EasyChair Preprint

№ 8434

A Rust-like Type System for Cooperative Threads

Darine Rammal, Wadoud Bousdira and Frédéric Dabrowski

July 10, 2022

A Rust-like type system for cooperative threads

Darine Rammal, Wadoud Bousdira, and Frédéric Dabrowski

{darine.rammal,wadoud.semmar, frederic.dabrowski}@univ-orleans.fr

Abstract. We propose a Rust-like type system for a kernel programming language named MSSL and featuring a cooperative threading model. Our proposal allows the sharing of mutable data between cooperative threads while preserving the type and borrowing safety. This result is obtained by introducing a new abstraction, named `Trc`, which combines the ownership safety of Rust references with the reference counting mechanism of Rust smart pointers. We present a subset of MSSL semantics and type system and prove that the latter ensures types and borrowing safety.

Keywords: Cooperative scheduling, reactive programming, memory safety, Rust

1 Introduction

Reliability of system software is a challenge in the computer world. Even today, most software vulnerabilities stem from programming errors related to memory management[23, 1]. Such errors can be memory leak errors, use after free errors or double free errors. To achieve memory safety, many high-level programming languages (e.g. Java and SML) use a garbage collector[12, 25] to automatically manage memory. At run-time, the garbage collector identifies blocks of memory that can never be used by the program again and reclaims their space for future allocations. However, dynamic garbage collection incurs an execution overhead that is not acceptable for some computer systems (system programming, embedded system, etc.). Thirty years of research[3, 20, 9] and the pragmatism of the Rust[2] development team have made it possible to combine the efficiency of high-level programming languages with the memory safety of high-level languages.

Sponsored by Mozilla and developed in the open by a large and diverse community of contributors, Rust is a statically-typed programming language designed for performance, reliability and safety, especially safe concurrency and memory management without requiring garbage collection. The type system of Rust restricts the possibilities of aliasing in such a way that upon exiting the scope of a variable, the memory addresses accessible from it can be safely deallocated. In practice, a strict application of this type system is too restrictive. This is where the pragmatism of Rust comes into play. Libraries can be developed in which the type system is disabled by using the unsafe keyword. Many features of Rust, such as RC and Mutexes, rely on this mechanism. RCs are a special type of read-only references whose safety is based on reference counting, which makes it

possible to get rid of lexical scope constraints. Yet, libraries using the unsafe feature must satisfy certain invariants in order not to break memory safety[13]. Rust comes with a multithreading library (or crate in Rust terminology) that relies on system threads. Although this concurrent programming model is widespread, it is too resource intensive for some systems (e.g. low resource embedded systems). Cooperative threads, which can be implemented in user space, are a good alternative. In this paper, we propose a Rust-like type system for a subset of MSSL (Memory Safe Synchronous Language) a reactive programming language based on cooperative threads and synchronous execution. MSSL is inspired by the family of synchronous reactive languages (Esterel[4],Fairthreads[8], Reactive ML[17], etc). In this programming model, threads run in turn. The change of control is done on demand through a yield command. A notion of signal allows threads to synchronise. Thanks to a notion of logical time, called instants, threads progress synchronously with a coherent view of the state of signals (present or absent) at each instant. Due to the use of cooperative scheduling, MSSL does not need a locking facility. However, even if the execution of threads is cooperative, the Rust type system requires adaptations to allow data sharing between threads since it introduces a new form of aliasing that must be controlled. Reference counting, as proposed by Rust, appears to be an interesting solution since it allows to get rid of lexical scope constraints. This is necessary to allow the sharing of data between different threads. This solution is however limited by the fact that these references are read-only. Removing this constraint would break the memory safety properties at the thread level. We therefore propose to combine the reference counting approach with the aliasing constraints of standard Rust references. This new type of smart pointer, which is the main contribution of this paper, is named `Trc`. We present an operational semantics for MSSL and Rust-like type system ensuring memory safety in presence of `Trc`. The formal developments presented in this paper build on top of the work of Pearce[21]. For lack of space, we omit signal-based synchronization and focus on the cooperate command, which is a special case. The rest of the paper is organized as follows: Section 2 describes the Rust feature and the reactive synchronous. Section 3 presents the syntax and the semantic of MSSL. Section 4 defines the type system. Section 5 presents the principal results of soundness. Section 6 we explore future work based on MSSL, and we conclude in Section 7.

2 About Rust and Concurrent Programming

Rust has common elements found in other languages, such as `struct, traits` (similar to interfaces in Java), lambda functions and modules. The main feature of Rust is its ownership and borrowing system [22]. This helps prevent many classes of errors arising from incorrect usage of memory, (e.g. use-after-free, double-free, memory leaks). Every value in Rust has a type, a mutability modifier, an owner and a lifetime. The owner is often the scope in which the value was declared, but ownership of the value can change when performing assignments or passing function arguments by value. In Rust, this is known as a *move*. A lifetime begins when the value is created and ends when it is dropped.

The compiler tracks lifetimes of every reference (borrow) to ensure all references are valid (i.e. the value could not possibly be dropped before using the reference, preventing a use-after-free error), and lifetimes are tied to the scope in which a value exists. The Rust type system includes a borrow checker that statically checks that for a given variable (1) one or more shared references ($\&\tau$) exist, and (2) no more than one mutable reference ($\&mut\ \tau$) exists. Borrowing prevents unsafe concurrent modifications of the same memory location thus ensuring memory safety. Another essential element of Rust is the lifetime concept. When borrowing something, misunderstanding can be prevented by agreeing up front on how long it can be borrowed. So, when a reference is created, a lifetime is assigned to it. Then, it is recorded in the full form of the reference type: $\&'a$ `mut` τ for lifetime $'a$. The compiler ensures that (1) the reference is used only during that lifetime, and (2) the referent is not used again until the lifetime is over. In the left part of Figure 1, the lifetime of x starts at line 5 and goes on until line 9 while that of y starts at line 6 and ends at line 7. The ownership of x is moved to y and then after line 7, the value of x becomes unreachable since the scope of y closes and y is automatically dropped. In the right part of Figure 1, at line 8, the inner scope closes and the variable y is dropped. Even though x is still available in the outer scope, the reference at line 9 is invalid because the value to which it points no longer exists. The Rust compiler reports an error.

```
1. struct Circle {
2.     a : i32, b : i32
3. }
4. fn newCircle(a : i32, b : i32) {
5.     let mut x = Circle{a, b};
6.     { let mut y = x;
7.     }
8.     return x; //Error: value used after moved
9. }
```

```
1. struct Circle {
2.     a : i32, b : i32
3. }
4. fn newCircle(a : i32, b : i32) {
5.     let mut x;
6.     { let mut y = Circle{a, b};
7.       x = &y;
8.     }
9.     println!("{}", x.a); //Error!
10. }
```

Fig. 1: Example 1

When using functions, Rust requires insurance that references used at runtime are valid. To achieve this, generic lifetimes in functions are used. A generic lifetime parameter imposes a lifetime constraint on the reference(s) and the return value(s) of a function. While compiling the code, a generic lifetime is substituted for a concrete lifetime, which is equal to the smaller of the passed references lifetimes. This enables Rust to identify a violation of the constraint by a parameter or the variable storing the return value. In Figure 2, the invocation of `compareCircle` at line 12 returns a reference whose lifetime is equal to the shortest lifetime among the references passed into the function (i.e. the lifetime of y). Calling print instruction at line 14 illustrates a compilation error because there may be cases when one of the returned references is invalid. To manipu-

```
1. struct Circle {
2.     a : i32, b : i32
3. }

4. fn compareCircle(a : &i32, b : &i32) → &i32 {
5.       if a > b { b }
6.       else { a }
7. }
```

```
8. fn main (){
9.     let x = 5;
10.    let result;
11.    { let y = 10;
12.      result = compareCircle(&x, &y);
13.    }
14.    println!("{}", result); // Error!
15. }
```

Fig. 2: Example 2

late dynamic data structures, Rust provides some types to allocate data in the heap. The simplest form is the type `Box`$\langle\tau\rangle$. Boxes provide ownership for this allocation and drop their contents automatically when they go out of scope. Two other types are defined that use shared data are `Rc` and `Arc`. These types allow shared ownership of values allocated in the heap by using reference counting operations. Unfortunately, inside both types, we cannot get a mutable reference to something even though `Arc` is thread-safe (unlike for `Rc`, reference counting are atomic operations for `Arc`). In order to share mutable data between threads, an alternative in Rust consists of using `Mutex`$\langle\tau\rangle$ where the data of type τ is only accessed through guards returned from blocking functions. However, using `Mutex` does not protect against the risk of creating deadlocks and it is too resource intensive for some systems. Instead, we propose a new type `Trc`(τ) to share data between cooperative synchronous threads which is an alternative to the concurrency as it is implemented in Rust. Let us present the main features of concurrent programming. The essential function of multi-threads systems, named scheduling is to manage the sequencing of tasks by optimizing CPU usage. The thread is the basic unit of concurrency. When they are managed according to a cooperative strategy, threads cooperate with each other and the scheduling policy must avoid to hog the CPU indefinitely. Various proposals for cooperative threads have been made recently. One of them named *fairthreads* consists of fair threads executed by specialized schedulers giving them equal possibilities to access the processor. Fairthreads can be finely controlled allowing the user to code their own execution strategy. Moreover, fairthreads can communicate through broadcast events, which simplifies programming and this style of programming is called reactive programming. Programs using fairthreads are deterministic: their result is independent of the scheduler of the machine which executes them, and this is in general not true with standard threads. Determinism greatly facilitates the porting and the debugging of programs. Fairthreads have been implemented in Java[7], in C[8] and in Scheme[16]. Finally, another approach exists, called reactive-synchronous approach which allows a simple and efficient programming of concurrency, by solving some problems of threads. Languages built on this approach have been developed, the best known of which are the synchronous languages *Esterel* [5], *Lustre* [10] and *Signal* [15].

