Formalizing Rust's Type System

Jacques-Henri Jourdan

February 15th, 2023

Date of the exam

The date of the exam is March 8th, 2023 (09:00).

If you cannot be available (another exam...), speak now!

Abstract from last weeks

During the last three weeks, we had an introduction to Rust programming

- Type system enforces: “mutation XOR aliasing”,
- Traits: an abstraction mechanism, sometimes at zero-cost.
- Unsafe blocks/functions: workaround strong static type-checking constraints,
- Encapsulation: clients can safely use libraries written with unsafe code.
- Interior mutability (the ability to mutate through shared borrows): a typical example of well-encapsulated unsafe code.
- Rust is a good fit for multithreading.
  - Protects against data races (with `Send` and `Sync`).
  - Provides abstractions for multithreading.

Today

I keep telling you Rust is type-safe.
How can I be so sure?

I.e., is this something one can prove formally?

1 Introduction

2 Syntactic type system

- λ_{Rust}
- Type system

The problem we are trying to solve

"Well-typed Rust programs do not go wrong."

Really?

The problem we are trying to solve

"Well-typed Rust programs **not using** `unsafe` do not go wrong."

Is that all?

The problem we are trying to solve

"Well-typed Rust programs **not using** `unsafe` do not go wrong."

A vast majority of real-life Rust programs (directly or indirectly) use unsafe code.
Example: `Vec`, interior mutability (e.g., `Cell`), low-level optimizations, ...

We want an extensible theorem to prove those programs safe too.

- Safe pieces of code are safe thanks to syntactic typing rules.
- Unsafe pieces are safe thanks to a specialized proof.

Both kinds of proofs will be linked together thanks to a logical relation.

Logical relations for type safety

The general approach

A type system is defined from:

- One (or several) typing judgment(s) $\cdots \vdash \cdots$
- Syntactic typing rules.
  Ex. for lambda-calculus: $\dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$

The logical relation approach to type soundness (i.e., semantic type soundness):

1. Define a semantic typing judgment $\cdots \models \cdots$ from the operational semantics.
2. Prove the fundamental theorem: "$\cdots \vdash \cdots \Rightarrow \cdots \models \cdots$"
3. Prove adequacy for the logical relation: "$\cdots \models \cdots \;\Rightarrow\;$ safety".
   - "Semantically well-typed programs do not go wrong."
   - Usually easy from the definition of $\cdots \models \cdots$.

Why is this approach more extensible than subject reduction+progress?

Extending semantic type safety

The fundamental theorem: by induction over the typing tree using semantic typing rules:

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{\dots \vdash \dots}{\dots \vdash \dots} \\ \vdots\end{array} \qquad \dots \vdash \dots}{\dots \vdash \dots} \qquad \dots \vdash \dots}{\dots \vdash \dots} \qquad \Rightarrow \qquad \dfrac{\dfrac{\begin{array}{c}\dfrac{\dots \vDash \dots}{\dots \vDash \dots} \\ \vdots\end{array} \qquad \dots \vDash \dots}{\dots \vDash \dots} \qquad \dots \vDash \dots}{\dots \vDash \dots}$$

Extending semantic type safety

The fundamental theorem: by induction over the typing tree using semantic typing rules:

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{\cdots\vdash\cdots}{\cdots\vdash\cdots}\\ \vdots\end{array}\qquad \cdots\vdash\cdots}{\cdots\vdash\cdots}\qquad \cdots\vdash\cdots}{\cdots\vdash\cdots} \quad\Rightarrow\quad \dfrac{\dfrac{\begin{array}{c}\dfrac{\cdots\models\cdots}{\cdots\models\cdots}\\ \vdots\end{array}\qquad \cdots\models\cdots}{\cdots\models\cdots}\qquad \cdots\models\cdots}{\cdots\models\cdots}$$

To prove $\cdots \models \cdots$ for the whole program, we don't need $\cdots \vdash \cdots$ everywhere.
We can fill some "holes" using custom proofs of $\cdots \models \cdots$.

Extending semantic type safety

`Cell` is not syntactically well-typed in safe Rust: “$\cdots \nvdash$ `Cell::get` $: \cdots$”.

