Temporal Resource Typing

Enriching Substructural Typing for **Liveness Reasoning**

Yiyuan Cao[1] Taro Sekiyama[2]

[1]Peking University

[2]National Institute of Informatics

IWACO 2025, Singapore

Outline

Resource Usage Verification

Resources are *stateful objects* that must be used according to specific protocols.

- The set of valid operations on a resource can *change* throughout its lifetime.
- Examples: files, locks, heap memory, network sockets, processes, etc.

We use temporal specifications to express such protocols, prescribing the valid usage traces.

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

A program is resource-usage correct if every resource it allocates has a usage trace δ that respects its assigned temporal specification Ψ, i.e., $\delta \in [\![\Psi]\!]$.

Valid Example: File-Processing Loop

```
let rec main_loop () =
  let file = open (input()) in
  let content = read file in
  (* process content *)
  close file;
  main_loop ()
```

The program runs *infinitely*:

- Every allocated file has a usage trace $\mathtt{open} \cdot \mathtt{read} \cdot \mathtt{close} \in [\![\Psi_{\mathtt{file}}]\!]$.

Thus, it is resource-usage correct.

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

Classify Violations: Safety vs Liveness

Every temporal specification can be decomposed into its *safety* and *liveness* parts (Alpern and Schneider 1985).

- Safety is about "Nothing bad ever happens". Violation:

$$\mathtt{open} \cdot \mathtt{close} \cdot \color{red}{\mathtt{close}} \cdot \ldots \notin [\![\Psi_{\mathtt{file}}]\!] \ .$$

- Liveness is about "Something good eventually happens". Violation:

$$\mathtt{open} \cdot \mathtt{read} \cdot \mathtt{read} \cdot \mathtt{read} \cdot \ldots \notin [\![\Psi_{\mathtt{file}}]\!] \ .$$

In general, liveness violations cannot be detected by checking prefixes only.

Safety Violation: Read After Close

```
let rec main_loop () =
  let file = open (input()) in
  close file;
  (* unsafe read after close *)
  let content = read file in
  main_loop ()
```

The program runs *infinitely*:

- Every allocated file has a usage trace $\mathtt{open} \cdot \mathtt{close} \cdot \mathtt{read} \notin [\![\Psi_{\mathtt{file}}]\!]$.
- The operation **read** is invoked after **close**, leading to a bad state.

Thus, it is not resource-usage correct (safety violation).

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

Liveness Violation: Infinite Read

```
let rec main_loop file =
  let content = read file in
  (* read infinitely *)
  main_loop file
in
  let file = open (input()) in
  main_loop file;
  close file
```

The program runs *infinitely*:

- The allocated file has a usage trace $\mathtt{open} \cdot \mathtt{read}^{\omega} \notin [\![\Psi_{\mathtt{file}}]\!]$.
- The resource is used in a safe way: every prefix $\mathtt{open} \cdot \mathtt{read}^{i}$ is good.
- The `file` is used infinitely by **read**, but missing the desired **close**.

Thus, it is not resource-usage correct (liveness violation).

$$\Psi_{\mathtt{file}} \stackrel{\mathrm{def}}{=} \mathtt{open} \cdot (\mathtt{read} \mid \mathtt{write})^{*} \cdot \mathtt{close}$$

Outline

Landscape: Verifying Resource Usage using Types

Safety of resource usage has been well-studied over decades.

- *Invariant-based reasoning* for type and resource safety.
- *Substructural typing* for strong update of resources' states.
- Strom and Yemini 1986; DeLine and Fähndrich 2001; Igarashi and Kobayashi 2002; Bierhoff and Aldrich 2007; Saffrich, Nishida, and Thiemann 2024...

Liveness of resource usage is under-served.

- Few works ensure liveness of resource usage for diverging programs.
- Detecting liveness violation requires reasoning about infinite usage.

A type system that enforces liveness of resource usage

Ensure resources produce not just valid prefixes but also valid full usage traces, even in infinite executions.