MSSL is inspired by the cooperative part of this work. Threads are implicitly linked to a scheduler and they are executed cooperatively according to the scheduler's clock. The execution of the threads is synchronous and is divided into logical instants like for Fairthreads[6]. In addition, these threads can also return control to their scheduler through the `cooperate` expression. At this point, the scheduler knows that the undergoing thread has finished its execution for the current instant, and that it will assume back control at the next instant.

3 MSSL

MSSL aims at improving the *Faithreads* programming model by providing memory safety in the manner of Rust. As in *Fraithreads*, threads are executed cooperatively by a round-robin scheduler, communicate through shared memory

and synchronize by means of signals. In the rest of the paper, we focus on the cooperative fragment of MSSL. Compared with FR, the new constructs are: two multithreading instructions (threads spawning and explicit cooperation) and a new kind of smart pointers called `TRC` and dedicated to communication. Intuitively, every shared data must be encapsulated in an `TRC`. The type system will enforce this property and will protect such data from concurrent corruption. The challenge in designing a new kind of smart pointers was to combine sharing and mutability without requiring a locking discipline (as is done in Rust). More precisely, `TRC` (1) allow sharing among threads, (2) ensure per thread unicity of mutable and (3) ensure that no thread possesses references to shared data at cooperation time. `TRC` pointers fall into two categories : active and inactive `TRC`. An active `TRC` is an entry point to the shared part of the heap, while an inactive `TRC` is a copy of the pointer aimed at being communicated to other threads. When communicated to another thread, an inactive `TRC` becomes active. Only active `TRC` can be accessed (required for (2)).

3.1 Syntax

The syntax of MSSL is given in Figure 3. A value may be one of the special constants ϵ_0 or ϵ_1, an integer n or a memory location. A value ϵ_0 is produced by a statement that finishes (e.g an assignment) whereas a value ϵ_1 is produced by a statement that cooperates. We distinguish four kinds of memory locations. A value $\ell^{\blacksquare}$ (resp. ℓ°) denotes an owned (resp. borrowed). A value $\ell^{\blacklozenge}$ (resp. $\ell^{\diamond}$) denotes an active (resp. inactive) `Trc`. Partial values extend values with a special constant denoting a moved value.

Values	v	$::= \epsilon_0 \mid \epsilon_1 \mid n \mid \ell^{\circ} \mid \ell^{\diamond} \mid \ell^{\blacksquare} \mid \ell^{\blacklozenge}$
Partial Values	$v^{\perp}$	$::= v \mid \perp$
Types	τ	$::= \epsilon \mid \texttt{int} \mid \&\texttt{mut}\ \overline{w} \mid \&\ \overline{w} \mid \diamond\overline{\omega} \mid \blacklozenge\tau \mid \blacksquare\tau$
Partial Types	$\tilde{\tau}$	$::= Types \mid \blacksquare\tilde{\tau} \mid \blacklozenge\tilde{\tau} \mid \lfloor\tau\rfloor$
LVals	ω	$::= x \mid *\omega$
Expressions	e	$::= v \mid w \mid \hat{w} \mid \{\overline{e}\}^1 \mid \texttt{let mut}\ x = e \mid \texttt{box}(e) \mid \&[\texttt{mut}]\ w \mid w = e$ $\mid \texttt{trc}(e) \mid w.clone \mid \texttt{spawn}(f(\overline{e})) \mid \texttt{cooperate}$
Functions	f	$::= \texttt{fn}\ f(\overline{\texttt{mut}\ x : S})\{\overline{e}\}^1$
Signatures	S	$::= \epsilon \mid \texttt{int} \mid \blacksquare\texttt{S} \mid \blacklozenge\texttt{S}$
Programs	p	$::= f\ p \mid e$

Fig. 3: Syntax of MSSL

Types include primitive types ϵ and `int`, reference types ($\&\texttt{mut}\ \overline{w}$ and $\&\ \overline{w}$) and box types ($\blacksquare\tau$), similar to their counterpart in FR. A type $\&\texttt{mut}\ \overline{w}$ (resp. $\&\ \overline{\omega}$) denotes a mutable (resp. immutable) reference to the location held at one of the lvals $\overline{w}$. A type $\blacksquare\tau$ denotes a heap allocated value of type τ. New types have the form $\diamond\overline{\omega}$ and $\blacklozenge\tau$. A value of type $\blacklozenge\tau$ denotes an active `TRC` while $\diamond\overline{\omega}$ denotes an inactive `TRC`. Similarly to references, the type of an inactive may refer to several paths (e.g. $\diamond x, y$). Finally, a partial type ($\lfloor\tau\rfloor$) denote a value that may contain moved locations.

We reuse the definition of the expressions of FR, augmented by the following ones: (1) $\texttt{trc}(e)$: allocates a new active `Trc`, initialized with the value denoted by e, (2) $\omega.clone$ returns an inactive copy of the `Trc` denoted by ω, (3) $\texttt{spawn}(f(\overline{e}))$ runs $f(\overline{v})$ as a new thread, $\overline{v}$ are the values denoted by $\overline{e}$ and (4) `cooperate` yields the control to other threads. Other expressions are identical to those in FR. $\{\overline{e}\}^{\texttt{l}}$ denotes a block, the scope of which is expressed by the lifetime `l` where $\overline{e}$ is a sequence of expressions separated by semicolons. The lifetimes form a partial ordering (i.e. $\texttt{l} \succeq \texttt{m}$ stands for `m` is inside `l` and $\texttt{l} \succeq \texttt{l}$ is always valid) that reflects the nesting property. The expression $\hat{\omega}$ (resp. ω) denotes a non destructive (resp. destructive) read of the value held at ω. $\texttt{box}(e)$ allocates a new `Box` in the heap, initialized with the value denoted by e. Finally, when declaring functions, parameters are defined with the `mut` keyword to be consistent with the variable declaration. The shape of the signatures is well defined in Figure 3. Since, threads communicate with each other, the inactive `Trc` becomes active, and its signature is $\blacklozenge S$. Note that with an active `Trc`, we cannot perform this communication. Therefore, a check is performed to ensure that in the signatures, there are no two active `Trc`s belonging to the same shared data (this adds further restrictions to the typing rules).

3.2 Operational Semantics

We introduce an operational semantics for MSSL. A state has the form $S \rhd T_1, T_2$ where S is a program store mapping locations to partial values, and T_1, T_2 are sets of threads. Concerning locations, we use three forms: (1) $\ell_{\texttt{m}::x}$ the location that is bound to the variable x with a lifetime `m`, (2) ℓ_n the location that is not bound to any variable and (3) ℓ_ω which is any location for a path ω. A thread is a pair $(t, \{\overline{e}\}^{\texttt{l}})$ where t is the thread's name and $\{\overline{e}\}^{\texttt{l}}$ is the code which is currently executed by t under the lifetime `l`. We use a pair of threads to schedule threads in a round robin scheduling (explained latter). Rules have the form $\langle S \triangleright T_1, T_2 \xrightarrow{T}_i S' \triangleright T_1', T_2' \rangle^{\texttt{l}}$ where T is a, possibly empty, set of threads spawned during the reduction. The index i, ranging over $0, 1$, denotes either a termination computation (0) or a cooperation (1). These rules rely on a auxiliary kind of rules denoting local computation of threads. Those rules have the form $S \rhd e \rightarrow_i S' \rhd e'$.

