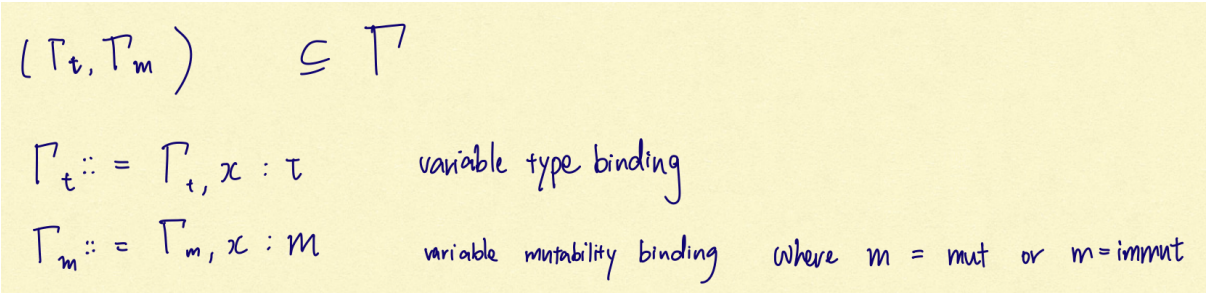
Developer's Guide

Introduction

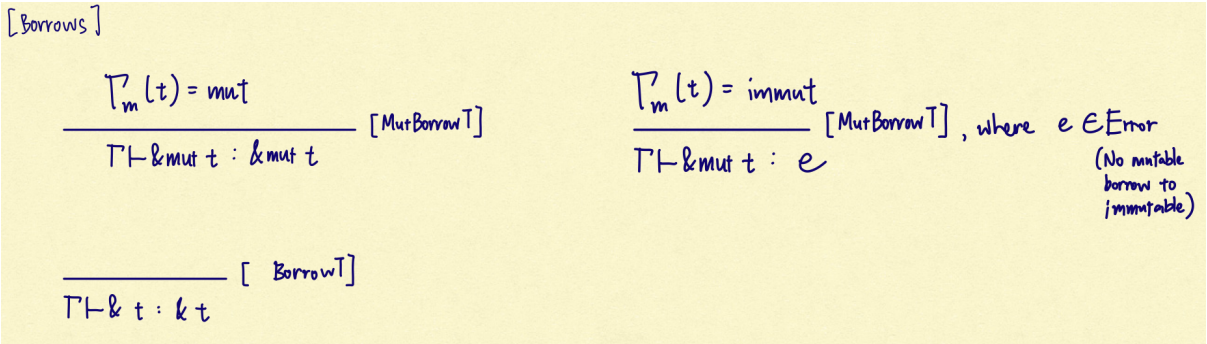
a. Parser
b. Static Checker

The static checker functions to statically check two things : 1) type correctness and 2)ownership-lifetime compatibility. The former is defined under the static semantics common to many programming languages. The only notable point in the static semantics for this Rust subset is that in addition to the usual variable-type binding context, that is, there is a variable mutability context. Essentially, the type checking requires additional information of whether every variable is mutable or immutable, so the environment stores this information.

Type-Correctness



This is so as to support the unique Rust feature of Borrows : either mutably or immutably.



A mutable borrow can only be made to mutable variables (variables, because we restrict the scope of our implementation, explained 3 images under this), while (immutable) borrow can be from anything.

The following shows the static semantics that the type checking ensures (nothing special to Rust).

[Variables]

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ [\text{VarT}]$$

[Literals]

$$\frac{}{\Gamma \vdash n : \text{Int64}}\ [\text{Int64T}] \qquad \frac{}{\Gamma \vdash \text{true} : \text{Bool}}\ [\text{TrueT}] \qquad \frac{}{\Gamma \vdash \text{false} : \text{Bool}}\ [\text{FalseT}]$$

$$\frac{}{\Gamma \vdash () : ()}\ [\text{UnitT}] \qquad \frac{}{\Gamma \vdash s : \text{Str}}\ [\text{StrT}] \qquad \frac{}{\Gamma \vdash s : \text{String}}\ [\text{StringT}]$$

[Assignment]

$$\frac{\Gamma \vdash E : \Gamma(E)}{\Gamma \vdash \text{let } \theta\ x = E : ()}\ [\text{LetT}]$$

E is not a function declaration, borrow/dereference to a non-variable literal

$$\frac{\Gamma_t \vdash x : t_1 \quad \Gamma_t \vdash y : t_2 \quad t_1 = t_2 \quad \Gamma_m(x) = \text{mut}}{\Gamma \vdash x = y; : ()}\ [\text{AssignT-1}]$$

$$\frac{\Gamma_t \vdash x : t_1 \quad \Gamma_t \vdash y : t_2 \quad t_1 = t_2 \quad \Gamma_m(x) = \text{immut}}{\Gamma \vdash x = y; : e}\ [\text{AssignT-2}] \quad \text{where } e \in \text{Error}$$

(x is immutable)

Slightly noteworthy above is the restriction we have made to our scope, such that assignments cannot be made to a borrow or dereference to a non-variable literal. That is, "let a = & 3;" is not supported, while "let a = 3; let b = &a;" is.

