Leveraging Rust Types for Program Synthesis: Appendix

JONÁŠ FIALA, Department of Computer Science, ETH Zurich, Switzerland
SHACHAR ITZHAKY, Technion, Israel
PETER MÜLLER, Department of Computer Science, ETH Zurich, Switzerland
NADIA POLIKARPOVA, University of California, San Diego, USA
ILYA SERGEY, National University of Singapore, Singapore

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: program synthesis, program logic, Rust, type systems

ACM Reference Format:
Jonáš Fiala, Shachar Itzhaky, Peter Müller, Nadia Polikarpova, and Ilya Sergey. 2023. Leveraging Rust Types for Program Synthesis: Appendix. *Proc. ACM Program. Lang.* 7, PLDI, Article 164 (June 2023), 6 pages. https://doi.org/10.1145/3591278

A SYNTHESIS RULES

This appendix contains the full set of synthesis rules: Fig. 1 lists ownership rules, Fig. 2 lists borrowing rules, and Fig. 3 lists structural rules.

Ownership Rules. The rules Drop, Rename, and Unreachable are straightforward: once a variable is dropped or renamed (moved), it is no longer accessible due to ownership-types semantics. The Unreachable rule may be encountered if a specification is inconsistent or, more importantly, in combination with Destr when synthesizing a `match` expression one of whose branches is unsatisfiable due to a contradiction with known assumptions (*e.g.* the `Nil` case when the matched list is known to be nonempty). This rule can only ever fire if a precondition is specified; otherwise, with no assumptions, it is impossible to deduce `false`.

The rules Primitive, Constr and Destr give the semantics of value construction and destructuring (pattern matching) for Rust data types, for the case where the given variable has an owned type. The rules take care to update the snapshots accordingly; in Destr the input binding is unconstrained so that it can be applied to arbitrary arguments—by applying the rule we learn facts about it the snapshot value. On the other hand, Primitive and Constr force a particular choice of snapshot on the right-hand-side since they create result objects where this value can be picked as required (when applied backwards).

Note that there is no construction rule for generic types `T` and that Constr cannot be applied to empty types (enums with no variants), since neither is constructable in Rust. While the Destr rule still cannot be applied to generics but is equivalent to Unreachable for empty types.

Authors' addresses: Jonáš Fiala, Department of Computer Science, ETH Zurich, Switzerland, jonas.fiala@inf.ethz.ch; Shachar Itzhaky, Technion, Israel, shachari@cs.technion.ac.il; Peter Müller, Department of Computer Science, ETH Zurich, Switzerland, peter.mueller@inf.ethz.ch; Nadia Polikarpova, University of California, San Diego, USA, nadia.polikarpova@ucsd.edu; Ilya Sergey, National University of Singapore, Singapore, ilya@nus.edu.sg.

2475-1421/2023/6-ART164
https://doi.org/10.1145/3591278

$$\textsc{Drop}\ \frac{T \text{ not ref}}{\{\mathtt{x}\colon T(v)\}\ \mathtt{drop!}(\mathtt{x})\ \{\mathtt{emp}\}}$$

$$\textsc{Primitive}\ \frac{T_a, \ldots, T_x \text{ are primitive}}{\{\phi \mid \mathtt{a}\colon T_a(v) * \ldots\}\ \textbf{let}\ \mathtt{x} = \mathtt{a} \oplus \ldots \left\{ \begin{array}{c} w = v \oplus \ldots \wedge \phi \mid \\ \mathtt{x}\colon T_x(w) * \mathtt{a}\colon T_a(v) * \ldots \end{array} \right\}}$$

$$\textsc{Rename}\ \frac{T \text{ not ref}}{\{\mathtt{x}\colon T(v)\}\ \textbf{let}\ \mathtt{y} = \mathtt{x}\ \{\mathtt{y}\colon T(v)\}}$$

$$\textsc{Constr}\ \frac{T = (T_0 \times \ldots)^{\mathsf{V}_0} + \ldots + (T_n \times \ldots)^{\mathsf{V}_n} + \ldots}{\{\mathtt{y}\colon T_n(v) * \ldots\}\ \textbf{let}\ \mathtt{x} = \mathsf{V}_\mathsf{n}(\mathtt{y}, \ldots)\ \{\mathtt{x}\colon T(\{\delta\colon \mathsf{V}_\mathsf{n}, \mathtt{y}\colon v, \ldots\})\}}$$

