Modal Crash Types for Intermittent Computing*

Farzaneh Derakhshan, Myra Dotzel, Milijana Surbatovich, and Limin Jia

Carnegie Mellon University, Pittsburgh PA, USA
{fderakhs, mdotzel, milijans, liminjia}@andrew.cmu.edu

Abstract. Intermittent computing is gaining traction in application domains such as Energy Harvesting Devices (EHDs) that experience arbitrary power failures during program execution. To make progress, programs require system support to checkpoint state and re-execute after power failure by restoring the last saved state. This 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. We define a Crash type system for a core calculus. We prove the correctness of intermittent systems by defining a novel logical relation for Crash types.

Keywords: intermittent computing · modal Crash type · logical relation

1 Introduction

Intermittent computing is gaining importance in application domains that require inaccessible or large-scale device deployments, such as wildlife monitoring [28], tiny satellites [22, 29], or smart civil infrastructure [1]. As battery maintenance may be infeasible in these environments, programs can instead run on batteryless Energy Harvesting Devices (EHDs). An EHD can run solely off energy harvested from its environment, at the cost of being powered intermittently. The device harvests energy (e.g., via solar panel) into a re-chargeable buffer. Once the energy buffer is full, the device turns on and begin to compute, consuming the stored energy. When the buffer drains, the device turns off at an arbitrary location until it can recharge and repeat this operational cycle. A power failure erases volatile execution state (e.g., the program counter), while nonvolatile state persists. For programs to make progress, they require *intermittent system* support to save state at checkpoints and restore the saved state after power failure, potentially causing re-execution from the last checkpoint.

* This work was generously funded in part through National Science Foundation (NSF) Award 2007998, NSF Graduate Research Fellowship Program grants DGE1745016 and DGE2140739, and the CMU CyLab Security & Privacy Institute. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsoring organizations.

As EHDs aim to enable long-term deployments with little or no maintenance, intermittent systems must execute programs reliably despite frequent power failures and partial executions. Initial systems [35, 43, 24] relied only on informal notions of correctness that left them susceptible to memory consistency bugs caused by reading the results of partial executions [23] or by allowing sensor reads from past executions to remain in the nonvolatile memory [39]. More recent work [41, 40, 9, 13] provides formal frameworks and correctness criteria for reasoning about intermittent execution. More concretely, all intermittent executions of a program must be simulated by some continuously-powered execution [41]. In other words, intermittent execution should be *idempotent*. Even if the system induces multiple partial executions of a program due to power failure, the program should not generate a different result than it would on a single execution.

The correctness of an intermittent execution relies on checkpointing, restoring, and finalizing state upon reaching the next checkpoint; mistakes in these operations can lead to incorrect, non-idempotent behavior. Few works have tried to understand the *fundamental logical underpinning* of these operations. This work fills this gap by formalizing checkpointing, crash, restoration, and re-execution as computation on *Crash types*. Crash types capture the core notion of intermittent computing: some values and computations persist across power failures and others do not. For instance, nonvolatile memory state persists across power failure and reboots, while volatile memory does not. Conversely, partially computed results do (or rather *should*) not persist across power failures, while completed/checkpointed computations do. We call the former *unstable* values and computations and the latter *stable* values and computations. Our key insight is that the interactions between these stable and unstable components bear close resemblance to shifts in adjoint logic [8, 36]. 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. Moreover, checkpoint and restore operations can turn values of one type to the other. We define terms and their associated types so that each of the key intermittent computing operations must be well-typed under our Crash types.

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 (the right disjunct). In the former case, the computation is suspended until energy arrives, after which it will again act as an intermittent computation. This recursive definition captures the multiple re-executions of a computation under repeated power failures. To prove the correctness of intermittent systems, we define a novel logical relation for Crash types, indexed by the number of power failures, which relates a continuously-powered execution to an intermittent execution. While intermittent computing motivates our results, the methods we develop are generally applicable to other system failures with the same effect on persistent and transient storage.

This paper makes the following technical 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.
– A novel logical relation to prove the correctness of intermittent executions.

Detailed proofs and definitions can be found in the extended TR [15].

2 Background

We provide background on intermittent computing and detail how checkpoint systems work to store and restore program state to handle power failures.

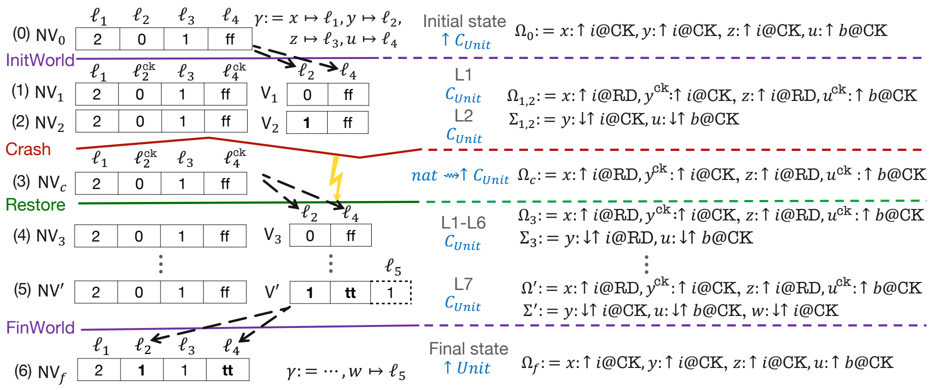
Intermittent Computing on EHDs. EHDs need intermittent system support to save necessary state before power failure and to restore it after reboot. When and where such checkpoints occur governs the *intermittent execution model* under which software executes. The two prevailing intermittent execution models are just-in-time (JIT) checkpoints [5, 4] and atomic execution [23, 24, 43, 37]. Under a JIT model, state is saved immediately prior to power failure so that execution resumes from the same point after reboot. Under an atomic execution model, state is saved at the beginning of an *atomic region*. If power fails before the end of the region, the system will reboot to the beginning of the region, re-executing until the region completes without power failure (akin to software transactions [38]). State-of-the-art intermittent systems use a hybrid "JIT + Atomics" model that defaults to JIT checkpoints except when there is an explicit atomic region [40, 25, 19]. Our core calculus follows this hybrid model.

To ensure idempotence, an intermittent system must save the value of volatile state and often a portion of the *nonvolatile* state. To illustrate why, consider an execution of the simple program in Fig. 1. The program has four variables stored in nonvolatile memory: x, y, and z of type `int` and u of type `bool`. It consists of two code blocks: an atomic region declared with the `Ckpt` construct (lines 1-7 on the left of Fig. 1) and a regular code block executed in JIT mode (lines 8-14 on the right). A continuous execution of the atomic region with initial state $x = 2, y = 0, z = 1, u = \mathsf{ff}$ ends in $x = 2, y = 1, z = 1, u = \mathsf{tt}$. Now, suppose power fails after the execution of Line 2. Once the device recharges, the program restarts from the start of the atomic region. If the system does not restore y's original value, this re-run computes an incorrect result: $x = 2, y = 2, z = 1, u = \mathsf{ff}$. Thus, to ensure idempotent execution, an intermittent system must checkpoint, i.e., save the value of, both volatile and nonvolatile memory. We next explain correct execution of the program in Fig 1 for atomic and JIT modes.

Atomic Region Execution. 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. In our example, x and z are read-only, so checkpointing y and u is enough to ensure correct intermittent execution. Many intermittent systems follow this design of checkpointing all variables that are not read-only [37, 19, 17,

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

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



(0) NV$_0$
InitWorld
(1) NV$_1$
(2) NV$_2$
Crash
(3) NV$_c$
Restore
(4) NV$_3$
(5) NV′
FinWorld
(6) NV$_f$
$\gamma := x \mapsto \ell_1, y \mapsto \ell_2, z \mapsto \ell_3, u \mapsto \ell_4$
$\gamma := \cdots, w \mapsto \ell_5$
Initial state
$\uparrow C_{Unit}$
L1
L2
$nat \rightsquigarrow \uparrow C_{Unit}$
L1-L6
L7
Final state
$\uparrow Unit$
$\Omega_0 := x{:}\uparrow i@\mathrm{CK}, y{:}\uparrow i@\mathrm{CK}, z{:}\uparrow i@\mathrm{CK}, u{:}\uparrow b@\mathrm{CK}$
$\Omega_{1,2} := x{:}\uparrow i@\mathrm{RD}, y^{\mathrm{ck}}{:}\uparrow i@\mathrm{CK}, z{:}\uparrow i@\mathrm{RD}, u^{\mathrm{ck}}{:}\uparrow b@\mathrm{CK}$
$\Sigma_{1,2} := y{:}\downarrow\uparrow i@\mathrm{CK}, u{:}\downarrow\uparrow b@\mathrm{CK}$
$\Omega_c := x{:}\uparrow i@\mathrm{RD}, y^{\mathrm{ck}}{:}\uparrow i@\mathrm{CK}, z{:}\uparrow i@\mathrm{RD}, u^{\mathrm{ck}}{:}\uparrow b@\mathrm{CK}$
$\Omega_3 := x{:}\uparrow i@\mathrm{RD}, y^{\mathrm{ck}}{:}\uparrow i@\mathrm{CK}, z{:}\uparrow i@\mathrm{RD}, u^{\mathrm{ck}}{:}\uparrow b@\mathrm{CK}$
$\Sigma_3 := y{:}\downarrow\uparrow i@\mathrm{RD}, u{:}\downarrow\uparrow b@\mathrm{CK}$
$\Omega' := x{:}\uparrow i@\mathrm{RD}, y^{\mathrm{ck}}{:}\uparrow i@\mathrm{CK}, z{:}\uparrow i@\mathrm{RD}, u^{\mathrm{ck}}{:}\uparrow b@\mathrm{CK}$
$\Sigma' := y{:}\downarrow\uparrow i@\mathrm{CK}, u{:}\downarrow\uparrow b@\mathrm{CK}, w{:}\downarrow\uparrow i@\mathrm{CK}$
$\Omega_f := x{:}\uparrow i@\mathrm{CK}, y{:}\uparrow i@\mathrm{CK}, z{:}\uparrow i@\mathrm{CK}, u{:}\uparrow b@\mathrm{CK}$

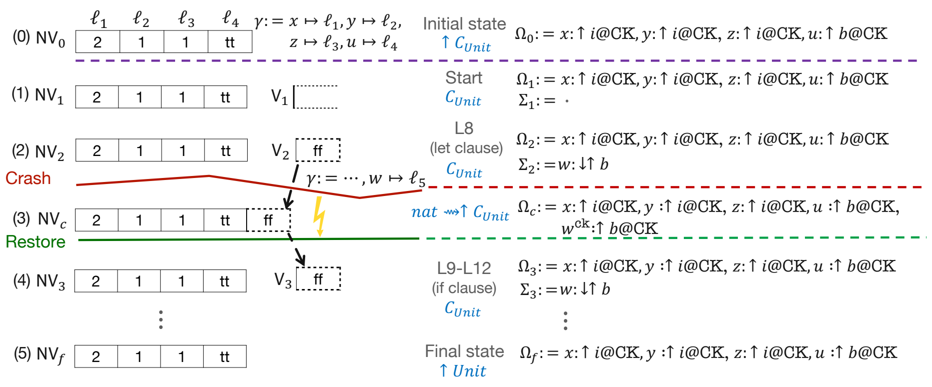
Fig. 2. Intermittent execution of an atomic region. We write i for `int` and b for `bool`.

