for our results to apply to real C programs, we would need to assume memory safety. Unsafe memory behavior would complicate and even invalidate the analysis of which memory location is written to. With

that said, programs written in a memory safe language like Rust can be analyzed using our type system (c.f. [60]).

Implementation. We plan to implement our modal crash type system, targeting Rust as its memory safety guarantees match our assumptions. Moreover, the existing Rust implementation of Curricle [60] serves as an attractive starting point, since it supports both type checking and inference for an extended set of access qualifiers compared to those used in this work. However, Curricle's type system uses only one store (so has simpler well-formedness conditions for memory), has simpler conditions for checking crashes/power failures, and does not support shift operations. Thus, developing an implementation that combines Curricle's more sophisticated access qualifiers and checkpointing policies with crash types' more sophisticated treatment of crashes and restores remains future work.

Forward Progress. Finally, in this article, we show that if there is enough energy in the symbolic energy channel, the program will eventually finish executing in a finite number of attempted power cycles. However, we cannot in general guarantee that there is always enough energy to do so. Reasoning about actual energy consumption for embedded systems is highly complex and we leave it as future work.

12 Conclusion

This work provides the first logical interpretation of intermittent execution for programs consisting of atomic and JIT regions. We apply adjoint logic to define crash types which internalize the dualities between stable and unstable values, and complete versus partial (re-)executions of intermittent programs. The typing constraints capture invariants of power failure, restoration, and re-execution in intermittent systems. The proofs of progress, preservation, the fundamental theorem, and adequacy imply the correctness of intermittent systems, i.e., idempotency of execution.

Acknowledgment

We especially thank the TOPLAS reviewers, whose feedback helped to improve the presentation and readability of this article.

References

[1] Joshua Adkins, Bradford Campbell, Branden Ghena, Neal Jackson, Pat Pannuto, and Prabal Dutta. 2016. The Signpost network: Demo abstract. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM (SenSys '16)*, 320–321. DOI: https://doi.org/10.1145/2994551.2996542

[2] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, 340–353. DOI: https://doi.org/10.1145/1480881.1480925

[3] Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State*. PhD thesis. Princeton University.

[4] D. Balsamo, A. Weddell, A. Das, A. Arreola, D. Brunelli, B. Al-Hashimi, G. Merrett, and L. Benini. 2016. Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 99 (2016), 1. DOI: https://doi.org/10.1109/TCAD.2016.2547919

[5] Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. 2015. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters* 7, 1 (2015), 15–18. DOI: https://doi.org/10.1109/LES.2014.2371494

[6] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest deadlock-freedom for shared session types. In *Proceedings of the 29th European Symposium on Programming*, 611–639.

[7] Nick Benton and Philip Wadler. 1996. Linear logic, monads and the lambda calculus. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 420–431.

[8] P. Nick Benton. 1994. A mixed linear and non-linear logic: Proofs, terms and models. In *Proceedings of the International Workshop on Computer Science Logic*. Springer, 121–135.

[9] Gautier Berthou, Pierre-Évariste Dagand, Delphine Demange, Rémi Oudin, and Tanguy Risset. 2020. Intermittent computing with peripherals, formally verified. In *Proceedings of the 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '20)*. ACM, New York, NY, 85–96. DOI: https://doi.org/10.1145/3372799.3394365

[10] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. 2009. Realizability semantics of parametric polymorphism, general references, and recursive types. In *Proceedings of the International Conference on Foundations of Software Science and Computational Structures (FOSSACS '09)*. Springer, 456–470.

[11] Guy E. Blelloch, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2018. The parallel persistent memory model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18)*, 247–258. DOI: https://doi.org/10.1145/3210377.3210381

[12] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and checking file system crash-consistency models. *ACM SIGARCH Computer Architecture News* 44, 2 (Mar. 2016), 83–98. DOI: https://doi.org/10.1145/2980024.2872406

[13] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, 18–37. DOI: https://doi.org/10.1145/2815400.2815402

[14] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*, 105–118. DOI: https://doi.org/10.1145/1950365.1950380

[15] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and channels for reliable intermittent programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '16)*, 514–530. DOI: https://doi.org/10.1145/2983990.2983995

[16] Alexei Colin and Brandon Lucia. 2018. Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*, 116–127.

[17] Manjeet Dahiya and Sorav Bansal. 2018. Automatic verification of intermittent systems. In *Verification, Model Checking, and Abstract Interpretation*. Dillig Isil and Palsberg Jens (Eds.), Springer International Publishing, Cham, 161–182.

[18] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2021. Resource-aware session types for digital contracts. In *Proceedings of the IEEE 34th Computer Security Foundations Symposium (CSF '21)*, 1–16.

[19] Farzaneh Derakhshan, Myra Dotzel, Milijana Surbatovich, and Limin Jia. 2023. Modal crash types for intermittent computing. In *Proceedings of the 32nd European Symposium on Programming Languages and Systems (ESOP '23), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS '23)*. Springer-Verlag, Berlin, 168–196. DOI: https://doi.org/10.1007/978-3-031-30044-8_7

[20] Emanuele D'Osualdo, Azalea Raad, and Viktor Vafeiadis. 2023. The path to durable linearizability. *Proceedings of the ACM on Programming Languages* 7, POPL, Article 26 (Jan. 2023), 27 pages. DOI: https://doi.org/10.1145/3571219

[21] Myra Dotzel, Farzaneh Derakhshan, Milijana Surbatovich, and Limin Jia. 2023. *Technical Report: Modal Crash Types for WAR-Aware Intermittent Computing*. Technical Report. Carnegie Mellon University. DOI: https://doi.org/10.1184/R1/24556708

[22] Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming* 22, 4–5 (2012), 477–528.

[23] Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, and Wolfgang Reif. 2016. Inside a verified flash file system: Transactions and garbage collection. In *Verified Software: Theories, Tools, and Experiments*. Arie Gurfinkel and Sanjit A. Seshia (Eds.), Springer International Publishing, Cham, 73–93. DOI: https://doi.org/10.1007/978-3-319-29613-5_5

[24] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 1–13. DOI: https://doi.org/10.1145/3131672.3131673

[25] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing*. Cyril Gavoille and David Ilcinkas (Eds.), Springer, Berlin, 313–327.

[26] Rajeev Joshi and Gerard Holzmann. 2007. A mini challenge: Build a verifiable filesystem. *Formal Aspects of Computing* 19 (Jun. 2007), 269–272. DOI: https://doi.org/10.1007/s00165-006-0022-3

[27] Achim Jung and Jerzy Tiuryn. 1993. A new characterization of lambda definability. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*. Springer, 245–257.

[28] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. 2020. Time-sensitive intermittent computing meets legacy software. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. ACM, New York, NY, 85–99. DOI: https://doi.org/10.1145/3373376.3378476

[29] Daniel R. Licata and Michael Shulman. 2016. Adjoint logic with a 2-category of modes. In *Proceedings of the International Symposium on Logical Foundations of Computer Science*. Springer, 219–235.

[30] Daniel R. Licata, Michael Shulman, and Mitchell Riley. 2017. A fibrational framework for substructural and modal logics. In *Proceedings of the 2nd International Conference on Formal Structures for Computation and Deduction (FSCD '17)*, 1–22. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[31] Brandon Lucia, Brad Denby, Zachary Manchester, Harsh Desai, Emily Ruppel, and Alexei Colin. 2021. Computational nanosatellite constellations: Opportunities and challenges. *GetMobile: Mobile Computing and Communications* 25, 1 (Jun. 2021), 16–23. DOI: https://doi.org/10.1145/3471440.3471446

[32] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*, 575–585. DOI: https://doi.org/10.1145/2737924.2737978

[33] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015. Architecture exploration for ambient energy harvesting nonvolatile processors. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 526–537. DOI: https://doi.org/10.1109/HPCA.2015.7056060

[34] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 96 (Oct. 2017), 96:1–96:30 pages. DOI: https://doi.org/10.1145/3133920

[35] Kiwan Maeng and Brandon Lucia. 2019. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*, 1101–1116. DOI: https://doi.org/10.1145/3314221.3314613

[36] Kiwan Maeng and Brandon Lucia. 2020. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. ACM, New York, NY, 1005–1021. DOI: https://doi.org/10.1145/3385412.3385998

[37] Eugenio Moggi. 1988. *Computational Lambda-Calculus and Monads*. PhD thesis. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science.

[38] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*, 401–410. DOI: https://doi.org/10.1145/2150976.2151018

[39] Matteo Nardello, Harsh Desai, Davide Brunelli, and Brandon Lucia. 2019. Camaroptera: A batteryless long-range remote visual sensing system. In *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems (ENSsys'19)*. ACM, New York, NY, 8–14. DOI: https://doi.org/10.1145/3362053.3363491

[40] NASA. 2022. KickSat-2. Retrieved January 23, 2025 from https://nssdc.gsfc.nasa.gov/nmc/spacecraft/display.action?id=2018-092G

[41] Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. 2015. Fault-tolerant resource reasoning. In *Proceedings of the Programming Languages and Systems (ESOP '15)*. Xinyu Feng and Sungwoo Park (Eds.), Springer International Publishing, Cham, 169–188.

[42] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA '14)*, 265–276. Piscataway, NJ. DOI: https://doi.org/10.1109/ISCA.2014.6853222

[43] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2015. Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. *IEEE Micro* 35, 3 (2015), 125–131.

[44] Frank Pfenning and Rowan Davies. 2001. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science* 11, 4 (2001), 511–540.

[45] Gordon Plotkin and John Power. 2001. Semantics for algebraic operations. *Electronic Notes in Theoretical Computer Science* 45 (2001), 332–345.

[46] Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *Proceedings of the 19th European Symposium on Programming*. Springer, 80–94.

[47] Matija Pretnar and Gordon D. Plotkin. 2013. Handling algebraic effects. *Logical Methods in Computer Science* 9 (2013), 1–36.

[48] Klaas Pruiksma and Frank Pfenning. 2021. A message-passing interpretation of adjoint logic. *Journal of Logical and Algebraic Methods in Programming* 120 (2021), 100637.

[49] Azalea Raad and Viktor Vafeiadis. 2018. Persistence semantics for weak memory: Integrating epoch persistency with the TSO memory model. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 137 (Oct. 2018), 27 pages. DOI: https://doi.org/10.1145/3276507

[50] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency semantics of the Intel-X86 architecture. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 11 (Dec. 2019), 31 pages. DOI:

https://doi.org/10.1145/3371079
[51] Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 135 (Oct. 2019), 27 pages. DOI: https://doi.org/10.1145/3360561
[52] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System support for long-running computation on RFID-scale devices. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*, 159–170. DOI: https://doi.org/10.1145/1950365.1950386
[53] Jason Reed. 2009. *A Judgmental Deconstruction of Modal Logic*. Carnegie Mellon University.
[54] Emily Ruppel and Brandon Lucia. 2019. Transactional concurrency control for intermittent, energy-harvesting computing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*, 1085–1100. DOI: https://doi.org/10.1145/3314221.3314583
[55] Gerhard Schellhorn, Gidon Ernst, Jörg Pfähler, Dominik Haneberg, and Wolfgang Reif. 2014. Development of a verified flash file system. In *Proceedings of the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z-Volume 8477 (ABZ '14)*. Springer-Verlag, Berlin, 9–24. DOI: https://doi.org/10.1007/978-3-662-43652-3_2
[56] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Savannah, GA, 1–16. DOI: https://doi.org/10.5555/3026877.3026879
[57] Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2019. I/O dependent idempotence bugs in intermittent systems. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 183 (Oct. 2019), 31 pages. DOI: https://doi.org/10.1145/3360609
[58] Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2021. Automatically enforcing fresh and consistent inputs in intermittent systems. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. ACM, New York, NY, 851–866. DOI: https://doi.org/10.1145/3453483.3454081
[59] Milijana Surbatovich, Brandon Lucia, and Limin Jia. 2020. Towards a formal foundation of intermittent computing. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 163 (Nov. 2020), 31 pages. DOI: https://doi.org/10.1145/3428231
[60] Milijana Surbatovich, Naomi Spargo, Limin Jia, and Brandon Lucia. 2023. A type system for safe intermittent computing. *Proceedings of the ACM on Programming Languages* 7, PLDI, Article 136 (Jun. 2023), 25 pages. DOI: https://doi.org/10.1145/3591250
[61] Jacob Thamsborg and Lars Birkedal. 2011. A Kripke logical relation for effect-based program transformations. *ACM SIGPLAN Notices* 46, 9 (2011), 445–456.
[62] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent computation without hardware support or programmer intervention. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 17–32. DOI: https://doi.org/10.5555/3026877.3026880
[63] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*, 91–104. DOI: https://doi.org/10.1145/1950365.1950379
[64] Kasim Sinan Yildirim, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. InK: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SenSys '18)*. ACM, New York, NY, 41–53. DOI: https://doi.org/10.1145/3274783.3274837

Appendices

A Store Typing

The rules in Figure A1 define the well-formedness of NV and V with respect to Ω which corresponds to a crash where V is dropped. The rules are similar, but here the typing judgments of the values v and v' handle the possibility of a crash. This definition is only used in proofs of minor lemmas, so it is included here in the appendix rather than in the main text.