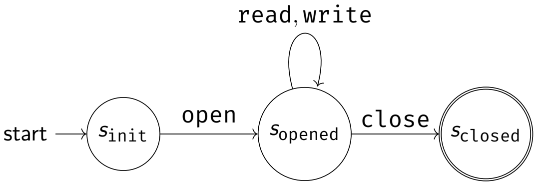
Our Contributions

- Formalize resource-usage correctness to include liveness.
- Temporal resource typing with a resettable timer mechanism for progressivity guarantee.
- Incorporate termination analysis for precise reasoning about implicit discarding.
- Soundness via a logical relation capturing the progressive nature of divergence.

Outline

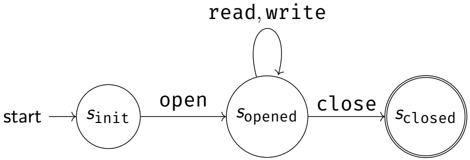
Model Specs as Transition Systems

Temporal specifications can be represented as transition systems. Consider $\Psi_{\texttt{file}}$:



A resource usage can then be represented as a sequence of visited states.

Revisit Safety vs Liveness



- **Safety**: *always* take a valid transition.
- **Liveness**: *eventually* reach some accepting state.

Valid usage: $s_{\texttt{init}} \xrightarrow{\texttt{open}} s_{\texttt{opened}} \xrightarrow{\texttt{read}} s_{\texttt{opened}} \xrightarrow{\texttt{close}} s_{\texttt{closed}}$.

Track Usage State in Types

To verify resource-usage correctness using types:

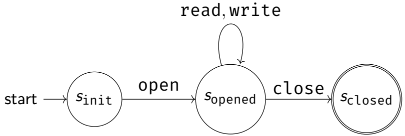
- The types of resources carry the current "state", e.g., for a just opened file f,

$$\mathtt{f} : \mathtt{file}[s_{\mathtt{opened}}] \,.$$

- Resource operations update the state information in types, e.g.,

$$\mathbf{open} : \mathtt{file}[s_{\mathtt{init}}] \to \mathtt{file}[s_{\mathtt{opened}}] \,.$$

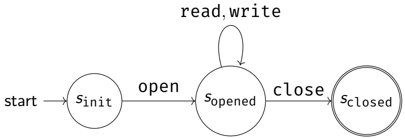
- For sound strong updates of state, apply *substructural typing* techniques to track aliasing. (Specifically, we use uniqueness typing.)



Revisit Safety Violation: Read After Close

```
let rec main_loop () =
  let x = new[file] in  // x: file[init]
  open x;               // x: file[opened]
  close x;              // x: file[closed]
  (* unsafe read after close *)
  read x;
  // error: expected file[opened] but got file[closed]
  main_loop ()
```

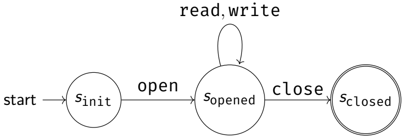
Unsafe usage: $s_{\texttt{init}} \xrightarrow{\texttt{open}} s_{\texttt{opened}} \xrightarrow{\texttt{close}} s_{\texttt{closed}} \overset{\texttt{read}}{\rightsquigarrow}$.



Revisit Liveness Violation: Infinite Read

Naively, function f can be assigned type $\texttt{file}[s_{\texttt{opened}}] \to \texttt{file}[s_{\texttt{opened}}]$:

```
let rec f x =              // x:file[opened]
  read x;                  // x:file[opened]
  f x
in
  let x=new[file] in       // x:file[init]
  open x;                  // x:file[opened]
  f x
```



The usage trace $\texttt{open} \cdot \texttt{read}^{\omega}$ is safe but violates liveness:

$$s_{\texttt{init}} \xrightarrow{\texttt{open}} s_{\texttt{opened}} \xrightarrow{\texttt{read}} s_{\texttt{opened}} \xrightarrow{\texttt{read}} s_{\texttt{opened}} \xrightarrow{\texttt{read}} \cdots$$

Root of Liveness Violation: Lack of Progressivity

The root of liveness violation is the *lack of progressivity*[1] of infinite resource usage.

$$s_{\texttt{init}} \xrightarrow{\texttt{open}} s_{\texttt{opened}} \xrightarrow{\texttt{read}} s_{\texttt{opened}} \xrightarrow{\texttt{read}} s_{\texttt{opened}} \xrightarrow{\texttt{read}} \cdots$$

The execution continues infinitely by recursion,
but the resource is stuck in $s_{\texttt{opened}}$ and never *progresses* towards the desired state $s_{\texttt{closed}}$.