26, 44, 12]. Given such a system, Fig. 2 shows an execution of the atomic region in Fig. 1. For now, ignore the last two columns about typing. To save and restore state, the system follows redo-log semantics. It records updates to checkpointed variables in a special volatile region, not main memory. This region 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 the atomic region (Row (1)), the system takes a snapshot of ℓ_2 and ℓ_4 and stores it in the volatile region V_1. We mark the original nonvolatile locations as checkpointed with the superscript `ck`. i.e., $\ell_2^{\mathtt{ck}}$ and $\ell_4^{\mathtt{ck}}$.Checkpointed locations $\ell_2^{\mathtt{ck}}$ and $\ell_4^{\mathtt{ck}}$ remain untouched for the remainder of the atomic region execution. Every access to variables y and u will instead be associated with their volatile copy ℓ_2 and ℓ_4, e.g., the assignment in Line 2 is applied to the volatile logs of Row (2).

On power failure, all volatile memory clears (Row (3)), throwing out the log. The system shuts down until more energy is harvested, at which point the system regenerates the volatile copies ℓ_2 and ℓ_4 (Row (4)) and resumes execution from Line 2. When the execution of the atomic region is complete (Row (5)), the system commits the updated values of the checkpointed locations (ℓ_2 and ℓ_4) from volatile memory to their original nonvolatile locations (Row (6)). During execution, local variables are stored to volatile memory via a `let` construct, e.g., location ℓ_5 for variable w on Line 3, corresponding to a volatile execution stack. On power failure, the device clears all volatile memory, but such stack allocated locations will be recreated upon re-execution.

JIT Region Execution. The JIT execution model prevents re-execution, so the intermittent system only saves and restores volatile state at checkpoints. Fig. 3 shows the details of executing the code on the right of Fig. 1 in JIT mode. Row (0) shows the initial nonvolatile locations, their values, and the mapping from variables to locations. The system starts 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 V_2 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 restoring the program, the system restores these copies to volatile memory and dismisses the nonvolatile backups (Row (4)). The program 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)).



(0) NV0
ℓ1 ℓ2 ℓ3 ℓ4
2 1 1 tt
γ := x ↦ ℓ1, y ↦ ℓ2, z ↦ ℓ3, u ↦ ℓ4
Initial state
↑ C_Unit
Ω0 := x:↑ i@CK, y:↑ i@CK, z:↑ i@CK, u:↑ b@CK
(1) NV1
V1
Start
C_Unit
Ω1 := x:↑ i@CK, y:↑ i@CK, z:↑ i@CK, u:↑ b@CK
Σ1 := ·
(2) NV2
V2
ff
L8
(let clause)
C_Unit
Ω2 := x:↑ i@CK, y:↑ i@CK, z:↑ i@CK, u:↑ b@CK
Σ2 := w: ↓↑ b
Crash
γ := ⋯, w ↦ ℓ5
(3) NVc
nat ⇝↑ C_Unit
Ωc := x:↑ i@CK, y:↑ i@CK, z:↑ i@CK, u:↑ b@CK, w^ck:↑ b@CK
Restore
(4) NV3
V3
L9-L12
(if clause)
C_Unit
Ω3 := x:↑ i@CK, y:↑ i@CK, z:↑ i@CK, u:↑ b@CK
Σ3 := w: ↓↑ b
(5) NVf
Final state
↑ Unit
Ωf := x:↑ i@CK, y:↑ i@CK, z:↑ i@CK, u:↑ b@CK

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

3 Key Ideas of Crash Types

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

3.1 Modal Store Types

An unstable value is an intermediate result of an execution towards a stable value and will be lost upon a power failure. However, if the result of a partial execution is committed to a nonvolatile location, it will persist and is thus stable. 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 $\uparrow_u^s \tau$ indicates that the location stores a stable value of type τ and $\downarrow_u^s \tau$ indicates that the location stores an intermediate result of an execution toward a value of type τ. To fully capture how intermittent execution interacts with a memory location, we also annotate the type of a memory location with an access qualifier, RD or CK, that represents whether the location is read-only or checkpointed by the system, respectively.

In our example in Fig. 2, the read-only variable x is stored in nonvolatile memory, so it has type $x :\uparrow_u^s \texttt{int@RD}$. The checkpointed variable y has type $y^{\mathsf{ck}} :\uparrow_u^s \texttt{int@CK}$ in the nonvolatile memory, while y's volatile copy has type $y :\downarrow_u^s\uparrow_u^s \texttt{int@CK}$. We use the context Ω to type nonvolatile memory and the context Σ to type volatile memory, as shown in the third columns of Figs. 2 and 3. We drop the superscript s and subscript u from the modalities for brevity.

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 is insufficient. One might expect the expression $x - y$ to have the type $\downarrow\uparrow\texttt{int}$ as it is a (partial) execution towards computing a stable 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 $\boxed{?} \vee \downarrow\uparrow\texttt{int}$, where $\boxed{?}$ is a type for computations that handle power failures. This type means that the expression either power fails, or completes its execution that evaluates to int. Next, we fill in $\boxed{?}$ for commands and expressions. $\boxed{?}$ is a recursive type since it handles re-execution.

Commands. The Crash type for commands is: $\mathtt{C}_{\mathtt{unit}} = \downarrow(\texttt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}}) \vee \downarrow\uparrow\texttt{unit}$. The right disjunct states that if no power failure occurs while executing a command, then it computes a stable value of type unit. The left disjunct states that on power failure, the computation continues as a function; after receiving a (logical) energy input from the environment, it becomes a computation that yields a stable value of a command type, i.e., $\mathtt{C}_{\mathtt{unit}}$. This computation will execute after the restore, which differs for atomic and JIT modes. In an atomic region, the system re-executes the region from the beginning, and in a JIT region, the system continues with the same command that was interrupted by the failure.

Expressions. The definition of the Crash type for expressions depends on the execution mode, just as the continuation of the program after a power failure depends on the mode. In an atomic region, the system restores an interrupted run of the expression to the original command enclosed in the region, so the type of an atomic mode expression is $\mathtt{C}_A^{\mathsf{atom}} = \downarrow(\texttt{nat} \rightsquigarrow \uparrow \mathtt{C}_{\mathtt{unit}}) \vee \downarrow\uparrow A$, where the left disjunct is the same as that of a command. On the other hand, an interrupted run of an expression in JIT mode will be restored to the expression itself. Hence, the type of a JIT mode expression is $\mathsf{C}^{\mathsf{jit}}_A = {\downarrow}(\mathtt{nat} \rightsquigarrow {\uparrow}\mathsf{C}^{\mathsf{jit}}_A) \vee {\downarrow}{\uparrow}A$, where the left disjunct states that after power failure and reception of the energy input, the computation again yields a stable value of a JIT mode expression type.

3.3 Independence Principle for Typing Intermittent Execution

We design our typing rules to follow the rules for $\downarrow$ and $\uparrow$ modalities in adjoint logic. 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 and unstable 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 sequent calculus rules in the underlying logic govern this back-and-forth behavior in our system. The rules are derivable from the more general rules in prior work [8, 34, 36]—in particular, the ${\uparrow}L^*$ rule can be derived from a cut rule and ${\downarrow}L$. Typical of 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 illustrate this matching using the execution steps in Figs. 2 and 3.

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

Shifts in Atomic Mode (Fig. 2): A combination of ${\uparrow}R$ and two ${\uparrow}L^*$ rules 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 last two columns in Row (0) correspond to the conclusion of a ${\uparrow}R$ rule: $\Omega_0 \vdash {\uparrow}\,\mathsf{C}_{\mathtt{unit}}$. An application of ${\uparrow}R$ from bottom to top drops the $\uparrow$ modality from the type of the program and opens an empty volatile region, i.e., $\Omega_0; \cdot \vdash \mathsf{C}_{\mathtt{unit}}$. Next, one application of ${\uparrow}L^*$, copies the variable y of type $\uparrow \mathtt{int}$ to the volatile memory with the type $\downarrow\uparrow \mathtt{int}$. Similarly, the next application of ${\uparrow}L^*$ copies the variable u of type $\uparrow$ $\mathtt{bool}$ to the volatile memory with the type $\downarrow\uparrow\mathtt{bool}$. The same combination corresponds to creating a volatile log from a nonvolatile location when restarting the atomic region, i.e., the step from Row (3) to Row (4), again copying variables y and u to the volatile memory.

The ${\downarrow}R$ rule corresponds to a power failure, which erases the volatile memory Σ. From Row (2) to Row (3) in Fig. 2, the system loses the volatile locations of y and u and closes off the volatile context. Row (2) corresponds to the conclusion of the rule, and Row (3) corresponds to its premise. The type of the command in Row (2) changes from $\mathsf{C}_{\mathtt{unit}}$ to ${\downarrow}(\mathtt{nat} \rightsquigarrow {\uparrow}\mathsf{C}_{\mathtt{unit}})$ (by another $\vee$-R rule as a crash is detected), and then to the type $(\mathtt{nat} \rightsquigarrow {\uparrow}\mathsf{C}_{\mathtt{unit}})$ in Row (3).

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, i.e., stepping from Row (5) to Row (6), the nonvolatile context drops the locations y and u of types

Command, expression, and memory

$$\begin{array}{llll|llll}
\textit{values} & v & ::= & \mathsf{n} \mid \mathsf{tt} \mid \mathsf{ff} \mid x & \textit{access qualifier} & q & ::= & \mathtt{CK} \mid \mathtt{RD} \\
\textit{exprs} & e & ::= & v \mid e \odot e & \textit{var loc map} & \gamma & ::= & \cdot \mid \gamma, x \mapsto \ell \\
\textit{cmds} & c & ::= & \mathsf{skip} \mid \mathsf{let}\, x = e\, \mathsf{in}\, c \mid c; c & \textit{nonvolatile mem} & \mathsf{NV} & ::= & \cdot \mid \ell \mathbin{@} q \hookrightarrow v, \mathsf{NV} \\
 & & \mid & \mathsf{if}\, e\, \mathsf{then}\, c\, \mathsf{else}\, c \mid x ::= e & & & \mid & \ell_{\mathtt{ck}} \mathbin{@} \mathtt{CK} \hookrightarrow v, \mathsf{NV} \\
\textit{progs} & p & ::= & \mathsf{Ckpt}[\mathsf{aID}, \rho](c); p \mid c; p \mid \mathsf{skip} & \textit{volatile mem} & \mathsf{V} & ::= & \cdot \mid l \mathbin{@} \mathtt{CK} \hookrightarrow v, \mathsf{V}
\end{array}$$

Instructions, statements, and configurations.

$$\begin{array}{llll|llll}
\textit{commands} & c & ::= & \cdots c;_W c & \textit{crash instrs} & i & ::= & \downarrow\varepsilon \text{ \# } \mathsf{in}(b > 0, \uparrow\kappa) \\
\textit{continuations} & \kappa & ::= & c \mid e & & & \mid & \varepsilon \text{ \# } \mathsf{in}(b > 0, \uparrow\kappa) \mid \uparrow \kappa \\
\textit{statements} & s & ::= & \kappa \mid i \mid p & \textit{open config} & K_o & ::= & (\gamma \mid \mathtt{Md} \mid g \mid \mathsf{NV} \mid \mathsf{V} \mid s) \\
\textit{energy level} & g & ::= & \cdot \mid n & & & \mid & (\gamma \mid \mathtt{Md} \mid g \mid \mathsf{NV} \mid s) \\
\textit{charge stream} & \chi & ::= & n :: \chi & \textit{closed config} & K_c & ::= & [\chi \rhd \varepsilon] \otimes K_o \\
\textit{exec. mode} & \mathtt{Md} & ::= & \mathsf{aID}(c) \mid \mathsf{jit} & & & &
\end{array}$$

Fig. 4. Summary of syntax

$\uparrow$`int` and $\uparrow$`bool`, respectively, by a weakening rule. These two variables map to the locations with outdated values. Next, the volatile locations of y and u in Σ', which contain the up-to-date values, commit their values to the nonvolatile context by a $\downarrow L$ rule. Then, a $\downarrow R$ rule closes off the remaining volatile context, which contains w of type $\downarrow\uparrow$`int`. The type of the command in Row (2) changes from $\mathtt{C_{unit}}$ to $\downarrow\uparrow$`unit` (by a separate $\vee$-R rule as the system detects a successful execution) and from that to type $\uparrow$`int` in Row (6).

