Type-Based Temporal Resource Usage Analysis

YIYUAN CAO, Peking University, China
TARO SEKIYAMA, National Institute of Informatics, Japan

Resources—such as files, locks, and memory cells—are stateful objects that must be used according to specific protocols during program execution. These protocols can be formally expressed as *temporal properties* over event traces that represent resource usage, incorporating both safety and liveness requirements. Existing verification approaches to resource usage analysis ignore liveness requirements of resource usage. As a result, they are inadequate for verifying the resource usage correctness of divergent or indefinitely running programs, such as server-side programs or interactive applications.

To address this gap, we define the general problem of *temporal resource usage analysis* in a higher-order language with recursive functions and dynamically allocated resources. As a solution to this problem, we propose a *temporal resource type system*. A main novelty of our type system is a *resettable timer* mechanism that provides a progressivity guarantee for temporal resource usage even in the divergent program execution. We demonstrate the usefulness of our type system by examples, and prove its soundness. Our proof of the soundness is based on a logical relation that captures the progressive nature of divergent computations.

1 Introduction

1.1 Background: Correct Use of Resources

Programs use a variety of *resources*—such as heap memory, files, network sockets, and locks—for efficiency, interaction with the external environment, or to access special functionalities of the underlying operating system. These resources are *stateful* objects that must be used in a valid manner. Different types of resources may require specific usage protocols, including constraints on the order, reachability, and multiplicity of operations on the resources. For example, a file handle must be opened before reading from or writing to it and a memory block may be freed at most once. These requirements on file and memory resources are *safety* properties, which require that resource operations are only applied to resources in some appropriate states (or, "nothing bad happens to resources"). The task of verifying such safety properties about resource usage is called *resource usage analysis problem* [22] and has been studied actively for decades [2, 7, 17, 19, 22, 43, 50].

However, safety is not the only criterion for correct use of resources: another important criterion is *liveness*, which requires resources to eventually reach some desired states (or, "something good will eventually happen to resources"). For instance, consider the program (written in ML-like syntax) in Figure 1 that manipulates file and lock resources. Its functionality is to get a path name from the user interactively, open it as an input file, read from the file, and write the contents to a lock-protected log. The program dynamically allocates file resources and may run indefinitely (it terminates only when the user types `"EXIT"`). For such a potentially diverging program to be correct, it is desired to ensure liveness properties of resources—not only that every file is safely used, but also that the opened files are eventually closed to avoid resource leaks, and a lock that has been acquired should eventually be released so that other threads can enter the critical sections.[1] Although liveness properties for terminating programs can be reformulated as safety ones, ensuring liveness properties for diverging or indefinitely running programs, such as server-side programs and interactive applications, poses a distinct challenge [5, 39, 57].

[1]The formal system presented in the paper only focuses on single-threaded programs, and the support for multi-threaded programs is left as future work.

Authors' Contact Information: Yiyuan Cao, Peking University, Beijing, China, cyy9447@pku.edu.cn; Taro Sekiyama, National Institute of Informatics, Tokyo, Japan, sekiyama@nii.ac.jp.

```
let rec main_loop (log_lock : lock) : unit =
  let path = input () in
  if path = "EXIT" then () else (
    let input_file = open path in
    let content = read input_file in
    close input_file;
    acquire log_lock;
    ... (* write content to the lock-protected log *)
    release log_lock;
    main_loop log_lock)
let main () =
  let log_lock = new_lock () in main_loop log_lock
```

Fig. 1. An Interactive File-Logger (Valid).

A unified view for resource usage specifications—including both safety and liveness properties—is obtained by formulating them as *linear-time temporal properties* (or temporal properties for short) [3, 41], that is, properties on the (potentially infinite) *traces* that represent the sequences of effectful operations performed on resources. For example, using the mixed form of regular and ω-regular expressions, valid usage traces of files and locks may be formally specified as:

$$\begin{aligned} \Psi_{\texttt{file}} &\stackrel{\text{def}}{=} \texttt{open} \cdot (\texttt{read} \mid \texttt{write})^* \cdot \texttt{close} \\ \Psi_{\texttt{lock}} &\stackrel{\text{def}}{=} (\texttt{acquire} \cdot \texttt{release})^* \mid (\texttt{acquire} \cdot \texttt{release})^\omega \,. \end{aligned}$$

Here, given expressions Ψ and Ψ' in the mixed form, $\Psi \cdot \Psi'$ (resp. $\Psi \mid \Psi'$) denotes the concatenation (resp. union) of traces in Ψ and Ψ', Ψ^* denotes any finite number of repetitions of traces in Ψ, and Ψ^ω denotes any infinite number of repetitions of traces in Ψ. The expression $\Psi_{\texttt{file}}$ specifies traces that start with opening a file, followed by any finite number of reading or writing, and end with closing the file. Thus, files used as specified by $\Psi_{\texttt{file}}$ are read and written only when they have been opened but have not been closed (the safety property), and they are closed eventually when they have been opened (the liveness property). Similarly, the expression $\Psi_{\texttt{lock}}$ specifies traces that repeat (finitely or infinitely) the pattern of acquiring a lock and then releasing it finally. Therefore, locks following this specification are acquired before being released (the safety property) and are eventually released once acquired (the liveness property). Traces on locks may be infinite because locks can be acquired again and again after being released. The program shown in Figure 1 satisfies these specifications for files and locks. However, if the `close` operation were moved after the recursive call to `main_loop`, the file specification would no longer be satisfied, since the recursive call might not return and then `close` would not be invoked. Note that the checking problem of safety properties essentially aims to verify that a generated trace is a *prefix* of some trace in the specification. When taking liveness properties into account, we have to check if the generated full trace *exactly matches* some trace in the specification.[2]

[2]This is more general than prefix checking. If one is only interested in safety properties of, e.g., files, it suffices to check that the generated trace exactly matches some trace in the prefix set of $\Psi_{\texttt{file}}$.

1.2 Our Work

In this paper, we focus on type-based verification for temporal properties of resources in a higher-order language with general recursive functions and dynamically allocated resources. For this goal, we make the following contributions.

Contribution 1: A formalization of the temporal resource usage analysis problem (Section 2). We formulate the verification problem we are interested in as the *temporal resource usage analysis problem*, or *temporal usage analysis* for short, which generalizes the resource usage analysis problem proposed by Igarashi and Kobayashi [22] to consider liveness properties of resources. We classify resources by their lifetimes (*finite*, *indefinite*, and *infinite*) and provide a correctness criterion for each category. Crucially, our formulation of the problem establishes correctness criteria that address liveness properties of resources in diverging programs.

Contribution 2: A temporal resource type system (Sections 3 and 4). We introduce a type system, which we call a *temporal resource type system*, for temporal usage analysis. The key challenge for that is threefold: *resource aliasing*, *progressivity guarantee*, and *termination analysis*.

The aliasing problem is that the same resource may be referenced by different names. Its presence together with mutability hinders program reasoning. This is common in programming with stateful objects (especially, in heap-manipulating programs), and many type-based approaches have been proposed to address it [10, 25, 32, 34, 37, 49, 54]. In this paper, for its simplicity and generality, we employ *uniqueness typing* [6, 16, 48] and guarantees that every resource can be referenced in a unique way. We discuss other possible mechanisms to address the aliasing problem in Section 6.

The progressivity guarantee concerns the problem of ensuring that the usage of resources eventually leads them to some desired states. Similar issues arise in reasoning about infinite data objects [4, 13, 33] and in liveness verification for a *global* trace [45] (which records all effectful operations during the program execution—unlike our setting of separate per-resource traces). Our key idea for ensuring progressivity in resource usage lies in the type representation of resources: we enrich resource types with *resettable timers*, which are a ghost mechanism that guarantees that resources will eventually perform some operations to reach the desired states.

The need for termination analysis stems from the subtle phenomenon of *implicit discarding* of resources. That is, resources that are not accessible from a diverging sub-computation are essentially discarded. In this case, a sound type system must ensure that these unused resources have reached the desired states just before the execution of the sub-computation starts. To precisely detect this phenomenon, we incorporate user-provided termination information into the type system.

In the literature, numerous type systems for ensuring safety properties of resources [1, 2, 7, 17, 19, 22, 23, 43, 50], as well as type-and-effect systems for temporal properties over global traces [21, 28, 36, 44, 45] have been proposed, but, to the best of our knowledge, our type system is the first to address temporal usage analysis for higher-order programs. We provide a detailed comparison with these prior works, as well as other approaches to temporal verification, in Section 7.

Contribution 3: Soundness of the temporal resource type system (Section 5). We prove soundness of our type system with respect to temporal usage analysis. The soundness for the finite- and indefinite lifetime criteria follows from standard syntactic type safety since they are safety properties. In contrast, the soundness for the infinite-lifetime criterion is more intricate as it essentially requires ensuring a liveness property. We prove the latter soundness using a logical relation that formalizes the progressive nature of divergent computations.

The organization of the paper. Section 2 formalizes the temporal usage analysis problem. Section 3 gives an overview of our type system, along with the technical challenges in its design. The

$$\textbf{Variables}\quad x, f \qquad \textbf{Locations}\quad \ell \qquad \textbf{Events}\quad a \in \mathbb{A}$$
$$\textbf{Integers}\quad n \in \mathbb{Z} \qquad \textbf{Arithmetic Operations}\quad \hat{\oplus} ::= + \mid - \mid \cdots$$
$$\begin{array}{lrcl} \textbf{Constants} & c & ::= & n \mid () \mid \cdots \quad \textbf{Patterns}\ P ::= x \mid () \mid (P_1, P_2) \\ \textbf{Values} & v & ::= & x \mid c \mid \mathsf{res}\,\ell \mid (v_1, v_2) \mid \lambda x.\, e \mid \mathsf{rec}\, f\, x.\, e \\ \textbf{Expressions} & e & ::= & v \mid v_1 \mathbin{\hat{\oplus}} v_2 \mid \mathsf{let}\, P = e_1\, \mathsf{in}\, e_2 \mid \mathsf{if0}\, v\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \mid v_1\ v_2 \mid \\ & & & \mathsf{new}_\Psi \mid \mathsf{acc}_a(v) \mid \mathsf{drop}(v) \end{array}$$

Fig. 2. Syntax of λ_{res}.

formalization and soundness of the type system are presented in Sections 4 and 5, respectively. We discuss the limitations of our current type system and outline future directions in Section 6. We also situate our work within the existing literature in Section 7.

In the paper, we only state the main meta-theoretic properties and proof sketches of our type system. The full proofs and auxiliary definitions can be found in the supplementary material.

2 Temporal Resource Usage Analysis Problem

In this section, we formally define the problem of temporal resource usage analysis. We first introduce a standard higher-order language λ_{res} with dynamically allocated resources. We then classify resources based on their lifetime in the resource heap during an execution, and formalize the temporal resource usage analysis problem using a correctness criterion for each class.

2.1 Language λ_{res}

2.1.1 Syntax. Our language λ_{res} is a fine-grain call-by-value λ-calculus [31] extended with resource manipulation. Its syntax is presented in Figure 2. It mostly follows the language used in resource usage analysis [22, 23] except for deallocation constructs, which are newly added for explicit resource management.

The syntax of λ_{res} consists of values v and expressions e. A value is a variable x, constant c, resource $\mathsf{res}\,\ell$ allocated at a location ℓ, pair (v_1, v_2), λ-abstraction $\lambda x.\, e$, or recursive function $\mathsf{rec}\, f\, x.\, e$. We separate λ-abstractions and recursive functions to introduce certain specific typing rules for recursive functions in Section 4, but semantically they can be unified by considering $\lambda x.\, e$ as an abbreviation of $\mathsf{rec}\, f\, x.\, e$ for some f that does not occur free in the expression e. An expression is: a value; an arithmetic operation call; a let-expression $\mathsf{let}\, P = e_1\, \mathsf{in}\, e_2$ with pattern P for deconstructing the value of the expression e_1; an if-expression $\mathsf{if0}\, v\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$, which branches into subexpressions e_1 and e_2 depending on whether the integer value v is zero; a function application $v_1\ v_2$; or an application of a certain resource operation, i.e., *resource allocation* new_Ψ, *access* $\mathsf{acc}_a(v)$, or *deallocation* $\mathsf{drop}(v)$.

In the rest of this paper, we use the following syntactic sugars: $\mathsf{acc}_a(x); e \stackrel{\text{def}}{=} \mathsf{let}\, x = \mathsf{acc}_a(x)\, \mathsf{in}\, e$ for single-threaded resource access (acc returns the same resource as the argument, as explained in detail shortly); and $e_1; e_2 \stackrel{\text{def}}{=} \mathsf{let}\, () = e_1\, \mathsf{in}\, e_2$ for sequential composition (if $e_1 \neq \mathsf{acc}_a(x)$).[3] Also, we often place a non-value expression at the position where a value is expected. In such a case, we assume that let constructs are inserted appropriately. For example, a function application $e_1\ e_2$ is regarded as $\mathsf{let}\, x = e_1\, \mathsf{in}\, \mathsf{let}\, y = e_2\, \mathsf{in}\, x\ y$ implicitly if neither e_1 nor e_2 is a value.

A resource allocation new_Ψ is accompanied by a *temporal specification* Ψ, which should be assumed to be provided by the user and prescribes valid usage traces of the allocated resource. We

[3] Also for the $\mathsf{reset}^m(x)$ construct introduced later in Section 4, we write $\mathsf{reset}^m(x); e \stackrel{\text{def}}{=} \mathsf{let}\, x = \mathsf{reset}^m(x)\, \mathsf{in}\, e$.

write $[\![\Psi]\!]$ for the set of such valid traces. Although our type system assumes temporal specifications in a certain specific syntactic form (see Section 4 for detail), the semantics relies only on the trace sets associated with them. The allocated resource is consumed by resource access constructs.

A resource access $\mathsf{acc}_a(v)$ takes a resource v and returns the same resource.[4] The *event* a is a symbol expressing the name of a certain resource operation, such as `open`, `read`, `write`, and `close` for files, and `acquire` and `release` for locks. Semantically, a resource accumulates as a trace the events specified in accessing to the resource: when allocated, a resource $\mathsf{res}\,\ell$ is associated with the empty trace; and every time a resource access $\mathsf{acc}_a(\mathsf{res}\,\ell)$ is performed, the event a is appended to the trace of the resource $\mathsf{res}\,\ell$. We call these event-accumulating traces *history traces*. For instance, consider an expression $e_{\mathtt{file1}} \stackrel{\text{def}}{=} \mathsf{let}\, x = \mathsf{new}_{\Psi_{\mathtt{file}}}\, \mathsf{in}\, \mathsf{acc}_{\mathtt{open}}(x); \mathsf{acc}_{\mathtt{close}}(x)$ with the operations `open` and `read` and the temporal specification $\Psi_{\mathtt{file}}$ for files. When it terminates, the history trace of the allocated resource is $\mathtt{open} \cdot \mathtt{close}$. An access $\mathsf{acc}_a(\mathsf{res}\,\ell)$ succeeds only when the history trace extended with an event a is a prefix of some trace in $[\![\Psi]\!]$ for the temporal specification Ψ of the resource $\mathsf{res}\,\ell$. The above expression $e_{\mathtt{file1}}$ successfully terminates. By contrast, an expression $e_{\mathtt{file2}} \stackrel{\text{def}}{=} \mathsf{let}\, x = \mathsf{new}_{\Psi_{\mathtt{file}}}\, \mathsf{in}\, \mathsf{acc}_{\mathtt{open}}(x); \mathsf{acc}_{\mathtt{close}}(x); \mathsf{acc}_{\mathtt{read}}(x)$ gets stuck because it tries to apply `read` after `close`, but $\Psi_{\mathtt{file}}$ does not accommodate the trace $\mathtt{open} \cdot \mathtt{close} \cdot \mathtt{read}$.

