Temporal Resource Typing (Extended Abstract)

Enriching Substructural Typing for Liveness Reasoning

Yiyuan Cao
Peking University
Beijing, China
cyy9447@pku.edu.cn

Taro Sekiyama
National Institute of Informatics
Tokyo, Japan
tsekiyama@acm.org

1 Introduction

Programs use a variety of *resources*—such as heap memory, files, network sockets, and locks. These resources are *stateful* objects that must be used in a valid manner. 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 are *safety* properties, which require that resource operations are applied to resources in some appropriate states. The task of verifying safety of resource usage is called the *resource usage analysis problem* [9] and has been studied actively for decades [1, 4, 7–9, 12, 14].

However, safety is not the only aspect of correct use of resources: the other aspect is *liveness*, which requires resources to eventually reach some desired states. For example, consider the program below, which is an interactive file logger:

```
1  let rec main_loop log_lock =
2    let path = input () in
3    if path = "EXIT" then () else (
4      let input_file = open path in
5      let content = read input_file in
6      close input_file;
7      acquire log_lock;
8      ... (* write content to a log *)
9      release log_lock;
10     main_loop log_lock)
11 let main () =
12   let log_lock = new_lock () in
       main_loop log_lock
```

The program may run indefinitely (it terminates only when the user types `"EXIT"`). For such a potentially diverging program to be correct, it is desirable to ensure liveness properties of resources it creates—the opened files should eventually be 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.

In this short paper, we focus on the problem of verifying temporal properties of resource usage. We call this problem *temporal resource usage analysis*, extending resource usage analysis [9, 10] with consideration of liveness properties.

2 Language $\lambda_{\mathbf{res}}$

Variables x, f **Locations** ℓ **Events** a **Integers** n

Values	$v ::= x \mid n \mid () \mid \mathsf{res}\,\ell \mid \mathsf{rec}\,f\,x.\,e$
Expressions	$e ::= v \mid v_1\,v_2 \mid \mathsf{new}_\Psi \mid \mathsf{acc}_a(v) \mid \mathsf{drop}(v)$
	$\mid \mathsf{let}\,x = e_1\,\mathsf{in}\,e_2 \mid \mathsf{if0}\,v\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2$

Our language λ_{res} is a call-by-value λ-calculus extended with resource operations. Resources $\mathsf{res}\,\ell$ are created by new_Ψ, where Ψ is some syntax (made abstract for now) for a user-assigned *temporal specification*. It prescribes a set of valid full usage traces of the allocated resource, which we call the *usage specification* and denote by $[\![\Psi]\!]$. For example, using a mixed form of regular and ω-regular expressions, temporal specifications of files and locks may be specified as:

$$\Psi_{\mathtt{file}} \stackrel{\text{def}}{=} \mathtt{open} \cdot (\mathtt{read} \mid \mathtt{write})^* \cdot \mathtt{close}$$
$$\Psi_{\mathtt{lock}} \stackrel{\text{def}}{=} (\mathtt{acquire} \cdot \mathtt{release})^* \mid (\mathtt{acquire} \cdot \mathtt{release})^\omega .$$

A resource access $\mathsf{acc}_a(v)$ takes a resource v and an event a, which is a symbolic name of a certain resource operation, such as open and `close` for files. A deallocation construct $\mathsf{drop}(v)$ discards all resources accessible from the value v.

Semantically, each resource accumulates accessed events as a *history trace* in the resource heap. When allocated, a resource is associated with the empty trace; and every time an access $\mathsf{acc}_a(\mathsf{res}\,\ell)$ is performed, the event a is appended to the trace of $\mathsf{res}\,\ell$ and this updated trace is checked to be a prefix of some trace in the usage specification; and when a resource is explicitly discarded by $\mathsf{drop}(v)$, the history trace must conform to the usage specification of the resource.