Shifts in JIT Mode (Fig. 3): A $\uparrow R$ rule corresponds to creating an empty volatile context Σ_1 when starting the JIT region, i.e., the step from Row (0) to Row (1). A combination of the $\downarrow L$ rule and $\downarrow R$ rule corresponds to a power failure, i.e., the stepping from Row (2) to Row (3). A $\downarrow L$ rule copies the location w of type $\downarrow\uparrow$`bool` from volatile memory Σ_2 to nonvolatile memory Ω_c. A $\downarrow R$ rule closes off the (empty) nonvolatile memory. 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 clears a portion of volatile memory, and the $\uparrow L^*$ rule copies variable w from nonvolatile memory into volatile memory. We need an extra weakening rule to eliminate the remaining variable w in nonvolatile memory. The dropping of volatile memory at the end of execution (Row (5)) is not a modal step, but rather follows from a standard rule for the let clause.

4 A Basic Calculus for Intermittent Execution

We present the syntax, semantics, and the Crash type system for a basic calculus.

4.1 Syntax

The syntactic constructs are summarized in Fig. 4. Expressions include constants, variables, and binary operations while commands include assignments, mutable let bindings, sequencing, and if branching. A program consists of sequenced blocks of commands and atomic regions, denoted $\mathsf{Ckpt}[\mathsf{aID}, \rho](c)$ with a unique identifier aID, read-only variables ρ, and the enclosed command c.

Nonvolatile memory (NV) and volatile memory (V) map locations ℓ to values. Each location is annotated with its access mode q ($\mathtt{RD}$ or $\mathtt{CK}$). The nonvolatile memory location ℓ_{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 have constrained values (to be discussed in the semantics).

The runtime instruction $c_1;_W c_2$ is used for evaluating c_1 under the execution context W. 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. 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 $\mathtt{Md}$ (i.e., atomic or JIT), energy available for this execution g, 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 Ξ of natural numbers, which models available energy the configuration harvests from the environment at each power failure point for re-execution.

We call a configuration that cannot take a step a value configuration (value for short). An open configuration of form $(\cdots \mid g \mid \cdots \mid s)$ is a value, i.e., $\mathit{Val}(\cdots \mid g \mid \cdots \mid s)$, if either s is a constant or skip, it has depleted all energy for this execution ($g = 0$), or s is a crash instruction. The latter two cases are values because they cannot take a step without interacting with the environment or perform operations on the volatile and novolatile memory specific to handling power failures. A closed configuration is a value only if the statement s is skip with some energy left ($g > 0$). We list all values in the extended TR [15].

4.2 Operational Semantics

Top-level Program Execution. The top-level semantic rules for setting up and finalizing the atomic and JIT execution contexts are shown in Fig. 5. The P-Ckpt rule applies if the next code block is an atomic region. The nonvolatile NV_0 and volatile V_0 locations are initialized based on a given NV, declared read-only variables ρ, and their mapping γ to locations. The $\mathsf{InitWorld}_d$ function (a) changes the qualifier of locations in NV that are declared as read-only in ρ from $\mathtt{CK}$ to $\mathtt{RD}$, (b) creates V_0 by copying the rest of the locations of NV that still have qualifier $\mathtt{CK}$, and (c) marks the original version of the locations ℓ in NV that still have qualifier $\mathtt{CK}$ as checkpointed (ℓ_{ck}). This part corresponds to the step from Row (0) to Row (1) in Fig. 2. The closed configuration of c_0 is evaluated

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

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

Fig. 5. Closed configuration semantics for programs

until completion, using the rules in Fig. 6. This execution may undergo several power failures and corresponds to the steps from Row (1) to Row (5) in Fig. 2. 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', (b) removes `CK` from the locations in NV', i.e., converts $\ell_{\mathtt{ck}}$ to ℓ, and (c) replaces the `RD` qualifier of the locations in NV' with `CK`. This corresponds to the step from Row (5) to Row (6) in Fig. 2.

The 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. This corresponds to the steps from Row (0) to Row (1) and Row (1) to Row (5) in Fig. 3. Then the resulting volatile locations V' scoped in c are dropped, corresponding to the step from Row (5) to Row (6) in Fig. 3.

Command Execution (Closed Config). We summarize rules for a closed configuration in the top part of Fig. 6. Rule D-STEP steps the closed command configuration when the corresponding open configuration steps. Next, we explain the trio of power failure, charge, and restore rules. When the energy for this execution is depleted (i.e., $g = 0$), the D-CRASH rule applies, stepping the system to the crash instruction $\downarrow\!\varepsilon$ `#` $\mathsf{in}(b > 0; \uparrow\!\kappa)$. Next, D-S-JIT or D-S-aID rules apply and operate on volatile memory based on the execution mode `Md`. In JIT mode, D-S-JIT checkpoints and stores all volatile memory in nonvolatile locations. In atomic mode, D-S-aID drops all volatile memory locations. Then, D-CHARGE applies and inputs a natural number $n > 0$ from the energy channel, replenishing the configuration's energy level for re-execution. Finally, the program is restored via D-RESTORE-JIT and D-RESTORE-aID which copy checkpointed locations into volatile memory. D-RESTORE-JIT drops the checkpointed regions and steps to the interrupted command κ, while D-RESTORE-aID keeps the checkpointed regions and steps to the original command c_0 in the atomic region.

Command/Expression Execution (Open Config). The rules for executing commands and expressions in an open configuration are standard. We present a selection of them on the bottom of Fig. 6. 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.

Closed Configuration Semantics for Commands and Crash Instructions

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

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

$$\frac{\mathtt{Md} = \mathsf{jit}}{\begin{array}{l}[\chi \rhd \varepsilon] \otimes \gamma \,|\, \mathtt{Md} \,|\, \cdot \,|\, \mathsf{NV} \,|\, \mathsf{V} \,|\, \downarrow\varepsilon \,\#\, \mathsf{in}(b > 0; \uparrow\kappa) \\ \Rightarrow\ [\chi \rhd \varepsilon] \otimes \gamma \,|\, \mathtt{Md} \,|\, \mathsf{NV}, \mathsf{V}_{\mathsf{ck}} \,|\, \varepsilon \,\#\, \mathsf{in}(b > 0; \uparrow\kappa)\end{array}} \text{ (D-S-JIT)}$$

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

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

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

$$\frac{\mathsf{NV} = \mathsf{NV}', \mathsf{NV}''_{\mathsf{ck}}}{[\chi \rhd \varepsilon] \otimes \gamma \,|\, \mathsf{aID}(c_0) \,|\, n \,|\, \mathsf{NV} \,|\, \uparrow \kappa \Rightarrow [\chi \rhd \varepsilon] \otimes \gamma \,|\, \mathsf{aID}(c_0) \,|\, n \,|\, \mathsf{NV} \,|\, \mathsf{NV}'' \,|\, c_0} \text{ (D-RESTORE-AID)}$$

Selected expression and command semantics

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

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

Fig. 6. Statement steps

4.3 Types, Typing Contexts, and Judgments

This section introduces the typing judgments used in our static typing.

Types and Static Context. Our types are summarized below. The two modalities stratify types into the varieties stable (τ^s) and unstable (τ^u). The base store types `int` and `bool` are considered unstable. A type variable v_t denotes a type in the set $\{\mathtt{C}_{\mathtt{unit}}, \mathtt{C}^{\mathsf{atom}}_A, \mathtt{C}^{\mathsf{jit}}_A\}$, and implements the recursive nature of Crash types. 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 Sec. 5 based on the intended meaning of its index type. Some well-formed types, e.g., $\mathtt{nat} \rightsquigarrow \mathtt{nat} \rightsquigarrow \uparrow\mathtt{unit}$,

(J_u)	$\texttt{Md} \mid b\,\mathcal{R}\,0 : \texttt{nat} \mid \Omega; \Sigma \vdash c :: \mathtt{C}_{\texttt{unit}}$	c could crash
(J_u)	$\texttt{Md} \mid b : \texttt{nat} \mid \Omega; \Sigma \vdash \mathsf{skip} :: \downarrow\uparrow\texttt{unit}$	c will not crash
(J_s)	$\texttt{Md} \mid b : \texttt{nat} \mid \Omega \vdash \mathsf{skip} :: \uparrow\texttt{unit}$	after commit
(J_u)	$\texttt{Md} \mid b\,\mathcal{R}\,0 : \texttt{nat} \mid \Omega; \Sigma \vdash_{\texttt{RD}} e :: \mathtt{C}_A^{\texttt{Md}}$	e read, could crash
(J_s)	$\texttt{Md} \mid b : \texttt{nat} \mid \Omega; \Sigma \vdash_{\texttt{RD}} v :: \downarrow\uparrow A$	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
(J_s)	$\texttt{Md} \mid b : \texttt{nat} \mid \Omega \vdash p :: \uparrow\mathtt{C}_{\texttt{unit}}$	before execution
(J_u)	$\texttt{Md} \mid b = 0 : \texttt{nat} \mid \Omega; \Sigma \vdash \kappa :: \mathtt{C}_T^{\texttt{Md}}$	about to crash
(J_u)	$\texttt{Md} \mid \cdot \mid \Omega; \Sigma \vdash \downarrow\varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa) :: \downarrow(\texttt{nat} \rightsquigarrow \uparrow \mathtt{C}_T^{\texttt{Md}})$	crash state
(J_s)	$\texttt{Md} \mid \cdot \mid \Omega \vdash \varepsilon \,\#\, \mathsf{in}(b > 0, \uparrow\kappa) :: \texttt{nat} \rightsquigarrow \uparrow \mathtt{C}_T^{\texttt{Md}}$	waiting for energy
(J_s)	$\texttt{Md} \mid b > 0 : \texttt{nat} \mid \Omega \vdash \uparrow\kappa :: \uparrow\mathtt{C}_T^{\texttt{Md}}$	before re-execution

Table 1. Typing judgment summary

are not accepted by our type system introduced in Sec. 4.4. These types have no inhabitants, i.e., no well-typed configuration is of these types.

$$\begin{array}{llll}
\textit{store types } A := \texttt{int} \mid \texttt{bool} & \textit{stable types} & \tau^s := \texttt{nat} \rightsquigarrow \tau^s \mid\uparrow \tau^u \\
\textit{basic types } T := \texttt{unit} \mid A & \textit{unstable types} & \tau^u := T \mid\downarrow \tau^s \mid \tau^u \vee \tau^u \mid v_t \\
\textit{Volatile store typing context} & \Sigma := \cdot \mid x : \downarrow_u^s\uparrow_u^s A@Ck,\ \Sigma \\
\textit{Nonvolatile store typing context} & \Omega := \cdot \mid x : \uparrow_u^s A@Rd,\ \Omega \mid x_{\texttt{ck}} : \uparrow_u^s A@\texttt{CK}, \Omega \\
& \quad \mid x : \uparrow_u^s A@\texttt{CK}, \Omega
\end{array}$$

A nonvolatile store typing context Ω assigns stable types to nonvolatile location variables, i.e. all variables in Ω have a type of the form $\uparrow_u^s A$. A volatile store typing context Σ assigns unstable types to volatile location variables, i.e., variables in Σ are of the type $\downarrow_u^s\uparrow_u^s A$. $x_{\texttt{ck}}$ refers to a location that has been checkpointed. In the atomic mode, $x_{\texttt{ck}}$ has an active volatile log in Σ.