Although resource access constructs take and return only resources, commonly used resource operations may take and return other values (e.g., `write` and `read` for files may take and return strings, respectively). We made this design choice to make our technical development as simple as possible while still being able to express temporal usage analysis, as in the work of Igarashi and Kobayashi [22], which only allows resource accesses to return Boolean values nondeterministically. We believe that it poses no challenge to support resource operations in a "common" form.

A deallocation construct $\mathsf{drop}(v)$ discards resources involved in the value v. It is allowed only when the history traces of the discarded resources conform to the temporal specifications (i.e., only when the resources have been *used-up* as described by the specifications). The deallocation is a common instruction in many resources by assuming that it is performed together with operations that finalize the resources, such as `close` for files and `free` for manually managed heap memory. The deallocated value v is not restricted to be a resource; it can be an arbitrary value, e.g., a resource-capturing λ-abstraction $\lambda x.\, \mathsf{acc}_{\mathtt{open}}(\mathsf{res}\,\ell)$.

We can define the notions of free and bound variables and capture-avoiding substitution in a standard way. Given a value substitution γ, which is a finite mapping from variables to values, the expression $\gamma(e)$ is obtained by applying γ to the expression e in a capture-avoiding manner. A value substitution that only maps a variable x to a value v is expressed by $\{x \mapsto v\}$. The domain of a value substitution γ is denoted by $dom(\gamma)$, and the concatenation of value substitutions γ_1 and γ_2 with distinct domains is by $\gamma_1 \uplus \gamma_2$. We use similar notation for other mappings (such as resource heaps introduced in Section 2.1.2).

2.1.2 Semantics. We define the operational semantics of λ_{res} using two relations: pure reduction relation $\rightsquigarrow$ and evaluation relation $\twoheadrightarrow$, which are defined in Figure 3 along with other auxiliary definitions. Given the event set $\mathbb{A}$, we write $\mathbb{A}^*$ and $\mathbb{A}^\omega$ for the sets of finite and infinite, respectively, traces consisting of events in $\mathbb{A}$. We designate finite, infinite, and indefinite (either finite or infinite) traces by ϖ, π, and δ, respectively. We use the notations ϵ and $\varpi \cdot \delta$ for the empty trace and the concatenation of a finite trace ϖ with an indefinite trace δ, respectively. We also introduce *resource heaps* because resources in our language are stateful. A resource heap σ is a finite mapping that associates a resource location ℓ with a history trace ϖ and a *usage specification* S, which is the trace set representing the denotation $[\![\Psi]\!]$ of a temporal specification Ψ given when allocating ℓ.

[4] The design decision to return the same resource as the argument stems from our type system with uniqueness typing.

$$\textbf{Traces}\ \varpi \in \mathbb{A}^* \quad \pi \in \mathbb{A}^\omega \quad \delta \in \mathbb{A}^\infty \stackrel{\text{def}}{=} \mathbb{A}^* \cup \mathbb{A}^\omega \quad \textbf{Trace Sets}\ S \subseteq \mathbb{A}^\infty$$

$$\textbf{Resource Heaps}\quad \sigma ::= \{\ell_1 \mapsto_{S_1} \varpi_1, \ldots, \ell_n \mapsto_{S_n} \varpi_n\}$$

Value Pattern Matching $\boxed{[v/P]}$

$$[v/x] \stackrel{\text{def}}{=} \{x \mapsto v\} \qquad [()/()] \stackrel{\text{def}}{=} \emptyset \qquad [(v_1, v_2)/(P_1, P_2)] \stackrel{\text{def}}{=} [v_1/P_1] \uplus [v_2/P_2]$$

Pure Reduction Rules $\boxed{e_1 \rightsquigarrow e_2}$

$$(\lambda x.\, e)\ v \rightsquigarrow \{x \mapsto v\}(e) \qquad \text{if0 } 0 \text{ then } e_1 \text{ else } e_2 \rightsquigarrow e_1$$
$$(\text{rec } f\ x.\, e)\ v \rightsquigarrow (\{f \mapsto \text{rec } f\ x.\, e\} \uplus \{x \mapsto v\})(e) \qquad \text{if0 } n \text{ then } e_1 \text{ else } e_2 \rightsquigarrow e_2 \quad (\text{if } n \neq 0)$$
$$\text{let } P = v \text{ in } e \rightsquigarrow [v/P](e) \qquad n_1 \mathbin{\hat{\oplus}} n_2 \rightsquigarrow n_1 \oplus n_2$$

Discardable Heaps $\boxed{\models^\dagger \sigma}$

$$\models^\dagger \sigma \stackrel{\text{def}}{=} \forall\, \ell \mapsto_S \varpi \in \sigma.\, \varpi \in S$$

Evaluation Rules $\boxed{(e_1, \sigma_1) \rightsquigarrow (e_2, \sigma_2)}$

$$\frac{e \rightsquigarrow e'}{(e, \sigma) \rightsquigarrow (e', \sigma)} \qquad \frac{(e_1, \sigma) \rightsquigarrow (e_1', \sigma')}{(\text{let } P = e_1 \text{ in } e_2, \sigma) \rightsquigarrow (\text{let } P = e_1' \text{ in } e_2, \sigma')}$$

$$\frac{[\![\Psi]\!] \neq \emptyset}{(\text{new}_\Psi, \sigma) \rightsquigarrow (\text{res } \ell, \sigma \uplus \{\ell \mapsto_{[\![\Psi]\!]} \epsilon\})} \qquad \frac{dom(\sigma') = loc(v) \quad \models^\dagger \sigma'}{(\text{drop}(v), \sigma \uplus \sigma') \rightsquigarrow ((), \sigma)}$$

$$\frac{\varpi \cdot a \in pref(S)}{(\text{acc}_a(\text{res } \ell), \sigma \uplus \{\ell \mapsto_S \varpi\}) \rightsquigarrow (\text{res } \ell, \sigma \uplus \{\ell \mapsto_S \varpi \cdot a\})}$$

Fig. 3. Operational Semantics of λ_{res}.

The pure reduction relation $\rightsquigarrow$ is a binary relation over expressions, defined by pure reduction rules shown in Figure 3. We assume that α-conversion of binders is performed to avoid name clash. The rules are almost standard except the one for let-expressions $\text{let } P = e_1 \text{ in } e_2$, which relies on a value substitution $[v/P]$ obtained by matching a pattern P with a value v (it is also defined in Figure 3). We assume that the denotation of an arithmetic operation $\hat{\oplus}$ is given by $\oplus$.

The evaluation relation $\rightsquigarrow$ is a binary relation over *configurations*, which are pairs of an expression and a resource heap. Formally, the evaluation relation is defined as the smallest relation satisfying the evaluation rules presented in Figure 3. The first two evaluation rules are self-explanatory, meaning that an expression that manipulates no resource is evaluated via the pure reduction rule and that the evaluation of a let-expression $\text{let } P = e_1 \text{ in } e_2$ starts by evaluating the expression e_1. A resource allocation new_Ψ allocates a fresh location ℓ and sets the initial history trace and usage specification to the empty trace ϵ and the trace set $[\![\Psi]\!]$ given by the user-provided specification Ψ, respectively. We require that the trace set $[\![\Psi]\!]$ be nonempty as, otherwise, we cannot access nor deallocate the resource. A resource access $\text{acc}_a(\text{res } \ell)$ appends the event a to the history trace ϖ of the resource $\text{res } \ell$ if the appending result $\varpi \cdot a$ is a prefix of some trace in the usage specification S of the resource $\text{res } \ell$. This side condition dynamically ensures that the usage of the resource satisfies

its safety property. Formally, the set $\mathit{pref}(S)$ of finite prefixes of traces in a trace set S is defined as:

$$\mathit{pref}(S) \stackrel{\text{def}}{=} \{\varpi \mid \exists\, \delta.\ \varpi \cdot \delta \in S\}\ .$$

A resource deallocation $\mathsf{drop}(v)$ discards the resources involved in the argument v. The evaluation rule for $\mathsf{drop}(v)$ identifies such resources by $\mathit{loc}(v)$, splits the resource heap into fragments σ' that only contains resources to be discarded, and σ that should be retained. The heap σ' is checked to be discardable by the *discardable heap predicate* $\models^{\dagger} \sigma'$, which ensures that each history trace in σ' satisfies the corresponding usage specification. For instance, an expression $e_{\mathtt{file3}} \stackrel{\text{def}}{=} \mathsf{let}\ x = \mathsf{new}_{\Psi_{\mathtt{file}}}\ \mathsf{in}\ \mathsf{acc}_{\mathsf{open}}(x); \mathsf{acc}_{\mathtt{close}}(x); \mathsf{drop}(x)$ may discard the allocated resource via $\mathsf{drop}(x)$ because its history trace at the point of $\mathsf{drop}(x)$ is $\mathsf{open} \cdot \mathtt{close}$, which conforms to $\Psi_{\mathtt{file}}$. By contrast, for an expression $e_{\mathtt{file4}} \stackrel{\text{def}}{=} \mathsf{let}\ x = \mathsf{new}_{\Psi_{\mathtt{file}}}\ \mathsf{in}\ \mathsf{acc}_{\mathsf{open}}(x); \mathsf{drop}(x)$, the deallocation $\mathsf{drop}(y)$ should fail, because the created file resource is still opened at the point, but the usage specification $\Psi_{\mathtt{file}}$ requires files to be closed finally.

2.2 Problem Definition

In this section, we formally define the problem of temporal resource usage analysis for full *programs*. We assume a program to be an expression that returns some constant, such as the unit value $()$.

To properly define correctness of temporal resource usage, we first classify resources by their *lifetimes*. Every resource $\mathsf{res}\ \ell$ allocated during a program execution falls into one of three categories, depending on their duration in the resource heap:

- **Finite-Lifetime Resources**, which are explicitly deallocated by drop during the execution;
- **Infinite-Lifetime Resources**, which are never deallocated and remain in the resource heap forever when the execution diverges; and
- **Indefinite-Lifetime Resources**, which remain in the resource heap produced by the terminating execution.

Resources are *correctly used* if their history traces conform to their temporal specifications. For a finite-lifetime resource, its correct use is guaranteed if the history trace at the point of the deallocation is contained in the usage specification—this requirement is actually checked by the evaluation rule for drop. Therefore, the correct use of finite-lifetime resources are guaranteed by ensuring that *the execution of a program does not get stuck*. For an indefinite-lifetime resource, its correct use is guaranteed if the program execution terminates at a discardable resource heap because every history trace in such a heap satisfies the corresponding usage specification (see Figure 3). Therefore, the correct use of indefinite-lifetime resources are guaranteed by ensuring that *the terminating execution of a program produces a discardable resource heap*.

The correctness of the usage of infinite-lifetime resources is more subtle. The semantics of λ_{res} guarantees that they are safely used at any point (specifically, the evaluation rule for resource accesses does this check), but it does *not* guarantee that history traces satisfying their usage specifications are eventually generated during the divergent execution. For example, consider a program that alters the example in Figure 1 to invoke the `close` operation after the recursive call (it is concretely given in Example 2.5 with slight simplification). The execution of such a program does not get stuck under the semantics of λ_{res}. Furthermore, when the execution terminates, all the allocated resources should be discardable. However, the resource usage in the divergent execution of that program does not satisfy the file specification $\Psi_{\mathtt{file}} = \mathsf{open} \cdot (\mathtt{read} \mid \mathtt{write})^* \cdot \mathtt{close}$ because history traces for allocated file resources evolve to $\mathsf{open} \cdot \mathtt{read}$ but remain unchanged after that.

To formalize the correct use of infinite-lifetime resources, we introduce the notion of *evolution of history traces*. Hereinafter, we write $\overrightarrow{X_n}^{n \in \mathbb{N}}$ for an infinite sequence $X_0, X_1, X_2, \cdots$.

Definition 2.1 (Multi-Step Evaluation). *Multi-step evaluation* $(e, \sigma) \rightsquigarrow^* (e', \sigma')$ is the reflexive, transitive closure of the evaluation relation $\rightsquigarrow$.

Definition 2.2 (Evolution of History Traces). Given a program e, the set $\mathit{Trace}(e)^\infty$ of triples of the form $(\ell, S, \overrightarrow{\varpi_n}^{n\in\mathbb{N}})$ is defined as follows:

$$\begin{aligned}\mathit{Trace}(e)^\infty \stackrel{\text{def}}{=} \{(\ell, S, \overrightarrow{\varpi_n}^{n\in\mathbb{N}}) \mid \exists e', \sigma'.\ (e, \emptyset) \rightsquigarrow^* (e', \sigma') \wedge \ell \notin \mathit{dom}(\sigma') \wedge \\ \exists \overrightarrow{e_n}^{n\in\mathbb{N}}, \overrightarrow{\sigma_n}^{n\in\mathbb{N}}.\ (e', \sigma') \rightsquigarrow (e_0, \sigma_0 \uplus \{\ell \mapsto_S \varpi_0\}) \wedge \\ \forall n \in \mathbb{N}.\ (e_n, \sigma_n \uplus \{\ell \mapsto_S \varpi_n\}) \rightsquigarrow (e_{n+1}, \sigma_{n+1} \uplus \{\ell \mapsto_S \varpi_{n+1}\})\}\ .\end{aligned}$$

Let's see the detail of Definition 2.2. Given a program e, let $(\ell, S, \overrightarrow{\varpi_n}^{n\in\mathbb{N}}) \in \mathit{Trace}(e)^\infty$. Then, there exist some e' and σ' such that $(e, \emptyset) \rightsquigarrow^* (e', \sigma')$ and $\ell \notin \mathit{dom}(\sigma')$. This indicates that the configuration (e', σ') is reachable from the initial configuration $(e, \emptyset)$ and during the execution up to (e', σ'), no resource is allocated at the location ℓ (or, some resource has been allocated at ℓ, but it has been deallocated already). We are also given some $\overrightarrow{e_n}^{n\in\mathbb{N}}$ and $\overrightarrow{\sigma_n}^{n\in\mathbb{N}}$ such that

$$(e', \sigma') \rightsquigarrow (e_0, \sigma_0 \uplus \{\ell \mapsto_S \varpi_0\}) \wedge \forall n \in \mathbb{N}.\ (e_n, \sigma_n \uplus \{\ell \mapsto_S \varpi_n\}) \rightsquigarrow (e_{n+1}, \sigma_{n+1} \uplus \{\ell \mapsto_S \varpi_{n+1}\})\ .$$

Since the location ℓ is not contained in σ', the first conjunct indicates that the expression e' allocates a new resource at the location ℓ, its usage specification is S, and its initial history is ϖ_0 (hence, $\varpi_0 = \epsilon$). The second conjunct says that the execution starting from (e', σ') diverges, it continues to retain the resource at ℓ, and the infinite sequences of the history traces for the resource constructed during the execution match $\overrightarrow{\varpi_n}^{n\in\mathbb{N}}$. Thereby, $(\ell, S, \overrightarrow{\varpi_n}^{n\in\mathbb{N}}) \in \mathit{Trace}(e)^\infty$ indicates how the history traces of the resource res ℓ with the usage specification S are evolved.

Now, we formally define the temporal resource usage analysis problem. We write $(e, \sigma) \rightsquigarrow^\infty$ if the execution from the configuration (e, σ) diverges. That is,

$$(e, \sigma) \rightsquigarrow^\infty \stackrel{\text{def}}{=} \exists \overrightarrow{e_n}^{n\in\mathbb{N}}, \overrightarrow{\sigma_n}^{n\in\mathbb{N}}.\ (e, \sigma) = (e_0, \sigma_0) \wedge \forall n \in \mathbb{N}.\ (e_n, \sigma_n) \rightsquigarrow (e_{n+1}, \sigma_{n+1})\ .$$

Definition 2.3 (Temporal Resource Usage Analysis). A program e is *temporal resource usage correct* if it satisfies the following criteria:

Finite-Lifetime Criterion The execution of the program e terminates or diverges, that is, either of the following holds: $\exists v, \sigma.\ (e, \emptyset) \rightsquigarrow^* (v, \sigma)$; or $(e, \emptyset) \rightsquigarrow^\infty$.