[Statements]

$$\frac{\Gamma \vdash E : \Gamma(E) \quad E \text{ is not a function definition}}{\Gamma \vdash \text{let } \theta\ x = E : ()}\ [\text{LetT}]$$

$$\frac{\Gamma \vdash E : t}{\Gamma \vdash E; : ()}\ [\text{ExprStmtT}] \qquad \frac{\Gamma \vdash E; t}{\Gamma \vdash \text{return } E;\ S : \text{return}(t)}\ [\text{ReturnT}]$$

[Functions]

$$\frac{\Gamma(f) = t_1 * \ldots * t_n > t \quad \Gamma[x \leftarrow t_1]\Gamma_1 \cdots \Gamma_{n-1}[x_n \leftarrow t_n]\Gamma_n \quad \Gamma_n \vdash S : \text{return}(t)}{\Gamma \vdash fn\ f\ (x_1 \cdots x_n)\{S\} : ()}\ [\text{FuncDecl1}]$$

$$\frac{\Gamma(f) = t_1 * \ldots * t_n > t \quad \Gamma[x \leftarrow t_1]\Gamma_1 \cdots \Gamma_{n-1}[x_n \leftarrow t_n]\Gamma_n \quad \Gamma_n \vdash S : t \quad t \neq \text{return}(t')}{\Gamma \vdash fn\ f\ (x_1 \cdots x_n)\{S\} : ()}\ [\text{FuncDecl2}]$$

$$\frac{\Gamma \vdash S_1 : t_1 \quad \Gamma \vdash S_2 : t_2}{\Gamma \vdash S_1\ S_2 : t_2}\ [\text{Sequence}] \quad \text{where } S_1 \text{ is not a return statement}$$

$$\frac{\Gamma \vdash E : t_1 * \cdots t_n > t \quad \Gamma \vdash E_1 : t_1 \quad \cdots \Gamma \vdash E_n : t_n}{\Gamma \vdash E(E_1 \cdots E_n) : t}\ [\text{Appl}]$$

Unary Primitive Operations

$$\frac{\Gamma \vdash E : t_1}{\Gamma \vdash p[E] : t}\ [\mathbf{Prim_1T}]$$

where types t_1 and t are given by the following table.

p	t_1	t
!	Bool	Bool
*	&t	t
StringFrom	&Str	String
Len	String	*Int64*
Len	&Str	*Int64*
AsStr	&Str	String

Binary Primitive Operations

$$\frac{\Gamma \vdash E_1 : t_a \in \{t_1, \text{mut } t_1\} \quad \Gamma \vdash E_2 : t_b \in \{t_2, \text{mut } t_2\}}{\Gamma \vdash p[E_1, E_2] : t} \ [\mathbf{Prim_2T}]$$

where types t_1, t_2, and t are given by the following table.

p	t_1	t_2	t
$+$	Int64	Int64	Int64
$-$	Int64	Int64	Int64
$*$	Int64	Int64	Int64
&&	Bool	Bool	Bool
$\|\|$	Bool	Bool	Bool
$==$	Bool	Bool	Bool
$<$	Bool	Bool	Bool
$\leq$	Bool	Bool	Bool
$>$	Bool	Bool	Bool
$\geq$	Bool	Bool	Bool
p	t_1	t_2	t
PushStr	String	&Str	String

Ownership-lifetime compatibility (Borrow Checking)

The speciality of Rust comes in its static checks for ownership-lifetime compatibility. This idea of ownership-lifetime compatibility, as I understand it, translates to the idea that, every pointer to some content, and every other pointer derived from that pointer, can only be used, up till the content is taken back or destroyed.

This means we will not unexpectedly have another variable editing the content at the location. A variable cannot temporarily own, or borrow, (have edit rights to) some location, without some original owner permitting, and a temporary owner (borrower) cannot have others change its content once it takes the content back.

```
fn main(){
    let a = ....;
    let b = .....;
    whatever(b);
    a; // remains the same, expected a.
}
```

This is partly digression but as one useful implication of such a principle, take the above as an example. So long as we know a and b do not ever refer to the same location, a will remain the same and expectable, regardless of whatever happens in whatever(). As I read, such expectations can allow for greater optimizations in compilers and safety for the user in general. Such an idea is easily supported under the stronger principle mentioned earlier.