[1] **Progressivity:** Resource reaches desired state within bounded "time steps".

Valid Example: Progressive Usage

Conversely, consider the following program with valid usage:

```
let rec f x y =        // x:file[opened], y:file[init]
  close x; open y;     // x:file[closed], y:file[opened]
  let z=new[file] in   // z:file[init]
  f y z
in
  let x=new[file] in
  let y=new[file] in
  open x;              // x:file[opened], y:file[init]
  f x y
```

Every allocated resource *progresses* towards the desired state $s_{\mathtt{closed}}$ within two recursive calls:

$$\xrightarrow{\mathtt{new}} s_{\mathtt{init}} \xrightarrow{\mathtt{open}} s_{\mathtt{opened}} \xrightarrow{\mathtt{close}} s_{\mathtt{closed}}$$

Enforce Progressivity with Timers

To ensure progressivity, we add a *timer* m to the resource types (general form Res^m_Ψ).

- An initial timer m is set when a resource is created by `new[Ψ,m]`.
  E.g., `x=new[file,2]` gives $x : \mathsf{Res}^2_{\Psi_{\mathtt{file}}}$.
- When a resource is passed to a "recursive computation", the timer is decreased by 1.

$$\frac{\Gamma, f : T_1 \to T_2, x : T_1^{-1} \vdash e : T_2}{\Gamma \vdash \mathsf{rec}\, f\, x.\, e : T_1 \to T_2}\ \mathrm{T\_REC}^2$$

Count-down operation on types T^{-m}:

$$(\mathsf{Res}^m_\Psi)^{-1} = \mathsf{Res}^{m-1}_\Psi \qquad (T_1 \otimes T_2)^{-1} = T_1{}^{-1} \otimes T_2{}^{-1}$$

- The timers are required to be non-negative at all times.

[2] This is a simplified rule in the first-order setting.

Revisit Infinite Read with Timers

```
let rec f x =       // x:Res[m,opened]
  (* countdown *) // x:Res[m-1,opened]
  read x;         // x:Res[m-1,opened]
  f x
  // error: expected Res[m,opened] but got Res[m-1,opened]
in
  let x=new[file,m] in // x:Res[m,opened]
  f x
```

For arbitrary m, the countdown along recursion is:

$$(m, s_{\texttt{opened}}) \xrightarrow{f} (m-1, s_{\texttt{opened}}) \xrightarrow{f} (m-2, s_{\texttt{opened}}) \rightarrow \cdots$$

No transition to $s_{\texttt{closed}}$ occurs before m hits 0. Therefore the program must be rejected.

Revisit Progressive Usage with Timers

```
let rec f x y =          // x:Res[1,opened], y:Res[2,init]
  (* countdown *)        // x:Res[0,opened], y:Res[1,init]
  close x; open y;       // x:Res[0,closed], y:Res[1,opened]
  let z=new[file,2] in // z:Res[2,init]
  f y z
in
  let x=new[file,1] in // x : Res[1,init]
  let y=new[file,2] in // y : Res[2,init]
  open x;                // x : Res[1,opened]
  f x y
```

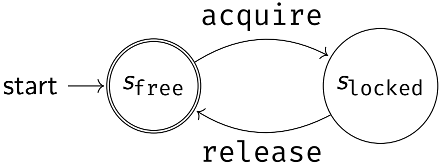
Intuitively, well-typed programs ensure that every resource reaches some desired state before it is passed into *m*-number of recursive calls.

$$\xrightarrow{\texttt{new}} (2, s_{\texttt{init}}) \xrightarrow{\texttt{f}} (1, s_{\texttt{init}}) \xrightarrow{\texttt{open}} (1, s_{\texttt{opened}}) \xrightarrow{\texttt{f}} (0, s_{\texttt{opened}}) \xrightarrow{\texttt{close}} (0, s_{\texttt{closed}})$$