Typing Judgments. Table 1 summarizes all the typing judgments. These 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 ($\texttt{nat}$) and is constrained by a relation $\mathcal{R} \in \{\geq, >\}$ or is set to 0; where $b \geq 0$ is unconstrained. The constraint on b determines whether or not a command can evaluate a value without power failure. 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. When the command reaches type $\downarrow\uparrow\texttt{unit}$, b no longer needs to be constrained as the execution succeeded without power failure. The second judgment invokes the third judgment to type the configuration after the volatile log is committed: in the typing rule for committing the volatile log, the conclusion is of the form of the second judgment and the premise is of the form of the third. For expression typing, we distinguish expressions on the right of an assignment (being read) from those on the left of an assignment (being written to) via subscripts RD and

$$\frac{\mathsf{jit} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \cdot \vdash_{\emptyset} c : \mathsf{C}_{\mathtt{unit}} \quad b : \mathtt{nat} \mid \Omega \vdash p : \uparrow\mathsf{C}_{\mathtt{unit}}}{b : \mathtt{nat} \mid \Omega \vdash c; p : \uparrow\mathsf{C}_{\mathtt{unit}}} \text{(T-P-SEQ)}$$

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

Fig. 7. Program typing

`WT`, respectively. The expressions that are being written to are only of the simple form x. As no execution is required to evaluate x, we consider its judgment crash free, so no constraint is required on b. 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 uses the constraint $b = 0$, which corresponds to the power failure condition. It invokes the second judgment, which types a state right after 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.

4.4 Typing Rules

Program Typing. Fig. 7 shows the typing rules for programs. The P-SEQ rule types program $c; p$ by first typing c under jit mode, requiring $b \geq 0$, and then typing the rest of the program. The volatile memory context is empty for now, but will be populated when the let commands allocate new volatile locations.

The 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, as illustrated in Fig. 2. The partial function $\mathsf{InitWorld}_t$ initializes the volatile memory by creating a log of variables in Ω that are not read-only. Ω can be uniquely split into Ω^c and Ω^r, where Ω^r is the set of all read-only locations in Ω, and Ω^c is the set of all locations that are not read-only. This function is defined below:

$\Omega_0 \mid \Sigma_0 = \mathsf{InitWorld}_t(\Omega; \rho)$ iff $\rho \subseteq \mathsf{dom}(\Omega)$, $\Omega_0 = \Omega^r, \Omega^c_{\mathsf{ck}}$ and $\Sigma_0 = \downarrow\Omega^c$
where $\Omega = \Omega^c, \Omega^r$ and $\Omega^r = \Omega\lceil\rho$.

Here $\Omega^r = \Omega\lceil\rho$ is a subset of Ω where locations are declared in ρ to be read-only, and Ω^c are all other locations in Ω. The context Ω^c_{ck}, is defined as $\Omega^c_{\mathsf{ck}} = \{x_{\mathsf{ck}} : \uparrow A@q \mid x : \uparrow A@q \in \Omega^c\}$, and the context $\downarrow\Omega^c$, is defined as $\downarrow\Omega^c = \{x : \downarrow\uparrow A@q \mid x : \uparrow A@q \in \Omega^c\}$. If the set of read only variables, ρ, is not in the domain of Ω, then the function $\mathsf{InitWorld}_t$ is not defined.

In rules P-SEQ and P-CKPT, the command typing judgment in the premise makes use of a signature (subscripts $\emptyset$ and `Sig`, respectively) to type check the command relative to the signature. The signature is populated at different stages of type checking the JIT and atomic regions. In an atomic region, rule

Commands

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

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

$$\frac{\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathsf{skip} : \downarrow\uparrow\mathtt{unit}}{\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathsf{skip} : \tau \vee \downarrow\uparrow\mathtt{unit}} \;\text{(T-}\vee\text{-Succ)}$$

$$\frac{\begin{array}{c}\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e_1 : \mathtt{C}^{\mathtt{Md}}_{A} \\ \mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma, x{:}\downarrow\uparrow A@\mathtt{CK} \vdash_{\mathtt{Sig}} c : \tau\end{array}}{\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} \mathsf{let}\, x = e_1 \,\mathsf{in}\, c : \tau} \;\text{(T-Let)}$$

$$\frac{\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{RD;Sig}} e : \mathtt{C}^{\mathtt{Md}}_{A} \quad \mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{WT}} x : \downarrow\uparrow A}{\mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} x := e : \mathtt{C}^{\mathtt{Md}}_{\mathtt{unit}}} \;\text{(T-Assign)}$$

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

Expressions

$$\frac{\Omega, \Sigma' = x{:}\uparrow A@q, \Omega'_2 \quad q \neq \mathtt{RD}}{\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega, \Sigma' \vdash_{Wt} x : \uparrow A} \;\text{(T-Loc-Write)}$$

$$\frac{\Omega = x : \uparrow A@q, \Omega'}{\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathtt{RD}} x : \uparrow A} \;\text{(T-Loc-Read)} \qquad \frac{}{\mathtt{Md} \mid b : \mathtt{nat} \mid \Omega \vdash_{\mathtt{RD}} \mathsf{tt} :\uparrow \mathtt{bool}} \;\text{(T-Bool-t)}$$

Fig. 8. Selected command and expression typing

T-P-Ckpt populates the signature at the beginning of the region with the initial judgment which includes the region's original command c_0 and static memory context $\Omega_0; \Sigma_0$. The region is then typed relative to the signature. In JIT mode, the signature is populated later with the judgment just at the point of the failure (rule T-enough?). The program remembers that it built a typing derivation for the judgment in the signature such that when it restores from a power failure, it refers to the signature and checks that the restored judgment matches the one stored in the signature without needing to derive it again. This makes the typing derivations finitary and inductive.

Command and Expression Typing. Fig. 8 shows selected typing rules for commands. The T-skip rule declares the command skip as the stable type $\uparrow\mathtt{unit}$. Rule T-$\vee$-Succ applies when the command successfully completes its execution and still has one unit of energy available ($b > 0$) to conclude the execution. In this case, we close off the energy level variable and continue typing the command against the type $\downarrow\uparrow$ `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 Fig. 2.

The rules T-LET and T-ASSIGN, are mostly standard except that we consider crashes. For example, in typing the assign command $x := e$, the first premise of T-ASSIGN considers the type of expression e to be the Crash type $C_A^{\mathtt{Md}}$, but in the second premise we require the location x to be of type $\downarrow\uparrow A$, i.e., the location only considers the type corresponding to the case where execution of e can be completed successfully. The reason is that the assignment only occurs if the execution of e is successful. The constraint on the energy levels for premises goes back to $b \geq 0$, as we use one energy unit to deconstruct these commands.

The rule T-ENOUGH? checks two premises based on the value of $b \geq 0$. The third premise, a crash judgment, corresponds to the case where $b = 0$ (typing rules for crash judgments are given later in this section) and the fourth premise corresponds to the case where $b > 0$. The condition $b > 0$ states that there is at least one unit of energy available to decompose one command construct, e.g., via T-LET or T-ASSIGN. This rule populates the signature for JIT commands. The second premise states that the signature remains intact if the mode is atomic, but is populated by $\mathtt{Sig}'$ if the mode is JIT. In the JIT mode, after a power failure, the command c is restored to itself, and $\mathtt{Sig}'$ remembers that the well-typedness of the command when the energy level is non-negative has been checked already.

Expression typing rules are very similar to those of the commands. Fig. 8 shows a few selected rules. The T-LOC-WRITE and T-LOC-READ rules match the location variable x with an existing variable inside the context. T-LOC-WRITE performs an extra check to make sure that x is not a read-only variable.

Statement typing Fig. 9 presents the typing rules for crash instructions. The crash is detected by the depleted energy level $b = 0$ in the T-$\vee$-CRASH rule. In the premise, the crash instruction $\downarrow\varepsilon$ `#` $\mathsf{in}(b > 0, \uparrow\kappa')$ is typed. In JIT mode, the T-JIT-STOP rule brings a checkpointed version of all the volatile variables in Σ inside Ω since they are checkpointed then. In atomic mode, 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, regardless of the mode. The first premise shows that the energy channel is needed to provide a natural number greater than zero. Finally, the T-JIT-RESTORE and T-AID-RESTORE rules prepare and check rebooted system in JIT and atomic modes, respectively. In both modes, volatile memory is restored from the checkpointed locations in Ω. In the atomic mode, the checkpointed locations persist in Ω as we may need them for the next power failure. Alternatively, in the JIT mode, checkpoints are dropped from Ω and execution continues with the expression or command κ, which was running right before the crash. In the atomic mode, 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.

$$\frac{\mathtt{Md} \mid \cdot \mid \varOmega; \varSigma \vdash_{\mathtt{Sig}} {\downarrow}\varepsilon \ \texttt{\#}\ \mathsf{i\c{n}}(b > 0, {\uparrow}\kappa') : {\downarrow}(\mathtt{nat} \rightsquigarrow {\uparrow}\mathtt{C}^{\mathtt{Md}}_{T'})}{\mathtt{Md} \mid b = 0 : \mathtt{nat} \mid \varOmega; \varSigma \vdash_{\mathtt{Sig}} \kappa' : {\downarrow}(\mathtt{nat} \rightsquigarrow {\uparrow}\mathtt{C}^{\mathtt{Md}}_{T'}) \vee {\downarrow}{\uparrow}T} \ (\textsc{T-}\vee\textsc{-Crash})$$

$$\frac{\varSigma = {\downarrow}{\uparrow}\varSigma' \quad \mathsf{jit} \mid \cdot \mid \varOmega, {\uparrow}\varSigma'_{\mathtt{ck}} \vdash_{\mathtt{Sig}} \varepsilon \ \texttt{\#}\ \mathsf{i\c{n}}(b > 0, {\uparrow}\kappa') : (\mathtt{nat} \rightsquigarrow {\uparrow}\mathtt{C}^{s}_{T})}{\mathsf{jit} \mid \cdot \mid \varOmega; \varSigma \vdash_{\mathtt{Sig}} {\downarrow}\varepsilon \ \texttt{\#}\ \mathsf{i\c{n}}(b > 0, {\uparrow}\kappa') : {\downarrow}(\mathtt{nat} \rightsquigarrow {\uparrow}\mathtt{C}^{s}_{T})} \ (\textsc{T-Jit-stop})$$

$$\frac{\mathsf{aID}(c_0) \mid \cdot \mid \varOmega \vdash_{\mathtt{Sig}} \varepsilon \ \texttt{\#}\ \mathsf{i\c{n}}(b > 0, {\uparrow}\kappa') : (\mathtt{nat} \rightsquigarrow {\uparrow}\mathtt{C}^{s}_{\mathtt{unit}})}{\mathsf{aID}(c_0) \mid \cdot \mid \varOmega; \varSigma \vdash_{\mathtt{Sig}} {\downarrow}\varepsilon \ \texttt{\#}\ \mathsf{i\c{n}}(b > 0, {\uparrow}\kappa') : {\downarrow}(\mathtt{nat} \rightsquigarrow {\uparrow}\mathtt{C}^{s}_{\mathtt{unit}})} \ (\textsc{T-aID-stop})$$

$$\frac{\varepsilon \ \texttt{\#}\ \mathsf{i\c{n}}() : \mathtt{nat} > 0 \quad \mathtt{Md} \mid b > 0 : \mathtt{nat} \mid \varOmega \vdash_{\mathtt{Sig}} {\uparrow}\kappa' : {\uparrow}\mathtt{C}^{s}_{T}}{\mathtt{Md} \mid \cdot \mid \varOmega \vdash_{\mathtt{Sig}} \varepsilon \ \texttt{\#}\ \mathsf{i\c{n}}(b > 0, {\uparrow}\kappa') : (\mathtt{nat} \rightsquigarrow {\uparrow}\mathtt{C}^{s}_{T}))} \ (\textsc{T-Charge})$$

$$\frac{\varOmega = \varOmega', \varOmega''_{\mathtt{ck}} \quad \mathsf{jit} \mid b \geq 0 : \mathtt{nat} \mid \varOmega'; {\downarrow}\varOmega'' \vdash \kappa' : \mathtt{C}_T \in \mathtt{Sig}}{\mathsf{jit} \mid b > 0 : \mathtt{nat} \mid \varOmega \vdash_{\mathtt{Sig}} {\uparrow}\kappa' : {\uparrow}\mathtt{C}_T} \ (\textsc{T-Jit-Restore})$$