Crucially, whereas safety of resource usage can be checked by inspecting the prefixes of the full trace, liveness requires checking that the full trace exactly matches some trace in the usage specification to ensure desired events are eventually accessed [2]. For this reason, temporal resource usage analysis requires that resources staying in the resource heap infinitely (we call these *infinite-lifetime* resources) during a divergent execution must have an evolution of history traces whose limit trace conforms to the usage specification. For example, consider the following program:

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

This program first creates and opens a file and then passes it to the recursive function f, which continues to `read` the file

infinitely. The history traces of the file resource evolve as

$$\epsilon,\ \mathtt{open},\ \mathtt{open}\cdot\mathtt{read},\ \mathtt{open}\cdot\mathtt{read}^2,\ \mathtt{open}\cdot\mathtt{read}^3,\ \cdots$$

and have a limit infinite trace $\mathtt{open}\cdot\mathtt{read}^\omega \notin [\![\Psi_{\mathtt{file}}]\!]$ as it violates the requirement that `close` must eventually be applied. This liveness violation cannot be detected by inspecting the prefixes without reasoning about the full trace.

3 Overview

We identify the key challenges in type-based temporal resource usage analysis and present our ideas to address them.

3.1 Alias Tracking

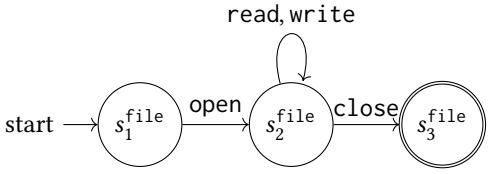
Common to prior work on resource usage analysis [4, 10, 12] is that (1) the types of resources are enriched with usage information that encodes the current state of the resource, and (2) substructural typing techniques, e.g., linear, uniqueness, or ordered typing, are applied to support strong update of statically tracked resource states.

For the former, we similarly introduce resource types of the form Res_Ψ that reuse the syntax of temporal specifications to encode the allowed *future usage* of the resource. We call these Ψ *usage prophecies*. For example, after an `open` event, the usage prophecy for a file resource can be updated from $\Psi_{\mathtt{file}}$ to $\Psi_{\mathtt{opened}} \stackrel{\text{def}}{=} (\mathtt{read} \mid \mathtt{write})^* \cdot \mathtt{close}$.

For the latter, among various established approaches, we adopt *uniqueness typing* [3, 6, 13] due to its simplicity and generality. Our use of uniqueness typing guarantees each resource has a unique reference, facilitating sound strong update of usage prophecies.

3.2 Progressivity Guarantee

Uniqueness typing alone is insufficient for verifying liveness of infinite-lifetime resources when programs diverge. Recall the program e_1 given in Section 2. It does not satisfy the temporal specification for files because the created file is 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 this in more detail, consider the finite automaton representation of the file specification:



In the program e_1, the state of the created file at the point immediately before the recursive call is $s_2^{\mathtt{file}}$, and it is then stuck in this non-accepting state: the resource enters $s_2^{\mathtt{file}}$ again and again with a self-loop.

Our rationale for identifying *unprogressiveness* as problematic is that it hinders ensuring that the resources reach certain desired states, namely, the accepting states in the automaton representations of their specifications (e.g., $s_3^{\mathtt{file}}$ above for files). At its core, liveness 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 liveness requirements are met.*

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:

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

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 towards $s_3^{\mathtt{file}}$ 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* $s_3^{\mathtt{file}}$.

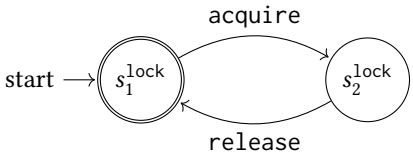
Based on this observation, we introduce *timers* to the type representation of resources as a technical device to guarantee the progressivity of resource states towards the desired states. Specifically, our *temporal resource types* take the form Res_Ψ^m accompanied by a timer m, which is a natural number, and a usage prophecy Ψ. An initial timer is assigned to each resource when it is created by new_Ψ^m. Because, as seen above, recursive functions are the source of infinite usage of resources that obfuscates 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 computations. The timer is 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 and then reaches the desired state. Therefore, we can assign a inital timer of 2 to these resources.