We introduce few notations in Figure 4 and explain the semantics rules which are given in Figure 5. The notation ℓ^* stands for any possible kind of location $(\ell^{\blacksquare},\ \ell^{\blacklozenge},\ \ell^{\diamond}, \ell^{\circ})$. The notation $S[\ell_{\texttt{m}::\omega} \mapsto \langle v^{\perp} \rangle^{\texttt{m}}]$ returns a program store by modifying in S the value of the lval location ω by a partial value $v^{\perp}$. The function $drop(S, \texttt{m})$ is responsible for deallocating memory locations associated with the lifetime `m`. $drop(S, \texttt{m})$ is defined as $drop(S, \psi)$ where $\psi = \{\ell^{\blacksquare}, \ell^{\blacklozenge} \mid \ell \mapsto \langle v^{\perp} \rangle^{\texttt{m}} \in S\}$. As illustrated in Figure 4, $\text{drop}(S, \psi)$ recursively traverses the owning references dropping the location if necessary. Hence, ψ denotes a *drop set*, which identifies the locations allocated by a given `box` or `trc` $(\ell^{\blacksquare}, \ell^{\blacklozenge})$ and should be dropped when its lifetime ends. Since the deallocation of slots allocated by a `Trc` depends on the counter (denoted by i), the deallocation is performed only if the counter

$$
\begin{array}{ll}
loc(S,x) & = \ell_{\mathtt{m}::x} \text{ where}(\ell_{\mathtt{m}::x}) = \langle . \rangle^{\mathtt{m}} \text{ and } \neg\exists \mathtt{n}.(\mathtt{m} \trianglerighteq \mathtt{n} \wedge S(\ell_{\mathtt{n}::x}) = \langle . \rangle^{\mathtt{n}} \\
loc(S,*\omega) & = \ell \text{ where } loc(S,\omega) = \ell_\omega \text{ and } S(\ell_\omega) = \langle \ell^* \rangle^{\mathtt{m}} \\
read(S,\omega) & = S(\ell_\omega) \text{ where } loc(S,\omega) = \ell_\omega \text{ and } S(\ell_\omega) = \langle v \rangle^{1} \\
write(S,\omega,v^\perp) & = S[\ell_\omega \mapsto \langle v^\perp \rangle^{\mathtt{m}}] \text{ where } loc(S,\omega) = \ell_\omega \text{ and } S(\ell_\omega) = \langle . \rangle^{\mathtt{m}} \\
drop(S,\varnothing) & = S \\
drop(S,\psi \cup \{v^\perp\}) & = drop(S,\psi) \;\; where \; (v^\perp \neq \ell^{\blacksquare} \wedge v^\perp \neq \ell^{\blacklozenge} \wedge v^\perp \neq \ell^{\diamond}) \\
drop(S,\psi \cup \ell^{\blacksquare}) & = drop(S - \{\ell \mapsto \langle v^\perp \rangle^*\}, \psi \cup \{v^\perp\}) \text{ where } S(\ell) = \langle v^\perp \rangle^* \\
drop(S,\psi \cup \ell^{\blacklozenge}) & = \begin{cases} drop(S - \{\ell \mapsto \langle v^\perp \rangle^1\}, \psi \cup \{v^\perp\}) \;\; where \; S(\ell) = \langle v^\perp \rangle^1 \\ drop(S - \{\ell \mapsto \langle v^\perp \rangle^{i-1}\}, \psi\} \;\; where \; S(\ell) = \langle v^\perp \rangle^i \end{cases} \\
drop(S,\psi \cup \ell^{\diamond}) & = drop(S - \{\ell \mapsto \langle v^\perp \rangle^i\}, \psi\} \;\; where \; S(\ell) = \langle v^\perp \rangle^{i+1}
\end{array}
$$

Fig. 4: Notations

falls to 1. Indeed, when the counter is 1, it means that there is only one reference for that slot and thus the deallocation is safely performed. Furthermore, we assume that only active `Trc`'s are responsible for deallocating the locations to which they refer. Specifically, when the lifetime of an inactive `Trc` expires, it is sufficient to decrement the counter by 1.

Now, let us present and explain the MSSL reduction rules for a given thread t according to Figure 4. **R-Copy** is the first reduction rule. The borrow checker should allow this rule if and only if the lval ω has copy semantics (see Section 4). Otherwise, if ω has move semantics, the T-Move rule should be applied. In comparison with **R-Copy**, **R-Move** enforces a *destructive read* by making lval ω inaccessible. Thus, this rule is responsible for reducing lval by effectively removing its location from the resulting program store (i.e. S_2), we use $(\perp)$ to indicate that ω is read-inaccessible. Regarding dynamic allocation, MSSL handles heap allocation in two ways depending on whether it is a `Box` or a `Trc`. The **R-Box** rule creates a $fresh$ location in S to represent the new box. We note that the slots in the heap receive the *global lifetime* *, i.e. corresponds to a *static* variable in C/C++. In contrast, the **R-Trc** creates a *fresh* location in the program store representing the active `Trc` by setting the counter i to 1. As we can produce multiple instances of inactive `Trc`, which point to the same allocation on the heap as the active `Trc` source, we can use **R-Clone**. This rule increments the number of references that point to the same allocation on the heap, e.g. $\langle v \rangle^{i+1}$. The **R-Borrow** rule determines the location of the borrowed lval that already exists in S. **R-Assign** updates the value of a given lval by its new value using the $write(S,\omega,v)$ function after it removes the old value using $drop(S,\{v^\perp\})$ function. This function is responsible for dropping any location belonging to $v^\perp$. To declare a new variable in a program store, it is therefore necessary to determine its corresponding location and **R-Declare** is then introduced. With **R-Declare**, we create a new lval and we add it to S_1 where the new value takes the lifetime `l` of the enclosing block. At this point, $write(S,\omega,v)$ cannot be used because it takes as a parameter an lval that already exists in S. For reducing a sequence of expressions, we define two rules: (1) **R-Seq** which reduces the expression if the indicator i is 0 (i.e. when e is different from `cooperate` and the value ϵ_1 denotes the reduction of the `cooperate` expression). In this case, the

$$\frac{read(S,\omega) = \langle v \rangle^{\mathtt{m}}}{\langle S \triangleright \hat{\omega} \rightarrow_0 S \triangleright v \rangle^{\mathtt{1}}}\ \textbf{(R - Copy)} \quad \frac{read(S_1,\omega) = \langle v \rangle^{\mathtt{m}} \quad S_2 = write(S_1,\omega,\bot)}{\langle S_1 \triangleright \omega \rightarrow_0 S_2 \triangleright v \rangle^{\mathtt{1}}}\ \textbf{(R - Move)}$$

$$\frac{\ell_m \notin dom(S_1) \quad S_2 = S_1[\ell_{1::m} \mapsto \langle v \rangle^{*}]}{\langle S_1 \triangleright \mathtt{box}(v) \rightarrow_0 S_2 \triangleright \ell^{\blacksquare}_m \rangle^{\mathtt{1}}}\ \textbf{(R - Box)} \quad \frac{\ell_m \notin dom(S_1) \quad S_2 = S_1[\ell_{1::m} \mapsto \langle v \rangle^{\mathtt{i}}]}{\langle S_1 \triangleright \mathtt{trc}(v) \rightarrow_0 S_2 \triangleright \ell^{\blacklozenge}_m \rangle^{\mathtt{1}}}\ \textbf{(R - Trc)}$$

$$\frac{\ell^{\blacklozenge}_m = loc(S_1,\omega) \quad S_1(\ell m) = \langle v \rangle^{\mathtt{i}} \quad S_2 = S_1[\ell \mathtt{l} :: m \mapsto \langle v \rangle^{\mathtt{i+1}}]}{\langle S_1 \triangleright \omega.clone \rightarrow_0 S_2 \triangleright \ell^{\circ}_{1::m} \rangle^{\mathtt{1}}}\ \textbf{(R - Clone)}$$

$$\frac{loc(S,\omega) = \ell_\omega}{\langle S \triangleright \&[\mathtt{mut}]\omega \rightarrow_0 S \triangleright \ell^{\circ}_\omega \rangle^{\mathtt{1}}}\ \textbf{(R - Borrow)} \quad \frac{S_2 = S_1[\ell_{1::x} \mapsto v]}{\langle S_1 \triangleright \mathtt{let\ mut}\ x = v \rightarrow_0 S_2 \triangleright \epsilon \rangle^{\mathtt{1}}}\ \textbf{(R - Declare)}$$

$$\frac{read(S_1,\omega) = \langle v_1^{\bot} \rangle^{\mathtt{m}} \quad S_2 = drop(S_1,\{v_1^{\bot}\}) \quad S_3 = write(S_2,\omega,v_2)}{\langle S_1 \triangleright \omega = v_2 \rightarrow_0 S_3 \triangleright \epsilon \rangle^{\mathtt{1}}}\ \textbf{(R - Assign)}$$

$$\frac{S_2 = drop(S_1,\{v\}) \quad v \neq \epsilon_1}{\langle S_1 \triangleright v;\overline{e} \rightarrow_0 S_2 \triangleright \overline{e} \rangle^{\mathtt{1}}}\ \textbf{(R - Seq)} \quad \frac{}{\langle S_1 \triangleright \epsilon_1;\overline{e} \rightarrow_1 S_1 \triangleright \overline{e} \rangle^{\mathtt{1}}}\ \textbf{(R - SeqTerm)}$$

$$\frac{\langle S_1 \triangleright \overline{e}_1 \rightarrow_i S_2 \triangleright \overline{e}_2 \rangle^{\mathtt{m}}}{\langle S_1 \triangleright \{\overline{e}_1\}^m \rightarrow_i S_2 \triangleright \{\overline{e}_2\}^m \rangle^{\mathtt{1}}}\ \textbf{(R - BlockA)} \quad \frac{S_2 = drop(S_1,m)}{\langle S_1 \triangleright \{\overline{v}\}^m \rightarrow_0 S_2 \triangleright v \rangle^{\mathtt{1}}}\ \textbf{(R - BlockB)}$$

$$\frac{}{\langle S \triangleright \mathtt{cooperate} \rightarrow_1 S \triangleright \epsilon_1 \rangle^{\mathtt{1}}}\ \textbf{(R - Coop)}$$

