Supplementary Material on "Translating C to Safer Rust"

MEHMET EMRE, University of California Santa Barbara, USA
RYAN SCHROEDER, University of California Santa Barbara, USA
KYLE DEWEY, California State University Northridge, USA
BEN HARDEKOPF, University of California Santa Barbara, USA

This document gives more detail into the translation method in our paper "Translating C to Safer Rust" to appear in PACMPL, OOPSLA issue on October 2021. We give an introduction to Rust's ownership system, followed by a precise list of the rewrite rules used by the lifetime resolution algorithm. Finally, we give two code snippets to (1) complete the example used in the paper, and (2) to show an example of borrow conflicts mentioned in the paper.

Additional Key Words and Phrases: Rust, C, Automatic Translation, Memory-Safety, Empirical Study

This document supplements "Translating C to Safer Rust" [Emre et al. 2021]. The following sections contain (1) a description of Rust's ownership system, (2) detailed rewrite rules used in ResolveLifetimes (see Section 3 of [Emre et al. 2021]), (3) a snippet from the `bzip2` corpus program showing a borrow conflict that is resolved by promoting the related references to raw pointers, and (4) the full version of the running example from the paper after all steps of our method are applied.

1 RUST'S OWNERSHIP SYSTEM

This section serves a short primer to how Rust handles *ownership* and *borrowing*. Both of these features are central to Rust's memory model, and enable it to statically ensure memory safety in safe code without resorting to garbage collection at runtime. Given that our work must work with Rust's memory model closely, it is necessary to have some understanding of Rust's memory model in order to understand the significance of our own work. That said, this section is intended only as a quick introduction; readers curious for more details are directed to the online Rust book for basics [Klabnik and Nichols 2018], as well as as a more formal alias-based formulation at [Matsakis 2018].

1.1 Motivation

Rust's memory model ensures memory safety statically, without resorting to potentially expensive runtime memory management techniques like garbage collection. In Rust, well-typed programs are memory-safe by construction. As with a garbage collected language, users explicitly perform memory allocation, but do not explicitly perform deallocation. Unlike with garbage collection, the Rust compiler statically inserts routines to deallocate heap-allocated memory when it is no longer needed. The type system of Rust is designed in such a manner that the compiler statically

Authors' addresses: Mehmet Emre, emre@cs.ucsb.edu, University of California Santa Barbara, Santa Barbara, CA, USA; Ryan Schroeder, rschroeder@ucsb.edu, University of California Santa Barbara, Santa Barbara, CA, USA; Kyle Dewey, kyle.dewey@csun.edu, California State University Northridge, Northridge, CA, USA; Ben Hardekopf, benh@cs.ucsb.edu, University of California Santa Barbara, Santa Barbara, CA, USA.

XXXX-XXXX/2021/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

knows exactly where these memory deallocations need to be performed. This knowledge of when to perform deallocation is based around *ownership*.

1.2 Ownership

By default, data is said to be *owned* in Rust. For example, consider the following function definition `f`, which uses type `Vec` from the Rust standard library (representing a vector):

```
1 fn f(v: Vec<i32>) {}
```

`f` is said to take ownership of `v`. This is indicated by the fact that `v` is directly of type `Vec<i32>`. Whoever owns the data is ultimately responsible for deallocating any heap-allocated data held. Deallocation implicitly occurs whenever the variable bound to the data falls out of scope. With this in mind, any heap-allocated data held in `v` is deallocated immediately after the call to `f`, as `v` will no longer be accessible.

Within a scope, ownership can be transferred from one variable to another. For example, consider the following code snippet:

```
1 fn example() {
2   let v1 = vec![1, 2, 3]; // creates a vector holding 1, 2, 3
3   let v2 = v1;
4 }
```

In this case, `v1` initially holds the underlying vector. Ownership is then transferred to variable `v2`. Because ownership is never transferred away from `v2`, `v2` will have all heap-allocated memory deallocated at `example`'s termination. Because ownership was transferred away from `v1`, there is no similar deallocation performed for `v1`, beyond typical stack deallocation of `v1`.