B Progress and Preservation for Open Configurations

Lemma B.1 (Progress for Shifted Expressions). *If*

$$\texttt{Md} \mid b{:}\texttt{nat} \mid \Omega \vdash_{Rd} e : \uparrow A$$

$\boxed{\vdash^{\mathsf{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega}$

$$\frac{}{\vdash_{\gamma} \cdot \mid \cdot : \cdot}\ (\textsc{-Empty})$$

$$\frac{\vdash^{\mathsf{Jit}}_{\gamma'} \mathsf{NV}' \mid \mathsf{V} : \Omega \quad \mathsf{NV} = \mathsf{NV}', \ell \mathbin{@} q \hookrightarrow v \quad q = \mathsf{Ck} \quad \gamma = \gamma', [x \mapsto \ell] \quad \mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathsf{RD}} v : \uparrow A^{s}}{\vdash^{\mathsf{Jit}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega, (x{:}\uparrow A@q)}\ (\textsc{nv-loc-Jit})$$

$$\frac{\vdash^{\mathsf{Md}}_{\gamma'} \mathsf{NV} \mid \mathsf{V}' : \Omega \quad \mathsf{V} = \mathsf{V}', \ell \mathbin{@} q \hookrightarrow v \quad q = \mathsf{Ck} \quad \gamma = \gamma', [x \mapsto \ell] \quad \mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathsf{RD}} v : \uparrow A^{s}}{\vdash^{\mathsf{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega, (x{:}\uparrow A@q)}\ (\textsc{v-loc})$$

$$\frac{\vdash^{\mathsf{aID}}_{\gamma'} \mathsf{NV}' \mid \mathsf{V} : \Omega \quad \mathsf{NV} = \mathsf{NV}', \ell \mathbin{@} q \hookrightarrow v \quad q \neq \mathsf{Ck} \quad \gamma = \gamma', [x \mapsto \ell] \quad \mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathsf{RD}} v : \uparrow A^{s}}{\vdash^{\mathsf{aID}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega, (x{:}\uparrow A@q)}\ (\textsc{nv-loc-aID-1})$$

$$\frac{\begin{array}{c}\vdash^{\mathsf{aID}}_{\gamma'} \mathsf{NV}' \mid \mathsf{V}' : \Omega \quad \mathsf{NV} = \mathsf{NV}', \ell_{\mathsf{ck}} \mathbin{@} q \hookrightarrow v \\ \mathsf{V} = \mathsf{V}', \ell \mathbin{@} q \hookrightarrow v' \quad q = \mathsf{Ck} \quad \gamma = \gamma', [x \mapsto \ell] \quad \mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathsf{RD}} v : \uparrow A^{s} \quad \mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathsf{RD}} v' : \uparrow A^{s}\end{array}}{\vdash^{\mathsf{aID}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega, (x_{\mathsf{ck}}{:}\uparrow A@q), (x{:}\uparrow A@q)}\ (\textsc{nv-loc-aID-2})$$

Fig. A1. Well-formedness of NV | V w.r.t. Ω.

then $\forall\ n : \mathtt{nat}$ *with* $n > 0$ *and* $\forall\, \mathsf{NV}, \mathsf{V}, \gamma$ *with* $\vdash^{\mathsf{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega$, *either*

—$Val(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e)$ *or*
—$\exists(\gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid e')$ *such that* $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e \ \longrightarrow\ \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid e'$.

Proof. The proof proceeds by structural induction over $\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{Rd} e : \uparrow A$. We include the full proof in the extended TR [21]. □

Theorem B.2 (Progress for Expressions). *If* $\mathtt{Md} \mid b\ \mathcal{R}\ m : \mathtt{nat} \mid \Omega; \Sigma \vdash_{Rd;\mathtt{Sig}} e : \tau$, *then* $\forall\ n : nat$ *with* $n\mathcal{R}m$ *and* $\forall\ \mathsf{NV}, \mathsf{V}, \gamma$ *with* $\vdash^{\mathsf{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$, *either*

—$Val(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e)$ *or*
—$\exists(\gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid e')$ *such that* $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e \ \longrightarrow\ \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid e'$.

Proof. The proof is by structural induction over $\mathtt{Md} \mid b\ \mathcal{R}\ m : \mathtt{nat} \mid \Omega; \Sigma \vdash_{Rd;\mathtt{Sig}} e : \tau$. We consider a specific (co-)natural number $n\mathcal{R}m$ and contexts $\mathsf{NV}, \mathsf{V}, \gamma$ with $\vdash^{\mathsf{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$. We include the full proof in the extended TR [21]. □

Lemma B.3. *If* $\vdash^{\mathsf{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$ *and* $\Sigma = \downarrow\Sigma'$, *then* $\vdash^{\mathsf{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega, \Sigma'$.

Proof. The proof is by induction on the structure of $\vdash^{\mathsf{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$. For each step in the derivation, we build the corresponding step of a derivation for $\vdash^{\mathsf{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega, \Sigma'$ according to the well-formedness definition. □

Theorem 6.1 (Progress for Commands). *If* $\mathtt{Md} \mid b\ \mathcal{R}\ m : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau \dashv \Omega'$, *then* $\forall\ n : \mathtt{nat}$ *with* $n\mathcal{R}m$ *and* $\forall\, \gamma, \mathsf{NV}, \mathsf{V}$ *with* $\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$, *either*

—$Val(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c)$ *or*
—$\exists(\gamma' \mid \mathtt{Md}' \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c')$ *such that* $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c \ \longrightarrow \gamma' \mid \mathtt{Md}' \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'$.

Proof. The proof is by structural induction over $\mathtt{Md} \mid b\ \mathcal{R}\ m : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau \dashv \Omega'$. We include the full proof in the extended TR [21]. □

Axiom 1 (Positive Input to Generation Channel). $\varepsilon\ \#\ \mathsf{in}() : \mathtt{nat} > 0$.

Lemma B.4 (Well-Typedness of Expressions Under Crash in jit). $\mathsf{jit} \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}'} e : \mathsf{C}_{\mathsf{A}}^{\mathsf{jit}}$ *for* $\mathtt{Sig}' = \{\mathsf{jit} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD}} e : \mathsf{C}_{\mathsf{A}}^{\mathsf{jit}}\}$.

Proof. We include the full proof in the extended TR [21]. □

Lemma B.5 (Well-Typedness of Expressions Under Crash in aID). *If* $\mathsf{aID}(c_0) \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma' \vdash_{\mathtt{RD;Sig}} e' : \tau$ *then for all* e *and* Σ, $\mathsf{aID}(c_0) \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e : \tau$.

Proof. We include the full proof in the extended TR [21]. □

Lemma B.6 (Well-Typedness of Commands Under Crash in jit). $\mathsf{jit} \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}'} c : \mathsf{C}_{\mathtt{unit}}^{\mathsf{jit}}$ *for* $\mathtt{Sig}' = \{\mathsf{jit} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash c : \mathsf{C}_{\mathtt{unit}}^{\mathsf{jit}}\}$.

Proof. We include the full proof in the extended TR [21]. □

Lemma B.7 (Well-Typedness of Commands Under Crash in aID). *If* $\mathsf{aID}(c_0) \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma' \vdash_{\mathtt{Sig}} c' : \tau$ *then for all* c *and* Σ, $\mathsf{aID}(c_0) \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau$.

Proof. We include the full proof in the extended TR [21]. □

Theorem B.8 (Preservation for Expressions). *If*

$$(\dagger) \quad \mathtt{Md} \mid b \geq 0 : \mathit{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e : \tau$$

and for some $\vdash_{\gamma}^{\mathtt{Md}} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$ *and (co-)natural number* $n \geq 0$, *we have*

$$\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e \longrightarrow \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid e'$$

then

$$\mathtt{Md} \mid b \geq 0 : \mathit{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e' : \tau$$

with $n' \geq 0$.

Proof. The proof is by induction on the size of e. We include the full proof in the extended TR [21]. □

Definition B.9. We write $\Sigma' = \mathsf{trim}(\Sigma, \mathsf{V}, \gamma)$ where $x{:}\tau@q \in \Sigma'$ iff $\gamma = [x \mapsto \ell], \gamma'$ and $x{:}\tau@q \in \Sigma$ and $\ell \in \mathit{dom}(\mathsf{V})$.

Lemma B.10 (Equality of Trimmed Volatile Contexts). *If*

(i) $\Sigma' = \mathsf{trim}(\Sigma, \mathsf{V}_0, \gamma_0)$
(ii) $\vdash_{\gamma}^{\mathtt{Md}} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$
(iii) $\vdash_{\gamma''}^{\mathtt{Md}} \mathsf{NV}' \mid \mathsf{V}' : \Omega \mid \Sigma''$
(iv) $\mathit{dom}(\mathsf{V}_0) \subseteq \mathit{dom}(\mathsf{V})$ *and* $\mathit{dom}(\mathsf{V}_0) \subseteq \mathit{dom}(\mathsf{V}')$
(v) $\gamma_0 \subseteq \gamma$ *and* $\gamma_0 \subseteq \gamma''$

then $\Sigma' = \mathsf{trim}(\Sigma'', \mathsf{V}_0, \gamma_0)$.

Proof. We include the full proof in the extended TR [21]. □

Lemma B.11 (Well-Formedness of Smaller Memories). *If*

(i) $\vdash_{\gamma}^{\mathtt{Md}} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$,
(ii) $\mathsf{V}'' = \mathsf{V} \restriction \mathit{dom}(\mathsf{V}')$,
(iii) $\Sigma' = \mathsf{trim}(\Sigma, \mathsf{V}', \gamma')$, *and*
(iv) $\gamma' \subseteq \gamma$

then $\vdash_{\gamma'}^{\mathtt{Md}} \mathsf{NV} \mid \mathsf{V}'' : \Omega \mid \Sigma'$.

Proof. The proof proceeds by induction on the size of $\mathsf{V} - \mathsf{V}''$. We include the full proof in the extended TR [21]. □

Lemma B.12 (Monotonicity of Volatile Memories). *If* $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c \rightarrow \gamma' \mid \mathtt{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'$ *where* $c \neq c_1;_W c_2$, *then* $dom(\mathsf{V}) \subseteq dom(\mathsf{V}')$ *and* $\gamma \subseteq \gamma'$.

Proof. The proof is straightforward, proceeding in cases on the dynamic rules. □

Definition B.13. $\Omega > \Omega'$ iff $\forall x{:}\tau@q \in$, we have $x{:}\tau@q' \in \Omega'$ where $q' \neq UN$ and either $q = q'$ or $\delta(q, Wt) = q'$.

Lemma B.14. *If* $\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \mathsf{C}^{\mathtt{Md}}_{\mathtt{unit}} \dashv \Omega'$, *then* $dom(\Omega) \subseteq dom(\Omega')$.

Proof. The proof is by induction on the size of c. We consider possible cases for $\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \mathsf{C}^{\mathtt{Md}}_{\mathtt{unit}} \dashv \Omega'$.

Case [T-C-Shift].

$$\frac{\Sigma = \downarrow\Sigma' \quad \Omega = \Omega', \Omega''_{\mathtt{ck}} \quad \mathtt{Md} \mid b : \mathtt{nat} \mid \Omega', \Sigma' \vdash_{\mathtt{Sig}} \mathsf{skip} : \uparrow\mathtt{unit} \dashv \Omega_1}{\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathsf{skip} : \downarrow\uparrow\mathtt{unit} \dashv \Omega_1 \restriction dom(\Omega)} \text{(T-C-Shift)}$$

By definition of $\restriction$, we obtain the desired result $dom(\Omega) \subseteq dom(\Omega_1 \restriction dom(\Omega))$.

Case [T-seq].

$$\frac{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c_1 : \mathsf{C}^{\mathtt{Md}}_{\mathtt{unit}} \dashv \Omega' \quad \mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega'; \Sigma \vdash_{\mathtt{Sig}} c_2 : \tau \dashv \Omega''}{\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c_1; c_2 : \tau \dashv \Omega''} \text{(T-seq)}$$

By inversion of T-seq, we learn that
— $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c_1 : \mathsf{C}^{\mathtt{Md}}_{\mathtt{unit}} \dashv \Omega'$
— $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega'; \Sigma \vdash_{\mathtt{Sig}} c_2 : \tau \dashv \Omega''$
By applying the inductive hypothesis to each of these judgments, we learn that $dom(\Omega) \subseteq dom(\Omega')$ and $dom(\Omega') \subseteq dom(\Omega'')$. By transitivity, we establish the desired result $dom(\Omega) \subseteq dom(\Omega'')$.

Case [T-seq-d]. This case is similar to T-seq.

Case [T-∨-Succ, T-Let, T-Assign, T-If, T-enough?]. In each case, we apply the inductive hypothesis to learn that $dom(\Omega) \subseteq dom(\Omega')$. □

Theorem 6.2 (Preservation for Commands). *If*

$$(\dagger) \quad \mathtt{Md} \mid b \geq 0 : nat \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau \dashv \Omega'$$

and $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c$ *is well-formed,* $\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$, $dom(\mathsf{NV}_{\mathtt{ck}}) \subseteq dom(\mathsf{V})$, *and for some (co-)natural number* $n \geq 0$, *we have*

$$\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c \rightarrow \gamma' \mid \mathtt{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'$$

then for some Σ', *and* Ω_0

$$\mathtt{Md} \mid b \geq 0 : nat \mid \Omega_0; \Sigma' \vdash_{\mathtt{Sig}} c' : \tau \dashv \Omega'$$

where $\vdash^{\mathtt{Md}}_{\gamma'} \mathsf{NV}' \mid \mathsf{V}' : \Omega_0 \mid \Sigma'$, $\Omega > \Omega_0$, *and* $n' \geq 0$. *Moreover* $\gamma' \mid \mathtt{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'$ *is well-formed. Moreover,* $dom(\mathsf{NV}) = dom(\mathsf{NV}')$, *and if* $\mathtt{Md} = \mathsf{aID}(c_0)$, *then* $\mathsf{NV}' \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\} = \mathsf{NV} \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\}$ *and* $dom(\mathsf{NV}'_{\mathtt{ck}}) \subseteq dom(\mathsf{V}')$.

Proof. The proof is by induction on the size of c. We consider possible cases for $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c \;\rightarrow\; \gamma' \mid \mathtt{Md}' \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'$ and only show the most interesting ones.

Case [D-Assign-V].

$$\frac{\begin{array}{c} \mathit{Val}(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e) \\ \mathsf{V} = \mathsf{V}', \ell@q \hookrightarrow v' \quad q' = \delta(q, \mathtt{wt}) \neq \mathtt{UN} \quad \gamma = \gamma', [x \rightarrow \ell] \quad n = n' + 1 \end{array}}{\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid x := e \;\rightarrow\; \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V}', \ell@q' \hookrightarrow e \mid \mathbf{skip}} \text{(D-Assign-V)}$$

By the last premise $n > 0$, and $n' \geq 0$. By inversion on (†) via T-Enough? rule, we have

$$(\dagger_1) \quad \mathtt{Md} \mid b \;>\; 0 : \mathtt{nat} \;\mid\; \Omega; \Sigma \vdash_{\mathtt{Sig}} x := e : \tau \dashv \Omega'$$

and

$$(\dagger_2) \quad \mathtt{Md} \mid b \;=\; 0 : \mathtt{nat} \;\mid\; \Omega; \Sigma \vdash_{\mathtt{Sig}''} x := e : \tau$$

where $\mathtt{Sig}'' = \mathit{if}\ \mathtt{Md} = \mathtt{jit}\ \mathit{then}\ \mathtt{Sig}'\ \mathit{else}\ \mathtt{Sig}$ and $\mathtt{Sig}' = \{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash x := e : \tau\}$.