$$\frac{t \in \ fresh \quad \mathcal{D}(f) = \lambda(\overline{x})\{\overline{e}\}^{\mathtt{m}} \quad \Theta(\mathtt{1} \Rightarrow \{\overline{e}\}^{\mathtt{m}}) = \{\overline{e}\}^{\mathtt{n}} \quad S_2 = S_1[\overline{\ell_{\mathtt{n}::x} \mapsto \langle v \rangle^{\mathtt{n}}}]}{\langle S_1 \triangleright \mathtt{spawn}(f(\overline{v})) \xrightarrow{\{(t,\{\overline{e}\}^{\mathtt{n}})\}}_0 S_2 \triangleright \epsilon_0 \rangle^{\mathtt{1}}}\ \textbf{(R - Spawn)}$$

$$\frac{\langle S_1 \triangleright e_1 \rightarrow_i S_2 \triangleright e_2 \rangle^{\mathtt{1}}}{\langle S_1 \rhd E[\![e_1]\!] \rightarrow_i S_2 \rhd E[\![e_2]\!] \rangle^{\mathtt{1}}}\ \textbf{(R - Sub)}$$

$$\frac{\langle S \triangleright e \xrightarrow{T_1}_0 S'' \triangleright e'' \rangle^{\mathtt{1}} \quad \langle S'' \triangleright e'' \overset{T_2}{\Longrightarrow} S' \triangleright e' \rangle^{\mathtt{1}}}{\langle S \rhd e \overset{T_1 \cup T_2}{\Longrightarrow} S' \rhd e' \rangle^{\mathtt{1}}}\ \textbf{(R - Steps1)}$$

$$\frac{\langle S \triangleright e \xrightarrow{T}_1 S' \triangleright e' \rangle^{\mathtt{1}}}{\langle S \rhd e \overset{T}{\Longrightarrow} S' \rhd e' \rangle^{\mathtt{1}}}\ \textbf{(R - Steps2)}$$

$$\frac{}{\varnothing, T_2, S \Longrightarrow T_2, \varnothing, S}\ \textbf{(R - End)}$$

$$\frac{(t,\{e\}^{\mathtt{1}}) \in T_1 \quad \langle S \rhd e \overset{T}{\Longrightarrow} S' \rhd e' \rangle^{\mathtt{1}} \quad T_{1|t}, T_2 \cup \{(t,\{e'\}^{\mathtt{1}})\} \cup T, S' \Longrightarrow T_1', T_2', S'}{T_1, T_2, S \Longrightarrow T_1', T_2', S'}\ \textbf{(R - Thread)}$$

Fig. 5: MSSL Reduction Rules

reduction is done by removing those completed from the left using $drop(S, \{v\})$. Otherwise, with (2) **R-SeqTerm**, nothing occurs if the indicator i is 1 (i.e. when e is a *cooperative* expression). The reduction of a block using the **R-BlockA** rule only terminates if there is only one value remaining. At this point, the block is completely reduced by the **R-BlockB** rule. In **R-blockB**, we deallocate any remaining owned locations using the $drop(S, \mathtt{m})$ function previously defined. **R-Coop** rule terminates the execution of the thread for the current round. We consider the **R-Spawn** rule for reducing the expression $\mathtt{spawn}(f(\overline{v}))$. This rule creates a new thread (i.e. a pair $\{(t, \{e\}^{\mathtt{n}})\}$ where t is the name of the thread and $\{e\}^{\mathtt{n}}$ the expression that t should execute with the lifetime $\mathtt{n}$. The function $\Theta(\mathtt{l} \Rightarrow \{\overline{e}\})$ is responsible for *instantiating* the lifetime of an expression $\overline{e}$. Hence, it recursively *instantiates* all lifetimes of an expression $\overline{e}$ into fresh lifetimes in $\mathtt{l}$. As usual, we rely on evaluation context to specifify the evaluation strategy. Contexts are defined below. The evaluation under contexts is defined in rule **R-Sub**.

$$E ::= [\![.]\!] \mid E; \overline{e} \mid \mathtt{let\ mut}\ x = E \mid \omega = \ E \mid \mathtt{box}(E) \mid \mathtt{trc}(E) \mid \mathtt{spawn}(f(\overline{v}, E, \overline{e}))$$

Rules R-**Steps1** and **R-Steps2** shows how an expression is evaluated upon termination (index 0) or cooperation (index 1). Rules **R-end** and **R-Thread** shows how we execute all threads in a round-robin manner. All threads are executed once upon termination or cooperation and are moved in T_2. Once all threads have been executed we then move all threads from T_2 to T_1 and restart the process. Spawned threads are added to T_2 in order to be scheduled in the next round. An execution of a program is a, possibly infinite, sequence of states $(S_1, T_1, T_1'), (S_2, T_2, T_2'), \ldots$ such that $(S_1, T_1, T_1') \Longrightarrow (S_2, T_2, T_2') \Longrightarrow \ldots$.

4 Type System

In this section, we concentrate on type and borrowing safety between threads while preventing type and borrowing errors. Here is an example statically checked in MSSL: $\{\mathtt{let\ mut}\ x = 0; \mathtt{let\ mut}\ y = \&x;\ \{\mathtt{let\ mut}\ z = 5;\ y\ =\ \&z;\}^{\mathtt{n}}\ y;\}^{\mathtt{m}}$ This program is not considered to be a safe borrowed type. However, the assignment "$y = \&z$" changes the value of y by creating a borrowed reference to the variable z that exists outside its lifetime.

We present now the shape of our typing judgment as follows: $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma}^{\mathtt{l}} \dashv \Gamma_2$ where Γ_1 is the typing environment mapping variables to a slot type $\langle \widetilde{\tau} \rangle^{\mathtt{m}}$ with an allocated lifetime $\mathtt{m}$. Afterwards, evaluating the expression e under the typing environment Γ_1 produces the environment Γ_2. The difference between the two environments forms the *effect* of the expression e, $\mathtt{l}$ is the *context lifetime* and σ is the typing store. The presence of σ in typing judgment is necessary to keep track of the heap-allocation location as described in [14]. For example: $\varnothing \triangleright \{\mathtt{let\ mut}\ x = trc(1)\};^{\mathtt{l}} \rightarrow \{\ell_1 \mapsto \langle 1 \rangle^{\mathtt{l}}\} \triangleright \{\mathtt{let\ mut}\ x = \ell_1^{\blacklozenge}; \}^{\mathtt{l}}$
As explained previously, Γ maps each variable to its type. However, in this example, ℓ_1 is not represented in any typing environment. Specifically, it refers to the location ℓ_1 allocated to the heap and this appears in the program store S. As a result, the typing store in this case has the form : $\sigma \vdash \ell_1^{\blacklozenge} : \blacklozenge\mathtt{int}$.

We present the typing rules for our calculus. Keep in mind that variables in our language can have either copy semantics or move semantics. Having said that our type system encodes both: type checking and borrow checking rules that are required to determine when it is safe to copy or move a variable.
The following introduces some support functions necessary for the typing rules:

Copy and Move Types. A type τ has a copy semantics, denoted by $copy(\tau)$, when τ is a base type (int) or τ is a shared reference $\&\overline{\omega}$. Otherwise, all other types, (mutable references, boxes and `Trc`s types) have move semantics.
To ensure MSSL safety, it is imperative to be able to determine when a location is mutable or immutable borrowed or cloned. Thus, we require a mechanism to determine if a location is borrowed as mutable (readprohibited) or as immutable (writeprohibited) or if a location is cloned (Trcprohibited):

Path, Path Conflict and Type Containment. A path π is defined as a sequence of zero ($\pi \triangleq \epsilon$) or more dereferences ($\pi \triangleq \pi'.*$). $u \triangleq \pi_u \mid x$ denotes a *destructuring* of an lval u into its base x and path π. Let $u \triangleq \pi_u \mid x$ and $\omega \triangleq \pi_u \mid y$ be lvals. Then, u is said in conflict with ω, denoted $u \bowtie \omega$, if $x = y$. Finally, let Γ be an environment where $\Gamma(x) = \langle \tilde{\tau}_1 \rangle^1$ for some `1`. Then, $\Gamma \vdash x \rightsquigarrow \tau$ indicates that x contains the type τ.

Read Prohibited. The $readProhibited(\Gamma, \omega)$ function is responsible to determine for a given lval ω if it is read prohibited where it exists for x the following: $\Gamma \vdash x \rightsquigarrow \&mut\ \overline{u} \wedge \exists\ i.(u_i \bowtie \omega)$.

Write Prohibited. The $writeProhibited(\Gamma, \omega)$ function is responsible to determine for a given lval ω if it is write prohibited where it exists for x the following: $\Gamma \vdash x \rightsquigarrow \& \overline{u} \wedge \exists\ i.(u_i \bowtie \omega)$ or $readProhibited(\Gamma, \omega)$.

SafeTrc. An environment Γ is said to be safeTrc, denoted safeTrc(Γ) if for all $\omega \in$ LVAL where $\Gamma(\omega) = \langle \blacklozenge \tau \rangle^1$, then we have $\neg writeProhibited(\Gamma, \omega)$.

Active. The $active(\Gamma, \omega)$ partial function verifies that the path for a given lval ω never crosses an inactive trc (e.g. $\blacklozenge \blacksquare \diamond \omega$).