Ownership can also be transferred between scopes. For example, consider the following:

```
1 fn identity(v: Vec<i32>) -> Vec<i32> { return v; }
```

In this case, like the prior `f` example, `identity` takes ownership over `v`. However, because `identity` later returns `v`, it transfers ownership to `identity`'s caller. Any heap-allocated memory bound to `v` then becomes the concern of `identity`'s caller.

1.3 Borrowing and Lifetimes

While the ownership model unambiguously allows the compiler to safely statically deallocate all heap-allocated memory, it is nonetheless very restrictive. For example, if you wanted to define a function that merely printed the contents of a vector, it would need to transfer ownership back to the caller. This would mean having an unintuitive type signature like:

```
1 fn print_all(v: Vec<i32>) -> Vec<i32> { ... }
```

With this in mind, the more data a function needs to do its job, the more data the very same function needs to return. There are also negative performance implications of ownership transfer, since barring compiler optimizations, it entails copying any stack-allocated memory behind a variable.

To address these issues around ownership transfer, Rust also has a concept known as *borrowing*. As the name suggests, data can be temporarily borrowed without changing ownership. Data is borrowed through a reference, which bear similarity to references in other languages. Borrowed data can be used like owned data, with some restrictions. One important restriction is that borrowed data cannot outlive the actual data being borrowed. Using C/C++ terminology, Rust must ensure that there are no dangling pointers to any allocated data.

To ensure that the underlying data being borrowed is always valid, Rust introduces the concept of a *lifetime*. Lifetimes are type-level variables which abstractly define how long the underlying data being borrowed will be in memory. For example, consider the following code:

```
fn has_lifetime<'a>(v: &'a Vec<i32>) { ... }
```

Instead of having ownership of v transferred to has_lifetime, this instead borrows the underlying Vec<i32> for lifetime 'a. Rust will ensure that the underlying Vec<i32> is in memory for the duration of the call to has_lifetime. Because has_lifetime merely borrows the Vec<i32>, there is no memory deallocation of v performed; has_lifetime does not own the vector, and so it is not has_lifetime's responsibility to deallocate the vector.

Like regular type variables, data structure definitions themselves can take lifetimes, as with:

```
struct SomeData<'a, 'b> {
  first: &'a i32,
  second: &'b i32
}
```

With the above code in mind, Rust will make sure that no allocated instance of SomeData will outlive anything it borrows. That is, the data referred to by first and second will always be in memory at least as long as the SomeData data structure itself.

To show this in practice, consider the following example, which is rejected by the Rust compiler:

```
fn rejected() {
  let the_data;
  let first_int = 1;
  {
    let second_int = 2;
    the_data = SomeData { first: &first_int, second: &second_int };
  }
  print!("{}'', *the_data.second);
}
```

The above code is rejected by the Rust compiler, with an error message stating that second_int does not live long enough. To understand why, first understand that each block in Rust corresponds to a separate lifetime variable. That is, an enclosing scope maps directly to object lifetimes. For speaking purposes, the outer scope of rejected will be called 'a, and the inner scope (where second_int is declared) will be called 'b. With this in mind, the_data has type SomeData<'a, 'b>, and it itself has lifetime 'a. However, 'b does not live as long as 'a. As such, we have attempted to create a data structure with a lifetime longer than its constituents, which is not permitted. As such, Rust rejects the program. Thinking in terms of C/C++, this rejection makes sense - second_int is allocated on the stack and subsequently deallocated after the_data is initialized, so the_data.second would be a dangling pointer.

1.3.1 Restrictions. All borrows seen so far are immutable borrows, meaning that the underlying object cannot be changed through these borrows. Furthermore, the underlying object may not be changed at all while any immutable borrows are active. Similarly, Rust disallows ownership transfers while any borrows are active. This can be statically checked at compile time, as shown in the code below:

```
struct MyStruct {
    first: i32
}

fn involves_borrows<'a>(datum: &'a MyStruct) -> &'a MyStruct {
  return datum;
}
fn performs_transfer(x: MyStruct) {}
```

```
 9
10 fn main() {
11     let x = MyStruct { first: 42 };
12     let r = involves_borrows(&x);
13     performs_transfer(x);
14     print!("{}", r.first)
15 }
```

The above code fails to compile, as the the transfer performed by `performs_transfer` is disallowed because reference `r` still refers to the same data structure. Specifically, Rust tracks that `x` has an active borrow at the call to `performs_transfer`, disallowing the call. As an aside, the subsequent use of `r.first` is required to get this code to compile, as this forces the compiler to internally keep the borrow of `x` around after the call to `performs_transfer`; effectively, Rust will permit the existence of a dangling pointer, but not the access of a dangling pointer.