However, an ever-decreasing timer forbids resources to be used infinitely. For example, consider the following program

with lock resources:

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

This program acquires and releases the lock infinitely, which is allowed by the lock temporal specification, even though the resource x is passed to the recursive function infinitely. 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 when a resource is in a desired state*. Consider the automaton for lock specification $\Psi_{\mathtt{lock}}$:



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,[1] 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 e_3', which our type system accepts:

$$\begin{array}{rcl} e_3' & \stackrel{\text{def}}{=} & \mathsf{let}\, f = (\mathsf{rec}\, f\, x.\, e_{\mathsf{body}})\, \mathsf{in} \\ & & \mathsf{let}\, x = \mathsf{new}^{1}_{\Psi_{\mathtt{lock}}}\, \mathsf{in}\, f\, x \\ e_{\mathsf{body}} & \stackrel{\text{def}}{=} & \mathsf{acc}_{\mathtt{acquire}}(x); \mathsf{acc}_{\mathtt{release}}(x); \mathsf{reset}^1(x); f\, x\,. \end{array}$$

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

3.3 Termination Analysis

The progressivity guarantee ensures resources that are used infinitely—i.e., being passed to recursive functions infinitely—reach the desired states eventually. However, another possibility for the use of infinite-lifetime resources is that they can be left unused in a divergent sub-computation. We call this phenomenon *implicit discarding* of resources. Consider the function v given below (e_{cond} is unspecified):

$$\begin{array}{rcl} v & \stackrel{\text{def}}{=} & \mathsf{rec}\, f\, n.\, \mathsf{if0}\, e_{\mathsf{cond}}\, \mathsf{then}\, ()\, \mathsf{else}\, e \\ e & \stackrel{\text{def}}{=} & \mathsf{let}\, y = e_{\mathsf{open\cdot read}}\, \mathsf{in}\, f\, (n-1); e_{\mathtt{close}} \\ e_{\mathsf{open\cdot read}} & \stackrel{\text{def}}{=} & \mathsf{let}\, y = \mathsf{new}^{0}_{\Psi_{\mathtt{file}}}\, \mathsf{in}\, \mathsf{acc}_{\mathtt{open}}(y); \mathsf{acc}_{\mathtt{read}}(y) \\ e_{\mathtt{close}} & \stackrel{\text{def}}{=} & \mathsf{acc}_{\mathtt{close}}(y); \mathsf{drop}(y)\,. \end{array}$$

Termination behavior of the application of the function v depends on the expression e_{cond}. If $e_{\mathsf{cond}} \stackrel{\text{def}}{=} n \geq 0$, the function call might diverge. In these cases, the resource y will not be closed. On the other hand, if $e_{\mathsf{cond}} \stackrel{\text{def}}{=} n \leq 0$, the function call always terminates. In this case, every created file resource y reaches the desired state $s_3^{\mathtt{file}}$.

[1] Our type system tracks the current state of a resource by the usage prophecy Ψ in its type.

This example shows the importance of termination analysis in detecting implicit discarding. A sound type system must ensure that unused resources have reached the desired states just before the execution of a divergent sub-computation starts. Although assuming that recursive function calls always diverge enables sound reasoning, it is often too conservative. Termination analysis is a fundamental problem in computer science and has been extensively studied for decades. Rather than incorporating some specific termination analysis method into the type system, we *assume* that programs are annotated to indicate whether expressions terminate (e.g., we write $\mathsf{rec}^{\lightning}\, f\, x.\, e$ for always terminating recursive functions) and propagate the information as a *termination effect* in computation types.

4 Type System

In this section, we first present simplified versions of our type syntax and typing rules, and then present typing examples to illustrate how our type system works. We have proven the soundness of a full version of our type system but we omit the details here.