Mutable. An lval ω is said to be mutable if it is recursively mutable, i.e. the path it describes never crosses an immutable borrow (e.g. $\blacklozenge \blacklozenge \&x$) . The $\texttt{mut}(\Gamma, \omega)$ partial function identifies that for a given ω, ω is mutable or not.

Move. The $move(\Gamma, \omega)$ partial function returns the resulting environment after moving the value of an lval ω where $\omega \triangleq \pi_x \mid x$ and $\Gamma(x) = \langle \tilde{\tau}_1 \rangle^1$. Then, $\Gamma[x \mapsto \langle \tilde{\tau}_2 \rangle^1] = move(\Gamma, \omega)$ such that $\tilde{\tau}_2 = strike(\pi_x \mid \tilde{\tau}_1)$ defined as:

$strike(\epsilon \mid \tau) = \lfloor \tau \rfloor$

$strike((\pi.*) \mid \blacksquare \tilde{\tau}_1) = \blacksquare \tilde{\tau}_2 \quad where\ \tilde{\tau}_2 = strike(\pi \mid \tilde{\tau}_1)$

The following feature is helpful to avoid dereferencing an inactive `Trc`:

Deref Prohibited. The $derefProhibited(\Gamma, \omega)$ function is responsible to determine for a given lval ω if it is dereference prohibited where if $\exists\ u, u'$ such that $\Gamma(u) = \langle \diamond u' \rangle^1$ *and* $(*u \bowtie \omega)$.

Trc Prohibited. The $TrcProhibited(\Gamma, \omega)$ function is responsible to determine for a given lval ω if it is move prohibited where if it exists for x the following: $\Gamma \vdash x \rightsquigarrow \diamond u \wedge (u \bowtie \omega)$ or derefProhibited(Γ, ω).

$$\frac{\Gamma(x) = \langle\tilde{\tau}\rangle^{\mathtt{m}}}{\Gamma \vdash x : \langle\tilde{\tau}\rangle^{\mathtt{m}}}\ (\mathbf{T-LvVar}) \quad \frac{\Gamma \vdash \omega : \langle\blacklozenge\tilde{\tau}\rangle^{\mathtt{m}}}{\Gamma \vdash *\omega : \langle\tilde{\tau}\rangle^{\mathtt{m}}}\ (\mathbf{T-LvTrc})$$

$$\frac{\Gamma \vdash \omega : \langle\&[mut]\overline{u}\rangle^{\mathtt{n}} \quad \overline{\Gamma \vdash u : \langle\tau\rangle^{\mathtt{m}}}}{\Gamma \vdash *\omega : \langle\sqcup_i\tau_i\rangle^{\sqcap_i\mathtt{m}_i}}\ (\mathbf{T-LvBorrow})$$

$$\frac{\sigma \vdash v : \tau}{\Gamma \vdash \langle v : \tau\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma}\ (\mathbf{T-Const}) \frac{\Gamma \vdash \omega : \langle\blacksquare\tilde{\tau}\rangle^{\mathtt{m}}}{\Gamma \vdash *\omega : \langle\tilde{\tau}\rangle^{\mathtt{m}}}\ (\mathbf{T-LvBox})$$

$$\frac{\Gamma \vdash \omega : \langle\tau\rangle^{\mathtt{m}} \quad copy(\tau) \quad \neg readProhibited(\Gamma,\omega)}{\Gamma \vdash \langle\hat{\omega} : \tau\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma}\ (\mathbf{T-Copy})$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \omega : \langle\tau\rangle^{\mathtt{m}} \quad \neg writeProhibited(\Gamma_1,\omega) \\ \neg TrcProhibited(\Gamma_1,\omega) \quad \Gamma_2 = move(\Gamma_1,\omega)\end{array}}{\Gamma_1 \vdash \langle\omega : \tau\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_2}\ (\mathbf{T-Move})$$

$$\frac{\Gamma \vdash \omega : \langle\tau\rangle^{\mathtt{m}} \quad \mathtt{mut}(\Gamma,\omega) \quad \neg writeProhibited(\Gamma,\omega)}{\Gamma \vdash \langle\&\mathtt{mut}\ \omega : \&\mathtt{mut}\ \omega\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma}\ (\mathbf{T-MutBorrow})$$

$$\frac{\Gamma \vdash \omega : \langle\tau\rangle^{\mathtt{m}} \quad \neg readProhibited(\Gamma,\omega)}{\Gamma \vdash \langle\&\ \omega : \&\ \omega\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma}\ (\mathbf{T-ImmBorrow})$$

$$\frac{\Gamma_1 \vdash \langle e : \tau\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_2}{\Gamma_1 \vdash \langle\mathtt{trc}(e) : \blacklozenge\tau\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_2}\ (\mathbf{T-Trc}) \quad \frac{\Gamma_1 \vdash \langle e : \tau\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_2}{\Gamma_1 \vdash \langle\mathtt{box}(e) : \blacksquare\tau\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_2}\ (\mathbf{T-Box})$$

$$\frac{\Gamma \vdash \omega : \langle\blacklozenge\tau\rangle^{\mathtt{m}} \quad active(\Gamma,\omega)}{\Gamma \vdash \langle\omega.clone : \diamond\omega\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma}\ (\mathbf{T-Clone})$$

$$\frac{\Gamma_1 \vdash \langle e_1 : \tau_1\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_2 \quad \ldots \quad \Gamma_n \vdash \langle e_n : \tau_n\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_{n+1}}{\Gamma_1 \vdash \langle\overline{e} : \tau_n\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_{n+1}}\ (\mathbf{T-Sequence})$$

$$\frac{\Gamma_1 \vdash \langle\overline{e} : \tau\rangle^{\mathtt{m}}_{\sigma} \dashv \Gamma_2 \quad \Gamma_2 \vdash \mathtt{m} \geq \mathtt{1} \quad \Gamma_3 = drop(\Gamma_2,\mathtt{m})}{\Gamma_1 \vdash \langle\{\overline{e}\}^{\mathtt{m}} : \tau\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_3}\ (\mathbf{T-Block})$$

$$\frac{x \notin dom(\Gamma_1) \quad \Gamma_1 \vdash \langle e : \tau\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_2 \quad \Gamma_3 = \Gamma_2\,[x \mapsto \langle\ \tau\ \rangle^{\mathtt{m}}]}{\Gamma_1 \vdash \langle\mathtt{let\ mut}\ x = e : \epsilon\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_2}\ (\mathbf{T-Declare})$$

$$\frac{\begin{array}{c}\Gamma \vdash \omega : \langle\tilde{\tau}_1\rangle^{\mathtt{m}} \quad \Gamma_1 \vdash \langle e : \tau_2\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_2 \quad \Gamma_2 \vdash_{\mathtt{ff}} \tilde{\tau}_1 \approx \tau_2 \\ \Gamma_2 \vdash \tau_2 \geq \mathtt{m} \quad \Gamma_3 = write^0(\Gamma_2,\omega,\tau_2) \\ \neg writeProhibited(\Gamma_3,\omega) \quad \neg TrcProhibited(\Gamma_3,\omega)\end{array}}{\Gamma_1 \vdash \langle\omega = e : \epsilon\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_2}\ (\mathbf{T-Assign})$$

$$\frac{safeTrc\ (\Gamma)}{\Gamma \vdash \langle\mathtt{cooperate} : \epsilon_1\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma}\ (\mathbf{T-Coop})$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \langle\overline{e : T}\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_2 \\ \Gamma_2 \vdash (\overline{S}) \Longleftarrow (\overline{\tau})\end{array}}{\Gamma_1 \vdash \langle\mathtt{spawn}(f(\overline{e})) : \epsilon\rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_2}\ (\mathbf{T-Spawn})$$

Fig. 6: MSSL Typing Rules

We introduce more notions in Figure 7 as follows: The $drop(\Gamma, \omega)$ function deallocates locations with a lifetime $\mathtt{m}$, by removing them from an environment Γ. The $write^k(\Gamma, \omega, \tau)$ partial function updates the type of a given lval ω where $\omega \triangleq \pi_x \mid x$ and $\Gamma(x) = \langle\tilde{\tau}_1\rangle^{\mathtt{l}}$. For some $k \geq 0$, this function is defined as $\Gamma_2[x \mapsto \langle\tilde{\tau}_2\rangle^{\mathtt{l}}]$ where $(\Gamma_2, \tilde{\tau}_2) = update^k(\Gamma, \pi_x \mid \tilde{\tau}_1, \tau)$ presented in Figure 7. Then, for an environment Γ, two partial types $\tilde{\tau}_1$ and $\tilde{\tau}_2$ are said to be *shape compatible*, denoted as $\Gamma \vdash \tilde{\tau}_1 \approx \tilde{\tau}_2$, according to the rules (**S-***). Moreover, Γ is said to be *well-formed* with respect to a lifetime $\mathtt{l}$, denoted $\Gamma \vdash \tau \geq \mathtt{l}$, according to the rules (**L-***): Now we describe the typing rules for lvals and ex-