Indefinite-Lifetime Criterion If the execution of the program e terminates, the final resource heap is discardable, that is, $\forall v, \sigma.\ (e, \emptyset) \rightsquigarrow^* (v, \sigma) \Longrightarrow \models^\dagger \sigma$.

Infinite-Lifetime Criterion For any $(\ell, S, \overrightarrow{\varpi_i}^{i\in\mathbb{N}}) \in \mathit{Trace}(e)^\infty$, either of the following holds:

- The resource res ℓ is never accessed after reaching some desired state, that is, there exist some $n \in \mathbb{N}$ and finite trace $\varpi \in S$ such that $\forall m \geq n.\ \varpi = \varpi_m$; or
- The resource res ℓ continues to be accessed infinitely and the limit of its history traces conforms to the usage specification, that is, there exists some infinite trace $\pi \in S$ such that $\forall \varpi \in \mathit{pref}(\{\pi\}).\ \exists n \in \mathbb{N}.\ \varpi = \varpi_n$.

The finite- and indefinite-lifetime criterion are formulated as discussed above. In fact, these criteria are exactly the same as the ones considered in the resource usage analysis problem [22]. For example, a program only executing the expression $e_{\texttt{file1}}$ in Section 2.1.1 or $e_{\texttt{file3}}$ in Section 2.1.2, which only creates finite- or indefinite-lifetime resources, meets the criteria. In contrast, a program executing the example $e_{\texttt{file4}}$ in Section 2.1.2 does not meet the finite-lifetime criterion, and a program let $x = \mathsf{new}_{\Psi_{\texttt{file}}}$ in $\mathsf{acc}_{\mathsf{open}}(x)$; () does not meet the indefinite-lifetime criterion.

To illustrate how the infinite-lifetime criterion works, we consider two example programs.

Example 2.4 (Temporal Resource Usage Correct Example with Infinite-Lifetime Resources). The first example simplifies the valid program presented in Figure 1, given as follows:

$$e_{\texttt{valid}} \stackrel{\text{def}}{=} \text{let } f = (\text{rec } f\ x.\ \text{let } y = e_{\texttt{file}} \text{ in let } x = e_{\texttt{lock}} \text{ in } f\ x) \text{ in let } x = \text{new}_{\Psi_{\texttt{lock}}} \text{ in } f\ x$$

where

$$\begin{aligned} e_{\texttt{file}} &\stackrel{\text{def}}{=} \text{let } y = \text{new}_{\Psi_{\texttt{file}}} \text{ in } \text{acc}_{\texttt{open}}(y); \text{acc}_{\texttt{read}}(y); \text{acc}_{\texttt{close}}(y) \\ e_{\texttt{lock}} &\stackrel{\text{def}}{=} \text{acc}_{\texttt{acquire}}(x); e; \text{acc}_{\texttt{release}}(x) \end{aligned}$$

with some unit-returning expression e representing a critical section (which we suppose that meets the criteria). For each file resource created during the execution of this program, $\mathit{Trace}(e_{\texttt{valid}})^{\infty}$ tells that its history traces are evolved as

$$\epsilon,\ \texttt{open},\ \texttt{open} \cdot \texttt{read},\ \texttt{open} \cdot \texttt{read} \cdot \texttt{close},\ \texttt{open} \cdot \texttt{read} \cdot \texttt{close},\ \texttt{open} \cdot \texttt{read} \cdot \texttt{close},\ \cdots .$$

Because $\texttt{open} \cdot \texttt{read} \cdot \texttt{close} \in [\![\Psi_{\texttt{file}}]\!]$, the file resource meets the infinite-lifetime criterion. For the lock resource created before calling the function f, $\mathit{Trace}(e_{\texttt{valid}})^{\infty}$ tells that its history traces are evolved as

$$\epsilon,\ \texttt{acquire},\ \texttt{acquire} \cdot \texttt{release},\ \texttt{acquire} \cdot \texttt{release} \cdot \texttt{acquire},\ (\texttt{acquire} \cdot \texttt{release})^2,\ \cdots .$$

Because $(\texttt{acquire} \cdot \texttt{release})^{\omega} \in [\![\Psi_{\texttt{file}}]\!]$ and any finite prefix of $(\texttt{acquire} \cdot \texttt{release})^{\omega}$ is contained in the evolution, the lock resource meets the infinite-lifetime criterion. Furthermore, the program satisfies the other criteria. Thus, it is temporal resource usage correct.

Example 2.5 (Temporal Resource Usage Incorrect Example with Infinite-Lifetime Resources). The second example alters the one in Example 2.4 to call `close` after the recursive call, given as follows:

$$\begin{aligned} e_{\texttt{invalid}} \stackrel{\text{def}}{=}\ & \text{let } f = (\text{rec } f\ x.\ \text{let } y = e_{\texttt{open}\cdot\texttt{read}} \text{ in let } x = e_{\texttt{lock}} \text{ in } f\ x; e_{\texttt{close}}) \text{ in} \\ & \text{let } x = \text{new}_{\Psi_{\texttt{lock}}} \text{ in } f\ x \end{aligned}$$

where

$$\begin{aligned} e_{\texttt{open}\cdot\texttt{read}} &\stackrel{\text{def}}{=} \text{let } y = \text{new}_{\Psi_{\texttt{file}}} \text{ in } \text{acc}_{\texttt{open}}(y); \text{acc}_{\texttt{read}}(y) \\ e_{\texttt{close}} &\stackrel{\text{def}}{=} \text{acc}_{\texttt{close}}(y); \text{drop}(y)\ . \end{aligned}$$

Note that the expression $e_{\texttt{lock}}$ is the same as the one given in Example 2.4. For each file resource created during the execution of this program, $\mathit{Trace}(e_{\texttt{invalid}})^{\infty}$ tells that its history traces are evolved as

$$\epsilon,\ \texttt{open},\ \texttt{open} \cdot \texttt{read},\ \texttt{open} \cdot \texttt{read},\ \texttt{open} \cdot \texttt{read},\ \cdots$$

since `close` is applied after the recursive call returns, although it never returns. Because $\texttt{open} \cdot \texttt{read} \notin [\![\Psi_{\texttt{file}}]\!]$, the program does not meet the infinite-lifetime criterion and is therefore not temporal resource usage correct.

3 Technical Overview

This section provides an overview of the challenges involved in solving the temporal resource usage analysis problem—especially, ensuring the infinite-lifetime criterion—using type systems, along with our solutions. Specifically, we identify three main challenges: (1) resource aliasing, (2) progressivity guarantee, and (3) termination analysis. In the following subsections, we describe each challenge and our corresponding solution.

3.1 Resource Aliasing

Aliasing is a classic yet central issue in the verification of stateful programs [10, 49]. Since the resources in our language are stateful, we must address the problem of *resource aliasing*. For example, consider a function $v \stackrel{\text{def}}{=} \lambda x.\, \lambda y.\, \mathsf{acc}_{\mathtt{close}}(x); \mathsf{drop}(x); \mathsf{acc}_{\mathtt{close}}(y); \mathsf{drop}(y)$. If the same file resource is passed to both x and y, the resource is accessed (at $\mathsf{acc}_{\mathtt{close}}(y)$) after deallocated (at $\mathsf{drop}(x)$). In contrast, when distinct file resources are passed, and they are both ready to be closed, the call to v safely terminates. Therefore, to effectively reason about programs with stateful resources, it is crucial to track the aliasing of resources.

The aliasing problem has been extensively studied in the literature [10, 25, 32, 34, 37, 49, 54]. Among various established approaches, we adopt *uniqueness typing* [6, 16, 48] due to its simplicity and generality. Uniqueness typing is a variant of linear typing that guarantees each resource has a unique reference, thus forbidding the aliasing of resources. We discuss how we can support aliasing of resources in Section 6.1.

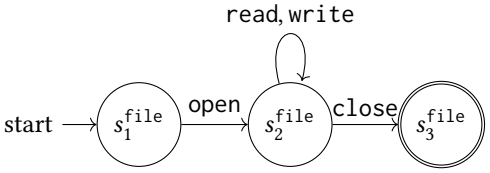
3.2 Progressivity Guarantee

Progressivity guarantee is the critical problem arising in ensuring the infinite-lifetime criterion. To see why it is necessary, consider the following program:

$$e_1 \stackrel{\text{def}}{=} \mathsf{let}\, f = (\mathsf{rec}\, f\, x.\, \mathsf{acc}_{\mathtt{read}}(x); f\, x)\, \mathsf{in}\, \mathsf{let}\, x = \mathsf{new}_{\Psi_{\mathtt{file}}}\, \mathsf{in}\, \mathsf{acc}_{\mathsf{open}}(x); f\, x\,.$$

This program first creates and opens a file resource and then passes it to the recursive function f, which continues to access the resource via `read` infinitely. It does not satisfy the temporal specification $\Psi_{\mathtt{file}}$ because the created file resource will be never closed.

The crux of the problem is that the resource's state does not *progress* even though the infinite execution of the program does. To see it in more detail, consider the finite automaton presented on the right, which represents the file specification $\Psi_{\mathtt{file}}$. With this automaton-based view, in the program e_1, the state of the created file resource at the point immediately before the recursive call is $s_2^{\mathtt{file}}$, and it is then stuck in this non-accepting state: although the execution does not stuck and progresses infinitely by repeatedly calling $\mathsf{acc}_{\mathtt{read}}(x)$, the resource state enters $s_2^{\mathtt{file}}$ again and again with a self-loop.



Our rationale for identifying this issue as problematic is that the absence of progressivity guarantees on resources hinders ensuring that the resources reach certain desired states—namely, the accepting states in the automaton representations of their specifications. At its core, the infinite-lifetime criterion requires that a resource eventually reach a desired state: either the resource should be left unaccessed forever in a desired state, or, over the course of the execution, it should visit the desired states infinitely such that the limit of the history traces at these states matches some trace in the usage specification. Thus, *the lack of the progressivity guarantee makes it difficult to ensure the infinite-lifetime criterion.*

Conversely, *by ensuring that the states of resources progress along with the execution, we can guarantee that they reach the desired states.* For example, consider the following program:

$$\begin{aligned} e_2 \stackrel{\text{def}}{=}\ & \mathsf{let}\, f = (\mathsf{rec}\, f\, (x, y).\, \mathsf{acc}_{\mathsf{open}}(x); \mathsf{acc}_{\mathtt{close}}(y); \mathsf{let}\, z = \mathsf{new}_{\Psi_{\mathtt{file}}}\, \mathsf{in}\, f\, (z, x))\, \mathsf{in} \\ & \mathsf{let}\, z_1 = \mathsf{new}_{\Psi_{\mathtt{file}}}\, \mathsf{in}\, \mathsf{let}\, z_2 = \mathsf{new}_{\Psi_{\mathtt{file}}}\, \mathsf{in}\, \mathsf{acc}_{\mathsf{open}}(z_2); f\, (z_1, z_2) \end{aligned}$$

where we use the notation $\mathsf{rec}\, f\, (x, y).\, e$ to denote $\mathsf{rec}\, f\, z.\, \mathsf{let}\, (x, y) = z\, \mathsf{in}\, e$ for some fresh z. This program first creates two file resources z_1 and z_2 and then calls the recursive function f with them. The states of the resources passed to f progress along with the execution—specifically, every time the function f is recursively called. For the first argument resource x, its state at the time of the call is supposed to be $s_1^{\mathtt{file}}$. Since it is accessed via `open`, its state is changed to $s_2^{\mathtt{file}}$. Finally, it is passed to the recursive call as the second argument. For the second argument y, its state at the time of the call is supposed to be $s_2^{\mathtt{file}}$. Since it is accessed via `close`, its state is changed to $s_3^{\mathtt{file}}$. Then it is left unaccessed forever in the *desired state* $s_3^{\mathtt{file}}$.

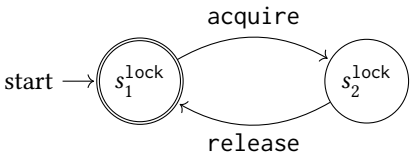
Based on this observation, we introduce the notion of *timers* to the type representation of resources as a technical device to guarantee the progressivity of resource states towards the desired states. Specifically, in our type system, the types of resources take the form Res_{Ψ}^{m} accompanied by a timer m, which is a natural number actually, and a temporal specification Ψ; we call such a type a *temporal resource type*. Because, as seen above, recursive functions are the source of obfuscating the progressivity guarantee, timers cooperate with recursive function calls. Once a resource of a type Res_{Ψ}^{m} is passed to a recursive function, the value of the timer m is decreased—namely, the timer represents the "potential" of how many times the resource can be passed to recursive functions. The timer is a natural number, so it must be nonnegative. Therefore, it disallows the resource to be passed to recursive functions infinitely many times. This mechanism enables rejecting the problematic example e_1 as it passes the resource x to the recursive function infinitely. In contrast, in the example e_2, each created resource is passed to the recursive function only twice. Therefore, we can assign a inital timer of 2 to these resources, to show they reach the desired state eventually.

However, only allowing timers to be decreased is not very useful. For example, consider the following program with lock resources:

$$e_3 \stackrel{\text{def}}{=} \mathsf{let}\, f = (\mathsf{rec}\, f\, x.\, \mathsf{acc}_{\mathtt{acquire}}(x); e; \mathsf{acc}_{\mathtt{release}}(x); f\, x)\, \mathsf{in}\, \mathsf{let}\, x = \mathsf{new}_{\Psi_{\mathtt{lock}}}^{m}\, \mathsf{in}\, f\, x$$

where we assume that the expression e is terminating and does not reference the lock resource x. Since the temporal specification $\Psi_{\mathtt{lock}}$ for lock resources accepts the infinite usage $(\mathtt{acquire} \cdot \mathtt{release})^{\omega}$, this program should be accepted, although the same resource x is passed to the recursive function infinitely many times.

Our idea to address this issue is that, since we introduce timers to ensure that resources reach certain desired states, *we can allow resetting the timer of a resource when it is in a desired state*. For instance, the automaton for lock resources is given on the right.[5] It illustrates that the timer of a lock resource can be reset if it has not been accessed yet or the last operation applied to it is `release`. For a resource v in a desired state,[6] we allow applying a *reset* construct $\mathsf{reset}^{m}(v)$, which resets the timer of the resource v to a new timer m. Using this mechanism, we can rewrite the example e_3 to the following one, which our type system accepts:



$$\mathsf{let}\, f = (\mathsf{rec}\, f\, x.\, \mathsf{acc}_{\mathtt{acquire}}(x); e; \mathsf{acc}_{\mathtt{release}}(x); \mathsf{reset}^{1}(x); f\, x)\, \mathsf{in}\, \mathsf{let}\, x = \mathsf{new}_{\Psi_{\mathtt{lock}}}^{1}\, \mathsf{in}\, f\, x\,.$$

The initial and reset timer is 1, which indicates that the lock resource reaches the desired state $s_1^{\mathtt{lock}}$ for each recursive call.

[5] This is an extended Büchi automaton (defined in Hofmann and Chen [21]) to allow both finite, e.g., $(\mathtt{acquire} \cdot \mathtt{release})^{*}$, and infinite, e.g., $(\mathtt{acquire} \cdot \mathtt{release})^{\omega}$, traces to be accepted by the same automaton.