But we can prove “$\cdots \vDash$ `Cell::get` $: \cdots$”.

Combining with an otherwise syntactically well-typed program:

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{\cdots \vdash \cdots}{\cdots \vdash \cdots} \\ \vdots\end{array} \qquad \cdots \vDash \texttt{Cell::get} : \cdots}{\cdots \vDash \cdots} \qquad \cdots \vdash \cdots}{\cdots \vDash \cdots}$$

And we can conclude safety thanks to adequacy!

What's left to be done...

- Define formally the language and its syntactic type system.
- Define the logical relation and prove adequacy+fundamental theorem.
- Extend the logical relation for types like `Cell`...

That's a lot of work.

We will only see a sketch here.
The details are in a paper, and the accompanying Coq development:
RustBelt: Securing the foundations of the Rust programming language. In POPL'18.

To make sure nothing is wrong, this is formalized with Coq. Particularly useful to design and maintain the proof.

1 Introduction

2 Syntactic type system

- λ_{Rust}
- Type system

The language: λ_{Rust}

Rust is way too complex to be formalized as-is.

Should we formalize MIR (recall: the language behind the borrow checker)?
It still has a lot of technicalities unrelated to the type system, refers to traits...

Instead, we formalize a core language.
It has most of the features of MIR relevant for type soundness, but idealized to make the theory simpler.

The language: λ_{Rust}

Paths, elementary values

$$
\begin{aligned}
\mathit{Val} \ni v &::= \texttt{false} \mid \texttt{true} \mid z \mid \ell \mid \texttt{funrec}\, f(\bar{x})\ \ \texttt{ret}\, k := F \\
\mathit{Path} \ni p &::= x \mid p.n
\end{aligned}
$$

Elementary values that can be stored in local registers $x, \ldots$

Complex values (e.g., `struct`) are stored in memory, and accessed through pointers.

Note: function values take a continuation in parameter (language in CFG, see later).

The language: λ_{Rust}

Paths, elementary values

$$
\begin{aligned}
\mathit{Val} \ni v &::= \texttt{false} \mid \texttt{true} \mid z \mid \ell \mid \texttt{funrec}\, f(\bar{x})\ \ \texttt{ret}\, k := F \\
\mathit{Path} \ni p &::= x \mid p.n
\end{aligned}
$$

Pointers to fields of complex values can be created with paths.

$p.n$: pointer to field at offset n of `struct` pointed to by p.

Operationally: just a pointer offset (i.e., an addition).

The language: λ_{Rust}

Instructions

$$
\begin{aligned}
\mathit{Instr} \ni I ::= {} & v \mid p \mid p_1 + p_2 \mid p_1 - p_2 \mid p_1 \leq p_2 \mid p_1 == p_2 \\
& \mid \mathtt{new}(n) \mid \mathtt{delete}(n, p) \mid {}^*p \mid p_1 := p_2 \mid p_1 :=_n {}^*p_2 \\
& \mid \ldots
\end{aligned}
$$

Instructions are the elementary operations of λ_{Rust}.

They take paths as operands.

The language: λ_{Rust}

Instructions

$$
\begin{aligned}
\mathit{Instr} \ni I ::= {} & v \mid p \mid p_1 + p_2 \mid p_1 - p_2 \mid p_1 \le p_2 \mid p_1 == p_2 \\
& \mid \texttt{new}(n) \mid \texttt{delete}(n, p) \mid {}^*p \mid p_1 := p_2 \mid p_1 :=_n {}^*p_2 \\
& \mid \ldots
\end{aligned}
$$

λ_{Rust}'s instructions include values, paths, arithmetic operations...

The language: λ_{Rust}

Instructions

$$\begin{aligned} \mathit{Instr} \ni I ::= {} & v \mid p \mid p_1 + p_2 \mid p_1 - p_2 \mid p_1 \leq p_2 \mid p_1 == p_2 \\ & \mid \texttt{new}(n) \mid \texttt{delete}(n, p) \mid {}^*p \mid p_1 := p_2 \mid p_1 :=_n {}^*p_2 \\ & \mid \ldots \end{aligned}$$

... and instruction to allocate/free memory, and read/write it.