1.3.2 Immutable and Mutable Borrows. All prior borrow examples are based on immutable borrows, meaning the underlying object cannot be changed through the borrow. Rust also supports mutable borrows, which use the `mut` reserved word, like so:

```
&'a mut Vec<i32>
```

The above snippet refers to a mutable borrow of a `Vec<i32>`, where the underlying vector is in memory for at least `'a` lifetime.

Mutable borrows work similarly to mutable borrows, with the following twists. With immutable borrows, the same data may be borrowed multiple times in the same context, as none of the borrows can change the underlying object. However, with mutable borrows, only one such mutable borrow may be active at any time. Furthermore, if a mutable borrow is active, all mutation must be done through the mutable borrow, and no immutable borrows or ownership transfers are permitted. While restrictive, these requirements prevent data races from occurring - all mutation is very carefully tracked and made explicit in the types; it is not possible for data to be modified "out from under you", as it is in most languages.

2 THE REWRITE RULES

This section gives a simplified fragment of Rust HIR that we perform rewrite operations on (Figure 1), and the specific rewrite rules we use to rewrite the initial program according to a given configuration and taint analysis results. We consider a core language (Figure 1), and we rewrite other constructs like unary/binary operations and method calls to function calls. We similarly rewrite fused assignment operators (e.g., `+=`) to equivalent unfused code. The notation $\overrightarrow{a}$ denotes a sequence of as. For example, a function call contains a sequence of expressions representing arguments. e_{guard} denotes the guard expression in pattern matches. Similar to Oxide, we maintain a context denoting whether an expression is used in a place where it is borrowed mutably or immutably, or owned; these contexts are defined in Figure 2. The context assignee denotes that the expression is on the left-hand side of an assignment-like expression or being borrowed mutably. move denotes that the expression's value should be moved, as it is used in a context that should own its value. The notation $c[d \mapsto e]$ is used for conditionally updating the context: if $c = d$ then $c[d \mapsto e]$ produces e, otherwise it produces c. AdjustedType is a function provided by the Rust compiler that gives the type of the expression in the context it is used (after coercions, and converting `&mut` to `&` if necessary).

RewriteProgram adds lifetimes to each struct according to the method described in Section 3.3 of the original paper [Emre et al. 2021], and assigns unique lifetimes to each lifetime variable needed

in a function signature. It adds the lifetime constraints to each function signature directly from the configuration. The function bodies are rewritten using the rewrite rules given in Figures 3 to 5. The helper PKind returns the kind (owned, borrowed, raw) of an expression as computed by the taint analyses from the current configuration. Other helper functions are defined in Figure 6. The rewrite rule $c \vdash e_1 \longrightarrow e_2$ denotes that e_1 is rewritten into e_2 under mutability context c. Similarly, $c \vdash s_1 \longrightarrow s_2$ denotes that the statement s_1 is rewritten into the statement s_2 under the context c as described in Figure 5. After rewriting an expression according to the rules given in Figures 3 and 4, we check the context of the expression and the pointer kind to decide whether to re-borrow the expression according to Section 3.3 of the original paper. Let c be the current context, $p = \text{PKind}(e)$, and $c \vdash e \longrightarrow e'$. Then, if e has a pointer type, we choose whether to re-borrow e' according to the following conditionally-applied rules (attempted from first to last):

- p = owned and $c \neq$ move. We borrow the box by rewriting e' to e'`.as_ref().map(|x| x.as_ref())` (we use `as_mut` instead of `as_ref` if the current context is mut).
- p = owned and c = move. We do not re-borrow the expression's value, as it should be moved.
- p = raw. We do not perform any re-borrowing, as p is a raw pointer.
- p = borrowed and c = mut. We rewrite e' into `borrow_mut(&mut` e'`)`.
- p = borrowed, c = not, and e has a mutable pointer type. We rewrite e' into `borrow(&` e'`)`.
- p = borrowed, c = not, and e has an immutable pointer type. We rewrite e' into e'`.clone()`.
- Otherwise, we do not re-borrow the value of the expression.