[6] Our type system tracks the current state of a resource by reusing the temporal specification Ψ in its type.

$$
\begin{array}{llll}
\textbf{Timers} & m, n \in \mathbb{N} & \textbf{Termination Annotations} & \zeta ::= \lightning \mid ? \\
\textbf{Patterns} & P & ::= & x \mid () \mid (P_1, P_2) \mid \,!\, P \mid \dagger\, P \\
\textbf{Values} & v & ::= & x \mid c \mid \mathsf{res}\,\ell \mid (v_1, v_2) \mid \,!\, v \mid \dagger\, v \mid \mathsf{next}(v) \mid \lambda^{\zeta}\, x.\, e \mid \mathsf{rec}^{\zeta}\, f\, x.\, e \\
\textbf{Expressions} & e & ::= & v \mid v_1 \mathbin{\hat{\oplus}} v_2 \mid \mathsf{let}\, P = e_1\, \mathsf{in}\, e_2 \mid \mathsf{if0}\, v\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \mid v_1\, v_2 \mid \\
 & & & \mathsf{new}^{m}_{\Psi} \mid \mathsf{acc}_a(v) \mid \mathsf{drop}(v) \mid \mathsf{dup}(v) \mid \\
 & & & \mathsf{reset}^{m}(v) \mid \mathsf{later}(e) \mid v_1 \circledast v_2 \mid \mathsf{prev}(v) \mid e :: \zeta
\end{array}
$$

Fig. 4. Syntax of $\lambda^{\textcircled{\scriptsize t}}_{\mathrm{res}}$ (Extends Figure 2).

3.3 Termination Analysis

To see the importance of termination analysis, let us revisit the program example $e_{\mathtt{invalid}}$ given in Example 2.5. Recall its definition:

$$
\begin{array}{rcl}
e_{\mathtt{invalid}} & \stackrel{\mathrm{def}}{=} & \mathsf{let}\, f = (\mathsf{rec}\, f\, x.\, \mathsf{let}\, y = e_{\mathtt{open{\cdot}read}}\, \mathsf{in}\, \mathsf{let}\, x = e_{\mathtt{lock}}\, \mathsf{in}\, f\, x; e_{\mathtt{close}})\, \mathsf{in} \\
 & & \mathsf{let}\, x = \mathsf{new}_{\Psi_{\mathtt{lock}}}\, \mathsf{in}\, f\, x \\
e_{\mathtt{open{\cdot}read}} & \stackrel{\mathrm{def}}{=} & \mathsf{let}\, y = \mathsf{new}_{\Psi_{\mathtt{file}}}\, \mathsf{in}\, \mathsf{acc}_{\mathsf{open}}(y); \mathsf{acc}_{\mathsf{read}}(y) \\
e_{\mathtt{close}} & \stackrel{\mathrm{def}}{=} & \mathsf{acc}_{\mathtt{close}}(y); \mathsf{drop}(y)\ .
\end{array}
$$

If we *assume* that the recursive call $f\ x$ diverges, we can conclude that this program should not be temporal resource usage correct because the resource x would not be in the desired state at that point. Although assuming that recursive function calls always diverge enables sound reasoning, it is often too conservative. For example, consider the following recursive function:

$$\mathsf{rec}\, f\, n.\, \mathsf{if0}\, n \le 0\, \mathsf{then}\, ()\, \mathsf{else}\, \mathsf{let}\, y = e_{\mathtt{open{\cdot}read}}\, \mathsf{in}\, f\, (n-1); e_{\mathtt{close}}\ .$$

This function terminates for any integer n, so each created file resource reaches the desired state and may therefore be discarded by drop. This example indicates that precise reasoning about temporal resource usage correctness often relies on precise termination analysis.

Termination analysis is a fundamental problem in computer science and has been extensively and still actively studied for decades, ranging from first-order programs [15, 30] to higher-order ones [20, 29, 46].

Rather than incorporating some specific termination analysis method to the type system, we assume that programs are annotated to indicate whether expressions terminate. Given a program equipped with such termination annotations and well-typed in our type system, we prove that the program is temporal resource usage correct, *provided that the termination annotations are correct.*

4 Temporal Resource Type System

Based on the ideas described in Section 3, we formally define our temporal resource type system in this section. Because our type system requires new constructs that work as type annotations, we first introduce an extension $\lambda^{\textcircled{\scriptsize t}}_{\mathrm{res}}$ of the language λ_{res} given in Section 2.1, and then present our type system for $\lambda^{\textcircled{\scriptsize t}}_{\mathrm{res}}$.

4.1 Extended Language $\lambda^{\textcircled{\scriptsize t}}_{\mathrm{res}}$

This section defines the syntax and semantics of the extended language $\lambda^{\textcircled{\scriptsize t}}_{\mathrm{res}}$ for temporal resource usage analysis.[7]

[7]The $\textcircled{\scriptsize t}$ symbol in the name of $\lambda^{\textcircled{\scriptsize t}}_{\mathrm{res}}$ hints the addition of timers to the language.

4.1.1 Syntax. The syntax of $\lambda_{\text{res}}^{\circlearrowright}$ is presented in Figure 4. Throughout the paper, constructs or rules unchanged from λ_{res} are shaded (or omitted); those extended with annotations are grayboxed; and newly introduced ones are shown with no special styling. In the remainder of this section, we explain how the extensions and modifications address each of the solutions described in Section 3.

Uniqueness Typing. Uniqueness typing is a type-based methodology to enforce that there are only unique references to mutable values [6, 16, 32, 48]. We introduce uniqueness in the style of the language L^3 [34]. Our extended language $\lambda_{\text{res}}^{\circlearrowright}$ is built atop uniqueness typing in that every *unannotated* value is enforced to be only referenced in a unique way. For a value v that retains no resource, we allow multiple references by annotating it with the !-constructor $!\, v$; we call such a value v *unrestricted*. Such an annotated value can be duplicated manually using duplication constructs dup. We also introduce another kind of annotation constructs $\dagger\, v$, which state that the value v is *discardable*. Naturally, we require all the resources involved in a discardable value v to be discardable, i.e., already used-up. By allowing deallocation (through drop) only for discardable values, our type system ensures the finite-lifetime criterion for well-typed programs. Note that, whereas unrestricted values are discardable, the converse does not hold. For example, while released lock resources are discardable, the references to them have to be unique as they may be acquired again. Patterns are also extended with $!\, P$ and $\dagger\, P$ to extract unrestricted and discardable values from these constructs, respectively.

Temporal Resource Types with Timers. To implement temporal resource types with timers, we introduce a new construct $\mathsf{reset}^m(v)$, which allows resetting timers, and equip resource allocations with initial timers new_Ψ^m.

We also have to track when resources are passed to recursive functions because their timers should be decreased then. One simple approach is doing it pure syntactically—namely, having the reduction of an application $(\mathsf{rec}\, f\, x.\, e)\, \mathsf{res}\, \ell$ decrease the timer of the resource $\mathsf{res}\, \ell$. However, this approach is not very flexible. For instance, consider an application $(\mathsf{rec}\, f\, x.\, \lambda y.\, e)\, ()\, \mathsf{res}\, \ell$, where, semantically, the resource $\mathsf{res}\, \ell$ is passed to the recursive function, while, literally, it is passed to the λ-abstraction $\lambda y.\, e$. Thus, if we adopted this purely syntactic approach, we would need to impose heavy or unnatural constraints on recursive functions (e.g., disallowing recursive functions that returns λ-abstractions taking resources).

Instead of such a syntactic approach, we employ a type-based mechanism, inspired by *later modalities* [35]. Specifically, we introduce a *later type* $\blacktriangleright T$, which is assigned to computations caused by recursive functions (T is a type representing the resulting value of the computation). This invariant is enforced by the following two typing disciplines: first, in typechecking a recursive function $\mathsf{rec}\, f\, x.\, e$, our type system assigns a function type $\blacktriangleright(T_1 \to T_2)$ to the self-referential variable f;[8] second, a function of a type $\blacktriangleright(T_1 \to T_2)$ can be applied to a value of the type $\blacktriangleright T_1$ and the application has the type $\blacktriangleright T_2$. As can be seen from the second discipline, for passing a value to a computation of a later type, our type system requires converting the type T of a value to the later type $\blacktriangleright T$. Intuitively, this conversion determines the point when the value is passed to a recursive function. Based on this observation, we introduce constructs $\mathsf{later}(e)$ and $\mathsf{next}(v)$, which make expressions and values, respectively, of later types. The semantics of $\lambda_{\text{res}}^{\circlearrowright}$ ensures that resources passed to e or v are used after their timers are decreased. We also introduce later applications in the form $v_1 \circledast v_2$, which applies the "later" function v_1 to the "later" value v_2. Furthermore, as the prior work on later types [4, 13, 45], we allow converting a type $\blacktriangleright T$ of a later value to the underlying type T through a construct prev, provided that the later value is not (nor does not capture) a recursive

[8] Function types in our type system take a different form to support additional typing information, as shown shortly.

function; we check the side condition on the type T. Note that this type-based approach imposes no syntactic restriction on recursive functions, unlike the aforementioned syntactic approach.

Termination Annotations. As described in Section 3.3, to annotate whether expressions terminate, $\lambda^{\circledcirc}_{\text{res}}$ supports constructs with termination annotations. A termination annotation ζ is either the terminating annotation $\lightning$, which indicates that expressions terminate, or unknown annotation ?, which indicates that expressions may not terminate. These annotations are attached to expressions by ascription $e :: \zeta$, λ-abstractions ($\lambda^{\zeta}\, x.\, e$), or recursive functions ($\text{rec}^{\zeta}\, f\, x.\, e$). The annotations in the last two constructs are of the function bodies.

$$\textbf{Heaps} \quad \sigma \quad ::= \quad \{\ell_1 \mapsto^{m_1}_{S_1} \varpi_1, \ldots, \ell_n \mapsto^{m_n}_{S_n} \varpi_n\}$$

Value Pattern Matching $\boxed{[v/P]}$

$$[!\, v/!\, P] \stackrel{\text{def}}{=} [v/P] \qquad [\dagger\, v/\dagger\, P] \stackrel{\text{def}}{=} [v/P]$$

Heap Timer Count-Down $\boxed{\sigma^{-m}}$

$$\sigma^{-m} \stackrel{\text{def}}{=} \{\ell \mapsto^{m_0 - m}_{S} \varpi \mid \ell \mapsto^{m_0}_{S} \varpi \in \sigma\}$$

Heap Timer Count-Up $\boxed{\sigma^{+m}}$

$$\sigma^{+m} \stackrel{\text{def}}{=} \{\ell \mapsto^{m_0 + m}_{S} \varpi \mid \ell \mapsto^{m_0}_{S} \varpi \in \sigma\}$$

Pure Reductions Rules $\boxed{e_1 \rightsquigarrow e_2}$

$$(\lambda^{\zeta}\, x.\, e)\, v \rightsquigarrow \{x \mapsto v\}(e) :: \zeta$$

$$(\text{rec}^{\zeta}\, f\, x.\, e)\, v \rightsquigarrow (\{f \mapsto \text{next}(!\, \text{rec}^{\zeta}\, f\, x.\, e)\} \uplus \{x \mapsto v\})(e) :: \zeta$$

$$v :: \zeta \rightsquigarrow v \qquad \text{dup}(v) \rightsquigarrow (v, v)$$

$$\text{next}(v_1) \circledast \text{next}(v_2) \rightsquigarrow \text{later}(v_1\, v_2) \qquad \text{prev}(\text{next}(v)) \rightsquigarrow v$$

Evaluation Rules $\boxed{(e_1, \sigma_1) \rightsquigarrow (e_2, \sigma_2)}$

$$\frac{[\![\Psi]\!] \neq \emptyset}{(\text{new}^{m}_{\Psi}, \sigma) \rightsquigarrow (\text{res}\, \ell, \sigma \uplus \{\ell \mapsto^{m}_{[\![\Psi]\!]} \epsilon\})} \qquad \frac{(e_1, \sigma \uplus {\sigma_1}^{-1}) \rightsquigarrow (e_2, \sigma \uplus \sigma_2) \quad dom(\sigma_1) = loc(e_1)}{(\text{later}(e), \sigma \uplus \sigma_1) \rightsquigarrow (\text{later}(e_2), \sigma \uplus {\sigma_2}^{+1})}$$

$$(\text{reset}^{m}(\text{res}\, \ell), \sigma \uplus \{\ell \mapsto^{m'}_{S} \varpi\}) \rightsquigarrow (\text{res}\, \ell, \sigma \uplus \{\ell \mapsto^{m}_{S} \varpi\}) \qquad \frac{(e, \sigma) \rightsquigarrow (e', \sigma')}{(e :: \zeta, \sigma) \rightsquigarrow (e' :: \zeta, \sigma')}$$

Fig. 5. Operational Semantics of $\lambda^{\circledcirc}_{\text{res}}$ (Excerpt).

4.1.2 Semantics. We present (an excerpt of) the operational semantics of $\lambda^{\circledcirc}_{\text{res}}$ in Figure 5. Resource heaps associate resource locations with their timers in addition to usage specifications and history traces. The pattern matching allows extracting values from new constructs. Note that we cannot freely extract the argument value v from a value construct $\text{next}(v)$ of a later type.

The pure reduction rules given in Figure 5 address the altered or newly added expression constructs that behave independently of resource heaps. Function applications are altered to annotate the application results with the termination annotations attached to the functions in order to indicate that the annotations are given by the user. Furthermore, after the reduction,

recursive functions are wrapped by the next and !-constructs. This is because recursive functions are of a later type, as discussed above, and may be called multiple times via recursive calls. For an termination ascription $e :: \zeta$, once the expression evaluates to a value v, the termination annotation ζ is taken away because the termination of e has been ensured. An expression $\mathsf{later}(v)$ of a later type reduces to the value $\mathsf{next}(v)$ of the later type. The reductions for duplication, later applications, and applications of prev are self-explanatory and follow the prior work [13, 34, 45].

The evaluation rules address the constructs whose behavior depends on resource heaps. The evaluation rule for resource allocations is modified to initialize the timer of the allocated resource with a given one. Reset constructs allow resetting the timer of a given resource. The evaluation of other resource operations (acc, drop) leaves the timers untouched (see the supplementary material for the formal rules). The evaluation of a later expression $\mathsf{later}(e)$ under a resource heap σ_0 proceeds by evaluating the expression e. Because the expression e performs the computation caused by a recursive function call, the timers of the resources referenced by e has to be decreased in evaluating e. In contrast, when the scope of later escapes, the remaining values of the timers are increased. We call these decreasing and increasing of the timer values *count-down* and *count-up*, respectively. Count-down and count-up on timers in resource heaps are defined in Figure 5. Note that, since timers are natural numbers, σ^{-m} is undefined if the timer of some resource in the heap σ is less than m.

Example 4.1 (Example Evaluation: Run Out of Timers). This example illustrates how timers count-down when the recursive calls happen. For brevity, we ignore termination annotations installed during evaluation. Consider the following program $e_{\mathtt{main}}$:

$$\begin{array}{rcl} e_{\mathtt{main}} & \stackrel{\text{def}}{=} & \mathsf{let}\, f = v_f \,\mathsf{in}\, \mathsf{let}\, x = \mathsf{new}^1_\Psi \,\mathsf{in}\, f\, x \\ v_f & \stackrel{\text{def}}{=} & \mathsf{rec}^?\, f\, x.\, \mathsf{let}\, y = \mathsf{apnext}\, f \,\mathsf{in}\, y\, \mathsf{next}(x) \end{array}$$

where

$$\begin{array}{rcl} \mathsf{unwrap} & \stackrel{\text{def}}{=} & \mathsf{next}(\lambda^{\lightning} f.\, \mathsf{let}\, !f = f \,\mathsf{in}\, f) \\ \mathsf{apnext} & \stackrel{\text{def}}{=} & \lambda^{\lightning} f.\, \lambda^{\lightning} x.\, \mathsf{let}\, y = \mathsf{unwrap} \circledast f \,\mathsf{in}\, y \circledast x\,. \end{array}$$

Intuitively, the functionality of unwrap is to unwrap a !-value into a normal value under a next-constructor. And apnext is used in the body of a recursive function to apply a recursive occurrence to a next-value.