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

Fig. 9. Crash, restore, and checkpoint typing

5 Logical Relation for Intermittent Execution

We establish a logical relation to prove idempotency, which states that every intermittent execution of a program can be simulated by a continuous execution. The logical relation relates an intermittent execution with a continuous one and is indexed by Crash types. A continuous run is one with an infinite energy level, ∞. Crash types are recursive, yielding possible infinite atomic region re-executions. Thus, we use the maximum number of executions (also power failures) as a step index to stratify our logical relation to ensure its well-foundedness.

The logical relation (defined in Sec. 5.1) relies on `PwOff`, `Restore`, and `Commit` functions, referred to as power failure, restore, and commit policies, respectively. We establish specific policies for atomic and JIT execution modes. We formalize *semantic typing* as every atomic and JIT region of the program being logically-related to themselves. We prove that the semantically well-typed programs are idempotent across power failures in Sec. 5.2. The definitions match the memory operations in the dynamic rules that deal with crash, restore, and re-execution (D-S-aID/ D-S-Jit, D-R-aID/ D-R-Jit, and D-P-Ckpt/ D-P-seq) for atomic and JIT regions, We prove that our syntactically well-typed programs are semantically well-typed. We generalize semantic typing rules, allowing custom power failure, restore, and commit policies (Sec. 5.3).

5.1 Semantic Typing via a Logical Relation

The logical relation, written $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \varOmega \mid \varSigma \Vdash c_1 \leq c_2 : \mathtt{C}_{\mathtt{unit}}$, is defined in Fig. 10 by a lexicographic induction on the index m and the structure of the

$$\begin{aligned}
&\texttt{Md} \mid b \geq 0 : \texttt{nat} \mid \Omega \mid \Sigma \Vdash c_1 \leq c_2 : \texttt{C}_{\texttt{unit}} \\
&\text{iff } \forall n, m \geq 0.\ \forall \gamma, \mathsf{NV}, \mathsf{V}. s.t.\ \mathsf{NV} \mid \mathsf{V} \Vdash \gamma :: \Omega \mid \Sigma. \\
&\qquad\qquad (\gamma \mid \texttt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c_1, \gamma \mid \texttt{Md} \mid \infty \mid \mathsf{NV} \mid \mathsf{V} \mid c_2) \in \mathcal{E}\llbracket \texttt{C}_{\texttt{unit}} \rrbracket^m
\end{aligned}$$

Term Relation

$$\begin{aligned}
\mathcal{E}\llbracket \texttt{C}_{\texttt{unit}} \rrbracket^{m+1} = \{&(\gamma_1 \mid \texttt{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1, \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2)\ s.t. \\
&\exists.(\gamma_1' \mid \texttt{Md}' \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1')\ s.t. \\
&\gamma_1 \mid \texttt{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1 \to^*_{irred} \gamma_1' \mid \texttt{Md}' \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1' \wedge \\
&\exists.(\gamma_2' \mid \texttt{Md}' \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2')\ s.t. \\
&\quad \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2 \to^* \gamma_2' \mid \texttt{Md}' \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2' \wedge \\
&(\gamma_1' \mid \texttt{Md}' \mid n_1' \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1', \gamma_2' \mid \texttt{Md}' \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2') \in \mathcal{V}\llbracket \texttt{C}_{\texttt{unit}} \rrbracket^{m+1}\} \\
\mathcal{E}\llbracket \texttt{C}_{\texttt{unit}} \rrbracket^{0} \quad = \{&(\gamma_1 \mid \texttt{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1, \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2)\}
\end{aligned}$$

Value Relation

$$\begin{aligned}
\mathcal{V}\llbracket \uparrow\texttt{unit} \rrbracket^m &= \{(\gamma \mid \texttt{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{skip}, \gamma \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{skip})\ \texttt{s.t.}\mathsf{NV}_1 = \mathsf{NV}_2\} \\
\mathcal{V}\llbracket \downarrow\uparrow\texttt{unit} \rrbracket^m &= \{(\gamma_1 \mid \texttt{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid \mathsf{skip}, \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid \mathsf{skip})\ \texttt{s.t.} \\
&\quad \texttt{Commit}(\gamma_i \mid \texttt{Md} \mid \mathsf{NV}_i \mid \mathsf{V}_i) = \gamma_1' \mid \mathsf{NV}_i' \wedge \\
&\quad (\gamma_1' \mid \texttt{Md} \mid n_1 \mid \mathsf{NV}_1' \mid \mathsf{skip}, \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{skip}) \in \mathcal{V}\llbracket \uparrow\texttt{unit} \rrbracket^m\} \\
\mathcal{V}\llbracket \uparrow\texttt{C}_{\texttt{unit}} \rrbracket^m &= \{(\gamma_1 \mid \texttt{Md} \mid n \mid \mathsf{NV}_1 \mid \uparrow\kappa, \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2)\ \texttt{s.t.} \\
&\quad \texttt{restore}(\gamma_1, \texttt{Md}, \mathsf{NV}_1, \kappa) = \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0 \wedge \\
&\quad (\gamma_1 \mid \texttt{Md} \mid n \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0, \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2) \in \mathcal{E}\llbracket \texttt{C}_{\texttt{unit}} \rrbracket^m\} \\
\mathcal{V}\llbracket \texttt{nat} \rightsquigarrow \uparrow\texttt{C}_{\texttt{unit}} \rrbracket^m &= \{(\gamma_1 \mid \texttt{Md} \mid \cdot \mid \mathsf{NV}_1 \mid \varepsilon\ \#\ \mathsf{in}(n > 0, \uparrow\kappa), \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2)\ \texttt{s.t.} \\
&\quad \forall n > 0.(\gamma_1 \mid \texttt{Md} \mid n \mid \mathsf{NV}_1 \mid \uparrow\kappa, \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2) \in \mathcal{V}\llbracket \uparrow \texttt{C}_{\texttt{unit}} \rrbracket^m\} \\
\mathcal{V}\llbracket \downarrow(\texttt{nat} \rightsquigarrow \uparrow\texttt{C}_{\texttt{unit}}) \rrbracket^m &= \{(\gamma_1 \mid \texttt{Md} \mid \cdot \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid \downarrow\varepsilon\ \#\ \mathsf{in}(n > 0, \uparrow\kappa), \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2) \\
&\quad \texttt{s.t. PwOff}(\gamma_1, \texttt{Md}, \mathsf{NV}_1, \mathsf{V}_1) = \gamma_1' \mid \mathsf{V}' \wedge \\
&\quad (\gamma_1' \mid \texttt{Md} \mid \cdot \mid \mathsf{V}', \mathsf{NV}_1 \mid \varepsilon\ \#\ \mathsf{in}(n > 0, \uparrow\kappa), \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2) \\
&\qquad\qquad\qquad \in \mathcal{V}\llbracket \texttt{nat} \rightsquigarrow \uparrow\texttt{C}_{\texttt{unit}} \rrbracket^m\} \\
\mathcal{V}\llbracket \texttt{C}_{\texttt{unit}} \rrbracket^{m+1} &= \{(\gamma_1 \mid \texttt{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1,\ \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2) \\
&\quad \texttt{s.t. either} \\
&\quad n_1 = 0 \wedge (\gamma_1 \mid \texttt{Md} \mid \cdot \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid \downarrow\varepsilon\ \#\ \mathsf{in}(n_1 > 0, \uparrow c_1), \\
&\qquad\qquad\qquad \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2) \in \mathcal{V}\llbracket \downarrow(\texttt{nat} \rightsquigarrow \uparrow \texttt{C}_{\texttt{unit}}) \rrbracket^m, \texttt{or} \\
&\quad n_1 > 0 \wedge (\gamma_1 \mid \texttt{Md} \mid n_1 \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1, \gamma_2 \mid \texttt{Md} \mid \infty \mid \mathsf{NV}_2 \mid \mathsf{V}_2 \mid c_2) \\
&\qquad\qquad\qquad \in \mathcal{V}\llbracket \downarrow\uparrow \texttt{unit} \rrbracket^m\}
\end{aligned}$$

Fig. 10. Logical relation

types. The judgment $\mathsf{NV} \mid \mathsf{V} \Vdash \gamma :: \Omega \mid \Sigma$ in the definition states that γ maps the variables in Σ and Ω to locations in V and NV resp., such that their qualifiers and types match. Similar to prior work [2, 16, 42], our definition consists of a term relation $\mathcal{E}\llbracket \texttt{C}_{\texttt{unit}} \rrbracket^m$ and a value relation $\mathcal{V}\llbracket \tau \rrbracket^m$.

Term Relation. A pair of open command configurations of type $\texttt{C}_{\texttt{unit}}$ are in the term relation of index m if any intermittent execution of the first one after m power failures is indistinguishable from a continuous execution of the second one. In particular, for index $m+1$, the term relation relates two configurations at type $\mathtt{C_{unit}}$ if the first configuration eventually steps to a value (or "irreducible") configuration, i.e., it either evaluates to skip or its energy level depletes ($n'_1 = 0$), and the second configuration can take zero or more steps such that the pair continue to be in the value relation of $\mathcal{V}[\![\mathtt{C_{unit}}]\!]^{m+1}$. When the index is $m = 0$, no execution is observed, so any two configurations are in the term relation. Here, *irred* refers to $\gamma'_1 \,|\, \mathtt{Md}' \,|\, n'_1 \,|\, \mathsf{NV}'_1 \,|\, \mathsf{V}'_1 \,|\, c'_1$ being an irreducible configuration, i.e. it cannot take any more steps. Since our semantics for commands is derministic, for each configuration $\gamma_1 \,|\, \mathtt{Md} \,|\, n_1 \,|\, \mathsf{NV}_1 \,|\, \mathsf{V}_1 \,|\, c_1$ there is exactly one such irreducible configuration.