By inversion on $(\dagger_1)$ via T-Assign

$$\frac{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \;\mid\; \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e : \mathsf{C}_A^{\mathtt{Md}} \quad \mathtt{Md} \mid b > 0 : \mathtt{nat} \;\mid\; \Omega; \Sigma \vdash_{\mathtt{WT}} x : \downarrow\uparrow A \dashv \Omega'}{\mathtt{Md} \mid b > 0 : \mathtt{nat} \;\mid\; \Omega; \Sigma \vdash_{\mathtt{Sig}} x := e : \mathsf{C}_{\mathtt{unit}}^{\mathtt{Md}} \dashv \Omega'} \text{(T-Assign)}$$

we have

(1) $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \;\mid\; \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e : \mathsf{C}_A^{\mathtt{Md}}$

(2) $\mathtt{Md} \mid b > 0 : \mathtt{nat} \;\mid\; \Omega; \Sigma \vdash_{\mathtt{WT}} x : \downarrow\uparrow A \dashv \Omega'$

By V-Loc and the definition of V, we have that $\vdash_{\gamma}^{\mathtt{Md}} \mathsf{NV} \mid \mathsf{V}', (\ell@\mathtt{Ck} \hookrightarrow v') : \Omega \mid \Sigma_0, (x :\downarrow\uparrow A@\mathtt{Ck})$ where $\Sigma = \Sigma_0, (x :\downarrow\uparrow A@\mathtt{CK})$. It follows by inversion on T-w-Shift and T-Loc-Write

$$\frac{\Omega = \Omega_0, \Omega_{1\,\mathtt{ck}} \qquad \dfrac{\begin{array}{c}\Sigma = \downarrow\Sigma' \\ \Omega_0, \Sigma' = x : \uparrow A@\mathtt{CK}, \Omega_2' = \Omega'' \quad \mathtt{CK} = \delta(\mathtt{CK}, Wt) \neq \mathit{UN}\end{array}}{\mathtt{Md} \mid b \;>\; 0 : \mathtt{nat} \;\mid\; \Omega_0, \Sigma' \vdash_{\mathtt{WT}} x : \uparrow A \dashv \Omega''} \text{(T-Loc-Write)}}{\mathtt{Md} \mid b \;>\; 0 : \mathtt{nat} \;\mid\; \Omega; \Sigma \vdash_{\mathtt{WT}} x : \downarrow\uparrow A \dashv \Omega'' \restriction dom(\Omega)} \text{(T-w-Shift)}$$

that

$-\,\Omega' = \Omega'' \restriction dom(\Omega)\ (\exists \Omega'')$

$-\,\Omega'' = \Omega_0, \Sigma'$

$-\,\Sigma = \downarrow\Sigma'$

$-\,\Omega = \Omega_0, \Omega_{1\,\mathtt{ck}}$

It follows by assumption $\mathsf{NV}_{\mathtt{ck}} \subseteq \mathsf{V}$ and the well-formedness condition $\vdash_{\gamma}^{\mathtt{Md}} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$ that $\Omega_{1\,\mathtt{ck}} \subseteq \Sigma'$. Hence $\Omega \subseteq \Omega_0, \Sigma'$. Since $\Omega'' = \Omega_0, \Sigma'$, it follows that $\Omega \subseteq \Omega''$, and hence $\Omega = \Omega'' \restriction dom(\Omega)$. Since $\Omega' = \Omega'' \restriction dom(\Omega)$, we have that $\Omega = \Omega'$.

By the premise $\mathit{Val}(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e)$, we can apply inversion on $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e : \mathsf{C}_A^{\mathtt{Md}}$ via T-Enough?, T-∨-Succ, and T-R-Shift to get

$$\mathtt{Md} \mid b : \mathtt{nat} \;\mid\; \Omega, \Sigma' \vdash_{\mathtt{RD;Sig}} e : \uparrow A,$$

where $\Sigma = \downarrow\Sigma'$.

Applying V-Loc, we can show that $\vdash_{\gamma}^{\mathtt{Md}} \mathsf{NV} \mid \mathsf{V}', (\ell@\mathtt{Ck} \hookrightarrow e) : \Omega \mid \Sigma_0, (x :\downarrow\uparrow A@\mathtt{Ck})$ where $\Sigma' = \Sigma_0, (x :\downarrow\uparrow A@\mathtt{Ck})$.

We want to show that

$$\mathtt{Md} \mid b \;>\; 0 : \mathtt{nat} \;\mid\; \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathbf{skip} : \mathsf{C}_{\mathtt{unit}} \dashv \Omega'' \restriction dom(\Omega).$$

Noting that $\Omega = \Omega'' \restriction dom(\Omega)$, it follows by T-Skip, T-C-Shift, and T-∨-Succ that

$$\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathbf{skip} : \mathtt{C}_{\mathtt{unit}} \dashv \Omega.$$

We consider two subcases based on Md.

Subcase 1. [Md = Jit]. Let $\mathtt{Sig}_1 = \{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash \mathbf{skip} : \mathtt{C}_{\mathtt{unit}}\}$. From Lemma B.6, we get $\mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathbf{skip} : \mathtt{C}_{\mathtt{unit}}$.

Subcase 2. [Md = aID(c_0)]. By ($\dagger_2$) via Lemma B.7, we get $\mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathbf{skip} : \mathtt{C}_{\mathtt{unit}}$.

In both subcases, the desired result follows by T-Enough?:

$$\frac{\begin{array}{c}\mathtt{Sig}_1 = \{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash \mathbf{skip} : \tau\} \\ \mathtt{Sig}_2 = \mathit{if}\ \mathtt{Md} = \mathsf{jit}\ \mathit{then}\ \mathtt{Sig}_1, \mathit{else}\ \mathtt{Sig} \quad \mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}_2} \mathbf{skip} : \mathtt{C}_{\mathtt{unit}} \\ \mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathbf{skip} : \mathtt{C}_{\mathtt{unit}} \dashv \Omega\end{array}}{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathbf{skip} : \mathtt{C}_{\mathtt{unit}} \dashv \Omega} \text{(T-enough?)}$$

Observe that the well-formedness of $\gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV} \mid \mathsf{V}', \ell@q' \hookrightarrow e \mid \mathbf{skip}$ follows by Definition 4.1, vacuously, and $\Omega > \Omega$ follows by Definition 8.3, both vacuously. Additionally, observe that $\mathsf{NV} \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\} = \mathsf{NV} \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\}$, as desired. Lastly, note that $dom(\mathsf{V}) = dom(\mathsf{V}', \ell@q' \hookrightarrow e)$. Therefore, it follows by assumption that $dom(\mathsf{NV}_{\mathtt{ck}}) \subseteq dom(\mathsf{V}', \ell@q' \hookrightarrow e)$.

Case [D-Assign-NV].

$$\frac{\begin{array}{c}\mathit{Val}(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e) \\ \mathsf{NV} = \mathsf{NV}', \ell@q \hookrightarrow v' \quad q' = \delta(q, Wt) \neq \mathtt{UN} \quad \gamma = \gamma', [x \to \ell] \quad n = n' + 1\end{array}}{\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid x := e \ \longrightarrow\ \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV}', \ell@q' \hookrightarrow e \mid \mathsf{V} \mid \mathbf{skip}} \text{(D-Assign-NV)}$$

By the last premise $n > 0$, and $n' \geq 0$. By inversion on (†) via T-Enough? rule, we have

$$(\dagger_1) \quad \mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} x := e : \tau \dashv \Omega'$$

and

$$(\dagger_2) \quad \mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}''} x := e : \tau$$

where $\mathtt{Sig}'' = \mathit{if}\ \mathtt{Md} = \mathsf{jit}\ \mathit{then}\ \mathtt{Sig}'\ \mathit{else}\ \mathtt{Sig}$ and $\mathtt{Sig}' = \{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash x := e : \tau\}$.

By inversion on ($\dagger_1$) via T-Assign

$$\frac{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e : \mathtt{C}_A^{\mathtt{Md}} \quad \mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{WT}} x : \downarrow\uparrow A \dashv \Omega'}{\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} x := e : \mathtt{C}_{\mathtt{unit}}^{\mathtt{Md}} \dashv \Omega'} \text{(T-Assign)}$$

we learn that

(1) $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e : \mathtt{C}_A^{\mathtt{Md}}$

(2) $\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{WT}} x : \downarrow\uparrow A \dashv \Omega'$

By inversion on (2) via T-w-Shift and T-Loc-Write:

$$\frac{\dfrac{\begin{array}{c}\Sigma = \downarrow\Sigma' \quad \Omega = \Omega_0, \Omega_{1\,\mathtt{ck}} \\ \Omega_0, \Sigma' = x : \uparrow A@q, \Omega_2' \quad \Omega'' = x : \uparrow A@q', \Omega_2' \quad q' = \delta(q, Wt) \neq UN\end{array}}{\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega_0, \Sigma' \vdash_{\mathtt{WT}} x : \uparrow A \dashv \Omega''} \text{(T-Loc-Write)}}{\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{WT}} x : \downarrow\uparrow A \dashv \Omega'' \restriction dom(\Omega)} \text{(T-w-Shift)}$$

we have

(i) $\Omega' = \Omega'' \restriction dom(\Omega)$ $(\exists \Omega'')$

(ii) $\Sigma = \downarrow\Sigma'$

(iii) $\Omega = \Omega_0, \Omega_{1\,\mathtt{ck}}$

(iv) $\Omega_0, \Sigma' = x : \uparrow A@q, \Omega_2'$
(v) $\Omega'' = x : \uparrow A@q', \Omega_2'$
(vi) $q' = \delta(q, Wt) \neq UN$

By $Val(\gamma \mid \texttt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e)$, we can apply inversion on $\texttt{Md} \mid b \geq 0 : \texttt{nat} \mid \Omega; \Sigma \vdash_{\texttt{RD;Sig}} e : \texttt{C}_A^{\texttt{Md}}$ via T-Enough?, T-∨-Succ, and T-R-Shift to get

$$\texttt{Md} \mid b : \texttt{nat} \mid \Omega, \Sigma' \vdash_{\texttt{RD;Sig}} e : \uparrow A,$$

where $\Sigma = \downarrow \Sigma'$. This is enough to prove $\vdash_{\gamma}^{\texttt{Md}} (\ell@\texttt{Ck} \hookrightarrow e, \mathsf{NV}'') \mid \mathsf{V} : (x : \uparrow A@q', \Omega_2') \mid \Sigma$. From here, we note that since $x : \uparrow A@q$, we have $x : \uparrow A@q \in \Omega_0$. Therefore, $\Omega'' = \Omega^0, \Sigma'$ for some $\Omega^0 < \Omega_0$ (i.e., $\Omega^0 = \Omega_0'', x : \uparrow A@q'$ where $\Omega_0 = \Omega_0'', x : \uparrow A@q$).
We now need to show that

$$\texttt{Md} \mid b > 0 : \texttt{nat} \mid \Omega'; \Sigma \vdash_{\texttt{Sig}} \textbf{skip} : \texttt{C}_{\texttt{unit}} \dashv \Omega'' \restriction dom(\Omega).$$

Let $\Omega' = \Omega^0, \Omega_{\texttt{ck}}^1$. By T-Skip, T-C-Shift, and T-∨-Succ, and recalling that $\Omega' = \Omega'' \restriction dom(\Omega)$ (as defined above), we have

$$\dfrac{\dfrac{\Omega' = \Omega^0, \Omega_{\texttt{ck}}^1 \quad \dfrac{\Sigma = \downarrow \Sigma'}{\texttt{Md} \mid b > 0 : \texttt{nat} \mid \Omega^0, \Sigma' \vdash_{\texttt{Sig}} \textbf{skip} : \uparrow \texttt{unit} \dashv \Omega''}\,(\text{T-Skip})}{\texttt{Md} \mid b > 0 : \texttt{nat} \mid \Omega'; \Sigma \vdash_{\texttt{Sig}} \textbf{skip} : \downarrow \uparrow \texttt{unit} \dashv \Omega'' \restriction dom(\Omega')}\,(\text{T-C-Shift})}{\texttt{Md} \mid b > 0 : \texttt{nat} \mid \Omega'; \Sigma \vdash_{\texttt{Sig}} \textbf{skip} : \texttt{C}_{\texttt{unit}} \dashv \Omega'' \restriction dom(\Omega')}\,(\text{T-∨-Succ})$$

We now need to show that $dom(\Omega') = dom(\Omega)$. From (i), we have that $dom(\Omega') \subseteq dom(\Omega)$. By Lemma B.14, we have that $dom(\Omega) \subseteq dom(\Omega')$.
We consider two subcases based on Md.
Subcase 1. $[\texttt{Md} = \texttt{Jit}]$. Let $\texttt{Sig}_1 = \{\texttt{Md} \mid b \geq 0 : \texttt{nat} \mid \Omega'; \Sigma \vdash \textbf{skip} : \texttt{C}_{\texttt{unit}}\}$. By Lemma B.6, we get $\texttt{Md} \mid b = 0 : \texttt{nat} \mid \Omega'; \Sigma \vdash \textbf{skip} : \texttt{C}_{\texttt{unit}}$.
Subcase 2. $[\texttt{Md} = \texttt{aID}(c_0)]$. By $(\dagger_2)$ via Lemma B.7, we get $\texttt{Md} \mid b = 0 : \texttt{nat} \mid \Omega'; \Sigma \vdash \textbf{skip} : \texttt{C}_{\texttt{unit}}$.
In both subcases, the desired result follows by T-Enough?:

$$\dfrac{\begin{array}{c}\texttt{Sig}_1 = \{\texttt{Md} \mid b \geq 0 : \texttt{nat} \mid \Omega; \Sigma \vdash \textbf{skip} : \texttt{C}_{\texttt{unit}}\} \\ \texttt{Sig}_2 = \textit{if}\ \texttt{Md} = \texttt{jit}\ \textit{then}\ \texttt{Sig}_1, \textit{else}\ \texttt{Sig} \quad \texttt{Md} \mid b = 0 : \texttt{nat} \mid \Omega'; \Sigma \vdash_{\texttt{Sig}_2} \textbf{skip} : \texttt{C}_{\texttt{unit}} \\ \texttt{Md} \mid b > 0 : \texttt{nat} \mid \Omega'; \Sigma \vdash_{\texttt{Sig}} \textbf{skip} : \texttt{C}_{\texttt{unit}} \dashv \Omega'' \restriction dom(\Omega')\end{array}}{\texttt{Md} \mid b \geq 0 : \texttt{nat} \mid \Omega'; \Sigma \vdash_{\texttt{Sig}} \textbf{skip} : \texttt{C}_{\texttt{unit}} \dashv \Omega'' \restriction dom(\Omega')}\,(\text{T-enough?})$$