3 SIMPLIFIED VERSION OF THE BORROW CHECKER VIOLATION IN BZIP2

In the snippet below, the object pointed to by `strm` is borrowed into `(*s).strm`, and it is later mutated while `s` and the first borrow is still alive. This is a simplified version of `BZ2_bzCompressInit` function from `bzip2` where all the code unrelated to the borrow conflict is removed.

```
1  unsafe fn BZ2_bzCompressInit<'a1>(mut strm: Option<&mut bz_stream>, /* removed */) {
2    let mut s: *mut EState = core::ptr::null();
3    // ...
4    (*s).strm = borrow_mut(&mut strm); // first mutable borrow occurs here
5    // ... other borrow conflicts also occur here, but we isolate only one below
6    (*borrow_mut(&mut strm).unwrap()).state = s; // Here, the object pointed to by strm is mutated by
       assigning to its state field.
7  }
```

Because of the conflict above, we promote the borrowed expression (`strm`) to be a raw pointer. In this specific case, a higher level rewriting approach might have reorganized the statements to resolve the borrow conflict.

4 THE FULL PROGRAM AFTER APPLYING ALL STEPS OF OUR TECHNIQUE

Figure 7 shows the full BST program after applying all steps of our technique.

REFERENCES

Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating C to Safer Rust. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 121 (Oct. 2021), 29 pages. https://doi.org/10.1145/3485498

S. Klabnik and C. Nichols. 2018. *The Rust Programming Language.* No Starch Press. https://doc.rust-lang.org/book/

Nicholas D Matsakis. 2018. An alias-based formulation of the borrow checker. https://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/

$e \in \text{Expression} ::=$	$[\overrightarrow{e}]$	array literals
$e \in \text{Expression} ::=$	$T\{\overrightarrow{fld : e}\} \mid$	struct construction
	$f(\overrightarrow{e}) \mid$	function call
	$(\overrightarrow{e}) \mid$	tuples
	$\&\mu\, e \mid$	address-of operator
	$*\, e \mid$	dereference
	$l \in \text{Literal} \mid$	
	e **as** $\tau \mid$	
	$x \in Variable \mid$	
	loop $e \mid$	
	match $e_{scrutinee}\ \overrightarrow{pe_{guard} \Rightarrow e} \mid$	pattern matching
	$e_1 = e_2 \mid$	assignment
	$e.fld \mid$	field access
	$e_1[e_2] \mid$	array access
	$\overrightarrow{s}\, e \mid$	blocks
	break $\mid$ **continue** $\mid$ **return** e	early return, control flow redirection
$s \in \text{Statement} ::=$	**let** $\mu x = e_1; e_2 \mid e;$	
$\mu \in \text{Mutability} ::=$	**mut** $\mid$ **not**	
$\tau \in \text{Type} ::=$	$*\, \mu\tau \mid$	raw pointers
	$\&\mu\tau \mid$	borrowing references
	$Option < Box < \tau >> \mid$	owned references
	$T \in \text{StructName} \mid$	structs
	$\ldots$	other types
$p \in \text{Pattern}$		

Fig. 1. Abstract syntax for the fragment of Rust HIR that is relevant to our rewrite rules for expressions. Because we rely on the compiler for lifetime inference, the lifetimes inside types are elided.

$$c \in \text{UseCtx} ::= \text{mut} \mid \text{not} \mid \text{move} \mid \text{assignee}$$

Fig. 2. The contexts for determining how a variable is used.