$$
\begin{array}{ll}
drop(\Gamma, \mathtt{m}) & = \Gamma - \{x \mapsto \langle\tilde{\tau}\rangle^{\mathtt{m}} \mid x \mapsto \langle\tilde{\tau}\rangle^{\mathtt{m}} \in \Gamma\} \\
update^0(\Gamma, \epsilon \mid \tilde{\tau}_1, \tau_2) & = (\Gamma, \tau_2) \\
update^{k\geq 1}(\Gamma, \pi \mid \tau_1, \tau_2) & = (\Gamma, \tau_1 \sqcup \tau_2) \\
update^k(\Gamma, (\pi.*) \mid \blacksquare\tilde{\tau}_1, \tau) & = (\Gamma_1, \blacksquare\tilde{\tau}_2) \quad where\ (\Gamma_1, \tilde{\tau}_2) = update^k(\Gamma, \pi \mid \tilde{\tau}_1, \tau) \\
update^k(\Gamma, (\pi.*) \mid \blacklozenge\tau_1, \tau) & = (\Gamma_1, \blacklozenge\tau_2) \quad where\ (\Gamma_1, \tau_2) = update^k(\Gamma, \pi \mid \tau_1, \tau) \\
update^k(\Gamma, (\pi.*) \mid \&\mathtt{mut}\ \overline{u_i}, \tau) & = (\sqcup_i \Gamma_i, \&\mathtt{mut}\ u_i)
\end{array}
$$

$$
\frac{}{\Gamma \vdash int \approx int}\ \textbf{(S-Int)} \quad \frac{\Gamma \vdash \tau_1 \approx \tilde{\tau}_2}{\Gamma \vdash \lfloor\tau_1\rfloor \approx \tilde{\tau}_2}\ \textbf{(S-UnL)} \quad \frac{\forall_{i,j}(\Gamma \vdash u_i : \tilde{\tau}_1 \approx \tilde{\tau}_2 : \omega_j \dashv \Gamma)}{\Gamma \vdash \&[\mathtt{mut}]\overline{u} \approx \&[\mathtt{mut}]\overline{\omega}}\ \textbf{(S-Bor)}
$$

$$
\frac{\Gamma \vdash \tilde{\tau}_1 \approx \tilde{\tau}_2}{\Gamma \vdash \blacksquare\tilde{\tau}_1 \approx \blacksquare\tilde{\tau}_2}\ \textbf{(S-Box)} \quad \frac{\Gamma \vdash \tilde{\tau}_1 \approx \tilde{\tau}_2}{\Gamma \vdash \blacklozenge\tilde{\tau}_1 \approx \blacklozenge\tilde{\tau}_2}\ \textbf{(S-ATrc)} \quad \frac{\Gamma \vdash \tilde{u} : \tau_1 \approx \tilde{\tau}_2 : \omega \dashv \Gamma}{\Gamma \vdash \diamond u \approx \diamond\omega}\ \textbf{(S-ITrc)}
$$

$$
\frac{\Gamma \vdash \tilde{\tau}_1 \approx \tau_2}{\Gamma \vdash \tilde{\tau}_1 \approx \lfloor\tau_2\rfloor}\ \textbf{(S-UnR)} \quad \frac{\Gamma \vdash \tilde{\tau}_1 \approx \tilde{\tau}_2 : \omega \dashv \Gamma}{\Gamma \vdash \blacklozenge\tilde{\tau}_1 \approx \diamond\omega}\ \textbf{(S-ATrcL)} \quad \frac{\Gamma \vdash u : \tilde{\tau}_1 \approx \tilde{\tau}_2 \dashv \Gamma}{\Gamma \vdash \diamond u \approx \blacklozenge\tilde{\tau}_2}\ \textbf{(S-ATrcR)}
$$

$$
\frac{}{\Gamma \vdash \mathtt{int} \geq \mathtt{l}}\ \textbf{(L-Int)} \quad \frac{\Gamma \vdash \tau \geq \mathtt{l}}{\Gamma \vdash \blacksquare\tau \geq \mathtt{l}}\ \textbf{(L-Box)} \quad \frac{\Gamma \vdash \tau \geq \mathtt{l}}{\Gamma \vdash \blacklozenge\tau \geq \mathtt{l}}\ \textbf{(L-AcTrc)}
$$

$$
\frac{\Gamma \vdash \tau \geq \mathtt{l}}{\Gamma \vdash \diamond\tau \geq \mathtt{l}}\ \textbf{(L-IncTrc)} \quad \frac{\overline{\Gamma \vdash u : \langle\tau\rangle^{\mathtt{m}}} \quad \overline{\mathtt{m} \geq \mathtt{l}}}{\Gamma \vdash \&[\mathtt{mut}]\,\overline{u} \geq \mathtt{l}}\ \textbf{(L-BoR)}
$$

Fig. 7: Notations of typing rules

pressions in MSSL according to Figure 6: An lval ω is *well-typed* with respect to Γ, denoted $\Gamma \vdash\ \omega : \langle\tilde{\tau}\rangle^{\mathtt{m}}$, by the following rules (**T-Lv***) presented in Figure 6. Consider the following example: $\Gamma = \{y \mapsto \blacklozenge int,\ b \mapsto \diamond y,\ c \mapsto \&y\}$ it follows that $*y : int$ and $*c : int$. However, *b cannot be typed. Additionally, lvals can have partial types as long as their internal "*path*" is defined, e.g. $\Gamma = \{x \mapsto \blacksquare\lfloor\blacksquare int\rfloor\}$ it follows that x and $*x$ can be typed, while $**x$ cannot. Finally, we denote by $\sqcap_{\mathtt{i}}\mathtt{m_i}$ (see **T-LvBorrow**) the lowest lifetime of $\mathtt{m}_0, \ldots, \mathtt{m}_\mathtt{n}$ such that the sequence of active lifetimes is always well defined in an expression. The **T-Copy** rule handles copying out of an lval that has copy semantics. Hence, it is safe to leave the output environment unchanged. This rule requires that ω is not borrowed as a mutable using $\neg readProhibited(\Gamma, \omega)$ function. The value returned will take on a new lifetime. With **T-Move**, instead of copying the value, we move it out of ω that has move semantics. Nonetheless, this rule requires two constraints: (1) ω must not be borrowed or cloned in its environment, which is checked by $\neg$

writeProhibited(Γ_1, ω) and $\neg$*TrcProhibited*(Γ_1, ω) respectively. (2) Since with an inactive `Trc` we cannot access the values of the heap, we then must prevent its dereferencing. This behavior is captured by the *derefProhibited*(Γ_1, ω) function imposed in $\neg$ *TrcProhibited*(Γ_1, ω) function. After validating these conditions, the output environment removes ω using $move(\Gamma_1, w)$. The **T-MutBorrow** rule requires that ω is not write prohibited in order to borrow it safely. However, the **T-ImmBorrow** rule requires that ω is not read prohibited. In both cases, the type of ω must be well defined $\langle\tau\rangle^{\mathtt{m}}$. In MSSL, dynamic allocation is handled either by returning a box type with the **T-Box** rule, or an active `Trc` type with **T-Trc**. These types represent an owned pointer to a dynamically allocated location in the heap. According to these rules, if e induces the move of a variable, we observe this effect in `box(`e`)` (or `trc(`e`)`) as well. Finally, **T-clone** returns an inactive `Trc` type by enforcing two requirements: (1) ω has to be of `Trc` type. (2) ω must be recursively active. This property is checked by the *active*(Γ, ω) function. The **T-Sequence** rule captures how variable environments are threaded into programs. The environment generated after evaluating an expression in a sequence is simply added to the next following environment, since the type checking of some expressions modifies the environment. Furthermore, the type of a sequence is determined by the final expression in it. The **T-Block** rule builds on **T-Sequence** for body processing. This rule exploits the lifetime associated with a given block to determine which variables should be dropped while ensuring that there are no dangling references: the *drop*$(\Gamma, \mathtt{m})$ function deallocates variables that have a lifetime `m`, as illustrated in Figure 7. With **T-Declare**, the creation of a new location (owner) requires that the variable does not already exists in the environment. Then, this rule produces an output environment by adding the new variable whose lifetime matches that of the enclosing block. **T-Assign** requires several conditions: (1) ω should not be borrowed or cloned. This is checked with the $\neg$*writeProhibited*(Γ_3, ω) and $\neg$*TrcProhibited*(Γ_3, ω) functions, to ensure that an unchecked mutation does not occur in the presence of aliasing, (2) ω can have a partial type, (3) $\tilde{\tau}_1$ and τ_2 must be *shape compatible* $\tilde{\tau}_1 \approx \tau_2$, (4) the new type τ_2 has to be *well-formed* with respect to lifetime `l` (i.e., $\tau_2 \geq \mathtt{m}$ as a subtype requirement). The expression `cooperate` means that the current thread has finished its execution at the current instant (round); in this case, another thread ready to execute assumes control. This implies that the latter can share the data with the current thread through `Trc` extension. Thus, to ensure memory safety and to avoid errors that may be produced, it is necessary to guarantee that in the current typing environment, there are no variables of `Trc` type that are borrowed. Therefore, this must be checked by the *safeTrc*(Γ) function with **T-Coop** rule. Finally, we introduce the **T-Spawn** rule which ensures the typing of arguments on $\overline{e}$ from left to right, denoted by $\Gamma_1 \vdash \langle\overline{e : T}\rangle_{\sigma}^{\mathtt{l}} \dashv \Gamma_2$. Afterwards, it is necessary to guarantee that the signatures in the declared function are compatible with the arguments types when the function is called. For this reason, we define the following mechanism: $\Gamma \vdash (\overline{S}) \Longleftarrow (\overline{\tau})$. The latter is used to verify two constraints, in order to avoid: (1) the presence of two inactives `Trc`s with the same