Observe that the well-formedness of $\gamma \mid \texttt{Md} \mid n' \mid \mathsf{NV}', \ell@q' \hookrightarrow e \mid \mathsf{V} \mid \textbf{skip}$ follows by Definition 4.1 vacuously. The detail $\Omega > \Omega'$ follows by Definition 8.3, (iv) and (v) from above, and the premise $q' = \delta(q, Wt) \neq \texttt{UN}$. If $\texttt{Md} = \texttt{aID}(c_0)$, we want to show that $\mathsf{NV} \setminus \{\texttt{MFstWt}, \texttt{Wtn}\} = (\mathsf{NV}', \ell@q' \hookrightarrow e) \setminus \{\texttt{MFstWt}, \texttt{Wtn}\}$. By definition, $\mathsf{NV} = \mathsf{NV}', \ell@q \hookrightarrow v'$. Since ℓ is written to, $q' = \texttt{Wtn}$, and either $q = \texttt{Wtn}$ or $q = \texttt{MFstWt}$. Now we need to show that $\mathsf{NV}', \ell@q \hookrightarrow v' \setminus \{\texttt{MFstWt}, \texttt{Wtn}\} = \mathsf{NV}', \ell@q' \hookrightarrow e \setminus \{\texttt{MFstWt}, \texttt{Wtn}\}$. From above, this reduces to showing $\mathsf{NV}' \setminus \{\texttt{MFstWt}, \texttt{Wtn}\} = \mathsf{NV}' \setminus \{\texttt{MFstWt}, \texttt{Wtn}\}$ which holds trivially. Lastly, note that $dom(\mathsf{NV}) = dom(\mathsf{NV}', \ell@q' \hookrightarrow e)$. Therefore, it follows by assumption, and the definition of δ (i.e., $q' = \texttt{Ck}$ if $q = \texttt{Ck}$), that $dom(\mathsf{NV}_{\texttt{ck}}^0) \subseteq dom(\mathsf{V})$ where $\mathsf{NV}', \ell@q' \hookrightarrow e = \mathsf{NV}_{\texttt{ck}}^0, \mathsf{NV}^1$.

Case [D-SEQ-STEP].

$$\frac{n > 0 \quad \gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c_1 \longrightarrow \gamma' \mid \mathtt{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c_1'}{\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c_1;_W c_2 \longrightarrow \gamma' \mid \mathtt{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c_1';_W c_2} \text{ (D-SEQ-STEP)}$$

The premises yield

(a) $n > 0$

(b) $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c_1 \longrightarrow \gamma' \mid \mathtt{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c_1'$

By assumption, note that

$-\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$

$-\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c_1;_W c_2 : \tau \dashv \Omega'$

$-\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c_1;_W c_2$ is well-formed.

Put $W = \gamma_0 \mid \mathsf{V}_0$. Then by Definition 4.1, we have $dom(\mathsf{V}_0) \subseteq dom(\mathsf{V})$ and $\gamma_0 \subseteq \gamma$. Observe that $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c_1$ is well-formed, which vacuously follows by Definition 4.1 because c_1 does not have the form $c';_{W'} c''$ and we always run the command c_1 before c_2.

By inversion of $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c_1;_W c_2 : \tau \dashv \Omega'$ via T-ENOUGH? rule, we have

$$(\dagger_1) \quad \mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c_1;_W c_2 : \tau \dashv \Omega'$$

and

$$(\dagger_2) \quad \mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}''} c_1;_W c_2 : \tau$$

where $\mathtt{Sig}'' = \mathit{if}\ \mathtt{Md} = \mathsf{jit}, \mathit{then}\ \mathtt{Sig}', \mathit{else}\ \mathtt{Sig}$ and $\mathtt{Sig}'' = \{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash c_1;_W c_2 : \tau\}$.

By inversion of $(\dagger_1)$ via T-SEQ-D

$$\frac{\begin{array}{c} W = \gamma_0 \mid \mathsf{V}_0 \quad \mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c_1 : \mathsf{C}_{\mathtt{unit}} \dashv \Omega' \\ \Sigma' = \mathsf{trim}(\Sigma, \mathsf{V}_0, \gamma_0) \quad \mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega'; \Sigma' \vdash_{\mathtt{Sig}} c_2 : \tau \dashv \Omega'' \end{array}}{\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c_1;_W c_2 : \tau \dashv \Omega''} \text{ (T-SEQ-D)}$$

we learn that

(1) $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c_1 : \mathsf{C}_{\mathtt{unit}} \dashv \Omega'$

(2) $\Sigma' = \mathsf{trim}(\Sigma, \mathsf{V}_0, \gamma_0)$

(3) $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega'; \Sigma' \vdash_{\mathtt{Sig}} c_2 : \tau \dashv \Omega''$

By the inductive hypothesis applied to (1), (b), the well-formedness of $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c_1$, $\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$, $n \geq 0$, and $dom(\mathsf{NV}_{\mathtt{ck}}) \subseteq dom(\mathsf{V})$, we get $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega_0; \Sigma'' \vdash_{\mathtt{Sig}} c_1' : \mathsf{C}_{\mathtt{unit}} \dashv \Omega'$, where

(i) $\vdash^{\mathtt{Md}}_{\gamma'} \mathsf{NV}' \mid \mathsf{V}' : \Omega_0 \mid \Sigma''$,

(ii) $n' \geq 0$,

(iii) $\gamma' \mid \mathtt{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c_1'$ is well-formed,

(iv) $\Omega > \Omega_0$,

(v) if $\mathtt{Md} = \mathsf{aID}(c_0)$, $\mathsf{NV} \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\} = \mathsf{NV}' \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\}$, and

(vi) $dom(\mathsf{NV}'_{\mathtt{ck}}) \subseteq dom(\mathsf{V}')$.

We now need to show that $dom(\mathsf{NV}'_{\mathtt{ck}}) \subseteq dom(\mathsf{V}_0) \subseteq dom(\mathsf{V}')$ and $\gamma_0 \subseteq \gamma'$. Observe that $c_1 \neq c';_W c''$ because $c_1;_W c_2$ and we always run c_1 first. By Lemma B.12 and $c_1 \neq c';_W c''$, it follows that $dom(\mathsf{V}) \subseteq dom(\mathsf{V}')$ and $\gamma \subseteq \gamma'$. Then it follows by $dom(\mathsf{V}_0) \subseteq dom(\mathsf{V})$ and $\gamma_0 \subseteq \gamma$ (as shown above), that $dom(\mathsf{V}_0) \subseteq dom(\mathsf{V}) \subseteq dom(\mathsf{V}')$ and $\gamma_0 \subseteq \gamma \subseteq \gamma'$. Additionally, it follows by assumption that $dom(\mathsf{NV}) = dom(\mathsf{NV}')$, and so it follows by the assumption $dom(\mathsf{NV}_{\mathtt{ck}}) \subseteq dom(\mathsf{V}_0)$ that $dom(\mathsf{NV}'_{\mathtt{ck}}) \subseteq dom(\mathsf{V}_0)$. Hence, $dom(\mathsf{NV}'_{\mathtt{ck}}) \subseteq dom(\mathsf{V}_0) \subseteq dom(\mathsf{V}')$ and $\gamma_0 \subseteq \gamma'$.

It suffices to show $\Sigma' = \mathsf{trim}(\Sigma'', \mathsf{V}_0, \gamma_0)$.

By Lemma B.10, it follows that $\Sigma' = \mathsf{trim}(\Sigma'', \mathsf{V}_0, \gamma_0)$.
Using $W = \gamma_0 \mid \mathsf{V}_0$, (2), (3), and $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega_0; \Sigma'' \vdash_{\mathtt{Sig}} c_1' : \mathtt{C}_{\mathtt{unit}} \dashv \Omega'$, we can apply T-SEQ-D

$$\frac{W = \gamma_0 \mid \mathsf{V}_0 \quad \mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega_0; \Sigma'' \vdash_{\mathtt{Sig}} c_1' : \mathtt{C}_{\mathtt{unit}} \dashv \Omega' \quad \Sigma' = \mathsf{trim}(\Sigma'', \mathsf{V}_0, \gamma_0) \quad \mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega'; \Sigma' \vdash_{\mathtt{Sig}} c_2 : \tau \dashv \Omega''}{\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega_0; \Sigma'' \vdash_{\mathtt{Sig}} c_1';_W c_2 : \tau \dashv \Omega''} \text{(T-SEQ-D)}$$

We now need to show that $\mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \Omega_0; \Sigma'' \vdash_{\mathtt{Sig}} c_1';_W c_2 : \tau$. The proof proceeds in two subcases based on Md:

Case $\mathtt{Md} = \mathsf{jit}$. Let $\mathtt{Sig}_1 = \{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega_0; \Sigma'' \vdash c_1';_W c_2 : \tau\}$. It follows by Lemma B.6 that $\mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \Omega_0; \Sigma'' \vdash c_1';_W c_2 : \tau$.

Case $\mathtt{Md} = \mathsf{aID}(c_0)$. By $(\dagger_2)$ via Lemma B.7, we can see that $\mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \Omega_0; \Sigma'' \vdash c_1';_W c_2 : \tau$.

In both cases, $\mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \Omega_0; \Sigma'' \vdash c_1';_W c_2 : \tau$.
The desired result follows by T-ENOUGH?:

$$\frac{\begin{array}{c}\mathtt{Sig}_1 = \{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega_0; \Sigma'' \vdash c_1';_W c_2 : \tau\} \\ \mathtt{Sig}_2 = \mathit{if}\ \mathtt{Md} = \mathsf{jit}, \mathit{then}\ \mathtt{Sig}_1, \mathit{else}\ \mathtt{Sig} \quad \mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \Omega_0; \Sigma'' \vdash c_1';_W c_2 : \tau \\ \mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega_0; \Sigma'' \vdash c_1';_W c_2 : \tau \dashv \Omega''\end{array}}{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega_0; \Sigma'' \vdash c_1';_W c_2 : \tau \dashv \Omega''} \text{(T-ENOUGH?)}$$

and (iv), which asserts that $\Omega > \Omega_0$.
Observe that by Definition 4.1 applied to $\mathit{dom}(\mathsf{NV}'_{\mathsf{ck}}) \subseteq \mathit{dom}(\mathsf{V}_0) \subseteq \mathit{dom}(\mathsf{V}')$ and $\gamma_0 \subseteq \gamma'$ (as shown above), $\gamma' \mid \mathtt{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c_1';_W c_2$ is well-formed. In the case where c_1' is of the form $c';_W c''$, the rule stepping c_1 must be D-SEQ since $c_1 \neq c';_W c''$. Thus, observe that $\gamma' \subseteq \gamma'$ and $\mathit{dom}(\mathsf{NV}'_{\mathsf{ck}}) \subseteq \mathit{dom}(\mathsf{V}') \subseteq \mathit{dom}(\mathsf{V}')$, and hence it follows by Definition 4.1 that $\gamma' \mid \mathtt{Md} \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c_1';_W c_2$ is well-formed.

Note that $\mathit{dom}(\mathsf{NV}) = \mathit{dom}(\mathsf{NV}')$ holds by observation, and by the inductive hypothesis in D-SEQ-STEP. □

LEMMA B.15. *If* $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV}_1 \mid \mathsf{V} \mid c \to \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV}'_1 \mid \mathsf{V}' \mid c'$ *and* $\mathsf{NV}_1 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}_2 \setminus \{\mathtt{MFstWt}\}$, *then* $\gamma \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V} \mid c \to \gamma \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}'_2 \mid \mathsf{V}' \mid c'$ *and* $\mathsf{NV}'_1 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}'_2 \setminus \{\mathtt{MFstWt}\}$.

PROOF. The proof is by induction on the size of c. We proceed by considering possible cases for $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV}_1 \mid \mathsf{V} \mid c \to \gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV}'_1 \mid \mathsf{V}' \mid c'$. The most interesting case is [D-ASSIGN-NV].