The evaluation of $e_{\mathtt{main}}$ proceeds as follows (the choice of the resource location is arbitrary):

$$\begin{array}{ll} & (e_{\mathtt{main}}, \emptyset) \\ \rightsquigarrow^* & (v_f\ (\mathsf{res}\,\ell), \{\ell \mapsto^1_{[\![\Psi]\!]} \epsilon\}) \\ \rightsquigarrow^* & (\mathsf{let}\, y = \mathsf{apnext}\ (\mathsf{next}(!\, v_f))\,\mathsf{in}\, y\, \mathsf{next}(\mathsf{res}\,\ell), \{\ell \mapsto^1_{[\![\Psi]\!]} \epsilon\}) \\ \rightsquigarrow^* & (\mathsf{next}(v_f) \circledast \mathsf{next}(\mathsf{res}\,\ell), \{\ell \mapsto^1_{[\![\Psi]\!]} \epsilon\}) \\ \rightsquigarrow^* & (\mathsf{later}(v_f\ (\mathsf{res}\,\ell)), \{\ell \mapsto^1_{[\![\Psi]\!]} \epsilon\}) \\ \rightsquigarrow^* & (\mathsf{later}(\mathsf{later}(v_f\ (\mathsf{res}\,\ell))), \{\ell \mapsto^1_{[\![\Psi]\!]} \epsilon\}) \not\rightsquigarrow\ . \end{array}$$

It might seem that the evaluation of $e_{\mathtt{main}}$ diverges, but it is actually stuck at this point. Although timers are not counted down by looking at the resource heap in the main evaluation, it is counted down during evaluation of the sub-program inside later-constructors. Our timer assignment of 1 allows the resource to occur only inside a single later-constructor; when more are introduced, evaluation gets stuck.

$$
\begin{array}{lrcl}
\textbf{Basic Types} & B & ::= & \mathsf{Unit} \mid \mathsf{Int} \mid \dots \\
\textbf{First-Order Types} & \iota & ::= & B \mid \mathsf{Res}^{m}_{\Psi} \mid !\,\iota \mid \dagger\,\iota \mid \blacktriangleright\,\iota \mid \iota \otimes \iota \\
\textbf{Value Types} & T & ::= & B \mid \mathsf{Res}^{m}_{\Psi} \mid !\,T \mid \dagger\,T \mid \blacktriangleright\,T \mid T \otimes T \mid T \multimap^{m} C \\
\textbf{Computation Types} & C & ::= & T \,\&\, \zeta \\
\textbf{Typing Contexts} & \Gamma & ::= & \{x_1 : T_1, \dots, x_n : T_n\} \\
\textbf{Store Typing Contexts} & \Sigma & ::= & \{\ell_1 : \mathsf{Res}^{m_1}_{\Psi_1}, \dots, \ell_n : \mathsf{Res}^{m_n}_{\Psi_n}\} \\
\\
\textbf{Finite-Trace Sets} & s & \subseteq & \mathbb{A}^{*} \\
\textbf{Finite Specifications} & \phi & ::= & s \\
\textbf{Infinite Specifications} & \psi & ::= & \{\rho_1, \dots, \rho_n\} \\
\textbf{Lassos} & \rho & ::= & \langle s_{\mathrm{init}}, s_{\mathrm{rep}} \rangle \\
\textbf{Temporal Specifications / Usage Prophecies} & & & \\
 & \Psi & ::= & \langle \phi, \psi \rangle
\end{array}
$$

Fig. 6. Syntax of Types.

4.2 Types

The syntax of types, given in Figure 6, consists of *value types* T and *computation types* C, which are used to type values and expressions, respectively. A value type is: a basic type B for constants; a temporal resource type Res^{m}_{Ψ} for resources; an of-course type $!\,T$, which is assigned to unrestricted values of the type T; a *discardable* type $\dagger\,T$, which is assigned to discardable values of the type T; a product type $T \otimes T$; or a function type $T \multimap^{m} C$, where m is no greater than the minimum timer among those of resources captured by functions of this type.[9] A computation type is composed of a value type T and a termination *effect* ζ, which describes the values produced by the expressions of the computation type (if any) and their termination behavior, respectively. Typing contexts Γ and store typing contexts Σ associate variables and resource locations, respectively, with their types.

Now, we also formally define temporal specifications, as in Figure 6, to exploit them in our type system. Here, a *temporal specification* Ψ is a pair of a *finite specification* ϕ and an *infinite specification* ψ, which specify traces on resources generated during their lifetimes. We project Ψ to ϕ and ψ by Ψ^{fin} and Ψ^{inf}, respectively. A finite specification ϕ is a set s of finite traces, determining the finite usage of a resource: when the resource is explicitly discarded by drop or implicitly discarded, its history trace should be in the set s. In contrast, an infinite specification ψ determines the infinite usage of a resource: if a resource remains accessible forever in an infinite execution, the limit of its history traces should be in the interpretation of ψ. In general, ψ is a finite set of the form $\{\langle s_{11}, s_{12}\rangle, \cdots, \langle s_{n1}, s_{n2}\rangle\}$, where each pair $\langle s_{i1}, s_{i2}\rangle$ is called a *lasso*. The interpretation $[\![\psi]\!]$ of such ψ is formulated as below.

Definition 4.2 (Interpretations of Temporal Specifications). The interpretation $[\![\Psi]\!]$ of a temporal specification Ψ, along with that of its infinite specification Ψ^{inf}, is defined as follows:

$$
[\![\langle s_{\mathrm{init}}, s_{\mathrm{rep}}\rangle]\!] \stackrel{\mathrm{def}}{=} s_{\mathrm{init}} \cdot s^{\infty}_{\mathrm{rep}} \qquad [\![\psi]\!] \stackrel{\mathrm{def}}{=} \bigcup_{\rho \in \psi} [\![\rho]\!] \qquad [\![\Psi]\!] \stackrel{\mathrm{def}}{=} \Psi^{\mathsf{fin}} \cup [\![\Psi^{\mathsf{inf}}]\!]
$$

where $s^{\infty} \stackrel{\mathrm{def}}{=} \{\varpi_0 \cdot \varpi_1 \cdot \dots \mid \forall\, i \in \mathbb{N}.\ \varpi_i \in s\} \subseteq \mathbb{A}^{\infty}$ and $s \cdot S \stackrel{\mathrm{def}}{=} \{\varpi \cdot \delta \mid \varpi \in s \wedge \delta \in S\}$.

[9] Instead of attaching a timer m, we can possibly apply alternative techniques to enrich function types with explicit information about captured contexts, as in, e.g., capturing types [8].

Discardable Types $\boxed{\vdash^{\dagger} T}$

$$\frac{}{\vdash^{\dagger} B} \qquad \frac{\epsilon \in \Psi^{\mathsf{fin}}}{\vdash^{\dagger} \mathsf{Res}_{\Psi}^{m}} \qquad \frac{}{\vdash^{\dagger} !\,T} \qquad \frac{}{\vdash^{\dagger} \dagger T} \qquad \frac{\vdash^{\dagger} T}{\vdash^{\dagger} \blacktriangleright T} \qquad \frac{\vdash^{\dagger} T_1 \quad \vdash^{\dagger} T_2}{\vdash^{\dagger} T_1 \otimes T_2}$$

Unrestricted Types $\boxed{\vdash^{!} T}$

$$\frac{}{\vdash^{!} B} \qquad \frac{}{\vdash^{!} !\,T} \qquad \frac{\vdash^{!} T}{\vdash^{!} \blacktriangleright T} \qquad \frac{\vdash^{!} T_1 \quad \vdash^{!} T_2}{\vdash^{!} T_1 \otimes T_2}$$

Type Timer Count-Down $\boxed{T^{-m}}$

$$\begin{aligned}
B^{-m} &\stackrel{\text{def}}{=} B \\
(\mathsf{Res}_{\Psi}^{m_0})^{-m} &\stackrel{\text{def}}{=} \mathsf{Res}_{\Psi}^{m_0 - m} \\
(!\,T)^{-m} &\stackrel{\text{def}}{=} !\,T \\
(\dagger\, T)^{-m} &\stackrel{\text{def}}{=} \dagger(T^{-m}) \\
(\blacktriangleright T)^{-m} &\stackrel{\text{def}}{=} \blacktriangleright(T^{-m}) \\
(T_1 \otimes T_2)^{-m} &\stackrel{\text{def}}{=} {T_1}^{-m} \otimes {T_2}^{-m} \\
(T \multimap^{m_0} C)^{-m} &\stackrel{\text{def}}{=} T \multimap^{m_0 - m} C
\end{aligned}$$

Type Timer Count-Up $\boxed{T^{+m}}$

$$\begin{aligned}
B^{+m} &\stackrel{\text{def}}{=} B \\
(\mathsf{Res}_{\Psi}^{m_0})^{+m} &\stackrel{\text{def}}{=} \mathsf{Res}_{\Psi}^{m_0 + m} \\
(!\,T)^{+m} &\stackrel{\text{def}}{=} !\,T \\
(\dagger\, T)^{+m} &\stackrel{\text{def}}{=} \dagger(T^{+m}) \\
(\blacktriangleright T)^{+m} &\stackrel{\text{def}}{=} \blacktriangleright(T^{+m}) \\
(T_1 \otimes T_2)^{+m} &\stackrel{\text{def}}{=} {T_1}^{+m} \otimes {T_2}^{+m} \\
(T \multimap^{m_0} C)^{+m} &\stackrel{\text{def}}{=} T \multimap^{m_0 + m} C
\end{aligned}$$

Fig. 7. Predicates and Operations on Value Types.

Namely, a trace in $[\![\psi]\!]$ is a finite trace in s_{i1} followed by infinite loops of traces in s_{i2}. We adopt this form of infinite specifications because it is both convenient and expressive. For convenience, it enables us to easily identify "desired" states of the resource—i.e., the states that the resource should reach infinitely many times during their infinite usage. We can consider that a resource reaches a desired state if its history trace is $\varpi \cdot \varpi_1 \cdot \cdots \cdot \varpi_m$ where $\varpi \in s_{i1}$ and $\varpi_1, \cdots, \varpi_m \in s_{2i}$ for some lasso $\langle s_{i1}, s_{i2} \rangle$ ($m \geq 0$). By ensuring that the history trace of the resource is evolved to ϖ, $\varpi \cdot \varpi_1$, $\varpi \cdot \varpi_1 \cdot \varpi_2$, $\cdots$ (again, $\varpi \in s_{i1}$, and $\varpi_1, \varpi_2, \cdots \in s_{2i}$) over the course of the execution, we can guarantee that the trace limit is in the interpretation $[\![\psi]\!]$. For expressivity, this form of infinite specifications can express arbitrary ω-regular expressions [40] and ω-context-free-grammars [14]. Although infinite specifications could be represented in a bit more abstract form [45], we think that the current form is more tractable and sufficiently expressive.

Temporal specifications are used not only to specify traces over the entire lifetimes of resources, but also to prescribe valid traces for their *remaining* lifetimes at each program point. While the former usage appears in temporal specifications associated with resource allocations, the latter arises in those associated with temporal resource types. However, the set of valid traces may change over the course of execution. For example, the file specification $\Psi_{\mathtt{file}}$ allows traces beginning with open, but once open is applied, it may not be applied again. Therefore, for precise reasoning, it is necessary to track the valid traces of *current* resources for each program point. In fact, temporal specifications involved in temporal resource types describe valid traces for the *future usage* of resources—rather than the traces over their entire lifetimes. To reflect this nature, we refer to temporal specifications associated with temporal resource types as *usage prophecies*. In Section 4.3, we show how the type system updates usage prophecies as resources are accessed.

4.3 Type System

In this section, we define our temporal resource type system for the language $\lambda_{\mathsf{res}}^{\circledcirc}$.

Typing for Values $\boxed{\Sigma \mid \Gamma \vdash v : T}$

$$\frac{\vdash_{\mathrm{WF}} T}{\emptyset \mid \{x : T\} \vdash x : T}\ \text{T\_Var} \qquad \frac{}{\emptyset \mid \emptyset \vdash c : \mathit{typeof}(c)}\ \text{T\_Const} \qquad \frac{\vdash_{\mathrm{WF}} \mathsf{Res}_\Psi^m}{\{\ell : \mathsf{Res}_\Psi^m\} \mid \emptyset \vdash \mathsf{res}\,\ell : \mathsf{Res}_\Psi^m}\ \text{T\_Res}$$

$$\frac{\emptyset \mid \Gamma \vdash v : T \quad \vdash^! \Gamma}{\emptyset \mid \Gamma \vdash\, !\, v : !\, T}\ \text{T\_Bang} \qquad \frac{\Sigma \mid \Gamma \vdash v : T \quad \vdash^\dagger \Sigma \quad \vdash^\dagger \Gamma}{\Sigma \mid \Gamma \vdash \dagger\, v : \dagger\, T}\ \text{T\_Dis}$$

$$\frac{\Sigma^{-1} \mid \Gamma^{-1} \vdash v : T}{\Sigma \mid \Gamma \vdash \mathsf{next}(v) : \blacktriangleright T}\ \text{T\_Next} \qquad \frac{\Sigma_1 \mid \Gamma_1 \vdash v_1 : T_1 \quad \Sigma_2 \mid \Gamma_2 \vdash v_2 : T_2}{\Sigma_1 \uplus \Sigma_2 \mid \Gamma_1 \uplus \Gamma_2 \vdash (v_1, v_2) : T_1 \otimes T_2}\ \text{T\_Pair}$$

$$\frac{\Sigma^{-m} \mid \Gamma^{-m} \uplus \{x : T_1\} \vdash e : T_2 \,\&\, \zeta}{\Sigma \mid \Gamma \vdash \lambda^\zeta\, x.\, e : T_1 \multimap^m T_2 \,\&\, \zeta}\ \text{T\_Lam} \qquad \frac{T_f \;=\; T_1 \multimap^0 T_2 \,\&\, \zeta \quad \vdash^! \Gamma \qquad \emptyset \mid \Gamma \uplus \{f : \blacktriangleright\, !\, T_f\} \uplus \{x : T_1\} \vdash e : T_2 \,\&\, \zeta}{\emptyset \mid \Gamma \vdash \mathsf{rec}^\zeta\, f\, x.\, e : T_f}\ \text{T\_Rec}$$

$$\frac{\Sigma \mid \Gamma \vdash v : T_1 \quad \vdash T_1 <: T_2 \quad \vdash_{\mathrm{WF}} T_2}{\Sigma \mid \Gamma \vdash v : T_2}\ \text{T\_Sub}$$

Fig. 8. Typing Rules for Values.

We first define certain predicates and operations on value types in Figure 7. The discardable type predicate $\vdash^\dagger T$ and unrestricted type predicate $\vdash^! T$ are used to check if the values of type T can be duplicated and discarded, respectively. Especially, a temporal resource type can be discarded if its current finite usage prophecy accommodates the empty trace, which means that the resources are used-up correctly. The count-down operation T^{-m} (which is partial) and the count-step operation T^{+m} on value types are used to reflect the changes on timers in the semantics to the type system. Note that $(!\,T)^{\pm m} = !\,T$ because the values of the type $!\,T$ retain no resource. We use the convention that predicates and operations on value types are applied to typing contexts and store typing contexts point-wise, thereby obtaining predicates $\vdash^! \Gamma$, $\vdash^\dagger \Gamma$, and $\vdash^\dagger \Sigma$.

We also define *type well-formedness*, written $\vdash_{\mathrm{WF}} T$ and $\vdash_{\mathrm{WF}} C$, which ensure that all the resource types involved in the type T and C have well-formed usage prophecies. A usage prophecy Ψ is *well-formed* if (1) for any lasso ρ in Ψ^{inf}, $[\![\rho]\!]$ is nonempty (note that Ψ^{inf} can be empty), and (2) $[\![\Psi]\!]$ is nonempty. To save space, the full definition of type well-formedness is omitted in the paper and can be found in the supplementary material.