Read/write one word from/to a register: ${}^*p$, $p_1 := p_2$.

- Operational semantics defined so that races $\Rightarrow$ undefined behavior.

Copy a complex value (several words) from one memory location to another: $p_1 :=_n {}^*p_2$.

The language: λ_{Rust}

Function bodies

$$
\begin{aligned}
\mathit{FuncBody} \ni F ::= {} & \texttt{letcont}\ k(\bar{x}) := F_1\ \ \texttt{in}\ F_2 \mid \texttt{jump}\ k(\bar{x}) \\
\mid {} & \texttt{let}\ x = I\ \ \texttt{in}\ F \mid \texttt{if}\ p\ \ \texttt{then}\ F_1\ \ \texttt{else}\ F_2 \\
\mid {} & \texttt{call}\ f(\bar{x})\ \ \texttt{ret}\ k \\
\mid {} & \texttt{newlft};\ F \mid \texttt{endlft};\ F \\
\mid {} & \ldots
\end{aligned}
$$

Function bodies combine instructions to build functions.

Code is written in CPS: a way to model code written as a CFG.
We can declare a new continuation (i.e., label in the CFG), and jump to it.

The language: λ_{Rust}

Function bodies

$$
\begin{aligned}
\mathit{FuncBody} \ni F ::= {} & \texttt{letcont}\, k(\bar{x}) := F_1 \ \ \texttt{in}\, F_2 \mid \texttt{jump}\, k(\bar{x}) \\
& \mid \texttt{let}\, x = I \ \ \texttt{in}\, F \mid \texttt{if}\, p \ \ \texttt{then}\, F_1 \ \ \texttt{else}\, F_2 \\
& \mid \texttt{call}\, f(\bar{x}) \ \ \texttt{ret}\, k \\
& \mid \texttt{newlft};\, F \mid \texttt{endlft};\, F \\
& \mid \ldots
\end{aligned}
$$

Basic function bodies:

- An instruction (+ binding the result to a variable).
- If-then-else.
- Calling a function (passing a continuation).

The language: λ_{Rust}

Function bodies

$$
\begin{aligned}
\mathit{FuncBody} \ni F ::= {} & \texttt{letcont}\, k(\bar{x}) := F_1 \;\; \texttt{in}\, F_2 \mid \texttt{jump}\, k(\bar{x}) \\
\mid {} & \texttt{let}\, x = I \;\; \texttt{in}\, F \mid \texttt{if}\, p \;\; \texttt{then}\, F_1 \;\; \texttt{else}\, F_2 \\
\mid {} & \texttt{call}\, f(\bar{x}) \;\; \texttt{ret}\, k \\
\mid {} & \texttt{newlft};\, F \mid \texttt{endlft};\, F \\
\mid {} & \ldots
\end{aligned}
$$

Ghost instructions for creating/ending a lifetime.

The language: λ_{Rust}

Handling sum types

$$
\begin{aligned}
\mathit{Instr} \ni I ::=\ & \dots \\
|\ & p \overset{\text{inj } i}{:==} () \mid p_1 \overset{\text{inj } i}{:==} p_2 \mid p_1 \overset{\text{inj } i}{:==}_n {}^*p_2 \\
|\ & \dots \\
\mathit{FuncBody} \ni F ::=\ & \dots \\
|\ & \texttt{case}\ {}^*p\ \ \texttt{of}\ \bar{F}
\end{aligned}
$$

λ_{Rust} has a notion of sum types, stored in memory, with a tag in the first word.

There are special instructions to write such values together with the tag.

And a case statement for "pattern matching" the in-memory tag.

Types

$$
\begin{aligned}
\mathit{Type} \ni \tau ::= {} & \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own}\,\tau \mid \&^{\kappa}_{\mathbf{mut}}\,\tau \mid \&^{\kappa}_{\mathbf{shr}}\,\tau \mid \lightning_{n} \\
& \mid \Pi\bar{\tau} \mid \Sigma\bar{\tau} \mid \forall\bar{\alpha}.\,\mathbf{fn}(\mathsf{F} : \mathsf{E}; \bar{\tau}) \to \tau \mid T \mid \mu\, T.\,\tau \\
& \mid \ldots
\end{aligned}
$$

Types for integer and Booleans.