Case [D-ASSIGN-NV].

$$\frac{\mathit{Val}(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e) \quad \mathsf{NV} = \mathsf{NV}', \ell@q \hookrightarrow v' \quad q' = \delta(q, \mathtt{wt}) \neq \mathtt{UN} \quad \gamma = \gamma', [x \to \ell] \quad n = n' + 1}{\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid x := e \ \to \gamma \mid \mathtt{Md} \mid n' \mid \mathsf{NV}', \ell@q' \hookrightarrow e \mid \mathsf{V} \mid \mathbf{skip}} \text{(D-ASSIGN-NV)}$$

By inversion of D-ASSIGN-NV, we learn that
(1) $\mathit{Val}(\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid e)$
(2) $\mathsf{NV} = \mathsf{NV}', \ell@q \hookrightarrow v'$
(3) $q' = \delta(q, \mathtt{wt}) \neq \mathtt{UN}$
(4) $\gamma = \gamma', [x \to \ell]$
(5) $n = n' + 1$

Let $\mathsf{NV} \setminus \{\texttt{MFstWt}\} = \mathsf{NV}_2 \setminus \{\texttt{MFstWt}\}$. We want to show that $\gamma \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V} \mid x := e \;\longrightarrow\; \gamma \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2', \ell@q' \hookrightarrow e \mid \mathsf{V} \mid \textbf{skip}$ and $\mathsf{NV}', \ell@q' \hookrightarrow e \setminus \{\texttt{MFstWt}\} = \mathsf{NV}_2', \ell@q' \hookrightarrow e \setminus \{\texttt{MFstWt}\}$ where $\mathsf{NV}_2 = \mathsf{NV}_2', \ell@q \hookrightarrow v_2'$.
From (5), we know that (1) is not a crash value (i.e., $n > 0$, and hence $n \neq 0$). Since $\infty > 0$, we have $\mathit{Val}(\gamma \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V} \mid e)$. By application of D-Assign-NV to $\mathit{Val}(\gamma \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V} \mid e)$, $\mathsf{NV}_2 = \mathsf{NV}_2', \ell@q \hookrightarrow v_2'$, (3), (4), and $\infty > 0$, we have

$$\gamma \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V} \mid x := e \;\longrightarrow\; \gamma \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2', \ell@q' \hookrightarrow e \mid \mathsf{V} \mid \textbf{skip}$$

Towards $\mathsf{NV}', \ell@q' \hookrightarrow e \setminus \{\texttt{MFstWt}\} = \mathsf{NV}_2', \ell@q' \hookrightarrow e \setminus \{\texttt{MFstWt}\}$, observe that the assumption $\mathsf{NV} \setminus \{\texttt{MFstWt}\} = \mathsf{NV}_2 \setminus \{\texttt{MFstWt}\}$ implies $\mathsf{NV}' \setminus \{\texttt{MFstWt}\} = \mathsf{NV}_2' \setminus \{\texttt{MFstWt}\}$. Thus, it follows that $\mathsf{NV}', \ell@q' \hookrightarrow e \setminus \{\texttt{MFstWt}\} = \mathsf{NV}_2', \ell@q' \hookrightarrow e \setminus \{\texttt{MFstWt}\}$, as desired. □

C Fundamental Theorem

Lemma C.1. *If* $\mathsf{aID}(c) \mid b \geq 0 : \texttt{nat} \mid \Omega'; \Sigma \vdash_{\texttt{Sig}} c : \mathsf{C}_{\texttt{unit}} \dashv \Omega''$ *and* $\vdash_{\gamma_0}^{\mathsf{aID}(c)} \mathsf{NV}_1 \mid \mathsf{V}_0 : \Omega' \mid \Sigma$, *then* $\exists.(\gamma_1 \mid \mathsf{aID}(\mathsf{c}) \mid n_1 \mid \mathsf{NV}_1' \mid \mathsf{V}_1 \mid c_1)$ *such that*

– $\gamma_0 \mid \mathsf{aID}(\mathsf{c}) \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c \longrightarrow^* \gamma_1 \mid \mathsf{aID}(\mathsf{c}) \mid n_1 \mid \mathsf{NV}_1' \mid \mathsf{V}_1 \mid c_1$,
– $\mathsf{NV}_1 \setminus \{\texttt{MFstWt}, \texttt{Wtn}\} = \mathsf{NV}_1' \setminus \{\texttt{MFstWt}, \texttt{Wtn}\}$, *and*
– $\mathit{dom}(\mathsf{NV}_1) = \mathit{dom}(\mathsf{NV}_1')$.

Proof. We prove this by induction on n_0:

Base Case. If $n_0 = 0$, then the configuration is a value.
Inductive Case. Suppose that $n_0 = n_0' + 1$ ($\exists n_0'$). Since $\mathsf{aID}(c) \mid b \geq 0 : \texttt{nat} \mid \Omega'; \Sigma \vdash_{\texttt{Sig}} c : \mathsf{C}_{\texttt{unit}} \dashv \Omega''$ and $\vdash_{\gamma_0}^{\mathsf{aID}(c)} \mathsf{NV}_1 \mid \mathsf{V}_0 : \Omega' \mid \Sigma$, it follows by the progress theorem (Theorem 8.1) that either $\gamma_0 \mid \mathsf{aID}(\mathsf{c}) \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c$ is a value or $\gamma_0 \mid \mathsf{aID}(\mathsf{c}) \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c$ is not a value, in which case $\exists\, \gamma_1'' \mid \mathsf{aID}(\mathsf{c}) \mid n_1'' \mid \mathsf{NV}_1'' \mid \mathsf{V}_1'' \mid c_1''$ such that

$$\gamma_0 \mid \mathsf{aID}(\mathsf{c}) \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c \longrightarrow \gamma_1'' \mid \mathsf{aID}(\mathsf{c}) \mid n_1'' \mid \mathsf{NV}_1'' \mid \mathsf{V}_1'' \mid c_1''$$

where
– $\mathsf{aID}(c) \mid b \geq 0 : \texttt{nat} \mid \Omega_0''; \Sigma' \vdash_{\texttt{Sig}} c_1'' : \mathsf{C}_{\texttt{unit}} \dashv \Omega''$,
– $\Omega' > \Omega_0''$,
– $\vdash_{\gamma_1''}^{\mathsf{aID}(c)} \mathsf{NV}_1'' \mid \mathsf{V}_1'' : \Omega_0'' \mid \Sigma'$,
– $\gamma_1'' \mid \mathsf{aID}(\mathsf{c}) \mid n_1'' \mid \mathsf{NV}_1'' \mid \mathsf{V}_1'' \mid c_1''$ is well-formed,
– if $\texttt{Md} = \mathsf{aID}(c)$, $\mathsf{NV}_1 \setminus \{\texttt{MFstWt}, \texttt{Wtn}\} = \mathsf{NV}_1'' \setminus \{\texttt{MFstWt}, \texttt{Wtn}\}$, and
– $\mathit{dom}(\mathsf{NV}_1) = \mathit{dom}(\mathsf{NV}_1'')$.
We get these conditions by applying the preservation theorem (Theorem 8.2) because $\gamma_0 \mid \mathsf{aID}(\mathsf{c}) \mid n_0 \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c$ is well-formed.
By the inductive hypothesis,

$$\gamma_1'' \mid \mathsf{aID}(\mathsf{c}) \mid n_1'' \mid \mathsf{NV}_1'' \mid \mathsf{V}_1'' \mid c_1'' \longrightarrow^* \gamma_1' \mid \mathsf{aID}(\mathsf{c}) \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1',$$

where
– $\gamma_1' \mid \mathsf{aID}(\mathsf{c}) \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1'$ is well-formed and a value,
– $\Omega_0'' > \Omega_0'$,
– $\mathsf{aID}(c) \mid b \geq 0 : \texttt{nat} \mid \Omega_0'; \Sigma'' \vdash_{\texttt{Sig}} c_1' : \mathsf{C}_{\texttt{unit}} \dashv \Omega''$,
– $\vdash_{\gamma_1'}^{\mathsf{aID}(c)} \mathsf{NV}_1' \mid \mathsf{V}_1' : \Omega_0' \mid \Sigma''$,
– if $\texttt{Md} = \mathsf{aID}(c)$, $\mathsf{NV}_1'' \setminus \{\texttt{MFstWt}, \texttt{Wtn}\} = \mathsf{NV}_1' \setminus \{\texttt{MFstWt}, \texttt{Wtn}\}$, and
– $\mathit{dom}(\mathsf{NV}_1'') = \mathit{dom}(\mathsf{NV}_1')$.

By head expansion, we establish that

$$\gamma_0 \mid \mathsf{aID(c)} \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c \rightarrow^* \gamma_1' \mid \mathsf{aID(c)} \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1'$$

where $\mathsf{NV}_1 \setminus \{\texttt{MFstWt}, \texttt{Wtn}\} = \mathsf{NV}_1' \setminus \{\texttt{MFstWt}, \texttt{Wtn}\}$ and $dom(\mathsf{NV}_1) = dom(\mathsf{NV}_1')$. □

Lemma C.2. *If* $\gamma_0 \mid \mathsf{aID(c)} \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c \rightarrow^m \gamma_1 \mid \mathsf{aID(c)} \mid n_1 \mid \mathsf{NV}_1' \mid \mathsf{V}_1 \mid c_1$ *and* $\mathsf{NV}_1 \setminus \{\texttt{MFstWt}\} = \mathsf{NV}_2 \setminus \{\texttt{MFstWt}\}$, *then* $\gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c \rightarrow^m \gamma_1' \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_1' \mid c_1'$ *where* $\mathsf{NV}_1' \setminus \{\texttt{MFstWt}\} = \mathsf{NV}_2' \setminus \{\texttt{MFstWt}\}$.

Proof. The proof proceeds by induction on the number of steps m such that $\gamma_0 \mid \mathsf{aID(c)} \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c \rightarrow^m \gamma_1' \mid \mathsf{aID(c)} \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1'$.

Base Case ($m = 0$). By assumption, $\mathsf{NV}_1 \setminus \{\texttt{MFstWt}\} = \mathsf{NV}_2 \setminus \{\texttt{MFstWt}\}$. Trivially,

$$\gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c \rightarrow^0 \gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c$$

where $\mathsf{NV}_1 \setminus \{\texttt{MFstWt}\} = \mathsf{NV}_2 \setminus \{\texttt{MFstWt}\}$.

Inductive Case ($m = k + 1$). Suppose that $m = k + 1$ such that

$$\begin{aligned} &\gamma_0 \mid \mathsf{aID(c)} \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c \\ \rightarrow &\gamma_1'' \mid \mathsf{aID(c)} \mid n_1'' \mid \mathsf{NV}_1'' \mid \mathsf{V}_1'' \mid c_1'' \\ \rightarrow^k &\gamma_1' \mid \mathsf{aID(c)} \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1' \end{aligned}$$

By assumption, $\mathsf{NV}_1 \setminus \{\texttt{MFstWt}\} = \mathsf{NV}_2 \setminus \{\texttt{MFstWt}\}$. Hence, it follows by Lemma B.15

$$\gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c \rightarrow \gamma_1'' \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2'' \mid \mathsf{V}_1'' \mid c_1''$$

where $\mathsf{NV}_1'' \setminus \{\texttt{MFstWt}\} = \mathsf{NV}_2'' \setminus \{\texttt{MFstWt}\}$. By the inductive hypothesis, it follows that

$$\gamma_1'' \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2'' \mid \mathsf{V}_1'' \mid c_1'' \rightarrow^k \gamma_1' \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_1' \mid c_1'$$

where $\mathsf{NV}_1' \setminus \{\texttt{MFstWt}\} = \mathsf{NV}_2' \setminus \{\texttt{MFstWt}\}$. □

Definition C.3 (Subset Property). A judgment $\mathsf{aID}(c) \mid b \geq 0 : \texttt{nat} \mid \Omega;\ \Sigma \vdash_{\texttt{Sig}} c : \mathsf{C}_{\texttt{unit}} \dashv \Omega'$ has the subset property iff $\Omega^0_{\texttt{ck}} \subseteq \Sigma'$ where $\Sigma = \downarrow\Sigma'$ and $\Omega = \Omega^0_{\texttt{ck}}, \Omega^1$.

Theorem 6.3 (Fundamental Theorem). *If* $b : \texttt{nat} \mid \Omega \vdash p : \uparrow\mathsf{C}_{\texttt{unit}}$, *then* $b : \texttt{nat} \mid \Omega \Vdash p : \uparrow\mathsf{C}_{\texttt{unit}}$.

Proof. The proof is by induction on the static typing derivation for p and considering the last step in the derivation. We only show the aID case as it is the most interesting in this setting. The jit case is included in the extended TR [21].

Case 1. Suppose that $p = \mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c); p'$ such that T-P-Ckpt is the last step of the derivation.

$$\frac{\begin{array}{c}\Omega' \mid \Sigma = \mathsf{InitWorld}_t(\Omega; \rho; \mu; \omega) \quad \texttt{Sig} = \{\mathsf{aID}(c) \mid b \geq 0 : \texttt{nat} \mid \Omega'; \Sigma \vdash c : \mathsf{C}_{\texttt{unit}}\} \\ \mathsf{aID}(c) \mid b \geq 0 : \texttt{nat} \mid \Omega';\ \Sigma \vdash_{\texttt{Sig}} c : \mathsf{C}_{\texttt{unit}} \dashv \Omega'' \\ b : \texttt{nat} \mid \Omega \vdash p' : \uparrow\mathsf{C}_{\texttt{unit}} \quad \Omega'' \upharpoonright \{\texttt{MFstWt}\} = \emptyset\end{array}}{b : \texttt{nat} \mid \Omega \vdash \mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c); p' : \uparrow\mathsf{C}_{\texttt{unit}}}\ \text{(T-P-Ckpt)}$$

By inversion, we know that

(1) $\Omega' \mid \Sigma = \mathsf{InitWorld}_t(\Omega; \rho; \mu; \omega)$
(2) $\texttt{Sig} = \{\mathsf{aID}(c) \mid b \geq 0 : \texttt{nat} \mid \Omega'; \Sigma \vdash c : \mathsf{C}_{\texttt{unit}}\}$
(3) $\mathsf{aID}(c) \mid b \geq 0 : \texttt{nat} \mid \Omega';\ \Sigma \vdash_{\texttt{Sig}} c : \mathsf{C}_{\texttt{unit}} \dashv \Omega''$
(4) $b : \texttt{nat} \mid \Omega \vdash p' : \uparrow\mathsf{C}_{\texttt{unit}}$
(5) $\Omega'' \upharpoonright \{\texttt{MFstWt}\} = \emptyset$

By (1) and the definition of $\mathsf{InitWorld}_t$, we have that $\Omega' = \Omega'_1, \Sigma_{\mathtt{ck}}$. Observe that the inductive hypothesis asserts that $b : \mathtt{nat} \mid \Omega \vdash p' : \uparrow \mathtt{C}_{\mathtt{unit}}$ implies $b : \mathtt{nat} \mid \Omega \Vdash p' : \uparrow \mathtt{C}_{\mathtt{unit}}$. By applying the inductive hypothesis to (4), we learn that

$$b : \mathtt{nat} \mid \Omega \Vdash p' : \uparrow \mathtt{C}_{\mathtt{unit}}.$$

To complete the proof, we need to establish

$$\mathsf{aID}(c) \mid b \geq 0 : \mathtt{nat} \mid \Omega'; \Sigma \Vdash c \leq c : \mathtt{C}_{\mathtt{unit}}.$$

By definition of logical relation, this is equivalent to showing that c is related to itself in the term interpretation for arbitrary n_0, m_0, γ_0, NV_0, and V_0 where $\vdash_{\gamma_0}^{\mathsf{aID}(c)} \mathsf{NV}_0 \mid \mathsf{V}_0 : \Omega' \mid \Sigma$, and hence, $\mathsf{NV}_0 = \mathsf{NV}'_0, \mathsf{V}_{0\,\mathtt{ck}}$ and $\mathit{range}(\gamma_0) = \mathit{dom}(\mathsf{NV}_0)$. By assumption, p does not contain any worlds W, so it follows that c does not contain any worlds W. Therefore, it follows by Definition 4.1 that $\gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid n_0 \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c$ and $\gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c$ are well-formed.