4.1 Type Syntax

We present an excerpt of our type syntax:

$$\begin{array}{lrcl} \textbf{Value Types} & T & ::= & \ldots \mid \mathsf{Res}^m_\Psi \\ \textbf{Comp. Types} & C & ::= & T \,\&\, \zeta \\ \textbf{Termination Effects} & \zeta & ::= & \lightning \mid ? \\ \textbf{Finite Trace Sets} & s & \subseteq & \mathbb{A}^* \\ \textbf{Finite Spec.} & \phi & ::= & s \\ \textbf{Infinite Spec.} & \psi & ::= & \{\rho_1, \ldots, \rho_n\} \\ \textbf{Lassos} & \rho & ::= & \langle s_{\mathsf{init}}, s_{\mathsf{rep}} \rangle \\ \textbf{Temporal Specifications / Usage Prophecies} & & & \\ & \Psi & ::= & \langle \phi, \psi \rangle \end{array}$$

The syntax of types consists of *value types* T and *computation types* C, which are used to type values and expressions, respectively. As discussed in Section 3.2, the type of resources is Res^m_Ψ, containing a timer m and a usage prophecy Ψ. A computation type is composed of a value type T and a *termination effect* ζ, which describes the values produced by the expressions (if any) and their termination behavior, respectively. A *temporal specification* (used in new^m_Ψ) or *usage prophecy* (used in Res^m_Ψ) is a pair of a *finite specification* ϕ and an *infinite specification* ψ. A finite specification ϕ is a finite trace set s, 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*. Their interpretation is formulated as below.

Definition 4.1 (Interpretations of Temporal Specifications)**.** The interpretation of a temporal specification Ψ, infinite specification ψ, and lasso ρ is defined as:

$$[\![\langle \phi, \psi \rangle]\!] \stackrel{\text{def}}{=} \phi \cup [\![\psi]\!] \quad [\![\psi]\!] \stackrel{\text{def}}{=} \bigcup_{\rho \in \psi} [\![\rho]\!] \quad [\![\langle s_1, s_2 \rangle]\!] \stackrel{\text{def}}{=} s_1 \cdot s_2^\infty$$

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

Namely, a trace in $[\![\psi]\!]$ is a finite trace in s_{i1} followed by infinite repetitions of traces in s_{i2} for some lasso $\langle s_{i1}, s_{i2} \rangle$. 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. 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 [11] and ω-context-free-grammars [5].

4.2 Typing Rules

We show simplified versions of the typing rules that are crucial for the enforcement of progressiveness and the detection of implicit discarding.

4.2.1 Consumption of Usage Prophecy. To understand how usage prophecy in temporal resource types works, we present the typing rule for resource access:

$$\frac{\Gamma \vdash v : \mathsf{Res}^m_\Psi \qquad \vdash_{\mathrm{WF}} \mathsf{Res}^m_{\Psi^{-a}}}{\Gamma \vdash \mathsf{acc}_a(v) : \mathsf{Res}^m_{\Psi^{-a}} \,\&\, \lightning} \text{ C\_Acc}$$

where Ψ^{-a} consumes the finite specification as well as the initial part of the lassos in the infinite specification, removing the raised event a from the usage prophecy. The well-formedness premise $\vdash_{\mathrm{WF}} \mathsf{Res}^m_{\Psi^{-a}}$ ensures the access is safe, i.e., it leaves a non-empty usage prophecy for future usage.

4.2.2 Timer Count-Down. In a first-order setting,[2] the timer of a resource is decreased whenever it is passed to a recursive function call, giving rise to a derived typing rule for recursive functions:

$$\frac{\Gamma^! \uplus \{f : T_1 \multimap T_2 \,\&\, \zeta\} \uplus \{x : {T_1}^{-1}\} \vdash e : T_2 \,\&\, \zeta}{\Gamma^! \vdash \mathsf{rec}^\zeta f\, x.\, e : T_1 \multimap T_2 \,\&\, \zeta} \text{ T\_Rec}$$

where the count-down operation T^{-1} decreases timers of all temporal resource types in T by 1. The typing context $\Gamma^!$ is assumed to capture no resources for conforming to the uniqueness typing discipline (because recursive functions can be recursively applied multiple times).