Now, we give define the type system, which consists of typing judgments $\Sigma \mid \Gamma \vdash v : T$ for values and $\Sigma \mid \Gamma \vdash e : C$ for expressions, as well as subtyping judgments $\vdash T_1 <: T_2$ and $\vdash C_1 <: C_2$. We first explain the typing rules for values and expressions, and then the subtyping rules.

Value Typing. The typing rules for values are presented in Figure 8. The rules for variables, constants, resources, applications of arithmetic operations, and pairs are standard. We assume the metafunction *typeof*, which returns the basic type of a given constant. For unrestricted and discardable values, the corresponding typing rules (T_Bang) and (T_Dis) require the captured contexts to be unrestricted and discardable, respectively. Note that any resource is not unrestricted. For a "later" value $\mathsf{next}(v)$, the argument v is typechecked under the contexts where all the timers are counted down (T_Next) because the timers involved in v are counted down during evaluation inside a later-computation. In the typing of a λ-abstraction (T_Lam), all the captured resources *delegate* m counts in their timers to the function type $T \multimap^m C$. Our type system ensures that the λ-abstraction is applied before the delegated counts run out. For example, consider a λ-abstraction that captures a resource with timer 1. For such a λ-abstraction, our type system may give, say, a

Pattern Matching on Value Types $\boxed{[T/P]}$

$$[T/x] \stackrel{\text{def}}{=} \{x : T\} \qquad [\mathsf{Unit}/()] \stackrel{\text{def}}{=} \emptyset \qquad [!\,T/!\,P] \stackrel{\text{def}}{=} [T/P]$$
$$[\dagger\, T/\dagger\, P] \stackrel{\text{def}}{=} [T/P] \qquad [T_1 \otimes T_2/(P_1, P_2)] \stackrel{\text{def}}{=} [T_1/P_1] \uplus [T_2/P_2]$$

Typing for Expressions $\boxed{\Sigma \mid \Gamma \vdash e : C}$

$$\frac{\Sigma \mid \Gamma \vdash v : T}{\Sigma \mid \Gamma \vdash v : T \,\&\, \lightning}\ \text{C\_Val} \qquad \frac{\Sigma \mid \Gamma \vdash e : T \,\&\, \zeta'}{\Sigma \mid \Gamma \vdash e :: \zeta : T \,\&\, \zeta}\ \text{C\_Annot} \qquad \frac{\vdash_{\text{WF}} C_2 \quad \Sigma \mid \Gamma \vdash e : C_1 \quad \vdash C_1 <: C_2}{\Sigma \mid \Gamma \vdash e : C_2}\ \text{C\_Sub}$$

$$\frac{\begin{array}{c}\Sigma_1 \mid \Gamma_1 \vdash e_1 : T_1 \,\&\, \zeta_1 \\ \Sigma_2 \mid \Gamma_2 \uplus [T_1/P] \vdash e_2 : T_2 \,\&\, \zeta_2 \\ \zeta_1 = ? \Longrightarrow \vdash^{\dagger} \Sigma_2 \ \wedge\ \vdash^{\dagger} \Gamma_2\end{array}}{\Sigma_1 \uplus \Sigma_2 \mid \Gamma_1 \uplus \Gamma_2 \vdash \mathsf{let}\, P = e_1 \,\mathsf{in}\, e_2 : T_2 \,\&\, \zeta_1 \triangleright \zeta_2}\ \text{C\_Let} \qquad \zeta_1 \triangleright \zeta_2 \stackrel{\text{def}}{=} \begin{cases} \lightning & \text{if } \zeta_1 = \lightning \text{ and } \zeta_2 = \lightning \\ ? & \text{otherwise} \end{cases}$$

$$\frac{\begin{array}{c}\Sigma_1 \mid \Gamma_1 \vdash v_1 : \mathsf{Int} \\ \Sigma_2 \mid \Gamma_2 \vdash v_2 : \mathsf{Int}\end{array}}{\Sigma_1 \uplus \Sigma_2 \mid \Gamma_1 \uplus \Gamma_2 \vdash v_1 \mathbin{\hat{\oplus}} v_2 : \mathsf{Int} \,\&\, \lightning}\ \text{C\_Op} \qquad \frac{\begin{array}{c}\Sigma_1 \mid \Gamma_1 \vdash v : \mathsf{Int} \\ \Sigma_2 \mid \Gamma_2 \vdash e_1 : C \quad \Sigma_2 \mid \Gamma_2 \vdash e_2 : C\end{array}}{\Sigma_1 \uplus \Sigma_2 \mid \Gamma_1 \uplus \Gamma_2 \vdash \mathsf{if0}\ v\,\mathsf{then}\ e_1\,\mathsf{else}\ e_2 : C}\ \text{C\_IfZero}$$

$$\frac{\begin{array}{c}\Sigma_1 \mid \Gamma_1 \vdash v_1 : T \multimap^{m} C \\ \Sigma_2 \mid \Gamma_2 \vdash v_2 : T\end{array}}{\Sigma_1 \uplus \Sigma_2 \mid \Gamma_1 \uplus \Gamma_2 \vdash v_1\, v_2 : C}\ \text{C\_App} \qquad \frac{\begin{array}{c}\Sigma_1 \mid \Gamma_1 \vdash v_1 : \blacktriangleright(T_1 \multimap^{m} T_2 \,\&\, \zeta) \\ \Sigma_2 \mid \Gamma_2 \vdash v_2 : \blacktriangleright T_1\end{array}}{\Sigma_1 \uplus \Sigma_2 \mid \Gamma_1 \uplus \Gamma_2 \vdash v_1 \circledast v_2 : \blacktriangleright T_2 \,\&\, \zeta}\ \text{C\_NextApp}$$

$$\frac{\Sigma^{-1} \mid \Gamma^{-1} \vdash e : T \,\&\, \zeta}{\Sigma \mid \Gamma \vdash \mathsf{later}(e) : \blacktriangleright T \,\&\, \zeta}\ \text{C\_Later} \qquad \frac{\Sigma \mid \Gamma \vdash v : \blacktriangleright \iota}{\Sigma \mid \Gamma \vdash \mathsf{prev}(v) : \iota^{+1} \,\&\, \lightning}\ \text{C\_Prev}$$

$$\frac{\Sigma \mid \Gamma \vdash v : T \quad \vdash^{!} T}{\Sigma \mid \Gamma \vdash \mathsf{dup}(v) : T \otimes T \,\&\, \lightning}\ \text{C\_Dup} \qquad \frac{\Sigma \mid \Gamma \vdash v : T \quad \vdash^{\dagger} T}{\Sigma \mid \Gamma \vdash \mathsf{drop}(v) : \mathsf{Unit} \,\&\, \lightning}\ \text{C\_Drop}$$

$$\frac{\vdash_{\text{WF}} \mathsf{Res}^{m}_{\Psi}}{\emptyset \mid \emptyset \vdash \mathsf{new}^{m}_{\Psi} : \mathsf{Res}^{m}_{\Psi} \,\&\, \lightning}\ \text{C\_New} \qquad \frac{\Sigma \mid \Gamma \vdash v : \mathsf{Res}^{m}_{\Psi} \quad \vdash_{\text{WF}} \mathsf{Res}^{m}_{\Psi - a}}{\Sigma \mid \Gamma \vdash \mathsf{acc}_a(v) : \mathsf{Res}^{m}_{\Psi - a} \,\&\, \lightning}\ \text{C\_Acc}$$

$$\frac{\Sigma \mid \Gamma \vdash v : \mathsf{Res}^{m'}_{\langle \phi, \psi \rangle} \qquad \psi' = \{\langle s_{\text{rep}}, s_{\text{rep}} \rangle \mid \langle s_{\text{init}}, s_{\text{rep}} \rangle \in \psi \wedge \epsilon \in s_{\text{init}}\} \neq \emptyset}{\Sigma \mid \Gamma \vdash \mathsf{reset}^{m}(v) : \mathsf{Res}^{m}_{\langle \phi, \psi' \rangle} \,\&\, \lightning}\ \text{C\_Reset}$$

Fig. 9. Typing Rules for Expressions.

type $T \multimap^{1} C$. A function of this type can be referenced by the value v of a later value $\mathsf{next}(v)$ (if the reference is not under next). However, the value v of a *nested* later value $\mathsf{next}(\mathsf{next}(v))$ cannot refer to the function because $(T \multimap^{1} C)^{-2}$ is undefined. In this way, when a λ-abstraction of a type $T \multimap^{m} C$ is applied, it is ensured that the timers of the resources captured by the λ-abstraction have been decreased by at most m counts. For typing a recursive function $\mathsf{rec}\, f\, x.\, e$ (T_Rec), we disallow capture of resources in the contexts. Furthermore, when resources are passed to the recursive function, their times are decremented. Therefore, given a function type T_f of the recursive function, the type $\blacktriangleright !\, T_f$ is assigned to the self-referential variable f in type checking the body e. Our type system supports subtyping (T_Sub); we defer its presentation to the end of this section.

Expression Typing. The typing rules for expressions are shown in Figure 9. For ascription expressions, we only trust the given annotations (C_Annot). The rule (C_Let) typechecks a let expression let $P = e_1$ in e_2 by splitting given contexts for typing the subexpressions e_1 and e_2, as usual in uniqueness typing [32, 34]; the other typing rules for compound expressions, like (T_App), follow this convention. Because the value of the expression e_1 is deconstructed according to the pattern P, (C_Let) matches P with the type T of the value to make a typing context $[T/P]$ that binds the variables involved in P. Furthermore, the rule also considers the possibility that the expression e_1 diverges; in that case, the resources captured by the expression e_2 would be implicitly discarded. Thus, if the expression e_1 is not trusted to terminate, (C_Let) ensures that the resources captured by e_2 are discardable. The termination effect of the let expression is derived from those of the subexpressions: the let expression is guaranteed to terminate if so are the subexpressions; otherwise, it may diverge. The rule (C_NextApp) is standard for later function applications [4, 13, 33, 35]. The rule (C_Later) decreases the timers of captured resources for the same reason as (T_Next). The rule (C_Prev) allows removing the outermost later type constructor ▶ from first-order values. The timers of the resources involved in the values are counted up because they escape the scope of next (see the reduction rule for prev in Figure 5). Duplication and deallocation are only allowed for unrestricted and discardable values ((C_Dup) and (C_Drop)). The rule (C_Acc) means that, given a resource access $\mathrm{acc}_a(v)$, the event a is filtered out from the set of the valid traces for the remaining lifetime of the resource v. This nature is expressed via the *filter-out operation* $\Psi^{-\varpi}$, which is defined as follows:[10]

Definition 4.3 (Filter-Out Operation). The filter-out operation for lassos, infinite specifications, and usage prophecies is defined as follows:

$$s^{-\varpi} \stackrel{\text{def}}{=} \{\varpi' \mid \varpi \cdot \varpi' \in s\} \qquad \langle s_{\text{init}}, s_{\text{rep}}\rangle^{-\varpi} \stackrel{\text{def}}{=} \langle {s_{\text{init}}}^{-\varpi}, s_{\text{rep}}\rangle$$
$$\psi^{-\varpi} \stackrel{\text{def}}{=} \{\, \rho^{-\varpi} \mid \rho \in \psi \wedge [\![\rho^{-\varpi}]\!] \neq \emptyset \,\} \qquad \langle \phi, \psi\rangle^{-\varpi} \stackrel{\text{def}}{=} \langle \phi^{-\varpi}, \psi^{-\varpi}\rangle\ .$$

The rule (C_Reset) ensures that the timer of a resource is reset only when it is in some desired state. Recall that, given a resource with a temporal specification Ψ at allocation, it reaches desired states when its history trace is one of $\varpi, \varpi \cdot \varpi_1, \varpi \cdot \varpi_1 \cdot \varpi_2, \cdots$ for some lasso $\langle s_{\text{init}}, s_{\text{rep}}\rangle \in \Psi^{\text{inf}}$ and finite traces $\varpi \in s_{\text{init}}$ and $\varpi_1, \varpi_2, \cdots \in s_{\text{rep}}$. Because events that have been emitted are removed from the usage prophecy of the resource, the lasso in the usage prophecy at the point when the history trace is ϖ takes the form $\langle s_{\text{init}}, s_{\text{rep}}\rangle$, where s_{init} must include the empty trace. Furthermore, by replacing the lasso $\langle s_{\text{init}}, s_{\text{rep}}\rangle$ with $\langle s_{\text{rep}}, s_{\text{rep}}\rangle$, we can ensure that the lasso becomes $\langle s_{\text{rep}}, s_{\text{rep}}\rangle$ again when the history trace becomes $\varpi \cdot \varpi_1$—i.e., the resource reaches a desired state. Therefore, in general, we allow resetting the timer of a resource if its usage prophecy involves a lasso of the form $\langle s_{\text{init}}, s_{\text{rep}}\rangle$ (where s_{init} must include the empty trace), and after the reset, we replace the lasso by $\langle s_{\text{rep}}, s_{\text{rep}}\rangle$. This makes sure that the timer is reset only when the resource reaches a desired state.

Subtyping. We define subtyping for computation types, value types, usage prophecies, and lassos, as in Figure 10. For computation types, the supertype can involve the terminating effect only if the subtype also does. In the value subtyping, timers can be underapproximated because the underapproximation only allows values involving resources to be passed to recursive functions less times. The other type constructors are covariant in terms of subtyping, omitted in the paper. The basic idea of subtyping for usage prophecies is that $\vdash \Psi_1 <: \Psi_2$ holds if $[\![\Psi_2]\!] \subseteq [\![\Psi_1]\!]$. However, it requires two more things: if a resource reaches a desired state under the super-prophecy Ψ_2, it should also reach a desired state under the super-prophecy Ψ_1; and if the resource reaches the

[10] In the supplementary material, the definition is slightly generalized to prove the value inversion lemma in a nice form.

Subtyping Rules $\boxed{\vdash C_1 <: C_2}$ $\boxed{\vdash T_1 <: T_2}$ $\boxed{\vdash \Psi_1 <: \Psi_2}$ $\boxed{\vdash \rho_1 <: \rho_2}$

$$\frac{\vdash T_1 <: T_2 \quad \zeta_2 = \lightning \Longrightarrow \zeta_1 = \lightning}{\vdash T_1 \,\&\, \zeta_1 <: T_2 \,\&\, \zeta_2}$$

$$\frac{m_2 \leq m_1 \quad \vdash \Psi_1 <: \Psi_2}{\vdash \mathsf{Res}_{\Psi_1}^{m_1} <: \mathsf{Res}_{\Psi_2}^{m_2}} \qquad \frac{\vdash T_2 <: T_1 \quad m_2 \leq m_1 \quad \vdash C_1 <: C_2}{\vdash T_1 \multimap^{m_1} C_1 <: T_2 \multimap^{m_2} C_2}$$

$$\frac{{\Psi_2}^{\mathsf{fin}} \subseteq {\Psi_1}^{\mathsf{fin}} \quad \forall \rho_2 \in \psi_2.\, \exists \rho_1 \in \psi_1.\, \vdash \rho_1 <: \rho_2}{\vdash \Psi_1 <: \Psi_2} \qquad \frac{S_{\mathrm{init2}} \subseteq S_{\mathrm{init1}} \cdot {S_{\mathrm{rep1}}}^* \quad S_{\mathrm{rep2}} \subseteq {S_{\mathrm{rep1}}}^+}{\vdash \langle S_{\mathrm{init1}}, S_{\mathrm{rep1}} \rangle <: \langle S_{\mathrm{init2}}, S_{\mathrm{rep2}} \rangle}$$

Fig. 10. Subtyping Rules (Excerpt).

"next" desired state under Ψ_2, it should also reach the "next" desired state under Ψ_1. The subtyping rules for usage prophecies and lassos ensure these (informal) requirements.

4.4 Typing Examples

In this section, we demonstrate how our type system accepts temporal resource usage correct programs and rejects incorrect ones through some typing examples. More examples are given in the supplementary material.