Additionally, by (1), we know that $\Omega' = \Sigma$ and hence $\mathsf{aID}(c) \mid b \geq 0 : \mathtt{nat} \mid \Omega';\ \Sigma \vdash_{\mathtt{Sig}} c : \mathtt{C}_{\mathtt{unit}} \dashv \Omega''$ has the subset property (by Definition C.3). It then follows by $\vdash_{\gamma_0}^{\mathsf{aID}(c)} \mathsf{NV}_0 \mid \mathsf{V}_0 : \Omega' \mid \Sigma$ that $\mathsf{NV}_0^0 \subseteq \mathsf{V}_0$ where $\mathsf{NV}_0 = \mathsf{NV}_{0\,\mathtt{ck}}^0, \mathsf{NV}_0^1$.

We need to show that $\forall n_0$:

$$(\gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid n_0 \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c, \gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{m_0}$$

Observe that $\mathsf{NV}_0 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}_0 \setminus \{\mathtt{MFstWt}\}$ vacuously. Additionally, observe that $\mathsf{NV}_0 = \mathsf{NV}_0 \setminus \{\mathtt{Wtn}\}$ by the definition of $\mathsf{InitWorld}_t$ which states that Ω' contains no $\mathtt{Wtn}$ variables, and $\vdash_{\gamma_0}^{\mathsf{aID}(c)} \mathsf{NV}_0 \mid \mathsf{V}_0 : \Omega' \mid \Sigma$ establishes that NV_0 contains no $\mathtt{Wtn}$ variables due to well-formedness.

To this end, we instead show a generalized version holds. That is, $\forall n_0$:

$$(\gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c, \gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{m_0}$$

where $\mathsf{NV}_1 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}_2 \setminus \{\mathtt{MFstWt}\}$, $\mathit{range}(\gamma_0) = \mathsf{NV}_1$, and $\mathsf{NV}_1 = \mathsf{NV}_1 \setminus \{\mathtt{Wtn}\}$.

The proof proceeds by induction on m_0:

Base Case ($m_0 = 0$). When $m_0 = 0$, the proof is trivial and the desired result follows immediately by the value interpretation at type $\mathtt{C}_{\mathtt{unit}}$:

$$(\gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c, \gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{0}$$

Inductive Case ($m_0 = k + 1$ ($\exists k$)). If $m_0 = k + 1$, we need to show that

$$(\gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c, \gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{k+1}$$

such that

(i) $\exists.(\gamma_1 \mid \mathsf{aID}(\mathtt{c}) \mid n_1 \mid \mathsf{NV}'_1 \mid \mathsf{V}_1 \mid c_1)$ such that $\gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c \rightarrow^* \gamma_1 \mid \mathsf{aID}(\mathtt{c}) \mid n_1 \mid \mathsf{NV}'_1 \mid \mathsf{V}_1 \mid c_1$ AND

(ii) $\exists.(\gamma_2 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}'_2 \mid \mathsf{V}_2 \mid c_2)$ such that $\gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c \rightarrow^* \gamma_2 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}'_2 \mid \mathsf{V}_2 \mid c_2$ AND

(iii) $(\gamma_1 \mid \mathsf{aID}(\mathtt{c}) \mid n_1 \mid \mathsf{NV}'_1 \mid \mathsf{V}_1 \mid c_1, \gamma_2 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}'_2 \mid \mathsf{V}_2 \mid c_2) \in \mathcal{V}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{k+1}$.

By the progress and preservation for commands (Theorems 8.1 and 8.2) applied to (2) and (3), we know that the first configuration

$$\gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c$$

can take multiple steps until it becomes a value configuration that continues to be well-typed. Observe that in the mode $\mathsf{aID}(c)$, $\mathsf{NV}_1 \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\} = \mathsf{NV}'_1 \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\}$.

By application of Lemma C.1 to $\mathsf{aID}(c) \mid b \geq 0 : \mathtt{nat} \mid \Omega'; \Sigma \vdash_{\mathtt{Sig}} c : \mathtt{C}_{\mathtt{unit}} \dashv \Omega''$ and $\vdash_{\gamma_0}^{\mathsf{aID}(c)} \mathsf{NV}_1 \mid \mathsf{V}_0 : \Omega' \mid \Sigma$, observe that (i) holds. Additionally, $\mathsf{NV}_1 \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\} = \mathsf{NV}_1' \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\}$ and $dom(\mathsf{NV}_1) = dom(\mathsf{NV}_1')$.

The proof proceeds in two subcases, depending on the value of n_1':

Subcase $n_1' = 0$. To show (ii), we pick the post-step such that it holds vacuously, i.e., $\gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c \rightarrow^* \gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c$ in 0 steps. We then show (iii) for the post step:

$$(\gamma_1' \mid \mathsf{aID(c)} \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1', \gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c) \in \mathcal{V}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{k+1}$$

By the value interpretation at type $\mathtt{C}_{\mathtt{unit}}$, and because $n_1' = 0$, this is equivalent to showing

$$(\gamma_1' \mid \mathsf{aID(c)} \mid \cdot \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid \downarrow \varepsilon\#\mathsf{in}(n_1' > 0, \uparrow c_1'), \gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c) \in \mathcal{V}[\![\downarrow (\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}})]\!]^k$$

At this step we show the above relation holds by its definition:

(iv) $\mathtt{PwOff}(\gamma_1', \mathsf{aID}(c), \mathsf{NV}_1', \mathsf{V}_1') = \gamma_1'' \mid \emptyset$ AND

(v) $(\gamma_1'' \mid \mathsf{aID}(c) \mid \cdot \mid \mathsf{NV}_1' \mid \varepsilon\#\mathsf{in}(n_1' > 0, \uparrow c_1'), \gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c) \in \mathcal{V}[\![\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}}]\!]^k$

To show (vi), we need to show that $range(\gamma_1'') = dom(\mathsf{NV}_1')$ where γ_1'' is the largest restriction of γ_1' such that this condition holds. Observe that the desired result follows immediately by the assumptions $\gamma_0 \subseteq \gamma_1'$ and $range(\gamma_0) = dom(\mathsf{NV}_1)$, where $\gamma_1'' = \gamma_0$, and also $dom(\mathsf{NV}_1) = dom(\mathsf{NV}_1')$ (from above).

Hence, we need to show

$$(\gamma_0 \mid \mathsf{aID}(c) \mid \cdot \mid \mathsf{NV}_1' \mid \varepsilon\#\mathsf{in}(n_1' > 0, \uparrow c_1'), \gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c) \in \mathcal{V}[\![\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}}]\!]^k$$

By definition of the value relation at the type $\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}}$, this is equivalent to showing the following:

$$\forall n_1' > 0.(\gamma_0 \mid \mathsf{aID}(c) \mid n_1' \mid \mathsf{NV}_1' \mid \uparrow c_1', \gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c) \in \mathcal{V}[\![\uparrow \mathtt{C}_{\mathtt{unit}}]\!]^k$$

Fix an arbitrary n_1. We need to show that

$$(\gamma_0 \mid \mathsf{aID}(c) \mid n_1 \mid \mathsf{NV}_1' \mid \uparrow c_1', \gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c) \in \mathcal{V}[\![\uparrow \mathtt{C}_{\mathtt{unit}}]\!]^k$$

By the definition of value relation at the type $\uparrow \mathtt{C}_{\mathtt{unit}}$, this is equivalent to showing

(viii) $\mathtt{Restore}(\gamma_0 \mid \mathsf{aID}(c) \mid \mathsf{NV}_1' \mid c_1') = \mathsf{NV}_1' \mid \mathsf{V}_0' \mid c_0'$ (for some V_0', c_0') AND

(ix) $(\gamma_0 \mid \mathsf{aID}(c) \mid n_1 \mid \mathsf{NV}_1' \mid \mathsf{V}_0' \mid c_0', \gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^k$

To show (ix), note that by the definition of `Restore`, we have that $\mathtt{Restore}(\gamma_0 \mid \mathsf{aID}(c) \mid \mathsf{NV}_1' \mid c_1') = \mathsf{NV}^2_{\mathtt{RD,MFstWt}}, \mathsf{V}'_{0\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{MFstWt}} \mid \mathsf{V}_0' \mid c$ where $\mathsf{NV}_1' = \mathsf{NV}^2_{\mathtt{RD,MFstWt}}, \mathsf{V}'_{0\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{Wtn}}$. In particular, we need to show that

$$(\gamma_0 \mid \mathsf{aID}(c) \mid n_1 \mid \mathsf{NV}^2_{\mathtt{RD,MFstWt}}, \mathsf{V}'_{0\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{MFstWt}} \mid \mathsf{V}_0' \mid c, \gamma_0 \mid \mathsf{aID(c)} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^k.$$

Additionally, observe that the nonvolatile memory $\mathsf{NV}^2_{\mathtt{RD,MFstWt}}, \mathsf{V}'_{0\,\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{MFstWt}}$ contains no Wtns (i.e., $\mathsf{NV}^2_{\mathtt{RD,MFstWt}}, \mathsf{V}'_{0\,\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{MFstWt}} = \mathsf{NV}^2_{\mathtt{RD,MFstWt}}, \mathsf{V}'_{0\,\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{MFstWt}} \setminus \{\mathtt{Wtn}\}$). Therefore, it follows that

$$\begin{aligned}
&\mathsf{NV}^2_{\mathtt{RD,MFstWt}}, \mathsf{V}'_{0\,\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{MFstWt}} \setminus \{\mathtt{MFstWt}\} \\
&= \mathsf{NV}^2_{\mathtt{RD,MFstWt}}, \mathsf{V}'_{0\,\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{Wtn}} \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\} \\
&= \mathsf{NV}'_1 \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\} \\
&= \mathsf{NV}_1 \setminus \{\mathtt{MFstWt}, \mathtt{Wtn}\} \\
&= \mathsf{NV}_1 \setminus \{\mathtt{MFstWt}\} \\
&= \mathsf{NV}_2 \setminus \{\mathtt{MFstWt}\}
\end{aligned}$$

Each step holds by an established identity, and the second to last step holds by the assumption $\mathsf{NV}_1 = \mathsf{NV}_1 \setminus \{\mathtt{Wtn}\}$, and the last step holds by the assumption $\mathsf{NV}_1 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}_2 \setminus \{\mathtt{MFstWt}\}$. To set up the inductive step, note that $\mathsf{NV}^2_{\mathtt{RD,MFstWt}}, \mathsf{V}'_{0\,\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{MFstWt}}$ contains no Wtns. Additionally, observe that

$$\begin{aligned}
dom(\mathsf{NV}^2_{\mathtt{RD,MFstWt}}, \mathsf{V}'_{0\,\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{MFstWt}}) &= dom(\mathsf{NV}^2_{\mathtt{RD,MFstWt}}, \mathsf{V}'_{0\,\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{Wtn}}) \\
&= dom(\mathsf{NV}'_1) \\
&= dom(\mathsf{NV}_1) \\
&= range(\gamma_0)
\end{aligned}$$

The desired result follows directly by the inductive hypothesis. Propagating up the cascade, we learn that since n_1 was arbitrary, the value relation holds for all n_1. In summary, we have just shown that

$$(\gamma_0 \mid \mathsf{aID}(c) \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c, \gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{k+1}$$

where $\mathsf{NV}_1 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}_2 \setminus \{\mathtt{MFstWt}\}$, $range(\gamma_0) = \mathsf{NV}_1$, and $\mathsf{NV}_1 = \mathsf{NV}_1 \setminus \{\mathtt{Wtn}\}$.

Subcase $n'_1 > 0$. Since $n'_1 > 0$ and $\gamma'_1 \mid \mathsf{aID}(\mathtt{c}) \mid n'_1 \mid \mathsf{NV}'_1 \mid \mathsf{V}'_1 \mid c'_1$ is a value, observe that $c'_1 = \mathsf{skip}$. It follows by Lemma C.2 that

$$\gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c \rightarrow^* \gamma'_1 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}'_2 \mid \mathsf{V}'_1 \mid c'_1$$

where $\mathsf{NV}'_1 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}'_2 \setminus \{\mathtt{MFstWt}\}$.
Now we want to show that

$$(\gamma'_1 \mid \mathsf{aID}(\mathtt{c}) \mid n'_1 \mid \mathsf{NV}'_1 \mid \mathsf{V}'_1 \mid c'_1, \gamma'_1 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}'_2 \mid \mathsf{V}'_1 \mid c'_1) \in \mathcal{V}[\![\downarrow\uparrow \mathtt{C}_{\mathtt{unit}}]\!]^{k+1}$$

By the value interpretation at type $\mathtt{C}_{\mathtt{unit}}$ and $n'_1 > 0$, we need to show that

$$(\gamma'_1 \mid \mathsf{aID}(\mathtt{c}) \mid n'_1 \mid \mathsf{NV}'_1 \mid \mathsf{V}'_1 \mid c'_1, \gamma'_1 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}'_2 \mid \mathsf{V}'_1 \mid c'_1) \in \mathcal{V}[\![\downarrow\uparrow \mathtt{unit}]\!]^{k}$$

By definition of the value interpretation at the type $\downarrow\uparrow \mathtt{unit}$, this is equivalent to showing

(x) $\mathtt{Commit}(\gamma'_1, \mathsf{aID}(c) \mid \mathsf{NV}'_1 \mid \mathsf{V}'_1) = \gamma_1 \mid \mathsf{NV}''_{1\,\mathtt{ck}}, \mathsf{V}''_1$ AND
(xi) $\mathtt{Commit}(\gamma'_1, \mathsf{aID}(c) \mid \mathsf{NV}'_2 \mid \mathsf{V}'_1) = \gamma_2 \mid \mathsf{NV}''_{2\,\mathtt{ck}}, \mathsf{V}''_2$ AND
(xii) $(\gamma_1 \mid \mathsf{aID}(c) \mid n'_1 \mid \mathsf{NV}''_1, \mathsf{V}''_1 \mid \mathsf{skip}, \gamma_2 \mid \mathsf{aID}(c) \mid \infty \mid \mathsf{NV}''_2, \mathsf{V}''_2 \mid \mathsf{skip})$
$\in \mathcal{V}[\![\uparrow \mathtt{unit}]\!]^k$