Types

$$\begin{aligned} \mathit{Type} \ni \tau ::= {} & \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own}\ \tau \mid \&^{\kappa}_{\mathbf{mut}}\ \tau \mid \&^{\kappa}_{\mathbf{shr}}\ \tau \mid \lightning_{n} \\ & \mid \Pi\bar{\tau} \mid \Sigma\bar{\tau} \mid \forall\bar{\alpha}.\, \mathbf{fn}(\mathsf{F} : \mathsf{E}; \bar{\tau}) \to \tau \mid T \mid \mu\, T.\, \tau \\ & \mid \ldots \end{aligned}$$

Types for pointers: `Box` is written **own** τ.

Types

$$
\begin{aligned}
\mathit{Type} \ni \tau ::= {} & \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own}\,\tau \mid \&^{\kappa}_{\mathbf{mut}}\,\tau \mid \&^{\kappa}_{\mathbf{shr}}\,\tau \mid \lightning\, n \\
& \mid \Pi\bar{\tau} \mid \Sigma\bar{\tau} \mid \forall\bar{\alpha}.\,\mathbf{fn}(\mathsf{F} : \mathsf{E}; \bar{\tau}) \to \tau \mid T \mid \mu\, T.\,\tau \\
& \mid \ldots
\end{aligned}
$$

Type of "uninitialized memory".

- When memory is just uninitialized, or when non-`Copy` values are moved.
- Used to replace the "initializedness" analysis of the borrow checker.

Types

$$\begin{aligned} \textit{Type} \ni \tau ::= {} & \textbf{bool} \mid \textbf{int} \mid \textbf{own}\,\tau \mid \&^{\kappa}_{\textbf{mut}}\,\tau \mid \&^{\kappa}_{\textbf{shr}}\,\tau \mid \lightning_{n} \\ & \mid \Pi\bar{\tau} \mid \Sigma\bar{\tau} \mid \forall\bar{\alpha}.\,\textbf{fn}(\mathsf{F} : \mathsf{E}; \bar{\tau}) \to \tau \mid T \mid \mu\, T.\,\tau \\ & \mid \ldots \end{aligned}$$

Complex types: sum, products.

Used to model `struct`, `enum` and tuples.

Types

$$\begin{aligned} \mathit{Type} \ni \tau ::= {} & \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own}\, \tau \mid \&^{\kappa}_{\mathbf{mut}}\, \tau \mid \&^{\kappa}_{\mathbf{shr}}\, \tau \mid \lightning_{n} \\ & \mid \Pi \bar{\tau} \mid \Sigma \bar{\tau} \mid \forall \bar{\alpha}.\, \mathbf{fn}(\mathsf{F} : \mathsf{E}; \bar{\tau}) \to \tau \mid T \mid \mu\, T.\, \tau \\ & \mid \ldots \end{aligned}$$

Type of function pointers, guarded equirecursive types...

Types

$$
\begin{aligned}
\mathit{Type} \ni \tau ::= {} & \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own}\,\tau \mid \&^{\kappa}_{\mathbf{mut}}\,\tau \mid \&^{\kappa}_{\mathbf{shr}}\,\tau \mid \lightning_{n} \\
& \mid \Pi\bar{\tau} \mid \Sigma\bar{\tau} \mid \forall\bar{\alpha}.\,\mathbf{fn}(\mathsf{F} : \mathsf{E}; \bar{\tau}) \to \tau \mid T \mid \mu\,T.\,\tau \\
& \mid \ldots
\end{aligned}
$$

Extensible for types providing safe abstraction over `unsafe`!

Typing judgments

There are two main typing judgments:

- One for instructions: $\mathrm{E}; \mathrm{L} \mid \mathrm{T}_1 \vdash I \dashv x.\, \mathrm{T}_2$
- One for function bodies: $\mathrm{E}; \mathrm{L} \mid \mathrm{K}; \mathrm{T} \vdash F$

They depend on four different kinds of contexts:

- T is the typing context. Elements:
  - $p \lhd \tau$: path p contains an elementary value of type τ.
  - $p \lhd^{\dagger\kappa} \tau$: same, but frozen until lifetime κ ends.
  - Substructural: elements cannot be duplicated (ownership tracking!).
- E and L are lifetime contexts. They contain:
  - information on lifetime inclusion, and
  - information on local lifetimes: which are they, and when they can be ended.
- K is the continuation context.
  - Describes the continuations we can jump to, and the required contexts.
  - Elements: $k \lhd \mathbf{cont}(\mathrm{L}; \bar{x}.\, \mathrm{T})$.
    "We can jump to k, passing parameters $\bar{x}$, if the contexts contain L and T."