In the examples presented in the paper, we use the following functions:

$$\begin{aligned} \mathsf{unwrap} \;:\; & \blacktriangleright(!\,T \multimap T \,\&\, \lightning) \stackrel{\mathrm{def}}{=} \mathsf{next}(\lambda^{\lightning} f.\, \mathsf{let}\; !f = f \;\mathsf{in}\; f) \\ \mathsf{ap} \;:\; & \blacktriangleright !(T \multimap^{m} \iota \,\&\, \zeta) \multimap (\blacktriangleright T \multimap^{m} \iota^{+1} \,\&\, \zeta) \,\&\, \lightning \\ & \qquad \stackrel{\mathrm{def}}{=} \lambda^{\lightning} f.\, \lambda^{\lightning} x.\, \mathsf{let}\; y = \mathsf{unwrap} \circledast f \;\mathsf{in}\; \mathsf{let}\; z = y \circledast x \;\mathsf{in}\; \mathsf{prev}(z)\;. \end{aligned}$$

where the entities T, C, ι, ζ, and m can be arbitrary, and the timer 0 assigned to function types is omitted for simplicity.

Example 4.4 (Well-Typed Temporal Resource Usage Correct Example). This well-typed example is a variant of the program given in Example 2.4.

$$e_{\mathsf{valid}} \stackrel{\mathrm{def}}{=} \mathsf{let}\, f = (\mathsf{rec}^{?}\, f\, x.\, e_{\mathsf{body}}) \;\mathsf{in}\; \mathsf{let}\, x = \mathsf{new}^{0}_{\Psi_{\mathtt{lock}}} \;\mathsf{in}\; f\, x$$

where

$$\begin{aligned} e_{\mathsf{body}} &\stackrel{\mathrm{def}}{=} e_{\mathtt{file}}; \mathsf{let}\, x = e_{\mathtt{lock}} \;\mathsf{in}\; \mathsf{ap}\, f\, \mathsf{next}(x) \\ e_{\mathtt{file}} &\stackrel{\mathrm{def}}{=} \mathsf{let}\, y = \mathsf{new}^{0}_{\Psi_{\mathtt{file}}} \;\mathsf{in}\; \mathsf{acc}_{\mathtt{open}}(y); \mathsf{acc}_{\mathtt{read}}(y); \mathsf{acc}_{\mathtt{close}}(y); \mathsf{drop}(y) \\ e_{\mathtt{lock}} &\stackrel{\mathrm{def}}{=} \mathsf{acc}_{\mathtt{acquire}}(x); \mathsf{acc}_{\mathtt{release}}(x); \mathsf{reset}^{1}(x) \\ \Psi_{\mathtt{file}} &\stackrel{\mathrm{def}}{=} \langle \{\mathtt{open}\} \cdot \{\mathtt{read}, \mathtt{write}\}^* \cdot \{\mathtt{close}\}, \emptyset \rangle \\ \Psi_{\mathtt{lock}} &\stackrel{\mathrm{def}}{=} \langle \{\mathtt{acquire} \cdot \mathtt{release}\}^*, \{\langle \{\mathtt{acquire} \cdot \mathtt{release}\}, \{\mathtt{acquire} \cdot \mathtt{release}\} \rangle\} \rangle\;. \end{aligned}$$

It is different from the program in Example 2.4 in that: (1) the file resource is explicitly dropped because of our strict typing rules; (2) it requires a reset operation to reset the timer and check for progressivity for the lock resource, because it continues to be passed to the recursive function within its body; (3) the recursive call is implemented by the ap combinator because the self-referential variable f is guarded by $\blacktriangleright$; and (4) (correct) termination annotation is given to the function.

We can type check the function $\mathsf{rec}^?\, f\, x.\, e_{\mathsf{body}}$ with the function type $T_f \stackrel{\text{def}}{=} \mathsf{Res}^0_{\Psi_{\mathsf{lock}}} \multimap \mathsf{Unit} \,\&\, ?$:

$$\frac{\emptyset \mid \{f : \blacktriangleright\, !\, T_f\} \uplus \{x : \mathsf{Res}^0_{\Psi_{\mathsf{lock}}}\} \vdash e_{\mathsf{body}} : \mathsf{Unit} \,\&\, ?}{\emptyset \mid \emptyset \vdash \mathsf{rec}^?\, f\, x.\, e_{\mathsf{body}} : T_f}\ \text{T\_Rec} \quad .$$

The crucial point in type checking e_{body} is at $\mathsf{reset}^1(x)$ in e_{lock}. There, we have the typing context:

$$\Gamma_{fx} \stackrel{\text{def}}{=} \{f : \blacktriangleright\, !\, T_f\} \uplus \{x : \mathsf{Res}^0_{\Psi_{\mathsf{lock1}}}\}$$

where $\Psi_{\mathsf{lock1}} \stackrel{\text{def}}{=} \langle \{\mathtt{acquire} \cdot \mathtt{release}\}^*, \{\langle\{\epsilon\}, \{\mathtt{acquire} \cdot \mathtt{release}\}\rangle\}\rangle$. The type of x matches the premises of (C_Reset)—especially, the prefix part of the (only) lasso in Ψ_{lock1} contains the empty trace ϵ, which means that a correct finite usage segment is produced: this is because the finite event segment of $\mathtt{acquire} \cdot \mathtt{release}$ has been produced in this iteration. The return type for $\mathsf{reset}^1(x)$ is then $\mathsf{Res}^1_{\Psi_{\mathsf{lock}}}$, with the prefix part regenerated using the repeating part of the lasso. Another crucial point is when $\mathsf{next}(x)$ is type checked: before this point the type assigned to x in the context is $\mathsf{Res}^1_{\Psi_{\mathsf{lock}}}$ (because $\mathsf{reset}^1(x)$ has been called). Thus, $\mathsf{next}(x)$ can be typechecked through (C_Next), and its type is $\blacktriangleright\, \mathsf{Res}^0_{\Psi_{\mathsf{lock}}}$, where the timer is decremented because x under next may be (and *is*) accessed by the computation caused by the recursive call. As a result, the type of $\mathsf{next}(x)$ matches the argument type of the function returned by $\mathsf{ap}\, f$. The remaining type checking is straightforward, and we conclude that the program is well-typed.

Example 4.5 (Ill-Typed Temporal Resource Usage Incorrect Example). This ill-typed example is a variant of the program given in Example 2.5. We make similar changes as in Example 4.4.

$$e_{\mathtt{invalid}} \stackrel{\text{def}}{=} \mathsf{let}\, f = (\mathsf{rec}^?\, f\, x.\, e_{\mathsf{body}})\, \mathsf{in}\, \mathsf{let}\, x = \mathsf{new}^0_{\Psi_{\mathsf{lock}}}\, \mathsf{in}\, f\, x$$

where

$$\begin{aligned} e_{\mathsf{body}} &\stackrel{\text{def}}{=} \mathsf{let}\, y = e_{\mathtt{open}\cdot\mathtt{read}}\, \mathsf{in}\, \mathsf{let}\, x = e_{\mathsf{lock}}\, \mathsf{in}\, \mathsf{ap}\, f\, \mathsf{next}(x); e_{\mathtt{close}} \\ e_{\mathtt{open}\cdot\mathtt{read}} &\stackrel{\text{def}}{=} \mathsf{let}\, y = \mathsf{new}^0_{\Psi_{\mathtt{file}}}\, \mathsf{in}\, \mathsf{acc}_{\mathtt{open}}(y); \mathsf{acc}_{\mathtt{read}}(y) \\ e_{\mathtt{close}} &\stackrel{\text{def}}{=} \mathsf{acc}_{\mathtt{close}}(y); \mathsf{drop}(y)\ . \end{aligned}$$

The expression e_{lock} and the specifications Ψ_{lock} and $\Psi_{\mathtt{file}}$ are the same as those in Example 4.4.

This function body will not type check. The issue is in type checking the sequential composition of $\mathsf{ap}\, f\, \mathsf{next}(x); e_{\mathtt{close}}$. Remember that this is syntactic sugar for $\mathsf{let}\, z = \mathsf{ap}\, f\, \mathsf{in}\, \mathsf{let}\, () = z\, \mathsf{next}(x)\, \mathsf{in}\, e_{\mathtt{close}}$ for some fresh variable z. At this point, we have the following instance of (C_Let):

$$\frac{\emptyset \mid \Gamma_{xz} \vdash z\, \mathsf{next}(x) : T_1 \,\&\, \zeta_1 \qquad \emptyset \mid \Gamma_y \vdash e_{\mathtt{close}} : T_2 \,\&\, \zeta_2 \qquad \zeta_1 = ? \Longrightarrow \vdash^\dagger \Gamma_y}{\emptyset \mid \Gamma_{xz} \uplus \Gamma_y \vdash \mathsf{let}\, () = z\, \mathsf{next}(x)\, \mathsf{in}\, e_{\mathtt{close}} : T_2 \,\&\, \zeta_1 \triangleright \zeta_2}\ \text{C\_Let}$$

where

$$\begin{aligned} \Gamma_{xz} &\stackrel{\text{def}}{=} \{x : \mathsf{Res}^1_{\Psi_{\mathsf{lock}}}\} \uplus \{z : \blacktriangleright\, \mathsf{Res}^0_{\Psi_{\mathsf{lock}}} \multimap \mathsf{Unit} \,\&\, ?\} \\ \Gamma_y &\stackrel{\text{def}}{=} \{y : \mathsf{Res}^0_{\Psi_{\mathsf{opened}}}\} \\ \Psi_{\mathsf{opened}} &\stackrel{\text{def}}{=} \langle \{\mathtt{read}, \mathtt{write}\}^* \cdot \{\mathtt{close}\}, \emptyset\rangle\ . \end{aligned}$$

It is easy to see that $\zeta_1 = ?$ since the function z has the return computation type $\mathsf{Unit} \,\&\, ?$. This requires us to check the premise $\vdash^\dagger \Gamma_y$, i.e., $\vdash^\dagger \mathsf{Res}^0_{\Psi_{\mathsf{opened}}}$. However, this requirement does not hold: ${\Psi_{\mathsf{opened}}}^{\mathsf{fin}} = \{\mathtt{read}, \mathtt{write}\}^* \cdot \{\mathtt{close}\}$ does not contain the empty trace ϵ, that is, the resource is not used-up yet. Therefore, the program is ill-typed. This example illustrates we correctly detect the violation of the infinite-lifetime criterion due to implicit discarding.

Note that, if the termination annotation given to the recursive function *were* $\lightning$, the type system would accept the program. This reliance on termination annotations allow effective reasoning

about programs where *terminating* computations involve recursive functions, such as the correct program presented in Section 3.3, which is similar to $e_{\texttt{invalid}}$ but the recursive function terminates and hence the file resource will be closed eventually. The soundness of our type system, shown in Section 5, rests on the correctness of the given termination annotations.

5 Soundness

In this section, we establish the soundness of our type system. We prove that any well-typed program correctly uses resources according to the criteria defined in Section 2.2. We present three main theorems, one for each category of resource lifetimes: finite, indefinite, and infinite.

The soundness for finite-lifetime and indefinite-lifetime resources almost directly follows from syntactic type safety, since our semantics does the checking for us.

Theorem 5.1 (Soundness: Finite-Lifetime Resources). *For any well-typed program e such that $\emptyset \mid \emptyset \vdash e : B \,\&\, \zeta$, the finite-lifetime criterion given in Section 2.2 is satisfied, i.e.,*

$$\exists\, v, \sigma.\ (e, \emptyset)\ \rightsquigarrow^*\ (v, \sigma)\ \textit{or}\ (e, \emptyset) \rightsquigarrow^{\infty}.$$

Proof Sketch. By the standard progress and preservation lemmas for our type system, which together establish ordinary syntactic type safety [55]. □

Theorem 5.2 (Soundness: Indefinite-Lifetime Resources). *For any well-typed program e such that $\emptyset \mid \emptyset \vdash e : B \,\&\, \zeta$, the indefinite-lifetime criterion given in Section 2.2 is satisfied, i.e.,*

$$\forall\, v, \sigma.\ (e, \emptyset)\ \rightsquigarrow^*\ (v, \sigma) \implies \models^{\dagger} \sigma.$$

Proof Sketch. The indefinite-lifetime criterion requires the resource heap to be discardable upon termination. By the preservation lemma, a terminating execution results in a well-typed configuration $\Sigma \vdash (v, \sigma) : B \,\&\, \zeta$ for some store typing Σ (see the supplementary material for the formal definition of well-typed configurations). We can finish by using a lemma stating that for any value configuration (v, σ) such that $\Sigma \vdash (v, \sigma) : T \,\&\, \zeta$, if the value type is discardable, i.e. , $\vdash^{\dagger} T$ (which all basic types satisfy), the accompanying heap is also discardable, i.e., $\models^{\dagger} \sigma$. □

The soundness for infinite-lifetime resources is more subtle, as it requires us to reason about the potential infinite usage of resources in divergent executions. Our main observation is that infinite-lifetime resources have only two possible usage behaviors under our type system: (1) being implicitly discarded, i.e., not accessible from the diverging part of the computation; or (2) being passed to reset infinitely. For the first case, we can separate the computation into some divergent expression e and its continuation such that the resource of interest only appears in the continuation. Then, our type system—more specifically, the rule (C_Let) and the assumption of the correctness of termination annotations—ensures that the resource is discardable, even implicitly. For the second case, we use a logical relation (deferred to supplementary material for its definition) to show that the divergent computation is caused by infinitely nested calls of recursive functions. Combined with the fact that our semantics decreases the timers for resources used inside later, we can then show that the resource is passed to reset infinitely. This is because the timers cannot be counted down infinitely—they are natural numbers. Thereby, the resource has to be reset eventually and infinitely many times, which enables us to conclude that the resource satisfies the infinite-lifetime criterion. We now present the statement of the soundness for infinite-lifetime resources.

Assumption 5.3 (Correctness of Termination Annotation). For the user-provided program e, its sub-evaluations under termination annotation contexts will not diverge, i.e.,

$$\forall E, e', \sigma.\ (e, \emptyset)\ \rightsquigarrow^*\ (E[e' :: \lightning], \sigma) \implies (e', \sigma) \not\rightsquigarrow^{\infty}$$

where E is an *evaluation context* with the hole $\square$, defined as:

$$E \;::=\; \square \mid \mathsf{let}\, P = E \,\mathsf{in}\, e \mid E :: \zeta \mid \mathsf{later}(E)\ .$$

Theorem 5.4 (Soundness: Infinite-Lifetime Resources). *For any well-typed program e such that $\emptyset \mid \emptyset \vdash e : B \,\&\, \zeta$, the infinite-lifetime criterion given in Section 2.2 is satisfied, i.e., for any $(\ell, S, \overrightarrow{\varpi_i}^{i \in \mathbb{N}}) \in \mathit{Trace}(e)^\infty$, either of the following holds:*

- *There exist some $n \in \mathbb{N}$ and finite trace $\varpi \in S$ such that $\forall\, m \geq n.\ \varpi = \varpi_m$;*
- *There exists some infinite trace $\pi \in S$ such that $\forall\, \varpi \in \mathit{pref}(\{\pi\}).\ \exists\, n \in \mathbb{N}.\ \varpi = \varpi_n$.*

6 Discussion

This section describes the current limitations of our type system.

6.1 Support for Resource Aliasing

Our type system relies on uniqueness typing to support sound *strong update* of static resource usage information, i.e., usage prophecies and timers. However, the current form of uniqueness typing eliminates aliasing entirely, which would limit practical applicability. Various techniques have been proposed in the literature to support strong update in the presence of aliasing [1, 9, 17, 18, 34, 37, 49]. We leave extending our temporal resource type system with these more advanced typing mechanisms as future work.