By (x) and (xi), we observe that $\gamma_1 = \gamma_2$. Additionally, it follows by $\mathsf{NV}'_1 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}'_2 \setminus \{\mathtt{MFstWt}\}$ and the definition of `Commit` (which asserts that $\mathsf{NV}'_1 = \mathsf{NV}''_{1\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{NV}^3_{\mathtt{ck}}$, $\mathsf{NV}'_2 = \mathsf{NV}''_{2\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{NV}^4_{\mathtt{ck}}$, $dom(\mathsf{NV}^3_{\mathtt{ck}}) = dom(\mathsf{V}''_1)$, $dom(\mathsf{NV}^4_{\mathtt{ck}}) = dom(\mathsf{V}''_2)$) that

$$\begin{aligned}&\mathsf{NV}'_1 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}'_2 \setminus \{\mathtt{MFstWt}\}\\ &\Rightarrow \mathsf{NV}''_{1\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{NV}^3_{\mathtt{ck}} \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}''_{2\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{NV}^4_{\mathtt{ck}} \setminus \{\mathtt{MFstWt}\}\\ &\Rightarrow \mathsf{NV}''_{1\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{V}''_1 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}''_{2\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{V}''_2 \setminus \{\mathtt{MFstWt}\}\end{aligned}$$

To complete the proof, we need to show that $\mathsf{NV}''_1, \mathsf{V}''_1 = \mathsf{NV}''_2, \mathsf{V}''_2$. To this end, we show that $\mathsf{NV}''_{1\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{V}''_1 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}''_{1\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{V}''_1$ and $\mathsf{NV}''_{2\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{V}''_2 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}''_{2\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{V}''_2$, i.e., that the final memories have no `MFstWt`.
It follows by Lemma 8.2 (preservation for commands) applied to
– $\gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c \rightarrow^* \gamma'_1 \mid \mathsf{aID}(\mathtt{c}) \mid n'_1 \mid \mathsf{NV}'_1 \mid \mathsf{V}'_1 \mid c'_1$,
– $\mathsf{aID}(c) \mid b \geq 0 : \mathtt{nat} \mid \Omega';\ \Sigma \vdash_{\mathtt{Sig}} c : \mathtt{C}_{\mathtt{unit}} \dashv \Omega''$, and
– $\vdash^{\mathsf{aID}(c)}_{\gamma_0} \mathsf{NV}_1 \mid \mathsf{V}_0 : \Omega' \mid \Sigma$
that

$$\mathtt{Md} \mid b : nat \mid \Omega''; \Sigma^1 \vdash \mathsf{skip} : \downarrow\uparrow\mathtt{unit} \dashv \Omega''$$

where $\vdash \mathsf{NV}'_1 \mid \mathsf{V}'_1 : \Omega'' \mid \Sigma^1$. Then, it follows by $\mathsf{NV}'_1 = \mathsf{NV}''_1, \mathsf{NV}^3_{\mathtt{ck}}$, $\mathsf{NV}'_2 = \mathsf{NV}''_2, \mathsf{NV}^4_{\mathtt{ck}}$, $dom(\mathsf{NV}^3_{\mathtt{ck}}) = dom(\mathsf{V}''_1)$, $dom(\mathsf{NV}^4_{\mathtt{ck}}) = dom(\mathsf{V}''_2)$, and the unicity of typing that

$$\vdash \mathsf{NV}''_1, \mathsf{V}''_1 \mid \mathsf{V}'_1 : \Omega'' \mid \Sigma^1$$

Since $\Omega'' \upharpoonright \{\mathtt{MFstWt}\} = \emptyset$ (i.e., Ω'' has no `MFstWts`), it follows that

$$\mathsf{NV}''_{1\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{V}''_1 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}''_{1\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{V}''_1.$$

By similar reasoning, we can show that

$$\mathsf{NV}''_{2\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{V}''_2 \setminus \{\mathtt{MFstWt}\} = \mathsf{NV}''_{2\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{V}''_2.$$

Hence, $\mathsf{NV}''_{1\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{V}''_1 = \mathsf{NV}''_{2\,\mathtt{RD,Wtn,MFstWt}}, \mathsf{V}''_2$. Therefore, $\mathsf{NV}''_{1\,\mathtt{ck}}, \mathsf{V}''_1 = \mathsf{NV}''_{2\,\mathtt{ck}}, \mathsf{V}''_2$. Therefore, we can use the value interpretation at type $\uparrow \mathtt{unit}$ to prove (xii):

$$(\gamma_1 \mid \mathsf{aID}(c) \mid n'_1 \mid \mathsf{NV}''_{1\,\mathtt{ck}}, \mathsf{V}''_1 \mid \mathsf{skip}, \gamma_2 \mid \mathsf{aID}(c) \mid \infty \mid \mathsf{NV}''_{2\,\mathtt{ck}}, \mathsf{V}''_2 \mid \mathsf{skip}) \in \mathcal{V}[\![\uparrow \mathtt{unit}]\!]^k$$

which holds by definition of logical relation. This is the last piece we needed in order to prove the desired result:

$$(\gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid n_0 \mid \mathsf{NV}_1 \mid \mathsf{V}_0 \mid c, \gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_0 \mid c) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{k+1} \qquad \square$$

In general, we have that $(\gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid n_0 \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c, \gamma_0 \mid \mathsf{aID}(\mathtt{c}) \mid \infty \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{m_0}$ where $\vdash^{\mathsf{aID}(c)}_{\gamma_0} \mathsf{NV}_0 \mid \mathsf{V}_0 : \Omega'', \Sigma_{\mathtt{ck}} \mid \Sigma$. Since $n_0, m_0 \geq 0$, γ_0, NV_0, and V_0 were arbitrarily chosen, this result holds for all $n, m \geq 0, \gamma, \mathsf{NV}, \mathsf{V}$. Therefore, it follows by definition of logical relation that $\mathsf{aID}(c) \mid b \geq 0 : \mathtt{nat} \mid \Omega'; \Sigma \Vdash c \leq c : \mathtt{C}_{\mathtt{unit}}$. Finally, the desired result follows by application of P-Ckpt-semantic.

$$\frac{\Omega' \mid \Sigma = \mathsf{InitWorld}_t(\Omega; \rho; \mu; \omega) \qquad \mathsf{aID}(c) \mid b \geq 0 : \mathtt{nat} \mid \Omega'; \Sigma \Vdash c \leq c : \mathtt{C}_{\mathtt{unit}} \quad b : \mathtt{nat} \mid \Omega \Vdash p' : \uparrow \mathtt{C}_{\mathtt{unit}}}{b : \mathtt{nat} \mid \Omega \Vdash \mathsf{Ckpt}[\mathsf{aID}, \rho, \mu, \omega](c); p' : \uparrow \mathtt{C}_{\mathtt{unit}}} \text{ (P-Ckpt-semantic)}$$

D Adequacy

Definition 6.4 (Idempotency). A triple of a program p, nonvolatile memory NV, and a mapping γ is idempotent, if every intermittent execution of the program can be simulated by a continuous execution of it: For all $n, n', \chi_1, \chi_1', \mathsf{NV}', p'$, if $[\chi_1 \rhd \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid\ p \Rightarrow [\chi_1' \rhd \varepsilon] \otimes \gamma \mid n' \mid \mathsf{NV}' \mid p'$, then $[\chi_2 \rhd \varepsilon] \otimes \gamma \mid \infty \mid \mathsf{NV} \mid\ p \Rightarrow [\chi_2 \rhd \varepsilon] \otimes \gamma \mid \infty \mid \mathsf{NV}' \mid p'$.

Theorem 6.5 (Adequacy). *Consider $b : \mathtt{nat} \mid \Omega \Vdash p : \mathtt{C}_{\mathtt{unit}}$, a nonvolatile memory NV, and a map γ such that $\vdash^{\mathsf{jit}}_{\gamma} \mathsf{NV} \mid \cdot : \Omega \mid \cdot$. The triple of p, NV, and γ is idempotent.*

Proof. The proof is by cases according to the execution mode. We show only the aID case as it is the most interesting case. The jit case is included in the extended TR [21].

Stepping an Atomic Region. Consider a program of form $[\chi \rhd \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid \mathsf{Ckpt}[\mathsf{aID}; \rho; \mu; \omega]\ (c_0); p'$ that can take a step using the D-P-Seq rule to $[\chi'' \rhd \varepsilon] \otimes \gamma \mid n' \mid \mathsf{NV}_1 \mid p'$. By inversion on the D-P-Ckpt rule,

$$\frac{\begin{array}{c} n > 0 \quad \mathsf{InitWorld}_d(\mathsf{NV}; \rho; \mu; \omega; \gamma) = \mathsf{NV}_0, V_0 \\ [\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{aID}(c_0) \mid n \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0 \Rightarrow^* [\chi'' \rhd \varepsilon] \otimes \gamma' \mid \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid \mathsf{skip} \\ n' > 0 \quad \mathsf{NV}_1 = \mathsf{FinWorld}_d(\mathsf{NV}'; \mathsf{V}') \end{array}}{[\chi \rhd \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid \mathsf{Ckpt}[\mathsf{aID}; \rho; \mu; \omega](c_0); p' \Rightarrow_p\ [\chi'' \rhd \varepsilon] \otimes \gamma \mid n' \mid \mathsf{NV}_1 \mid p'} \text{(D-P-Ckpt)}$$

we learn that

- $n > 0$
- $\mathsf{InitWorld}_d(\mathsf{NV}; \rho; \mu; \omega; \gamma) = \mathsf{NV}_0, V_0$
- $[\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{aID}(c_0) \mid n \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0 \Rightarrow^* [\chi'' \rhd \varepsilon] \otimes \gamma' \mid \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid \mathsf{skip}$
- $n' > 0$
- $\mathsf{NV}_1 = \mathsf{FinWorld}_d(\mathsf{NV}'; \mathsf{V}')$

Our goal is to simulate this execution in a continuous setting. In particular, we need to find a continuous execution such that $[\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0 \Rightarrow^* [\chi \rhd \varepsilon] \otimes \gamma' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid \mathsf{skip}$, where $\mathsf{NV}_1 = \mathsf{FinWorld}_d(\mathsf{NV}_2'; \mathsf{V}_2')$. To this end, we invert the assumption $b : \mathtt{nat} \mid \Omega \Vdash \mathsf{Ckpt}[\mathsf{aID}; \rho; \mu; \omega](c_0); p' :\uparrow \mathtt{C}_{\mathtt{unit}}$ via P-Ckpt-semantic,

$$\frac{\begin{array}{c} \Omega' \mid \Sigma = \mathsf{InitWorld}_t(\Omega; \rho; \mu; \omega) \\ \mathsf{aID}(c_0) \mid b \geq 0 : \mathtt{nat} \mid \Omega'; \Sigma \Vdash c_0 \leq c_0 : \mathtt{C}_{\mathtt{unit}} \quad b\ : \mathtt{nat} \mid \Omega \Vdash p' : \uparrow \mathtt{C}_{\mathtt{unit}} \end{array}}{b : \mathtt{nat} \mid \Omega \Vdash \mathsf{Ckpt}[\mathsf{aID}; \rho; \mu; \omega](c_0); p' : \uparrow \mathtt{C}_{\mathtt{unit}}} \text{(P-Ckpt-semantic)}$$

we learn that

(i) $\Omega' \mid \Sigma = \mathsf{InitWorld}_t(\Omega; \rho; \mu; \omega)$
(ii) $\mathsf{aID}(c_0) \mid b \geq 0 : \mathtt{nat} \mid \Omega'; \Sigma \Vdash c_0 \leq c_0 : \mathtt{C}_{\mathtt{unit}}$
(iii) $b\ : \mathtt{nat} \mid \Omega \Vdash p' : \uparrow \mathtt{C}_{\mathtt{unit}}$

By definition of logical relation applied to (ii), we have that $\forall n_1, m_1 \geq 0. \forall \gamma, \mathsf{NV}, \mathsf{V}$ s.t. $\vdash^{\mathsf{aID}(c_0)}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$ and $(\gamma \mid \mathsf{aID}(c_0) \mid n_1 \mid \mathsf{NV} \mid \mathsf{V} \mid c_0, \gamma \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV} \mid \mathsf{V} \mid c_0) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{m_1}$.

By instantiating the memories accordingly, and the index m_1 with the number of tries $m + 1$ (where m is the number of crashes), we have

$$(\gamma \mid \mathsf{aID}(c_0) \mid n \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0, \gamma \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{m+1}$$

To get our result, we first prove the following generalized statement: if

– $[\chi \triangleright \varepsilon] \otimes \gamma_1 \mid \mathsf{aID}(c_0) \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1 \Rightarrow^* [\chi' \triangleright \varepsilon] \otimes \gamma' \mid \mathsf{aID}(c_0) \mid n_1' \mid \mathsf{NV}' \mid \mathsf{V}' \mid \mathsf{skip}$ in m crashes and

– $(\gamma_1 \mid \mathsf{aID}(c_0) \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1, \gamma_2 \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{m+1}$

then for all energy streams χ^0, we have $[\chi^0 \triangleright \varepsilon] \otimes \gamma_2 \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2 \Rightarrow^* [\chi^0 \triangleright \varepsilon] \otimes \gamma_2'' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2'' \mid \mathsf{V}_2'' \mid \mathsf{skip}$, where $\mathsf{FinWorld}_d(\mathsf{NV}'; \mathsf{V}') = \mathsf{FinWorld}_d(\mathsf{NV}_2''; \mathsf{V}_2'')$

The proof proceeds by induction on the number of crashes:

Base Case: $m = 0$ (**# *tries*** $= 1$). If $m = 0$, then it follows by the term interpretation at type $\mathtt{C}_{\mathtt{unit}}$,

(1) $\exists(\gamma_1'' \mid \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1')$ s.t. $\gamma_1 \mid \mathsf{aID}(c_0) \mid n \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1 \rightarrow^* \gamma_1'' \mid \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1'$ AND

(2) $\exists(\gamma_2'' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2')$ s.t. $\gamma_2 \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2 \rightarrow^* \gamma_2'' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2'$ AND

(3) $(\gamma_1'' \mid \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1', \gamma_2'' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2') \in \mathcal{V}[\![\mathtt{C}_{\mathtt{unit}}]\!]^1$

The number of crashes is 0, so $n' > 0$ and the first configuration steps to completion via D-STEP where $\mathsf{NV}_1' = \mathsf{NV}'$ and $\mathsf{V}_1' = \mathsf{V}'$ and $\gamma_1'' = \gamma'$:

$$[\chi \triangleright \varepsilon] \otimes \gamma_1 \mid \mathsf{aID}(c_0) \mid n \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1 \Rightarrow^* [\chi \triangleright \varepsilon] \otimes \gamma_1'' \mid \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid \mathsf{skip}$$

Applying D-STEP to (2), we know that

$$[\chi^0 \triangleright \varepsilon] \otimes \gamma_2 \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2 \Rightarrow^* [\chi^0 \triangleright \varepsilon] \otimes \gamma_2'' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2'$$

and by (3) and $n' > 0$, we get that the post steps are related by the value interpretation at type $\downarrow\uparrow \mathtt{unit}$. This means that $c_2' = \mathsf{skip}$ and we have

$$[\chi^0 \triangleright \varepsilon] \otimes \gamma_2 \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2 \Rightarrow^* [\chi^0 \triangleright \varepsilon] \otimes \gamma_2'' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid \mathsf{skip}$$

and

$$(\gamma_1'' \mid \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid \mathsf{skip}, \gamma_2'' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid \mathsf{skip}) \in \mathcal{V}[\![\downarrow\uparrow \mathtt{unit}]\!]^0$$

It then follows that

(1) $\mathtt{Commit}(\gamma_1'', \mathsf{aID}(c_0), \mathsf{NV}_1', \mathsf{V}_1') = \gamma' \mid \mathsf{NV}_{\mathtt{ck}}', \mathsf{V}'$ where $\gamma' \subseteq \gamma_1''$, $\mathsf{NV}_1' = \mathsf{NV}'_{\mathtt{RD,Wtn,MFstWt}}$, $\mathsf{NV}'_{0_{\mathtt{ck}}}, \mathsf{V}_1' = \mathsf{V}_0', \mathsf{V}'$, $dom(\mathsf{V}') = dom(\mathsf{NV}_0')$, $range(\gamma') = dom(\mathsf{NV}_1') \cup dom(\mathsf{V}')$.