4.2.3 Timer Reset. Our typing rule for the reset construct is as follows:

$$\frac{\Gamma \vdash v : \mathsf{Res}^{m'}_{\langle \phi, \psi \rangle} \qquad \psi' = \{\langle s_2, s_2 \rangle \mid \langle s_1, s_2 \rangle \in \psi \wedge \epsilon \in s_1\} \neq \emptyset}{\Gamma \vdash \mathsf{reset}^m(v) : \mathsf{Res}^m_{\langle \phi, \psi' \rangle} \,\&\, \lightning} \text{ C\_Reset}$$

Note that the return type has a renewed timer of m instead of m'. Also, the second premise filters the lassos that have reached a desired state (by checking the initial part of the lasso contains the empty trace); it produces a new infinite specification with the repeated part of these lassos copied to the initial part. The new infinite specification must be non-empty, meaning at least one lasso is "realizable".

4.2.4 Discarding of Resources. The typing rule for the drop construct checks that the discarded resources have been used up correctly (by checking the finite specification in usage prophecy contains the empty trace). This check is also performed when implicit discarding happens, handled in the typing rule for the let-expressions:

$$\frac{\Gamma_1 \vdash e_1 : T_1 \,\&\, \zeta_1 \qquad \Gamma_2 \uplus \{x : T_1\} \vdash e_2 : T_2 \,\&\, \zeta_2 \qquad \zeta_1 = ? \Longrightarrow\!\vdash^\dagger \Gamma_2}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : T_2 \,\&\, \zeta_1 \triangleright \zeta_2} \text{ C\_Let}$$

where $\zeta_1 \triangleright \zeta_2$ is sequential composition of termination effects. Notably, the premise $\zeta_1 = ? \Longrightarrow\!\vdash^\dagger \Gamma_2$ ensures unused resources have been correctly used up when implicit discarding happens.

4.3 Typing Examples

Example 4.2 (Well-Typed Example)**.** The example program e'_3 in Section 3.2 is well-typed in our type system with the following temporal specification:

$$\Psi_{\mathsf{lock}} \stackrel{\text{def}}{=} \langle s_{\mathsf{lock}}{}^*, \{\langle s_{\mathsf{lock}}, s_{\mathsf{lock}} \rangle\} \rangle$$
$$s_{\mathsf{lock}} \stackrel{\text{def}}{=} \{\mathtt{acquire} \cdot \mathtt{release}\}\,.$$

Example 4.3 (Ill-Typed Example)**.** This ill-typed example illustrates the detection of invalid implicit discarding.

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

where (we reuse $e_{\mathtt{open{\cdot}read}}$ and $e_{\mathtt{close}}$ defined in Section 3.3)

$$e_{\mathsf{body}} \stackrel{\text{def}}{=} \mathsf{let}\, y = e_{\mathtt{open{\cdot}read}}\,\mathsf{in}\,\mathsf{let}\,() = f\, x\,\mathsf{in}\, e_{\mathtt{close}}$$
$$\Psi_{\mathtt{file}} \stackrel{\text{def}}{=} \langle \{\mathtt{open}\} \cdot \{\mathtt{read}, \mathtt{write}\}^* \cdot \{\mathtt{close}\}, \emptyset \rangle\,.$$

This function body will not type check. The issue is in type checking the sequential composition of $\mathsf{let}\,() = f\, x\,\mathsf{in}\, e_{\mathtt{close}}$. The resource bound to y is not closed when the divergent computation of $f\, x$ happens, hence invalid implicit discarding of resources occurs. This is detected by the typing rule for let-expressions.

[2] In a higher-order setting, i.e., functions may take and return functions, more sophisticated typing techniques are needed to handle timer decrement correctly. This is done in the full version of our type system.

References

[1] 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

[2] Bowen Alpern and Fred B. Schneider. 1985. Defining Liveness. *Inf. Process. Lett.* 21, 4 (1985), 181–185. doi:10.1016/0020-0190(85)90056-0

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

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

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

[12] 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

[13] 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

[14] 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