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 value relation relates two open command configurations at type $\mathtt{C_{unit}}$ and index $m+1$ if either (a) the first configuration has faced a power failure, and the two configurations continue to relate by $\mathcal{V}[\![\downarrow(\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C_{unit}})]\!]^m$, or (b) the first configuration executed successfully without any power failures, and the two configurations are related by $\mathcal{V}[\![\downarrow\uparrow\mathtt{unit}]\!]^m$. This definition matches the disjunctive nature of type $\mathtt{C_{unit}}$, which is recursively defined in the signature as $\downarrow(\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C_{unit}}) \vee \downarrow\uparrow\mathtt{unit}$. Since we unfold the recursive definition of $\mathtt{C_{unit}}$, 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 called for $\mathtt{C_{unit}}$ at index 0.

The value relations in the third, fourth, and fifth rows of Fig. 10 are defined based on the type of the *first configuration*; the second configurations in these relations continue to be of type $\mathtt{C_{unit}}$. Only in the relations defined in the first and second rows of Fig. 10 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 Fig. 10, the relation is *binary* while in the rest, the relation degenerates to *unary*, with the second configuration as its Kripke world [18].

The value relation at type $\downarrow(\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C_{unit}})$ relates two configurations if the first one runs the crash instruction $\downarrow\varepsilon \;\#\; \mathsf{in}(n > 0, \uparrow\kappa)$ and a power failure policy creates a checkpoint of volatile locations such that the configurations continue to be in the value relation at type $(\mathtt{nat} \rightsquigarrow \uparrow\mathtt{C_{unit}})$. The power failure function in an atomic mode is defined to checkpoint none of the volatile locations, 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 $\mathit{range}(\gamma') = \mathsf{dom}(\mathsf{NV}_1)$, and defined to checkpoint all volatile locations in JIT mode, i.e., $\mathtt{PwOff}(\gamma, \mathsf{jit}, \mathsf{NV}_1, \mathsf{V}_1) = \gamma \mid \mathsf{V}_1$.

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

The value relation at type $\uparrow\mathtt{C_{unit}}$ requires the first configuration to run the crash instruction $\uparrow\kappa$. The defined restore policy restores the nonvolatile memory NV_0, volatile memory V_0, and re-execution command c_0 such that the configurations continue to be related in the term interpretation at type $\mathtt{C_{unit}}$. In an atomic mode, the restore function is defined as $\mathtt{restore}(\gamma, \mathsf{aID}(c), \mathsf{NV}_1, \kappa) = \mathsf{NV}_1 \mid \mathsf{NV}'' \mid c$ where $\mathsf{NV}_1 = \mathsf{NV}', \mathsf{NV}''_{\mathsf{ck}}$. In the JIT mode, the restore function is defined as $\mathtt{restore}(\gamma, \mathsf{jit}(c), \mathsf{NV}_1, \kappa) = \mathsf{NV}'_1 \mid \mathsf{NV}'' \mid c$ where $\mathsf{NV}_1 = \mathsf{NV}', \mathsf{NV}''_{\mathsf{ck}}$.

We write $\mathsf{NV}_1 = \mathsf{NV}', \mathsf{NV}''_{\mathtt{ck}}$ to state that NV_1 can be uniquely partitioned into all locations ($\mathsf{NV}''_{\mathtt{ck}}$) that are checkpointed, i.e., 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 $\mathsf{NV}''_{\mathtt{ck}}$.

The value relation at type $\downarrow\uparrow\mathtt{unit}$ requires both configurations to run skip, and the defined commit policy creates nonvolatile memories for both runs such that they continue to be related at type $\uparrow\mathtt{unit}$. In an atomic mode, the commit function is defined to replace the checkpointed locations in the nonvolatile memory with their volatile log, i.e., $\mathtt{Commit}(\gamma \mid \mathsf{aID}(c_0) \mid \mathsf{NV}_1 \mid \mathsf{V}_1) = \gamma' \mid \mathsf{NV}'_1 \mid \mathsf{V}''$, where $\mathsf{NV}_1 = \mathsf{NV}'_1, \mathsf{NV}''_{\mathtt{ck}}$ and $\mathsf{V}_1 = \mathsf{V}'_1, \mathsf{V}''$ and $\mathsf{dom}(\mathsf{V}'') = \mathsf{dom}(\mathsf{NV}'')$. Moreover, $\gamma' \subseteq \gamma$, with $range(\gamma') = \mathsf{dom}(\mathsf{NV}_1) \cup \mathsf{dom}(V'')$. In the JIT mode, the commit function simply drops all volatile memory, i.e., $\mathtt{Commit}(\gamma \mid \mathsf{jit} \mid \mathsf{NV}_1 \mid \mathsf{V}_1) = \gamma' \mid \mathsf{NV}_1$, $\gamma' \subseteq \gamma$, with $\mathsf{range}(\gamma') = \mathsf{dom}(\mathsf{NV}_1)$.

The value relation at type $\uparrow\mathtt{unit}$ requires the successful executions to store the same values in their memories, i.e., $\mathsf{NV}_1 = \mathsf{NV}_2$.

Semantic Typing. A program is semantically well-typed if every JIT and atomic region of it is self-related under our logical relation.

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

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

5.2 Semantic Typing for Idempotency

The fundamental theorem of our logical relation states that syntactically well-typed programs are also semantically well-typed by proving that syntactically well-typed JIT and atomic regions are self-related. We state and prove the theorem in Sec. 6 but devote this section to explaining why being self-related implies idempotency. We explain it separately for JIT and atomic blocks.

Stepping a JIT block. Consider a program of form $[\chi_1 \rhd \varepsilon] \otimes \gamma_1 \mid n \mid \mathsf{NV}_1 \mid c_1; p$ that can take a step to $[\chi_k \rhd \varepsilon] \otimes \gamma \mid n'_k \mid \mathsf{NV}'_k \mid p$ via the D-P-SEQ rule. By the D-P-SEQ rule, we know that the command c_1 is successfully executed to completion with possibly m-many power failures along the way: $[\chi_1 \rhd \varepsilon] \otimes \gamma_1 \mid \mathsf{jit} \mid n \mid \mathsf{NV}_1 \mid \cdot \mid c_1 \Rightarrow^* [\chi_k \rhd \varepsilon] \otimes \gamma'_k \mid \mathsf{jit} \mid n'_k \mid \mathsf{NV}'_k \mid \mathsf{V}'_k \mid \mathsf{skip}$. Our goal is to simulate this execution in a continuous setting. To model a continuous run, we run the configuration with ∞, an energy level: $[\chi \rhd \varepsilon] \otimes \gamma_1 \mid \mathsf{jit} \mid \infty \mid \mathsf{NV}_1 \mid \cdot \mid c_1 \Rightarrow^* [\chi \rhd \varepsilon] \otimes \gamma'_j \mid \mathsf{jit} \mid \infty \mid \mathsf{NV}'_j \mid \mathsf{V}'_j \mid \mathsf{skip}$.

Fig. 12 shows the construction of the simulation. We start with the assumption that the configuration with n energy level is self-related when given energy level ∞ for every index, including $m+1$ (point (1) in Fig. 12). We show that if the first configuration takes one or more steps, the second configuration can take zero or more steps so that the intermediate regions continue to relate.

By definition of the term interpretation, c_1 in the first configuration is executed until the first power failure occurs. Moreover, by the relation, we can execute c_1 in the second configuration, too, such that the resulting configurations remain related (point (2) in Fig. 12) by the value interpretation at type $\mathtt{C}_{\mathtt{unit}}$. The first configuration takes a step from point (2) to point (3) using the D-CRASH rule by the computational semantics. 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) $\mathsf{V}' = \mathsf{V}'_1$ and $\gamma''_1 = \gamma$. This matches the definition of the power-off policy for JIT blocks (see Sec. 5.1), and thus 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 inputting a new energy level from the environment (n_2). By the definition of the value relations, the two configurations will remain related by the value interpretation at type $\uparrow \mathtt{C}_{\mathtt{unit}}$.

Finally, the configuration steps to point (6) by D-RESTORE-JIT that copies all checkpointed locations inside the volatile memory and continues by running the interrupted command κ, i.e., here $\mathsf{NV}_0 = \mathsf{NV}'_1$ and $\mathsf{V}_0 = \mathsf{V}' = \mathsf{V}'_1$ and $c_0 = \kappa$. 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 Fig. 12, but with fewer power failures remaining.

Now, when the first configuration finally steps to point (8), by the definition of the logical relation, we know that the second configuration steps into skip too. Thus, we can apply the D-Ckpt rule on the second configuration. The volatile memory V'_j is dropped, and the mapping is reset to γ, i.e., it matches the commit policy defined for JIT blocks. in the logical relation. By Fig. 12-d, we get $\mathsf{NV}'_j = \mathsf{NV}'_k$, which completes deriving our goal.

Stepping an atomic region. We can build the desired simulation by taking the same steps described for a JIT region. Similarly, the key point is that the power-off and restore policies exactly match how the rules D-S-AID and D-RESTORE-AID, respectively, handle nonvolatile and volatile memories, and the commit policy corresponds to the $\mathsf{FinWorld}$ function in the D-CKPT rule.

We showed that our logical relation ensures idempotency for JIT and atomic regions. In the next section, we show that our logical relation formalizes a semantic typing to ensure idempotency of more general policies.

5.3 More General Policies

We utilize our semantic typing to allow custom policies for power failure, restore, and commit. We extend the grammar of programs as $p := \cdot \mid \mathsf{Reg}[\mathsf{aID}, \overrightarrow{arg}](c); p$, where $\overrightarrow{arg}$ 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 $\mathtt{InitGeneral}_t$ and $\mathtt{InitGeneral}_d$ to initialize the static and dynamic memories, respectively. We add the following semantic typing rule for the general regions:

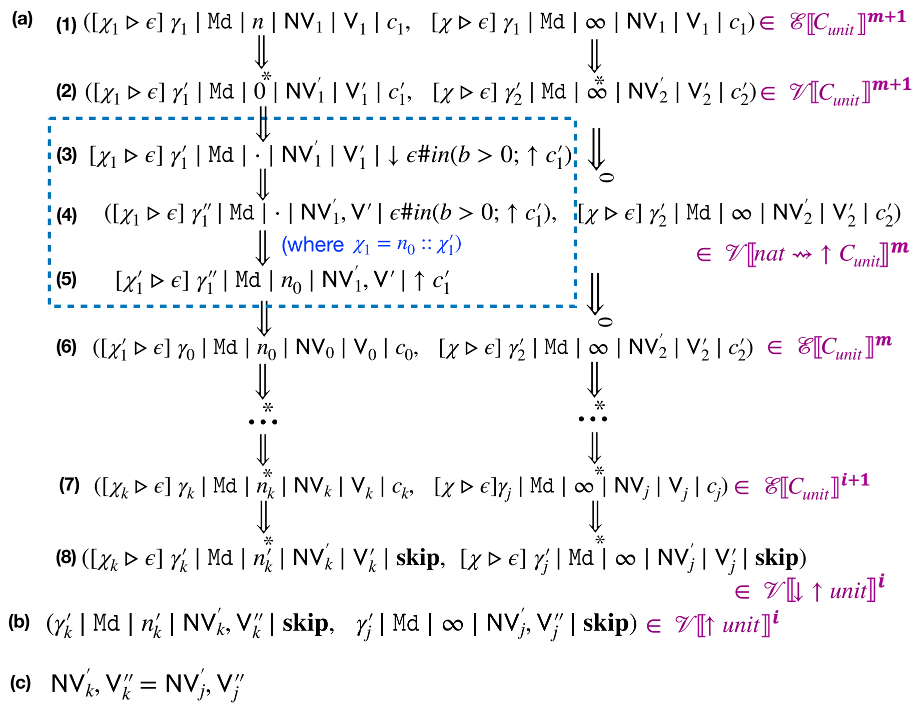

(a)
(1) $([\chi_1 \rhd \epsilon]\ \gamma_1 \mid \mathtt{Md} \mid n \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1,\ \ [\chi \rhd \epsilon]\ \gamma_1 \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_1 \mid \mathsf{V}_1 \mid c_1) \in \mathscr{E}[\![C_{unit}]\!]^{m+1}$
(2) $([\chi_1 \rhd \epsilon]\ \gamma_1' \mid \mathtt{Md} \mid 0 \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid c_1',\ \ [\chi \rhd \epsilon]\ \gamma_2' \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2') \in \mathscr{V}[\![C_{unit}]\!]^{m+1}$
(3) $[\chi_1 \rhd \epsilon]\ \gamma_1' \mid \mathtt{Md} \mid \cdot \mid \mathsf{NV}_1' \mid \mathsf{V}_1' \mid \downarrow \epsilon\#in(b > 0; \uparrow c_1')$
(4) $([\chi_1 \rhd \epsilon]\ \gamma_1'' \mid \mathtt{Md} \mid \cdot \mid \mathsf{NV}_1', \mathsf{V}' \mid \epsilon\#in(b > 0; \uparrow c_1'),\ \ [\chi \rhd \epsilon]\ \gamma_2' \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2')$
(where $\chi_1 = n_0 :: \chi_1'$)
$\in \mathscr{V}[\![nat \rightsquigarrow \uparrow C_{unit}]\!]^{m}$
(5) $[\chi_1' \rhd \epsilon]\ \gamma_1'' \mid \mathtt{Md} \mid n_0 \mid \mathsf{NV}_1', \mathsf{V}' \mid \uparrow c_1'$
(6) $([\chi_1' \rhd \epsilon]\ \gamma_0 \mid \mathtt{Md} \mid n_0 \mid \mathsf{NV}_0 \mid \mathsf{V}_0 \mid c_0,\ \ [\chi \rhd \epsilon]\ \gamma_2' \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_2' \mid \mathsf{V}_2' \mid c_2') \in \mathscr{E}[\![C_{unit}]\!]^{m}$
(7) $([\chi_k \rhd \epsilon]\ \gamma_k \mid \mathtt{Md} \mid n_k \mid \mathsf{NV}_k \mid \mathsf{V}_k \mid c_k,\ \ [\chi \rhd \epsilon]\gamma_j \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_j \mid \mathsf{V}_j \mid c_j) \in \mathscr{E}[\![C_{unit}]\!]^{i+1}$
(8) $([\chi_k \rhd \epsilon]\ \gamma_k' \mid \mathtt{Md} \mid n_k' \mid \mathsf{NV}_k' \mid \mathsf{V}_k' \mid \mathbf{skip},\ \ [\chi \rhd \epsilon]\ \gamma_j' \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_j' \mid \mathsf{V}_j' \mid \mathbf{skip})$
$\in \mathscr{V}[\![\downarrow \uparrow unit]\!]^{i}$
(b) $(\gamma_k' \mid \mathtt{Md} \mid n_k' \mid \mathsf{NV}_k', \mathsf{V}_k'' \mid \mathbf{skip},\ \ \gamma_j' \mid \mathtt{Md} \mid \infty \mid \mathsf{NV}_j', \mathsf{V}_j'' \mid \mathbf{skip}) \in \mathscr{V}[\![\uparrow unit]\!]^{i}$
(c) $\mathsf{NV}_k', \mathsf{V}_k'' = \mathsf{NV}_j', \mathsf{V}_j''$

Fig. 11. Why the logical relation is enough.

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

For a self-related region to be idempotent, its policies `Commit`, `PwOff`, and `Restore` must match the dynamics, so we add dynamic rules for custom regions in Fig. **??**. The JIT and atomic region policies and their dynamic rules are instances of these general policies. As an example, the programmer can customize the policies of the first block of Fig. 1 to not checkpoint variable u. The program remains idempotent as the atomic region never reads u before writing to it. This policy is implemented by real systems [23, 24, 41]. Our static typing rules can be extended to reason about them as shown in the companion technical report.

6 Metatheory

This section establishes the main properties of the system, which are progress and preservation, adequacy, and the most important result: the fundamental theorem

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

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

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

Fig. 12. Custom dynamic rules

where we prove that statically well-typed programs are semantically well-typed. The theorems and their complete proofs are provided in the companion TR [15].

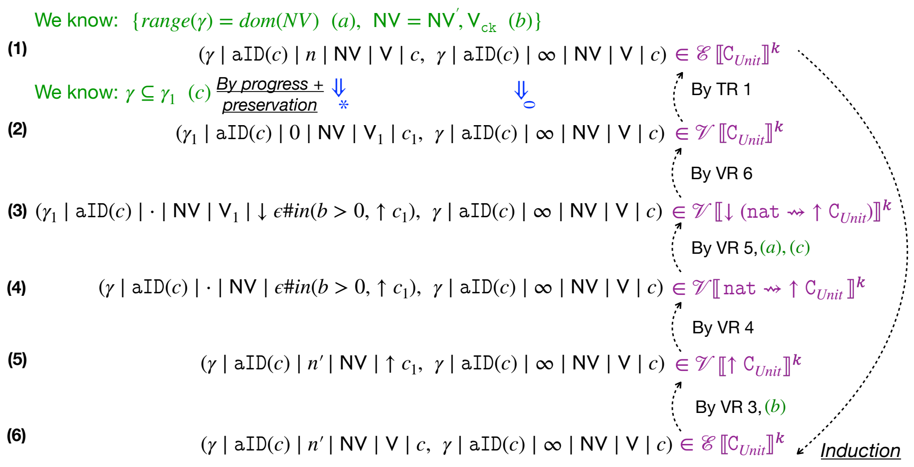
The progress and preservation theorems assume memory locations to be well-formed, $\vdash_{\gamma}^{\mathtt{Md}} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$, which is defined similarly to the $\mathsf{NV} \mid \mathsf{V} \Vdash \gamma : \Omega \mid \Sigma$ used in the logical relation, but imposes extra conditions based on the execution mode $\mathtt{Md}$. It states that γ maps variables in contexts Ω and Σ to the nonvolatile and volatile memories, NV and V, respectively, such that their qualifiers and the type of the stored values match. Moreover, it requires specific properties on the contexts depending on $\mathtt{Md}$; in atomic mode, each checkpointed location in NV and Ω must have copies in V and Σ. We state the theorems below.

Theorem 1 (Progress for Commands). *If* $\mathtt{Md} \mid b\, \mathcal{R}\, m : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau$, *then* $\forall\, n : \mathtt{nat}$ *with* $n \mathcal{R} m$ *and* $\forall\, \gamma, \mathsf{NV}, \mathsf{V}$ *with* $\vdash_{\gamma}^{\mathtt{Md}} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$, *either* $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c$ *is a value, or for some configuration* $\gamma' \mid \mathtt{Md}' \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid c'$ *we have* $\gamma \mid \mathtt{Md} \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c \;\rightarrow\; \gamma' \mid \mathtt{Md}' \mid n' \mid \mathsf{NV}' \mid \mathsf{V}' \mid c'$. *Moreover, if* $\mathtt{Md}$ *is an atomic mode, we have* $\mathsf{NV}' = \mathsf{NV}$.

Theorem 2 (Preservation for Commands). *If* $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma \vdash_{\mathtt{Sig}} c : \tau$, *and for some* $\vdash_{\gamma}^{\mathtt{Md}} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$ *and* $n{:}\mathtt{nat} \geq 0$, *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_1, *we have* $\mathtt{Md} \mid b \geq 0 : \mathtt{nat} \mid \Omega; \Sigma_1 \vdash_{\mathtt{Sig}} c' : \tau$, *where* $\vdash_{\gamma'}^{\mathtt{Md}} \mathsf{NV}' \mid \mathsf{V}' : \Omega \mid \Sigma_1$ *and* $n' \geq 0$.

Theorem 3 (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 3 is by induction on the static typing derivation for p and considers the last step in the derivation. Fig. 13 explains the idea of the proof for the case where P-Ckpt is the last step of the derivation. By inversion, $p = \mathsf{Ckpt}[\mathsf{aID}, \rho](c); p'$. Also, c is well-typed for static contexts Ω' and Σ, where $\Omega' = \Omega'', \Sigma_{\mathsf{ck}}$. The goal is to establish point (1) in the figure: c is related to itself in the term interpretation for arbitrary n, m, γ, NV and V where $\mathsf{NV} \mid \mathsf{V} \Vdash \gamma{:}\Omega'', \Sigma_{\mathsf{ck}} \mid \Sigma$. The last condition enforces that the static contexts match the



We know: $\{range(\gamma) = dom(NV)\ \ (a),\ \ \mathsf{NV} = \mathsf{NV}', \mathsf{V}_{\mathtt{ck}}\ \ (b)\}$
(1) $(\gamma \mid \mathtt{aID}(c) \mid n \mid \mathsf{NV} \mid \mathsf{V} \mid c,\ \gamma \mid \mathtt{aID}(c) \mid \infty \mid \mathsf{NV} \mid \mathsf{V} \mid c) \in \mathscr{E}[\![\mathtt{C}_{Unit}]\!]^k$
We know: $\gamma \subseteq \gamma_1\ \ (c)$
By progress + preservation $\Downarrow_*$
$\Downarrow_0$
By TR 1
(2) $(\gamma_1 \mid \mathtt{aID}(c) \mid 0 \mid \mathsf{NV} \mid \mathsf{V}_1 \mid c_1,\ \gamma \mid \mathtt{aID}(c) \mid \infty \mid \mathsf{NV} \mid \mathsf{V} \mid c) \in \mathscr{V}[\![\mathtt{C}_{Unit}]\!]^k$
By VR 6
(3) $(\gamma_1 \mid \mathtt{aID}(c) \mid \cdot \mid \mathsf{NV} \mid \mathsf{V}_1 \mid \downarrow \epsilon\#in(b > 0, \uparrow c_1),\ \gamma \mid \mathtt{aID}(c) \mid \infty \mid \mathsf{NV} \mid \mathsf{V} \mid c) \in \mathscr{V}[\![\downarrow (\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{Unit})]\!]^k$
By VR 5, $(a), (c)$
(4) $(\gamma \mid \mathtt{aID}(c) \mid \cdot \mid \mathsf{NV} \mid \epsilon\#in(b > 0, \uparrow c_1),\ \gamma \mid \mathtt{aID}(c) \mid \infty \mid \mathsf{NV} \mid \mathsf{V} \mid c) \in \mathscr{V}[\![\mathtt{nat} \rightsquigarrow \uparrow \mathtt{C}_{Unit}]\!]^k$
By VR 4
(5) $(\gamma \mid \mathtt{aID}(c) \mid n' \mid \mathsf{NV} \mid \uparrow c_1,\ \gamma \mid \mathtt{aID}(c) \mid \infty \mid \mathsf{NV} \mid \mathsf{V} \mid c) \in \mathscr{V}[\![\uparrow \mathtt{C}_{Unit}]\!]^k$
By VR 3, (b)
(6) $(\gamma \mid \mathtt{aID}(c) \mid n' \mid \mathsf{NV} \mid \mathsf{V} \mid c,\ \gamma \mid \mathtt{aID}(c) \mid \infty \mid \mathsf{NV} \mid \mathsf{V} \mid c) \in \mathscr{E}[\![\mathtt{C}_{Unit}]\!]^k$
Induction

Fig. 13. Proof of the fundamental theorem for P-Ckpt

dynamic context. The condition also establishes the more refined well-formedness condition that $\vdash^{\mathtt{Md}}_{\gamma} \mathsf{NV} \mid \mathsf{V} : \Omega \mid \Sigma$ in atomic mode, required by progress and preservation, since it enforces that each checkpointed location in NV and Ω have copies in V and Σ. In particular, $\mathsf{NV} = \mathsf{NV}', \mathsf{V}_{\mathtt{ck}}$ and $range(\gamma) = dom(\mathsf{NV})$. When $m = 0$, the proof is trivial. Consider the case where $m = k + 1$. By the progress and preservation theorems, the first configuration can take multiple steps until it becomes a value $\gamma_1 \mid \mathsf{aID}(c) \mid n' \mid \mathsf{NV} \mid \mathsf{V}_1 \mid c_1$ 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 not shown in the figure. If $n' = 0$, the second configuration does not step and instead reaches point (2) in Fig. 13. At point (2), the proof must show that the configurations are in the value interpretation at type $\mathtt{C}_{\mathtt{unit}}$.