source `Trc`, e.g. $\Gamma \vdash (\blacklozenge\texttt{int}, \blacklozenge\texttt{int}) \Longleftarrow (\diamond x, \diamond x)$; (2) the presence of an active `Trc` in the arguments of the function, e.g. $\Gamma \vdash (\blacklozenge\texttt{int}) \Longleftarrow (\blacklozenge\texttt{int})$, where, in this case, the signature $(\blacklozenge\texttt{int})$ is incompatible with the active `Trc` type $(\blacklozenge\texttt{int})$. For example, consider the following program: $\texttt{fn}\ f(a : \blacklozenge\texttt{int},\ b : \blacklozenge\texttt{int})\ \{..\}^{1}$ $\{\texttt{let mut}\ x = \texttt{e};\ \texttt{let mut}\ y = x.clone;\ \texttt{spawn}(f(x, y));\}^{\texttt{m}}$. When $f(x, y)$ is called and according to the type of a, $\Gamma(x) = \blacklozenge\texttt{int}$ and then $\Gamma(y) = \diamond x$. Consequently, $\texttt{spawn}(f(x, y))$ is not valid.

5 Soundness

In this section, we state our main results which are progress and preservation. Both notions are related to the notion of validity. Let S be a program store, let $\ell_1, \ell_2 \in S$. $S \vdash \ell_1 \rightsquigarrow \ell_2$ if $S(\ell_1) = \ell_2$ or $\exists\ \ell_3 \in S$ such that $S(\ell_1) = \ell_3$ and $S \vdash \ell_3 \rightsquigarrow \ell_2$. We note $S \vdash_{\text{u}} \ell_1 \rightsquigarrow \ell_2$ if $S \vdash \ell_1 \rightsquigarrow \ell_2$ and whenever there exists $\ell_3 \in S,\ \ell_3 \neq \ell_1$ such that $S \vdash \ell_3 \rightsquigarrow \ell_2$ then either $S \vdash \ell_3 \rightsquigarrow \ell_1$ or $S \vdash \ell_1 \rightsquigarrow \ell_3$.

- Let e be an expression where $\overline{v} \in e$ is the sequence of all distinct values it contains. We say that e is *well-formed* if $\neg\exists\ i, j.(i \neq j \wedge \exists\ \ell^{\blacklozenge}.(v_i = v_j = \ell^{\blacklozenge}))$ $\wedge\ \neg\exists\ i, j.(i \neq j \wedge \exists\ \ell.(v_i = v_j = \ell))$
- Let S be a program store. For every thread t, let $\overline{v^t}$ be the sequence of all distinct values reachable from t. Then S is said to be a *valid store* if for all t and $\ell^{\blacklozenge}$ we have $\ell^{\blacklozenge} = v_i^t = v_j^t \Rightarrow i = j$ and, moreover, for all t, t' and ℓ, if $\ell = v_i^t = v_j^{t'}$, where $(t, i) \neq (t', j)$, then there exists m, n and $\ell^{\blacklozenge}$ such that $v_m^t = v_n^{t'} = \ell^{\blacklozenge}$ and $S \vdash \ell^{\blacklozenge} \rightsquigarrow \ell$.
- Let $S \triangleright T_1, T_2$ be a program state and let $\overline{e}$ be the thread expressions in $T_1 \cup T_2$ where S and $e_1, \ldots, e_n$ are valid. For all thread t, let $\overline{v^t}$ and $\overline{u^t}$ be the sequence of all distinct values reachable from t in S and e^t respectively. Then $S \triangleright T_1, T_2$ is said to be valid if for all t and $\ell^{\blacklozenge}$ we have $\ell^{\blacklozenge} = v_i^t = u_j^t \Rightarrow i = j$ and, moreover, for all t, t' and ℓ, if $\ell = v_i^t = u_j^{t'}$, where $(t, i) \neq (t', j)$, then there are m, n and $\ell^{\blacklozenge}$ such that $v_m^t = u_n^{t'} = \ell^{\blacklozenge}$ and $S \vdash \ell^{\blacklozenge} \rightsquigarrow \ell$.

5.1 Safe Abstraction

An important connection exists between runtime program stores S, and typing environments Γ. Let us note that S is a safe abstraction by Γ when, for every variable in the typing environment, its value exists in the program store with the appropriate type. Let S be a program store, $v^{\perp}$ a partial value and $\tilde{\tau}$ a partial type. Then, $v^{\perp}$ is abstracted by $\tilde{\tau}$ in S, denoted $S \vdash v^{\perp} \sim \tilde{\tau}$, according to the following rules:

$$\frac{}{S \vdash \epsilon \sim \epsilon}(\textbf{V-Unit}) \quad \frac{}{S \vdash c \sim int}(\textbf{V-Int}) \quad \frac{}{S \vdash \perp \sim \lfloor \tau \rfloor}(\textbf{V-Undef}) \quad \frac{\exists\ i.(loc(S, \omega_i) = \ell)}{S \vdash \ell^{\circ} \sim \&\lfloor mut \rfloor \overline{\omega}}(\textbf{V-Borrow})$$

$$\frac{S(\ell) = \langle v^{\perp} \rangle^{1} \quad S \vdash v^{\perp} \sim \tilde{\tau}}{S \vdash \ell^{\blacklozenge} \sim \blacklozenge \tilde{\tau}}(\textbf{V-Trc}) \quad \frac{S(\ell) = \langle v^{\perp} \rangle^{1} \quad S \vdash v^{\perp} \sim \tilde{\tau}}{S \vdash \ell^{\blacksquare} \sim \blacksquare \tilde{\tau}}(\textbf{V-Box}) \quad \frac{\exists\ i.(loc(S, \omega_i) = \ell)}{S \vdash \ell^{\circ} \sim \diamond \overline{\omega}}(\textbf{V-Clone})$$

Let S be a program store, σ a store typing, and e an expression where $\overline{v} \in e$ is the sequence of distinct values it contains. Then, σ is valid for state S$\triangleright$e, denoted S$\triangleright$ e $\vdash \sigma$, if $\forall i.(S \vdash v_i \sim \sigma(v_i))$. Let Γ to be a typing environment for a given thread t and S a program store where S contains the locations of the variables reachable from t. Let $\mathcal{L}$ be the set of all heap locations, denoted ℓ_n. Then, S is *safely abstracted* by Γ, denoted $S \sim \Gamma$, iff (dom(S)-$\mathcal{L}$) = $\Theta(dom(\Gamma))$ and for all $x \in dom(\Gamma)$ we have $(S \vdash v^{\perp} \sim \tilde{\tau})$ where S(ℓ_x) = $\langle v^{\perp} \rangle^{\mathtt{l}}$ and $\Gamma(x) = \langle \tilde{\tau} \rangle^{\mathtt{l}}$. For a set of variable identifiers, ϕ, $\Theta(\phi) = \{\ell_x \mid x \in \phi\}$.

5.2 Progress and Preservation

An important invariant about typing environments is (1) to avoid the presence of invalid borrowings (2) to ensure that for every inactive `Trc` there is always one or more active `Trc`. This property is named the borrow invariant and it is captured in the *well-formedness* property over environments:
Let Γ be a typing environment, Γ is well-formed with respect to some lifetime `l` if :

1. for all $x \in dom(\Gamma)$ and $\overline{\omega} \in LVal^{+}$ where $\Gamma \vdash x \rightsquigarrow \&[mut]\overline{\omega} \wedge \Gamma(x) = \langle . \rangle^{\mathtt{n}}$, we have $\overline{\Gamma \vdash \omega : \langle \tau \rangle^{\mathtt{m}} \wedge \mathtt{m} \geq \mathtt{n}}$
2. for all $x \in dom(\Gamma)$ where $\Gamma(x) = \langle . \rangle^{\mathtt{n}}$, we have $\mathtt{n} \geq \mathtt{l}$;(3) for all $x \in dom(\Gamma)$ and $\overline{\omega} \in LVal^{+}$, if $\exists\ i$ *where* $\Gamma \vdash x \rightsquigarrow \diamond\omega_i$, then we have $\Gamma(\omega_i) = \langle \blacklozenge\tau \rangle^{\mathtt{m}}$ where $\mathtt{m} \geq \mathtt{l}$.