$$\frac{}{c \vdash x \to x}\ \textsc{Var} \qquad \frac{}{c \vdash l \to l}\ \textsc{Lit} \qquad \frac{}{c \vdash \textbf{break} \to \textbf{break}}\ \textsc{Break}$$

$$\frac{}{c \vdash \textbf{continue} \to \textbf{continue}}\ \textsc{Cont} \qquad \frac{c \vdash e_i \to e_i'}{c \vdash [\overrightarrow{e_i}] \to [\overrightarrow{e_i'}]}\ \textsc{Array}$$

$$\frac{c \vdash e_i \to e_i'}{c \vdash T\{\overrightarrow{fld_i : e_i}\} \to T\{\overrightarrow{fld_i : e_i'}\}}\ \textsc{Struct} \qquad \frac{c \vdash e_i \to e_i'}{c \vdash (\overrightarrow{e_i}) \to (\overrightarrow{e_i'})}\ \textsc{Tuple}$$

$$\frac{\mathsf{assignee} \vdash e \to e'}{c \vdash \textbf{return}\ e \to \textbf{return}\ e'}\ \textsc{Return} \qquad \frac{\mathsf{not} \vdash e \to e'}{c \vdash \textbf{loop}\ e \to \textbf{loop}\ e'}\ \textsc{Loop}$$

$$\frac{\textsc{Ctx}(e_{scrutinee}) \vdash e_{scrutinee} \to e'_{scrutinee} \quad \mathsf{not} \vdash e_{guard} \to e'_{guard} \quad c \vdash e \to e'}{c \vdash \textbf{match}\ e_{scrutinee}\ \overrightarrow{pe_{guard} \Rightarrow e} \to \textbf{match}\ e'_{scrutinee}\ \overrightarrow{pe'_{guard} \Rightarrow e'}}\ \textsc{Match}$$

$$\frac{(f \neq \texttt{malloc} \vee \textsc{PKind}(f(\overrightarrow{e_i})) = \mathsf{raw}) \quad \textbf{move} \vdash e_i \to e_i' \quad \textsc{PKind}(\textbf{param}fi) = \mathsf{owned}}{c \vdash f(\overrightarrow{e_i}) \to f(\overrightarrow{e_i'})}\ \textsc{Call-Mv}$$

$$\frac{(f \neq \texttt{malloc} \vee \textsc{PKind}(f(\overrightarrow{e_i})) = \mathsf{raw}) \quad \textsc{Ctx}(e_i) \vdash e_i \to e_i' \quad \textsc{PKind}(\textbf{param}fi) \neq \mathsf{owned}}{c \vdash f(\overrightarrow{e_i}) \to f(\overrightarrow{e_i'})}\ \textsc{Call-Br}$$

$$\frac{\textsc{PKind}(\texttt{malloc}(l)) \neq \mathsf{raw}}{c \vdash \texttt{malloc}(l)\ \textbf{as}\ *\mu T \to \texttt{Box::new}(T\texttt{::default()})}\ \textsc{Malloc}$$

$$\frac{c' = c[\mathsf{assignee} \mapsto \mathsf{mut}] \quad \textsc{PKind}(e) = \mathsf{raw} \quad c' \vdash e \to e'}{c \vdash *e \to *e'}\ \textsc{Deref-Raw}$$

$$\frac{c' = c[\mathsf{assignee} \mapsto \mathsf{mut}] \quad \textsc{PKind}(e) \neq \mathsf{raw} \quad c' \vdash e \to e'}{c \vdash *e \to *(e'\texttt{.unwrap()})}\ \textsc{Deref-Safe}$$

$$\frac{\textsc{PKind}(e) = \mathsf{raw} \quad c \vdash e \to e' \quad \tau \mapsto_{\{e\}} \tau'}{c \vdash e\ \textbf{as}\ \tau \to e'\ \textbf{as}\ \tau'}\ \textsc{Cast-Raw} \qquad \frac{\textsc{PKind}(e) \neq \mathsf{raw} \quad c \vdash e \to e'}{c \vdash e\ \textbf{as}\ *\mu\tau \to e'}\ \textsc{Cast-Rm}$$