The dashed line in the figure states that establishing point (2) implies the relation in point (1). The cascade of implications (dashed lines) follows the definition of the value relations at each type. 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_1 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. Again by the prior conditions established, we know the volatile memory created is the original volatile V. The goal at point (6) is similar to our original goal at point (1), except that the proof uses an inductive argument to relate the two configurations at k.

Finally the Adequacy Theorem states that semantically well-typed programs are idempotent, defined below. The proof is illustrated in Section 5.2.

Definition 1 (Idempotency). *A triple of a program p, nonvolatile memory* NV*, and a mapping γ is idempotent, if every intermittent execution of the pro-*

gram 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 4 (Adequacy). *Consider* $b : \mathtt{nat} \mid \Omega \Vdash p : \mathsf{C}_{\mathtt{unit}}$*, a nonvolatile memory* NV *and a bijective map* γ *that matches qualifiers and types from variables in* Ω *to locations in* NV*. The triple of* p*,* NV*, and* γ *is idempotent.*

7 Discussion & Related Work

Intermittent Computing. Surbatovich et al. [41] 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 some of these invariants; capturing all requires reasoning about the effects of non-deterministic sensor inputs, which we leave to future work. This work is the first to treat intermittent operations at the type level and explore the logical interpretation of intermittent execution. We speculate that our type-based approach using logical relations will provide a cleaner foundation for reasoning about the correctness of more complex intermittent systems, e.g., concurrent ones. 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 [40], which are orthogonal to ours.

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 [20, 21, 36]. There, the relation to Pfenning and Davies's [30] formulation of the lax $\bigcirc$ modality was noted; $\bigcirc$ 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, 14, 34], e.g., switching a processes' mode between shared and unshared [6], or adding multicasting, replicable services, and cancellation modes to a session-typed message passing system [34]. 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, 42] 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 [18]. 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, 16]. 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 [27, 31–33] 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 Hoare Logic. Crash Hoare logic (CHL) [11] 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 care about 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.

8 Conclusion

This work provides the first logical interpretation of intermittent execution. It shows that adjoint logic can be applied 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, and the fundamental theorem imply the correctness of intermittent systems, i.e. idempotency of execution.

References

1. Adkins, J., Campbell, B., Ghena, B., Jackson, N., Pannuto, P., Dutta, P.: The signpost network: Demo abstract. In: Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM. SenSys '16 (2016). https://doi.org/10.1145/2994551.2996542
2. Ahmed, A., Dreyer, D., Rossberg, A.: State-dependent representation independence. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 340–353. POPL '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1480881.1480925
3. Ahmed, A.J.: Semantics of types for mutable state. Princeton University (2004)
4. Balsamo, D., Weddell, A., Das, A., Arreola, A., Brunelli, D., Al-Hashimi, B., Merrett, G., Benini, L.: Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **PP**(99), 1–1 (2016). https://doi.org/10.1109/TCAD.2016.2547919
5. Balsamo, D., Weddell, A.S., Merrett, G.V., Al-Hashimi, B.M., Brunelli, D., Benini, L.: Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. IEEE Embedded Systems Letters **7**(1), 15–18 (2015). https://doi.org/10.1109/LES.2014.2371494
6. Balzer, S., Toninho, B., Pfenning, F.: Manifest deadlock-freedom for shared session types. In: Proceedings of the 29th European Symposium on Programming. pp. 611–639 (2019)
7. Benton, N., Wadler, P.: Linear logic, monads and the lambda calculus. In: Proceedings 11th Annual IEEE Symposium on Logic in Computer Science. pp. 420–431. IEEE (1996)
8. Benton, P.N.: A mixed linear and non-linear logic: Proofs, terms and models. In: International Workshop on Computer Science Logic. pp. 121–135. Springer (1994)
9. Berthou, G., Dagand, P.E., Demange, D., Oudin, R., Risset, T.: Intermittent computing with peripherals, formally verified. In: The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. pp. 85–96. LCTES '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372799.3394365
10. Birkedal, L., Støvring, K., Thamsborg, J.: Realizability semantics of parametric polymorphism, general references, and recursive types. In: International Conference on Foundations of Software Science and Computational Structures. pp. 456–470. FOSSACS '09, Springer (2009)
11. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Using crash hoare logic for certifying the fscq file system. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 18–37. SOSP '15 (2015)
12. Colin, A., Lucia, B.: 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 (2016). https://doi.org/10.1145/2983990.2983995
13. Dahiya, M., Bansal, S.: Automatic verification of intermittent systems. In: Dillig, I., Palsberg, J. (eds.) Verification, Model Checking, and Abstract Interpretation. VMCAI '18 (2018)
14. Das, A., Balzer, S., Hoffmann, J., Pfenning, F., Santurkar, I.: Resource-aware session types for digital contracts. In: IEEE 34th Computer Security Foundations Symposium. pp. 1–16. CSF '21 (2021)

15. Derakhshan, F., Dotzel, M., Surbatovich, M., Jia, L.: Technical report: Modal crash types for intermittent computing. Tech. rep., Carnegie Mellon University (2023)
16. Dreyer, D., Neis, G., Birkedal, L.: The impact of higher-order state and control effects on local relational reasoning. Journal of Functional Programming **22**(4-5), 477–528 (2012)
17. Hester, J., Storer, K., Sorber, J.: Timely execution on intermittently powered batteryless sensors. In: Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (2017). https://doi.org/10.1145/3131672.3131673
18. Jung, A., Tiuryn, J.: A new characterization of lambda definability. In: International Conference on Typed Lambda Calculi and Applications. pp. 245–257. Springer (1993)
19. Kortbeek, V., Yildirim, K.S., Bakar, A., Sorber, J., Hester, J., Pawełczak, P.: Time-sensitive intermittent computing meets legacy software. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 85–99. ASPLOS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3373376.3378476
20. Licata, D.R., Shulman, M.: Adjoint logic with a 2-category of modes. In: International Symposium on Logical Foundations of Computer Science. pp. 219–235. Springer (2016)
21. Licata, D.R., Shulman, M., Riley, M.: A fibrational framework for substructural and modal logics. In: 2nd International Conference on Formal Structures for Computation and Deduction. FSCD '17, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
22. Lucia, B., Denby, B., Manchester, Z., Desai, H., Ruppel, E., Colin, A.: Computational nanosatellite constellations: Opportunities and challenges. GetMobile: Mobile Comp. and Comm. **25**(1), 16–23 (Jun 2021). https://doi.org/10.1145/3471440.3471446
23. Lucia, B., Ransford, B.: 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 (2015). https://doi.org/10.1145/2737924.2737978
24. Maeng, K., Colin, A., Lucia, B.: Alpaca: Intermittent execution without checkpoints. Proc. ACM Program. Lang. **1**(OOPSLA), 96:1–96:30 (Oct 2017). https://doi.org/10.1145/3133920
25. Maeng, K., Lucia, B.: Supporting peripherals in intermittent systems with just-in-time checkpoints. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1101–1116. PLDI '19 (2019). https://doi.org/10.1145/3314221.3314613
26. Maeng, K., Lucia, B.: Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 1005–1021. PLDI '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3385998
27. Moggi, E.: Computational lambda-calculus and monads. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science (1988)
28. Nardello, M., Desai, H., Brunelli, D., Lucia, B.: Camaroptera: A batteryless long-range remote visual sensing system. In: Proceedings of the

7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems. pp. 8–14. ENSsys'19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3362053.3363491
29. NASA: What is KickSat-2? https://www.nasa.gov/ames/kicksat (2019), visited April 15th, 2022
30. Pfenning, F., Davies, R.: A judgmental reconstruction of modal logic. Mathematical structures in computer science **11**(4), 511–540 (2001)
31. Plotkin, G., Power, J.: Semantics for algebraic operations. Electronic Notes in Theoretical Computer Science **45**, 332–345 (2001)
32. Plotkin, G., Pretnar, M.: Handlers of algebraic effects. In: Proceedings of the 19th European Symposium on Programming. pp. 80–94. Springer (2009)
33. Pretnar, M., Plotkin, G.D.: Handling algebraic effects. Logical methods in computer science **9** (2013)
34. Pruiksma, K., Pfenning, F.: A message-passing interpretation of adjoint logic. Journal of Logical and Algebraic Methods in Programming **120**, 100637 (2021)
35. Ransford, B., Sorber, J., Fu, K.: Mementos: System support for long-running computation on RFID-scale devices. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XVI (2011). https://doi.org/10.1145/1950365.1950386
36. Reed, J.: A judgmental deconstruction of modal logic. Unpublished manuscript, January (2009)
37. Ruppel, E., Lucia, B.: Transactional concurrency control for intermittent, energy-harvesting computing systems. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1085–1100. PLDI '19 (2019). https://doi.org/10.1145/3314221.3314583
38. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing. pp. 204–213. PODC '95 (1995). https://doi.org/10.1145/224964.224987
39. Surbatovich, M., Jia, L., Lucia, B.: I/o dependent idempotence bugs in intermittent systems. Proc. ACM Program. Lang. **3**(OOPSLA), 183:1–183:31 (Oct 2019). https://doi.org/10.1145/3360609
40. Surbatovich, M., Jia, L., Lucia, B.: Automatically enforcing fresh and consistent inputs in intermittent systems. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 851–866. PLDI '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454081
41. Surbatovich, M., Lucia, B., Jia, L.: Towards a formal foundation of intermittent computing. Proc. ACM Program. Lang. **4**(OOPSLA) (Nov 2020). https://doi.org/10.1145/3428231
42. Thamsborg, J., Birkedal, L.: A kripke logical relation for effect-based program transformations. ACM SIGPLAN Notices **46**(9), 445–456 (2011)
43. Van Der Woude, J., Hicks, M.: Intermittent computation without hardware support or programmer intervention. In: Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation (2016). https://doi.org/10.5555/3026877.3026880
44. Yildirim, K.S., Majid, A.Y., Patoukas, D., Schaper, K., Pawelczak, P., Hester, J.: Ink: Reactive kernel for tiny batteryless sensors. In: Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems. pp. 41–53. SenSys '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3274783.3274837