Typing judgments

There are two main typing judgments:

- One for instructions: $\mathrm{E}; \mathrm{L} \mid \mathrm{T}_1 \vdash I \dashv x.\, \mathrm{T}_2$
- One for function bodies: $\mathrm{E}; \mathrm{L} \mid \mathrm{K}; \mathrm{T} \vdash F$

They depend on four different kinds of contexts:

- T is the typing context. Elements:
  - $p \lhd \tau$: path p contains an elementary value of type τ.
  - $p \lhd^{\dagger\kappa} \tau$: same, but frozen until lifetime κ ends.
  - Substructural: elements cannot be duplicated (ownership tracking!).
- E and L are lifetime contexts. They contain:
  - information on lifetime inclusion, and
  - information on local lifetimes: which are they, and when they can be ended.
- K is the continuation context.
  - Describes the continuations we can jump to, and the required contexts.
  - Elements: $k \lhd \mathbf{cont}(\mathrm{L}; \bar{x}.\, \mathrm{T})$.
    "We can jump to k, passing parameters $\bar{x}$, if the contexts contain L and T."

Typing judgments

There are two main typing judgments:

- One for instructions: $\mathrm{E};\mathrm{L} \mid \mathrm{T}_1 \vdash I \dashv x.\,\mathrm{T}_2$
- One for function bodies: $\mathrm{E};\mathrm{L} \mid \mathrm{K};\mathrm{T} \vdash F$

They depend on four different kinds of contexts:

- T is the typing context. Elements:
  - $p \lhd \tau$: path p contains an elementary value of type τ.
  - $p \lhd^{\dagger\kappa} \tau$: same, but frozen until lifetime κ ends.
  - Substructural: elements cannot be duplicated (ownership tracking!).
- E and L are lifetime contexts. They contain:
  - information on lifetime inclusion, and
  - information on local lifetimes: which are they, and when they can be ended.
- K is the continuation context.
  - Describes the continuations we can jump to, and the required contexts.
  - Elements: $k \lhd \mathbf{cont}(\mathrm{L}; \bar{x}.\,\mathrm{T})$.
    "We can jump to k, passing parameters $\bar{x}$, if the contexts contain L and T."

Typing judgments

There are two main typing judgments:

- One for instructions: $\mathrm{E};\mathrm{L} \mid \mathrm{T}_1 \vdash I \dashv x.\,\mathrm{T}_2$
- One for function bodies: $\mathrm{E};\mathrm{L} \mid \mathrm{K};\mathrm{T} \vdash F$

The typing context for instructions has two typing contexts an input typing context T_1 and an output typing context T_2.
The instruction I consumes T_1 and produces T_2.
$x.\mathrm{T}_2$ binds a special variable x: to give a type to the result of the instruction.

Function bodies never return (CPS). They only consume T.

Typing rules

Typing instructions

Selected rules:

$$\mathsf{E}; \mathsf{L} \mid \emptyset \vdash z \dashv x.\, x \lhd \textbf{int} \qquad \mathsf{E}; \mathsf{L} \mid p_1 \lhd \textbf{int}, p_2 \lhd \textbf{int} \vdash p_1 \leq p_2 \dashv x.\, x \lhd \textbf{bool}$$

$$\mathsf{E}; \mathsf{L} \mid \emptyset \vdash \texttt{new}(n) \dashv x.\, x \lhd \textbf{own}\ \lightning_n \qquad \frac{n = \mathrm{size}(\tau)}{\mathsf{E}; \mathsf{L} \mid p \lhd \textbf{own}\ \tau \vdash \texttt{delete}(n, p) \dashv \emptyset}$$