$$\frac{\textsc{PKind}(e) = \mathsf{raw} \quad \mathsf{not} \vdash e \to e'}{c \vdash \&\textbf{not}\ e \to \&\textbf{not}\ e'}\ \textit{\&}\textsc{-Raw} \qquad \frac{\textsc{PKind}(e) = \mathsf{borrowed} \quad \textbf{not} \vdash e \to e'}{c \vdash \&e \to \mathit{Some}(\&\textbf{not}\ e')}\ \textit{\&}\textsc{-Safe}$$

$$\frac{\textsc{PKind}(e) = \mathsf{raw} \quad \textbf{mut} \vdash e \to e'}{c \vdash \&\textbf{mut}\ e \to \textbf{mut}\mu e'}\ \textit{\&}\textsc{Mut-Raw}$$

$$\frac{\textsc{PKind}(e) = \mathsf{borrowed} \quad \textbf{assignee} \vdash e \to e'}{c \vdash \&\textbf{mut}e \to \mathit{Some}(\&\textbf{mut}\ e')}\ \textit{\&}\textsc{Mut-Safe}$$

Fig. 3. Rules for rewriting expressions, part I.

$$\frac{c \vdash e \rightarrow e'}{c \vdash e.fld \rightarrow e'.fld}\ \textsc{Field} \qquad \frac{c \vdash e_1 \rightarrow e_1' \quad \text{not} \vdash e_2 \rightarrow e_2'}{c \vdash e_1[e_2] \rightarrow e_1'[e_2']}\ \textsc{Index}$$

$$\frac{c \vdash e \rightarrow e' \quad \text{not} \vdash s \rightarrow s'}{c \vdash \overrightarrow{s}\, e \rightarrow \overrightarrow{s'}\, e'}\ \textsc{Block}$$

$$\frac{\text{assignee} \vdash e_1 \rightarrow e_1' \quad \textsc{PKind}(x) = \text{owned} \quad \textbf{move} \vdash e_2 \rightarrow e_2'}{c \vdash e_1 = e_2 \rightarrow e_1' = e_2'}\ \textsc{Assign-Move}$$

$$\frac{\begin{array}{c}\text{assignee} \vdash e_1 \rightarrow e_1' \quad \textsc{PKind}(x) \neq \text{owned} \quad \textsc{AdjustedType}(e_1) \neq *mut\ \tau \\ \textbf{not} \vdash e_2 \rightarrow e_2'\end{array}}{c \vdash e_1 = e_2 \rightarrow e_1' = e_2'}\ \textsc{Assign-Not}$$

$$\frac{\begin{array}{c}\text{assignee} \vdash e_1 \rightarrow e_1' \quad \textsc{PKind}(x) \neq \text{owned} \quad \textsc{AdjustedType}(e_1) = *mut\ \tau \\ \textbf{mut} \vdash e_2 \rightarrow e_2'\end{array}}{c \vdash e_1 = e_2 \rightarrow e_1' = e_2'}\ \textsc{Assign-Mut}$$

Fig. 4. Rules for rewriting expressions, part II.

$$\frac{c \vdash e_2 \rightarrow e_2' \quad \textsc{PKind}(x) = \text{owned} \quad \textbf{move} \vdash e_1 \rightarrow e_1'}{c \vdash \textbf{let}\ \mu x = e_1; e_2 \rightarrow \textbf{let}\ x = e_1'; e_2'}\ \textsc{S-Let-Move}$$

$$\frac{c \vdash e_2 \rightarrow e_2' \quad \textsc{PKind}(x) \neq \text{owned} \quad \textbf{mut} \vdash e_1 \rightarrow e_1'}{c \vdash \textbf{let mut}x = e_1; e_2 \rightarrow \textbf{let}\ x = e_1'; e_2'}\ \textsc{S-Let-Mut}$$

$$\frac{c \vdash e_2 \rightarrow e_2' \quad \textsc{PKind}(x) \neq \text{owned} \quad \textbf{not} \vdash e_1 \rightarrow e_1'}{c \vdash \textbf{let not}x = e_1; e_2 \rightarrow \textbf{let}\ x = e_1'; e_2'}\ \textsc{S-Let-Not}$$

$$\frac{\text{not} \vdash e \rightarrow e'}{c \vdash e; \rightarrow e';}\ \textsc{S-Semicolon}$$