Proposition 1 (Progress). *Let T_1, T_2, S be a global state; let σ be a store where $S \triangleright e \vdash \sigma$ for all $(t, \{e\}^{\mathtt{l}}) \in T_1 \cup T_2$. Let $\Gamma_1^1, \ldots \Gamma_1^n$, where n is the cardinality of $T_1 \cup T_2$, be well formed typing environments with respect to lifetimes $\mathtt{l}_1, \ldots, \mathtt{l}_n$ respectively; Let $\Gamma_2^1, \ldots \Gamma_2^n$ be typing environments and let $\tau_1, \ldots, \tau_n$ be some types. If for all $(t_i, \{e_i\}^{\mathtt{l}_i}) \in T_1$ we have $\Gamma_1^i \vdash \langle e_i : \tau_i \rangle_{\sigma}^{\mathtt{l}} \dashv \Gamma_2^i$ then either all threads are terminated or there exists T_1', T_2', S' such that $T_1, T_2, S \Longrightarrow T_1', T_2', S'$, or T_1, T_2, S diverges.*

Proposition 2 (Preservation). *Let T_1, T_2, S_1 be a valid state and $(t, \{e\}^{\mathtt{n}}) \in T_1$ and let σ be a valid store typing such that $S_1 \triangleright e \vdash \sigma$. Let Γ_1 be a well formed environment with respect a lifetime `l` where $S_1 \sim \Gamma_1$; let Γ_2 be a typing environment and let τ be a type and T be a set of threads. If $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma}^{\mathtt{l}} \dashv \Gamma_2$ and $\langle S \triangleright e \overset{T}{\Longrightarrow} S_2 \triangleright e' \rangle^{\mathtt{n}}$, then $T_{1|t}$, $T_2 \cup (t, \{e\}^{\mathtt{n}}) \cup T, S_2$ remains valid where $S_2 \sim \Gamma_2$ and $S_2 \vdash v \sim \tau$.*

6 Related Work

Several works have been up to date to carry out on the Rust type system and verification of Rust programs. Pearce [21] has developed FR, a lightweight formal programming language that represents a subset of the Rust language containing the safe part, including model boxes explicitly. This work is inspired by Featherweight [11] Java to produce a relatively lightweight formalization of Rust. The

FR most closely resembles Rust 1.0, which introduced lifetimes based on the lexical structure of programs. This work provides a type system that enforces type and borrowing safety and preserves the borrowing invariant. Weiss et al. [24] developed Oxide. Like FR, Oxide is a formalized programming language that represents a similar version of the Rust source code, especially on the formalization of the ownership system. Unlike FR, Oxide does not manage dynamic allocation of the heap and allows for Non-Lexical Lifetimes [19] by introducing a new view of lifetimes as an approximation of the provenances of references. Meanwhile, Jung et al. [13] developed a formal semantics and type system of a language called λ_{Rust}. Unlike Fr and oxide, λ_{Rust} is much closer to the Mid-level Intermediate Representation (MIR) [18] than to Rust source and it has been implemented in Coq. In addition, this work studies the ownership discipline of Rust in the presence of unsafe code.

7 Conclusion and Perspective

We have proposed an extension of the FR language dedicated to cooperative concurrent programming. This extension is a subset of our programming language MSSL. Compared to FR, we have proposed a new kind of smart pointers which allows threads to communicate without need for locking primitives. We provided an operational semantics and a type system ensuring memory safety of well-typed programs. In future work, we will provide an implementation of the type system for full MSSL and provide a translation, for well typed programs, of the latter to Fairthreads. We also plan to propose a Rust Crate based on our programming model. Such an extension will use [13] to ensure that safety is preserved.

References

1. Implications of rewriting a browser component in rust – mozilla hacks - the web developer blog.
2. The rust programming language. *http://www.rust-lang.org.*
3. Henry Baker. 'use-once' variables and linear objects - storage management, reflection and multi-threading. *SIGPLAN Notices*, 30:45–52, 01 1995.
4. Gérard Berry. The foundations of esterel. pages 425–454, 01 2000.
5. Gérard Berry, Philippe Couronne, and Georges Gonthier. Synchronous programming of reactive systems: an introduction to esterel. 08 1988.
6. F. Boussinot. "FairThreads: Mixing Cooperative and Preemptive Threads in C.". *Concurrency and Computation: Practice and Experience,*, 2005.
7. F. Boussinot and J-F. Susini. "Java Threads and SugarCubes". *Software Practice and Experience*, 2000.
8. Frédéric Boussinot. Fairthreads: Mixing cooperative and preemptive threads in c. *Concurrency and Computation: Practice and Experience*, 18:445–469, 04 2006.
9. Dave Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 33, 08 1998.

10. Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79:1305 – 1320, 10 1991.
11. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java - a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems*, 23, 11 1999.
12. Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. 08 1996.
13. Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2:1–34, 12 2017.
14. Mats Kindahl. Review of "types and programming languages by benjamin c. pierce", mit press, 2002. *SIGACT News*, 37:29–34, 01 2006.
15. Paul LeGuernic, Thierry Gautier, Michel Borgne, and Claude Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79:1321 – 1336, 10 1991.
16. F. Boussinot M. Serrano and B. Serpette. "Scheme Fair Threads.". *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming, pages 203–214, New York, NY, USA,*, 2004.
17. Louis Mandel and Marc Pouzet. ReactiveML : un langage fonctionnel pour la programmation réactive. *Technique et Science Informatiques (TSI)*, 27(9–10/2008):1097–1128, 2008.
18. Nicholas D. Matsakis. Introducing mir. *https://blog.rust-lang.org/2016/04/19/MIR.html.*, 2016.
19. Nicholas D. Matsakis. Non-lexical lifetimes: introduction. *http://smallcultfollowing.com/babysteps/blog/2016/04/27/non-lexical-lifetimes-introduction/. Accessed: 2019-02-28.*, 2016.
20. Naftaly Minsky. Towards alias-free pointers. 08 2001.
21. David Pearce. A lightweight formalism for reference lifetimes and borrowing in rust. *ACM Transactions on Programming Languages and Systems*, 43:1–73, 04 2021.
22. Mozilla Research. Rust book. 2019.
23. MSRC Team. We need a safer systems programming language – microsoft security response center.
24. Aaron Weiss, Daniel Patterson, Nicholas Matsakis, and Amal Ahmed. Oxide: The essence of rust. 03 2019.
25. Paul Wilson. *Uniprocessor Garbage Collection Techniques*, volume 637, pages 1–42. 04 2006.

A Appendix

Example 1: One of the main objectives of the newly introduced `Trc` extension is to ensure that two active `Trc`s do not point to the same location in the same thread. The following example reflects the aforementioned and of course it is rejected by our type system:

```
fn g(a : ◆ ◆ int, b : ◆ ◆ int) {...}[1]
fn f(r : ◆ ◆ int, q : ◆ ◆ int) { spawn(g(r.clone, q.clone)); }[1]
1. {                                                              (1)
2.     let mut x = trc(trc(0));
3.     spawn(f(x.clone, x.clone));
4. }[m]
```

In the block that covers lines 1 to 4, we declare one variable x of active `Trc` type (◆ ◆ `int`). Then we create a new thread that will execute the code of the function f. Both variable parameters in the function f are of type (◆ ◆ `int`). The **T-Spawn** rule returns an error on line 3, as the two arguments of the function f do not satisfy the *unicity of mutable* constraint because $x.clone$ and $x.clone$ both point to the same location as x. This leads to the creation of two active `Trc`s at the same location in the new thread environment.

Example 2: Another necessary constraint to ensure the *unicity of mutable* is to prevent threads to communicate between them using active `Trc`. To elaborate, we consider the following example:

```
fn g(a : ◆int, b : ◆int) {...}[1]
fn f(r : ◆ ◆ int, q : ◆ ◆ int) { spawn(g(∗r, ∗q)); }[1]
1. {
2.     let mut = x trc(trc(0));
3.     let mut = y trc(trc(0));                                   (2)
4.     spawn(f(x.clone, y.clone));
5.    let mut a = trc(5);
6.    ∗x = a.clone;
7.    ∗y = a.clone;
8. }[m]
```

In the function f that takes two parameters with (◆ ◆ `int`) as its signatures, we create a new thread that will execute the code of the function g. Since, the expression $*r$ moves the content of an active `Trc`, an error is displayed such as '`Arc`' or '`Rc`' in Rust, where also it cannot move out of an active '`Trc`'.

Example 3: One of the most important purpose of Rust borrow checker is to statically guarantee the validity of pointers (e.g., there are no dangling pointers). We ensure this property with the use of the cooperate expression where another thread assumes control. The following example explains how this works:

```
fn g(a : ◆ ◆ int, b : ◆ ◆ int) {...}[1]
fn f(r : ◆ ◆ int, q : ◆ ◆ int) {
   spawn(g(r.clone, q.clone));
   let mut a = &r; cooperate; }[1]
1. {
2.     let mut x = trc(trc(0));                    (3)
3.     let mut y = trc(trc(0));
4.     spawn(f(x.clone, y.clone));
5.    let mut a = trc(5);
6.    *x = a.clone;
7.    *y = a.clone;
8. }[m]
```

In the function f that takes two parameters with (◆ ◆ `int`) as its signatures, we create a new thread that will execute the code of the function g. No error occurs on this line. Subsequently, we create a reference to the variable r and we bind it to 'a' and the local thread terminates the function with the `cooperate` statement to give the control to another thread. The **T-Coop** rule returns an error because there is no reference to a `Trc` in the local environment (i.e. shared memory) to make certain that a thread will not overwrite the shared memory of other threads. Consequently, this example is rejected by the MSSL type system.