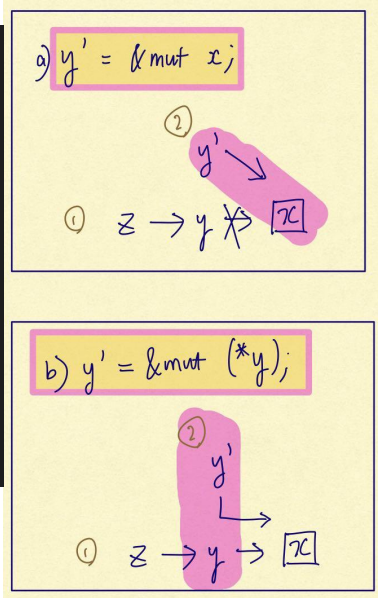
Back to the ownership-lifetime compatibility principle, it translates to the following rules

1. A reference and references derived from it, can only be used in its (whatever referred to's) lifetime.
2. A content will not be changed, until the reference gives the content up (i.e. a loan is given up and ownership returns).

```rust
fn main() {
    let mut x = 0;
    let mut y = &mut x;
    let mut z = &mut y;

    // a)
    let mut y_alt = &mut x;
    println!("{}", y); // ERROR! First borrow used after second!

    // b)
    let mut y_alt = &mut (*y);
    println!("{}", y); // Accepted!
```



This notion of one coming after the other, and the owner taking things back, is effectively showing the one-directional flow of control, out of some prior owner. Hence, this appears to give reason for the use of a stack to maintain this ownership information.

Implementation

Ownership-lifetime compatibility checks are done through the creation of stacks at every scope (sub-environment). At the end of every scope, the created stacks are all cleared, unless copied or moved to outside of the scope. As for each stack, they represent every location in memory, to support the earlier explained rules.

```
type ExpiredLifetimes = HashMap<usize, Vec<String>>;
type FunctionStore = (Vec<DataType>, DataType);
type DecAndBorrowStack = (Vec<&'static str>, LinkedList<&'static str>);
struct Environment {
    declared_functions_table : HashMap<&'static str, FunctionStore>,
    scope_info: LinkedList<DecAndBorrowStack>,
    variables_table : HashMap<&'static str, VariableProperties>,
    expired_lifetimes : ExpiredLifetimes,
}

struct VariableProperties{
    own_type : DataType,
    mutability : bool,
    is_copy_trait_mem : bool,
    // not accounting for immutable borrows for now
}
```

Hence of the environment that supports the whole static checking, it is Environment.scope_info that specifically contains information unique to supporting the ownership-lifetime compatibility checks.

1. Creation of scope
   For every scope, i.e. sequence, the environment is extended. A new DecAndBorrowStack is added to contain the information for that scope.
2. New memory location === Creation of stack
   For every Let expression a variable is set as either a borrower or a first-degree owner to some memory location.
   A first-degree owner to a location must exist at the bottom of the stack as the root of all borrows to the location, whereas any borrower must come after, i.e. push to the stack. Every use of a variable leads to the claiming back of ownership from the borrowers, hence the stack is popped above whatever variable is used.
3. Borrowing from outer scope
   Every variable declared within the scope is naturally allowed to borrow from whatever came from the earlier, outer scope. Hence, the abovementioned stack, whose purpose is to track the ownership to the memory location initialised in that scope is not sufficient. All variables declared in the scope, including those pointer to older memory locations, must be taken track of. The vector of variable names, ensures this.
4. Deletion of stack
   At the end of every scope, every declared (and by our restriction, will also have been instantiated) variable must be cleared.
   Every stack is deleted, and every declaration to the outside scope is also popped of the respective stacks of the outer scope.
5. Returning out of stack
   Either at the end or the middle of a sequence, the return statement causes the sequence to terminate, essentially ending the scope (Under our implementation, there is no support for conditionals, so the line-by-line processing until a return

statement is met is sufficient). In such cases, before the deletion of stack at 4 occurs,
1. The return variable is checked to be of first-degree, i.e. at the bottom of the stack.
2. The first-degree variable is given a special name that is unique, and added to the outer-scope. A hidden next line is added such that, if the new stack is of size one, only containing the special-named variable, the new stack, along with the variable name in the list of variable names of the scope, is deleted. Essentially, the returned variable is forced to have a lifetime of one line; if not moved or copied, it is deleted without assignment to anything.

Output

With the point of the static checker being to
a) check typed correctness, under the above hand-written rules, and
b) check ownership-lifetime compatibility,
It's key purpose is to provide the error message when the check fails.
It also provides the compiler with information about when to drop a variable, i.e. when to clear a memory location, which is subsidiary information given the clearing of variables at every scope, and in the line after the scope, if not re-assigned, as explained in 5.

The static checker returns

```
type ExpiredLifetimes = HashMap<usize, Vec<String>>;
```

The line after which the variable and the memory it points to must be cleared.

c. Compiler
d. VM