6.2 Value-Dependent Timers

Our type system currently uses constant natural numbers as timers in temporal resource types. This limits expressiveness: some correct programs become untypable. For example, consider the function $v_{\mathtt{main}}$ below:

$$\begin{aligned} v_{\mathtt{readn}} &\stackrel{\text{def}}{=} \mathsf{rec}\, f\ (n, x).\ \mathsf{if0}\ n \leq 0\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{acc}_{\mathtt{read}}(x); f\ (n-1, x) \\ v_{\mathtt{main}} &\stackrel{\text{def}}{=} \lambda n.\ \mathsf{let}\ x = \mathsf{new}^{?}_{\Psi_{\mathtt{file}}}\ \mathsf{in}\ v_{\mathtt{readn}}\ (n, x); \mathsf{acc}_{\mathtt{close}}(x); \mathsf{drop}(x)\ . \end{aligned}$$

The recursive function $v_{\mathtt{readn}}$ reads a given file resource x at most n times (zero times when n is negative). The main function $v_{\mathtt{main}}$ takes an argument n and calls $v_{\mathtt{readn}}$ with a newly opened file resource x and n as input, and finalizes the file resource x after this invocation. Evidently, the function is resource usage correct, because the file resource is finite-lifetime and produces a finite trace of $\mathtt{open} \cdot \mathtt{read}^n \cdot \mathtt{close}$.

However, we cannot type the function with the current type system: no concrete number suffices to be put in the placeholder ? for the timer in the allocation. Ideally, we would like a value-dependent timer, i.e., $\mathsf{new}^{n}_{\Psi_{\mathtt{file}}}$, where n is the argument of the main function $v_{\mathtt{main}}$. We believe that it is possible to enrich our type system with value dependency using standard refinement typing approaches [42, 51, 56], and leave it as future work.

6.3 Implementation

As seen in the examples given in Section 4.4, to verify a program in our language currently requires a lot of manual annotations, including explicit manipulation of unrestricted, discardable, and later values, the use of reset constructs with timers, etc.

Ideally, we would like a surface language where programs only require minimal annotations to verify temporal resource usage correctness, and other information is inferred by the type checker. Also, the choice of static representation of finite trace sets used in usage prophecies (e.g., regular expressions, context-free grammars, logical formulas, etc.) could lead to a trade-off between

expressiveness and automation of the verification process [52, 53]. We leave surface language design and automated type checker implementation as future work.

7 Related Work

Type-Based Methods for Resource Usage Analysis. Type-based approaches for the resource usage analysis has been extensively studied for decades [1, 2, 7, 17, 19, 22, 23, 43, 50]. Similar to our setting, these approaches target languages with dynamic allocation of resources, where each resource maintains a separate trace during its lifetime. The common feature of these approaches is that they verify per-resource traces by enriching the types of resources with static usage information. For example, in typestate systems [7], usage protocols are encoded as finite state machines embedded in the types of resources, e.g., a file resource type could be refined into two states, e.g., `File[opened]` and `File[closed]`, to distinguish whether the file has been closed. Our type system can similarly track resource states using usage prophecies (c.f. Section 3.2).

Another common feature is that, for modular reasoning in the presence of aliasing, these systems apply substructural typing techniques, e.g., linear, uniqueness, or ordered typing, to support strong update of tracked resource states. Safety of resource usage is guaranteed by checking the resource's state associated with the type before an effectful operation applies on the resource, and updating the state afterwards. For example, in typestate systems, when a file is closed, its type changes from `File[open]` to `File[closed]`. Our work can be aligned with this line of research as we adopt uniqueness typing to forbid aliasing of resources, although our type system aims to verify a more general problem, the temporal resource usage analysis. To the best of our knowledge, no existing approach supports reasoning about potential infinite usage of resources. This limitation prevents checking the correctness of infinite-lifetime resources. For example, it is not clear how to ensure a file is eventually closed, or the lock is eventually released whenever it is acquired. This is the most important difference between our type system and the existing ones.

Temporal Effect Systems. Temporal effect systems [21, 28, 36, 44, 45] verify temporal properties of a global trace shared by an entire program, recording all the effectful operations performed during the program execution. Therefore, temporal effect systems can only reason about per-resource traces in a coarse-grained manner.

These systems reason about the infinite behaviors of a program by identifying the *pattern of divergence*, i.e., the loop structures that may be repeated infinitely. Special typing rules for loop constructs, like recursive functions, manifest the loop structures as effects, thereby enabling sound temporal reasoning. For example, Nanjo et al. [36] utilize a greatest-fixpoint operator in the typing rule for recursive functions to capture the potential infinite computation brought by infinitely many recursive calls. In recent work, Sekiyama and Unno [45] generalize temporal effect systems to support recursive types. The presence of recursive types makes identifying the pattern of divergence harder, because loop structures are no longer lexical from the use of certain language constructs, e.g., recursive functions. To address this challenge, Sekiyama and Unno introduce a later modality [35] to their temporal effect system, and use it to delimit the stages of infinitely-unfolded computations in temporal effects. We follow Sekiyama and Unno's approach to add a later modality to our language, to make the progressive nature of divergent computation explicit, serving as a semantic foundation of the timer mechanism.

Higher-Order Model Checking. Higher-order model checking [27, 38] (or HOMC for short), an extension of *model checking* [11, 12] to higher-order programs, is a promising approach to fully automated temporal verification of higher-order programs (only with finite data domains). Kobayashi [26] showed that HOMC can encode the resource usage analysis. It would be thus an interesting future work to explore whether HOMC can also encode the temporal resource usage analysis. On

the other hand, the encoding of the resource usage analysis in HOMC relies on global program transformation. Therefore, while the encoding works well for *closed* programs, it is left unclear how it can be applied to *open* programs. In contrast, whereas our temporal resource type system requires type annotations, it can also verify open programs using type interfaces. Investing a balance between automated and modular verification is also valuable to be explored.

Type-Based Productivity Guarantee. Programming in dependent type theory with coinductive objects requires *productivity*, i.e., each element of a coinductive object must be produced in finite time [47]. An established type-based approach to guaranteeing the productivity is to guard recursion by a later modality [4, 13, 33, 35]. This technique has also been proven useful for verifying the functional correctness of higher-order programs with coinductive objects [24], as well as liveness properties of global traces [45]. In fact, the progressivity referred to in this paper can be regarded as the productivity of traces on resources, and the present work can be seen as an application of the type-based productivity guarantee to the verification of the infinite behavior of stateful objects.

References

[1] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. 2003. Checking and inferring local non-aliasing. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, 129–140. doi:10.1145/781131.781146

[2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 1015–1022. doi:10.1145/1639950.1640073

[3] Bowen Alpern and Fred B. Schneider. 1987. Recognizing Safety and Liveness. *Distributed Comput.* 2, 3 (1987), 117–126. doi:10.1007/BF01782772

[4] Robert Atkey and Conor McBride. 2013. Productive coprogramming with guarded recursion. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 197–208. doi:10.1145/2500365.2500597

[5] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2021. Diamonds are not forever: liveness in reactive programming with guarded recursion. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. doi:10.1145/3434283

[6] Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Math. Struct. Comput. Sci.* 6, 6 (1996), 579–612. doi:10.1017/S0960129500070109

[7] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 301–320. doi:10.1145/1297027.1297050

[8] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.* 45, 4 (2023), 21:1–21:52. doi:10.1145/3618003

[9] Arthur Charguéraud and François Pottier. 2008. Functional translation of a calculus of capabilities. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 213–224. doi:10.1145/1411204.1411235

[10] David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1998, Vancouver, British Columbia, Canada, October 18-22, 1998*, Bjørn N. Freeman-Benson and Craig Chambers (Eds.). ACM, 48–64. doi:10.1145/286936.286947

[11] Edmund M. Clarke and E. Allen Emerson. 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981 (Lecture Notes in Computer Science, Vol. 131)*, Dexter Kozen (Ed.). Springer, 52–71. doi:10.1007/BFB0025774

[12] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). 2018. *Handbook of Model Checking*. Springer.

[13] Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2015. Programming and Reasoning with Guarded Recursion for Coinductive Types. In *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9034)*, Andrew M. Pitts (Ed.). Springer, 407–421. doi:10.1007/978-3-662-46678-0_26

[14] Rina S. Cohen and Arie Y. Gold. 1977. Theory of omega-Languages. I. Characterizations of omega-Context-Free Languages. *J. Comput. Syst. Sci.* 15, 2 (1977), 169–184. doi:10.1016/S0022-0000(77)80004-4

[15] Michael Colón and Henny Sipma. 2002. Practical Methods for Proving Program Termination. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2404)*, Ed Brinksma and Kim Guldstrand Larsen (Eds.). Springer, 442–454. doi:10.1007/3-540-45657-0_36

[16] Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. 2007. Uniqueness Typing Simplified. In *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5083)*, Olaf Chitil, Zoltán Horváth, and Viktória Zsók (Eds.). Springer, 201–218. doi:10.1007/978-3-540-85373-2_12

[17] Robert DeLine and Manuel Fähndrich. 2001. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, Michael Burke and Mary Lou Soffa (Eds.). ACM, 59–69. doi:10.1145/378795.378811

[18] Manuel Fähndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 13–24. doi:10.1145/512529.512532

[19] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 36, 4 (2014), 12:1–12:44. doi:10.1145/2629609

[20] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. 2005. Proving and Disproving Termination of Higher-Order Functions. In *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3717)*, Bernhard Gramlich (Ed.). Springer, 216–231. doi:10.1007/11559306_12

[21] Martin Hofmann and Wei Chen. 2014. Abstract interpretation from Büchi automata. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 51:1–51:10. doi:10.1145/2603088.2603127

[22] Atsushi Igarashi and Naoki Kobayashi. 2002. Resource usage analysis. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, John Launchbury and John C. Mitchell (Eds.). ACM, 331–342. doi:10.1145/503272.503303

[23] Atsushi Igarashi and Naoki Kobayashi. 2005. Resource usage analysis. *ACM Trans. Program. Lang. Syst.* 27, 2 (2005), 264–313. doi:10.1145/1057387.1057390

[24] Guilhem Jaber and Colin Riba. 2021. Temporal Refinements for Guarded Recursive Types. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 548–578. doi:10.1007/978-3-030-72019-3_20

[25] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. doi:10.1145/3158154

[26] Naoki Kobayashi. 2009. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 416–428. doi:10.1145/1480881.1480933

[27] Naoki Kobayashi. 2013. Model Checking Higher-Order Programs. *J. ACM* 60, 3 (2013), 20:1–20:62. doi:10.1145/2487241.2487246

[28] Eric Koskinen and Tachio Terauchi. 2014. Local Temporal Reasoning. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS '14)*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 59:1–59:10. doi:10.1145/2603088.2603138

[29] Takuya Kuwahara, Tachio Terauchi, Hiroshi Unno, and Naoki Kobayashi. 2014. Automatic Termination Verification for Higher-Order Functional Programs. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 392–411. doi:10.1007/978-3-642-54833-8_21

[30] Chin Soon Lee. 2009. Ranking functions for size-change termination. *ACM Trans. Program. Lang. Syst.* 31, 3 (2009), 10:1–10:42. doi:10.1145/1498926.1498928

[31] Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Inf. Comput.* 185, 2 (2003), 182–210. doi:10.1016/S0890-5401(03)00088-9

[32] Danielle Marshall, Michael Vollmer, and Dominic Orchard. 2022. Linearity and Uniqueness: An Entente Cordiale. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the*

European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13240), Ilya Sergey (Ed.). Springer, 346–375. doi:10.1007/978-3-030-99336-8_13

[33] Rasmus Ejlers Møgelberg. 2014. A type theory for productive coprogramming via guarded recursion. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 71:1–71:10. doi:10.1145/2603088.2603132

[34] Greg Morrisett, Amal J. Ahmed, and Matthew Fluet. 2005. L^3: A Linear Language with Locations. In *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3461)*, Pawel Urzyczyn (Ed.). Springer, 293–307. doi:10.1007/11417170_22

[35] Hiroshi Nakano. 2000. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 255–266. doi:10.1109/LICS.2000.855774

[36] Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. 2018. A Fixpoint Logic and Dependent Effects for Temporal Property Verification. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 759–768. doi:10.1145/3209108.3209204

[37] Hiromi Ogawa, Taro Sekiyama, and Hiroshi Unno. 2025. Thrust: A Prophecy-Based Refinement Type System for Rust. *Proc. ACM Program. Lang.* 9, PLDI, Article 230 (June 2025), 25 pages. doi:10.1145/3729333

[38] C.-H. Luke Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 81–90. doi:10.1109/LICS.2006.38

[39] Susan S. Owicki and Leslie Lamport. 1982. Proving Liveness Properties of Concurrent Programs. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 455–495. doi:10.1145/357172.357178

[40] Dominique Perrin and Jean-Eric Pin. 2004. *Infinite words - automata, semigroups, logic and games*. Pure and applied mathematics series, Vol. 141. Elsevier Morgan Kaufmann.

[41] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 46–57. doi:10.1109/SFCS.1977.32

[42] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. doi:10.1145/1375581.1375602

[43] Hannes Saffrich, Yuki Nishida, and Peter Thiemann. 2024. Law and Order for Typestate with Borrowing. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1475–1503. doi:10.1145/3689763

[44] Taro Sekiyama and Hiroshi Unno. 2023. Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations. *Proc. ACM Program. Lang.* 7, POPL (2023), 2079–2110. doi:10.1145/3571264

[45] Taro Sekiyama and Hiroshi Unno. 2025. Algebraic Temporal Effects: Temporal Verification of Recursively Typed Higher-Order Programs. *Proc. ACM Program. Lang.* 9, POPL (2025), 2306–2336. doi:10.1145/3704914

[46] Damien Sereni and Neil D. Jones. 2005. Termination Analysis of Higher-Order Functional Programs. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3780)*, Kwangkeun Yi (Ed.). Springer, 281–297. doi:10.1007/11575467_19

[47] Ben A. Sijtsma. 1989. On the Productivity of Recursive List Definitions. *ACM Trans. Program. Lang. Syst.* 11, 4 (1989), 633–649. doi:10.1145/69558.69563

[48] Sjaak Smetsers, Erik Barendsen, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. 1993. Guaranteeing Safe Destructive Updates Through a Type System with Uniqueness Information for Graphs. In *Graph Transformations in Computer Science, International Workshop, Dagstuhl Castle, Germany, January 1993, Proceedings (Lecture Notes in Computer Science, Vol. 776)*, Hans Jürgen Schneider and Hartmut Ehrig (Eds.). Springer, 358–379. doi:10.1007/3-540-57787-4_23

[49] Frederick Smith, David Walker, and J. Gregory Morrisett. 2000. Alias Types. In *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1782)*, Gert Smolka (Ed.). Springer, 366–381. doi:10.1007/3-540-46425-5_24

[50] Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171. doi:10.1109/TSE.1986.6312929

[51] Hiroshi Unno, Yuki Satake, and Tachio Terauchi. 2018. Relatively complete refinement type system for verification of higher-order non-deterministic programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 12:1–12:29. doi:10.1145/3158100

[52] Hiroshi Unno, Tachio Terauchi, Yu Gu, and Eric Koskinen. 2023. Modular Primal-Dual Fixpoint Logic Solving for Temporal Verification. *Proc. ACM Program. Lang.* 7, POPL (2023), 2111–2140. doi:10.1145/3571265

[53] Moshe Y. Vardi. 1988. A Temporal Fixpoint Calculus. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and Peter

Mager (Eds.). ACM Press, 250–259. doi:10.1145/73560.73582

[54] Guannan Wei, Oliver Bracevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 393–424. doi:10.1145/3632856

[55] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. doi:10.1006/inco.1994.1093

[56] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 214–227. doi:10.1145/292540.292560

[57] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2024. Mostly Automated Verification of Liveness Properties for Distributed Protocols with Ranking Functions. *Proc. ACM Program. Lang.* 8, POPL (2024), 1028–1059. doi:10.1145/3632877