(2) $\mathtt{Commit}(\gamma_2'', \mathsf{aID}(c_0), \mathsf{NV}_2', \mathsf{V}_2') = \gamma^2 \mid \mathsf{NV}_{\mathtt{ck}}^2, \mathsf{V}^2$ where $\gamma^2 \subseteq \gamma_2''$, $\mathsf{NV}_2' = \mathsf{NV}^2_{\mathtt{RD,Wtn,MFstWt}}$, $\mathsf{NV}^2_{0_{\mathtt{ck}}}, \mathsf{V}_2' = \mathsf{V}_0^2, \mathsf{V}^2$, $dom(\mathsf{V}^2) = dom(\mathsf{NV}_0^2)$, $range(\gamma^2) = dom(\mathsf{NV}_2') \cup dom(\mathsf{V}^2)$.

(3) $(\gamma' \mid \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}_{\mathtt{ck}}', \mathsf{V}' \mid \mathsf{skip}, \gamma^2 \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_{\mathtt{ck}}^2, \mathsf{V}^2 \mid \mathsf{skip}) \in \mathcal{V}[\![\uparrow \mathtt{unit}]\!]^0$.
It follows by the value interpretation at type $\uparrow \mathtt{unit}$ that $\mathsf{NV}', \mathsf{V}' = \mathsf{NV}^2, \mathsf{V}^2$. We observe that by definition, the `Commit` function copies the values of the checkpointed volatile memory locations into the nonvolatile memory and changes the qualifiers of all other locations in the nonvolatile memory to `ck`. This corresponds to the semantics of the $\mathsf{FinWorld}_d$ function, and hence it follows by definition that $\mathsf{FinWorld}_d(\mathsf{NV}_1'; \mathsf{V}_1') = \mathsf{NV}', \mathsf{V}'$ and $\mathsf{FinWorld}_d(\mathsf{NV}_2'; \mathsf{V}_2') = \mathsf{NV}^2, \mathsf{V}^2$.
Therefore, we have

$$[\chi^0 \triangleright \varepsilon] \otimes \gamma_2 \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2 \Rightarrow^* [\chi^0 \triangleright \varepsilon] \otimes \gamma_2'' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid \mathsf{skip}$$

where $\mathsf{FinWorld}_d(\mathsf{NV}_1'; \mathsf{V}_1') = \mathsf{FinWorld}_d(\mathsf{NV}_2'; \mathsf{V}_2')$, and the proof of this subcase is complete.
Inductive Case: $m = k + 1 (\exists k)$ (**# *tries*** $= k + 2$). By the term interpretation at type $\mathtt{C}_{\mathtt{unit}}$, we have

(i) $\exists \gamma_1' \mid \mathsf{aID}(c_0) \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1'$ s.t. $\gamma_1 \mid \mathsf{aID}(c_0) \mid n \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1 \rightarrow^* \gamma_1' \mid \mathsf{aID}(c_0) \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1'$

(ii) $\exists\, \gamma_2' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2'$ s.t. $\gamma_2 \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2 \rightarrow^* \gamma_2' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2'$

(iii) $(\gamma_1' \mid \mathsf{aID}(c_0) \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1', \gamma_2' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2') \in \mathcal{V}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{k+2}$

From (i), we step the first configuration until it becomes a value. It follows by the rule D-STEP that

$$[\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{aID}(c_0) \mid n \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1 \Rightarrow^* [\chi \rhd \varepsilon] \otimes \gamma_1' \mid \mathsf{aID}(c_0) \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1'$$

Since there are $m > 0$ crashes, we know that $n_1' = 0$. By (ii), we step the second configuration via D-STEP:

$$[\chi^0 \rhd \varepsilon] \otimes \gamma_2 \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2 \Rightarrow^* [\chi^0 \rhd \varepsilon] \otimes \gamma_2' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2'$$

and by (iii), we have

$$(\gamma_1' \mid \mathsf{aID}(c_0) \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1', \gamma_2' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2') \in \mathcal{V}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{k+2}$$

By application of D-CRASH, we have

$$\begin{aligned}&[\chi \rhd \varepsilon] \otimes \gamma_1' \mid \mathsf{aID}(c_0) \mid 0 \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1' \\ &\Rightarrow [\chi \rhd \varepsilon] \otimes \gamma_1' \mid \mathsf{aID}(c_0) \mid \cdot \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid\!\downarrow \varepsilon\#\mathsf{in}(b > 0; \uparrow c_1')\end{aligned}$$

By the value interpretation at type $\mathtt{C}_{\mathtt{unit}}$, observe that the stepped configuration continues to be related to the second configuration:

$$\begin{aligned}(\gamma_1' \mid \mathsf{aID}(c_0) \mid \cdot \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid\!\downarrow \varepsilon\#\mathsf{in}(b > 0; \uparrow c_1'), \gamma_2' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2') \\ \in \mathcal{V}[\![\downarrow (\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}})]\!]^{k+1}\end{aligned}$$

By D-S-AID, for $\gamma^1 \subseteq \gamma_1'$ such that $range(\gamma^1) = dom(\mathsf{NV}_1')$, we have

$$\begin{aligned}&[\chi \rhd \varepsilon] \otimes \gamma_1' \mid \mathsf{aID}(c_0) \mid \cdot \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid\!\downarrow \varepsilon\#\mathsf{in}(b > 0; \uparrow c_1') \\ &\Rightarrow [\chi \rhd \varepsilon] \otimes \gamma^1 \mid \mathsf{aID}(c_0) \mid \cdot \mid \mathsf{NV}_1' \mid \varepsilon\#\mathsf{in}(b > 0; \uparrow c_1')\end{aligned}$$

Note that the semantics of D-S-AID match the semantics of the `PwOff` function as it drops the volatile memory locations V_1' just as the `PwOff` function does not checkpoint any volatile memory locations, i.e., $\mathtt{PwOff}(\gamma_1', \mathsf{aID}(c_0), \mathsf{NV}_1', \mathsf{V}_1') = \gamma^1 \mid \emptyset$. By the value interpretation at type $\downarrow (\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}})$, and the definition of `PwOff` in the atomic case, we have $\mathtt{PwOff}(\gamma_1', \mathsf{aID}(c_0)),$
$\mathsf{NV}_1', \mathsf{V}_1') = \gamma^1 \mid \emptyset$, and thus the stepped configuration continues to be related to the second configuration:

$$\begin{aligned}(\gamma^1 \mid \mathsf{aID}(c_0) \mid \cdot \mid \mathsf{NV}_1' \mid \varepsilon\#\mathsf{in}(b > 0; \uparrow c_1'), \gamma_2' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2') \\ \in \mathcal{V}[\![\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}}]\!]^{k+1}\end{aligned}$$

By stepping the first configuration according to D-CHARGE, we have for some $n'' > 0$ such that $\chi = n'' :: \chi'$:

$$\begin{aligned}&[\chi \rhd \varepsilon] \otimes \gamma^1 \mid \mathsf{aID}(c_0) \mid \cdot \mid \mathsf{NV}_1' \mid \varepsilon\#\mathsf{in}(b > 0; \uparrow c_1') \\ &\Rightarrow [\chi' \rhd \varepsilon] \otimes \gamma^1 \mid \mathsf{aID}(c_0) \mid n'' \mid \mathsf{NV}_1' \mid\!\uparrow c_1'\end{aligned}$$

By the value interpretation at type $\mathtt{nat} \rightsquigarrow\!\uparrow \mathtt{C}_{\mathtt{unit}}$, observe that the stepped configuration remains related to the second configuration for $n'' > 0$:

$$(\gamma^1 \mid \mathsf{aID}(c_0) \mid n'' \mid \mathsf{NV}_1' \mid\!\uparrow c_1', \gamma_2' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2') \in \mathcal{V}[\![\uparrow \mathtt{C}_{\mathtt{unit}}]\!]^{k+1}$$

Stepping the first configuration via D-RESTORE-AID, we have

$$[\chi \rhd \varepsilon] \otimes \gamma^1 \mid \mathsf{aID}(c_0) \mid n'' \mid \mathsf{NV}_1' \mid \uparrow c_1' \Rightarrow [\chi \rhd \varepsilon] \otimes \gamma^1 \mid \mathsf{aID}(c_0) \mid n'' \mid \mathsf{NV}'_{\mathtt{RD,MFstWt}}, \mathsf{NV}''_{\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{MFstWt}} \mid \cdot \mid c_0$$

where $\mathsf{NV}_1' = \mathsf{NV}'_{\mathtt{RD,MFstWt}}, \mathsf{NV}''_{\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{Wtn}}$.

By the value interpretation at type $\uparrow \mathtt{C}_{\mathtt{unit}}$, the first and second configurations remain related:

$$(\gamma^1 \mid \mathsf{aID}(c_0) \mid n'' \mid \mathsf{NV}'_{\mathtt{RD,MFstWt}}, \mathsf{NV}''_{\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{MFstWt}} \mid \cdot \mid c_0, \gamma_2' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2') \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{k+1}$$

since

— $\mathtt{Restore}(\gamma^1, \mathsf{aID}(c_0), \mathsf{NV}_1', c_1') = \mathsf{NV}'_{\mathtt{RD,MFstWt}}, \mathsf{NV}''_{\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{MFstWt}} \mid \cdot \mid c_0$ and

— $\mathsf{NV}_1' = \mathsf{NV}'_{\mathtt{RD,MFstWt}}, \mathsf{NV}''_{\mathtt{ck}}, \mathsf{NV}^3_{\mathtt{Wtn}}$

By assumption,

$$[\chi \rhd \varepsilon] \otimes \gamma^1 \mid \mathsf{aID}(c_0) \mid n'' \mid \mathsf{NV}_1' \mid \cdot \mid c_0 \Rightarrow^* [\chi'' \rhd \varepsilon] \otimes \gamma' \mid \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid \mathsf{skip}$$

in k crashes.

By induction hypothesis, we get

$$[\chi^0 \rhd \varepsilon] \otimes \gamma_2' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2' \Rightarrow^* [\chi^0 \rhd \varepsilon] \otimes \gamma_2'' \mid \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}_2'' \mid \mathsf{V}_2'' \mid \mathsf{skip}$$

such that $\mathsf{FinWorld}_d(\mathsf{NV}'; \mathsf{V}') = \mathsf{FinWorld}_d(\mathsf{NV}_2''; \mathsf{V}_2'')$. This combined with

$$[\chi^0 \rhd \varepsilon] \otimes \gamma_2 \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2 \Rightarrow^* [\chi^0 \rhd \varepsilon] \otimes \gamma_2' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2'$$

gives us

$$[\chi^0 \rhd \varepsilon] \otimes \gamma_2 \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2 \Rightarrow^* [\chi^0 \rhd \varepsilon] \otimes \gamma_2'' \mid \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}_2'' \mid \mathsf{V}_2'' \mid \mathsf{skip}$$

which completes the proof of this subcase.

With that established, we can apply the generalized statement on assumptions

$$(\gamma \mid \mathsf{aID}(c_0) \mid n \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0, \gamma \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0) \in \mathcal{E}[\![\mathtt{C}_{\mathtt{unit}}]\!]^{m+1}$$

and $[\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{aID}(c_0) \mid n \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0 \Rightarrow^* [\chi' \rhd \varepsilon] \otimes \gamma' \mid \mathsf{aID}(c_0) \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid \mathsf{skip}$ to get $[\chi \rhd \varepsilon] \otimes \gamma \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0 \Rightarrow^* [\chi \rhd \varepsilon] \otimes \gamma'' \mid \mathsf{aID}(c_0) \mid \infty \mid \mathsf{NV}'' \mid \mathsf{V}'' \mid \mathsf{skip}$, where $\mathsf{FinWorld}_d(\mathsf{NV}'; \mathsf{V}') = \mathsf{FinWorld}_d(\mathsf{NV}''; \mathsf{V}'')$. and apply D-P-CKPT rule to complete the proof of this case. □

E Preservation for Closed Configurations

THEOREM E.1 (PRESERVATION FOR PROGRAMS). *Consider* $b : \mathtt{nat} \mid \Omega \vdash p : \uparrow \mathtt{C}_{\mathtt{unit}}$, *a nonvolatile memory* NV *and a bijective map* γ *that matches qualifiers and types from variables in* Ω *to locations in* NV. *For any* $n{:}\mathtt{nat} \geq 0$, *if we have* $[\chi \rhd \varepsilon] \otimes \gamma \mid n \mid \mathsf{NV} \mid p \Rightarrow_p [\chi' \rhd \varepsilon] \otimes \gamma' \mid n' \mid \mathsf{NV}' \mid p'$, *then* $b : \mathtt{nat} \mid \Omega \vdash p' : \uparrow \mathtt{C}_{\mathtt{unit}}$, *with* γ *remaining a bijective map from* Ω *to* NV'.

PROOF. By a structural induction on the typing derivation, and case distinction on the step. We include the full proof in the extended TR [21]. □

Received 9 April 2024; revised 21 September 2024; accepted 15 January 2025