Permit Progressive Infinite Usage

More generally, we want to allow valid *infinite* progressive usage of resources.

```
let rec loop x= //x:Res[m,free]
  (*countdown*) //x:Res[m-1,free]
  acquire x;    //x:Res[m-1,locked]
  release x;    //x:Res[m-1,free]
  loop x     //error: type mismatch
in
  let x=new[lock,m] in
  // x:Res[m,free]
  loop x
```

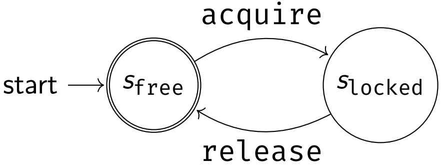


The resource reaches the desired state $s_{\mathtt{free}}$ infinitely, satisfying the liveness requirement.
But no initial timer suffices for an infinite usage.

Reset Timers in Desired States

We add a ghost construct **reset**[m](v) to the language. Its typing allows changing the timer of the passed-in resource to m, under the condition that *it is currently in a desired state*.

```
let rec loop x= //x:Res[1,free]
  (*countdown*) //x:Res[0,free]
  acquire x;    //x:Res[0,locked]
  release x;    //x:Res[0,free]
  reset[1](x);  //x:Res[1,free]
  loop x
in
  let x=new[lock,1] in
  // x:Res[1,free]
  loop x
```



The resource states and timers evolve as follows:

$$\xrightarrow{\mathtt{new}} (1, s_{\mathtt{free}}) \xrightarrow{\mathtt{loop}} (0, s_{\mathtt{free}}) \xrightarrow{\mathtt{acquire}\cdot\mathtt{release}} (0, s_{\mathtt{free}}) \xrightarrow{\mathtt{reset}} (1, s_{\mathtt{free}}) \xrightarrow{\mathtt{loop}} \cdots$$

Formalize Temporal Specs and States and Usage Prophecies

$$
\begin{array}{llcl}
\textbf{Temporal Specs} & \Psi & ::= & \langle \phi, \psi \rangle \\
\textbf{Infinite Specs} & \psi & ::= & \{\rho_1, \ldots, \rho_n\} \\
\textbf{Lassos} & \rho & ::= & \langle s_1, s_2 \rangle
\end{array}
$$

Note:

- This is a general representation that incorporates ω-regular and ω-context-free languages.
- A resource is considered to be in a *desired state* if its finite specification is satisfied or it finishes producing some initial finite trace in a lasso.

Outline

Summary

This work:

- *Problem*: Type-based resource usage verification of temporal properties.
- *Challenge*: Sound reasoning of infinite usage of resources with liveness requirements requires the progressivity guarantee.
- *Approach*: Add a *resettable timer* mechanism to the type system.
- More in paper: another source of unprogressivity is *implicit discarding* of resources.

Future work:

- Support value-dependency in timers and temporal specifications.
- Support aliasing of resources.
- Automatic inference of resets and timers.

Liveness Violation: Unreachable Close

```
1 let rec main_loop () =
2   let file = open (input()) in
3   let content = read file in
4   (* remain unused infinitely *)
5   main_loop ()
6   close file;
```

The program runs *infinitely*:

- Every allocated file has a usage trace $\mathtt{open} \cdot \mathtt{read} \notin [\![\Psi_{\mathtt{file}}]\!]$.
- The resource is used in a safe way.
- Since `main_loop()` never returns, `file` remains unused afterwards (**`close`** at line 6 is dead code).

Thus, it is not resource-usage correct (liveness violation).

Valid Example: Scheduler

```
let rec scheduler (active_list, wait_list) =
  if active_list != [] then
    (* schedule the first process in the active list *)
    let x::active_list = active_list in
    run x;
    (* put the process back to the wait list *)
    let wait_list = x::wait_list in
    (* continue to the round-robin scheduler *)
    scheduler(active_list, wait_list)
  else
    (* start over from the wait list *)
    scheduler(wait_list, [])
```

Temporal specification: $\Psi_{\mathtt{proc}} \stackrel{\text{def}}{=} \mathtt{run}^{\omega}$.

Classify Violations: Illustration