$$\frac{\tau \text{ copy} \qquad \mathrm{size}(\tau) = 1}{\mathsf{E}; \mathsf{L} \mid p \lhd \textbf{own}\ \tau \vdash {}^{*}p \dashv x.\, p \lhd \textbf{own}\ \tau, x \lhd \tau} \qquad \frac{\mathrm{size}(\tau) = 1}{\mathsf{E}; \mathsf{L} \mid p \lhd \textbf{own}\ \tau \vdash {}^{*}p \dashv x.\, p \lhd \textbf{own}\ \lightning_1, x \lhd \tau}$$

$$\frac{\mathsf{E}; \mathsf{L} \vdash \kappa \text{ alive}}{\mathsf{E}; \mathsf{L} \mid p_1 \lhd \&^{\kappa}_{\textbf{mut}}\ \tau, p_2 \lhd \tau \vdash p_1 := p_2 \dashv p_1 \lhd \&^{\kappa}_{\textbf{mut}}\ \tau}$$

Typing rules

Typing function bodies

Selected rules:

$$\frac{\mathsf{E};\mathsf{L} \mid \mathsf{T}_1 \vdash I \dashv x.\,\mathsf{T}_2 \qquad \mathsf{E};\mathsf{L} \mid \mathsf{K};\mathsf{T}_2,\mathsf{T} \vdash F}{\mathsf{E};\mathsf{L} \mid \mathsf{K};\mathsf{T}_1,\mathsf{T} \vdash \texttt{let}\,x = I\;\;\texttt{in}\,F} \qquad \frac{k \lhd \mathbf{cont}(\mathsf{L};\bar{x}.\,\mathsf{T}) \in \mathsf{K}}{\mathsf{E};\mathsf{L} \mid \mathsf{K};\mathsf{T}[\bar{y}/\bar{x}] \vdash \texttt{jump}\,k(\bar{y})}$$

$$\frac{\mathsf{E};\mathsf{L} \vdash \mathsf{T} \overset{\text{ctx}}{\Rrightarrow} \mathsf{T}' \qquad \mathsf{E};\mathsf{L} \mid \mathsf{K};\mathsf{T}' \vdash F}{\mathsf{E};\mathsf{L} \mid \mathsf{K};\mathsf{T} \vdash F}$$

Helper judgment for transforming a typing context: $\mathsf{E};\mathsf{L} \vdash \mathsf{T} \overset{\text{ctx}}{\Rrightarrow} \mathsf{T}'$.

Typing rules

Transforming typing environments

Some transformations can happen on environments before typing a piece of code:

$$\frac{\tau \text{ copy}}{\mathsf{E};\mathsf{L} \vdash p \lhd \tau \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \tau, p \lhd \tau}$$

$$\mathsf{E};\mathsf{L} \vdash p \lhd \&^{\kappa}_{\mu}(\tau_1 \times \tau_2) \overset{\mathsf{ctx}}{\Rrightarrow} p.0 \lhd \&^{\kappa}_{\mu}\,\tau_1,\ \ p.\mathsf{size}(\tau_1) \lhd \&^{\kappa}_{\mu}\,\tau_2$$

$$\frac{\mathsf{E};\mathsf{L} \vdash \kappa \text{ alive}}{\mathsf{E};\mathsf{L} \vdash p \lhd \&^{\kappa}_{\mathbf{mut}}\,\tau \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \&^{\kappa}_{\mathbf{shr}}\,\tau} \qquad \mathsf{E};\mathsf{L} \vdash p \lhd \mathbf{own}_n\,\tau \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \&^{\kappa}_{\mathbf{mut}}\,\tau, p \lhd^{\dagger\kappa} \mathbf{own}_n\,\tau$$

$$\frac{\mathsf{E};\mathsf{L} \vdash \kappa' \sqsubseteq \kappa}{\mathsf{E};\mathsf{L} \vdash p \lhd \&^{\kappa}_{\mathbf{mut}}\,\tau \overset{\mathsf{ctx}}{\Rrightarrow} p \lhd \&^{\kappa'}_{\mathbf{mut}}\,\tau, p \lhd^{\dagger\kappa'} \&^{\kappa}_{\mathbf{mut}}\,\tau}$$