Fig. 5. Rules for rewriting statements.

$$\textsc{Ctx}(e) = \textsc{Mutability}(\textsc{AdjustedType}(e))$$

$$\textsc{Mutability}(\tau) = \begin{cases} \text{mut} & \tau = *\mathbf{mut}\tau' \\ \text{not} & \text{otherwise} \end{cases}$$

$$\frac{\tau \neq *\mu\tau''}{\tau \mapsto_{loc} \tau'} \text{ T-NonPtr}$$

$$\frac{\tau \mapsto_{\textsc{PtsTo}(loc)} \tau' \quad \textsc{PKind}(loc) = raw}{*\mu\tau \mapsto_{loc} *\mu\tau'} \text{ T-RawPtr}$$

$$\frac{\tau \mapsto_{\textsc{PtsTo}(loc)} \tau' \quad \textsc{PKind}(loc) = owned}{*\mu\tau \mapsto_{loc} \texttt{Option<Box<}\tau'\texttt{>>}} \text{ T-OwnedPtr}$$

$$\frac{\tau \mapsto_{\textsc{PtsTo}(loc)} \tau' \quad \textsc{PKind}(loc) = borrowed}{*\mu\tau \mapsto_{loc} \texttt{Option<\&}\mu\tau'\texttt{>}} \text{ T-RawPtr}$$

Fig. 6. Helper functions for rewriting expressions and nested types. $\tau \mapsto_{loc} \tau'$ rewrites a type that is associated with the set of locations *loc*. PtsTo returns the points-to set of given set of locations.

```
// bst.rs
use std::os::raw::c_int;
// BST node
pub struct node_t<'a1, 'a2> {
    pub left: Option<&'a1 mut node_t<'a1, 'a2>>,
    pub right: Option<&'a1 mut node_t<'a1, 'a2>>,
    pub value: Option<&'a2 mut c_int>,
}
impl<'a1, 'a2> std::default::Default for node_t<'a1, 'a2> {
  // ...
}
pub fn insert<'a1, 'a2, 'a3>(mut value: c_int,
                             mut node: Option<&'a1 mut node_t<'a2, 'a3>>) {
    // ...
}
pub fn find<'a1, 'a2, 'a3, 'a4, 'a5, 'a6>(mut value: c_int, mut node: Option<&'a1 mut node_t<'a2, 'a3>>)
 -> Option<&'a4 mut node_t<'a5, 'a6>>
where 'a1: 'a4, 'a5: 'a2, 'a6: 'a3, 'a3: 'a6, 'a2: 'a5
{
    if value < **(**node.as_ref().unwrap()).value.as_ref().unwrap() && !(**node.as_ref().unwrap()).left.
      is_none() {
        return find(value, borrow_mut(&mut (*node.unwrap()).left))
    } else {
        if value > **(**node.as_ref().unwrap()).value.as_ref().unwrap() && !(**node.as_ref().unwrap()).
      right.is_none() {
            return find(value, borrow_mut(&mut (*node.unwrap()).right))
        } else { if value == **(**node.as_mut().unwrap()).value.as_mut().unwrap() { return node } }
    }
    return None;
}

// main.rs
use std::os::raw::c_int;
use bst::{node_t, insert, find};

pub fn main_0() -> int {
    // Using Box to avoid malloc clutter
    let mut tree = Some(Box::new(crate::node_t::default()));
    **(**tree.as_mut().unwrap()).value.as_mut().unwrap() = 3;
    // insert 2 nodes
    insert(1, tree.as_mut().map(|b| b.as_mut()));
    insert(2, tree.as_mut().map(|b| b.as_mut()));
    // change the value of node containing 3 to 4
    **(**find(3, tree.as_mut().map(|b| b.as_mut())).as_mut().unwrap()).value.as_mut().unwrap() = 4;
    return 0;
}
```

Fig. 7. The safe Rust program with no raw pointers after applying all steps of our technique.