$$\textsc{Unreachable}\ \frac{}{\{\mathtt{false} \mid P\}\ \mathtt{unreachable!}()\ \{Q\}}$$

$$\textsc{Destr}\ \frac{\begin{array}{cc} T = (T_y \times \ldots)^{\mathsf{V}_0} + (T_z \times \ldots)^{\mathsf{V}_1} + \ldots & \{v = \{\delta\colon \mathsf{V}_0, \mathtt{y}\colon u, \ldots\} \wedge \phi \mid \mathtt{y}\colon T_y(u) * \ldots * P\}\ c_0\ \{Q\} \\ \{\mathtt{r}, \ldots\} = \mathrm{returns}(Q) \quad \{v = \{\delta\colon \mathsf{V}_1, \mathtt{z}\colon w, \ldots\} \wedge \phi \mid \mathtt{z}\colon T_z(w) * \ldots * P\}\ c_1\ \{Q\} & \ldots \end{array}}{\{\phi \mid \mathtt{x}\colon T(v) * P\}\ \textbf{let}\ (\mathtt{r}, \ldots) = \textbf{match}\ \mathtt{x}\ \{\ \mathsf{V}_0(\mathtt{y}, \ldots) \Rightarrow c_0,\ \ \mathsf{V}_1(\mathtt{z}, \ldots) \Rightarrow c_1,\ \ldots\ \}\ \{Q\}}$$

Fig. 1. Ownership Rules.

Borrowing References. The borrowing rules were discussed in detail in Section 3, apart from BorrowIn and BorrowOut. These rules are analogous to DestrBorrow and ConstrBorrow, respectively, but when the referent is itself a reference instead of a data type (that is, these rules handle nested references).

Structural Rules. The rules Frame, Consequence, and Seq are familiar from SL and Hoare logic. One difference about our Frame rule is that it does not require a side condition that free variables of R are not modified by c, because in SOL a program variable can only appear in an assertion once. The Call is similar to the one in SuSLik, except that it is applied backwards.

$$\textsc{DropRef}\quad \frac{}{\left\{\phi \mid \mathtt{x}^{\theta} \mapsto T(v,\hat{x})\right\}\ \mathtt{drop!}(\mathtt{x})\ \{v = \hat{x} \wedge \phi \mid \mathsf{emp}\}}$$

$$\textsc{Write}\quad \frac{}{\left\{\begin{matrix}\mathtt{x} \longmapsto_{\mathtt{mut}} T(v,\hat{x}) * \\ \mathtt{new}\colon T(w)\end{matrix}\right\}\ {*}\mathtt{x} = \mathtt{new}\ \{\mathtt{x} \longmapsto_{\mathtt{mut}} T(w,\hat{x})\}}$$

$$\textsc{ReBorrow}\quad \frac{'a\colon 'b \qquad \mu = \mathtt{mut}\ ?\ \theta = \{'b\} : \nu = \mathtt{imm}}{\left\{\mathtt{x} \overset{'a}{\longmapsto}_{\mu} T(v,\hat{x})\right\}\ \mathbf{let}\ \mathtt{y} = \&\nu\ {*}\mathtt{x}\ \left\{\begin{matrix}\mathtt{y} \overset{'b}{\longmapsto}_{\nu} T(v,\hat{y}) * \\ \mathtt{x}^{\theta} \overset{'a}{\longmapsto}_{\mu} T(\hat{y},\hat{x})\end{matrix}\right\}}$$

$$\textsc{CopyOut}\quad \frac{T \text{ is primitive}}{\{\mathtt{x} \mapsto T(v,\hat{x})\}\ \mathbf{let}\ \mathtt{y} = {*}\mathtt{x}\ \left\{\begin{matrix}\mathtt{x} \mapsto T(v,\hat{x}) * \\ \mathtt{y}\colon T(v)\end{matrix}\right\}}$$

$$\textsc{DestrBorrow}\quad \frac{\begin{matrix} T = (T_y \times \ldots)^{\mathsf{V_0}} + (T_z \times \ldots)^{\mathsf{V_1}} + \ldots \qquad \{\mathsf{r},\ldots\} = \mathrm{returns}(Q) \qquad 'b \text{ is fresh} \\ \left\{v = \{\delta\colon \mathsf{V_0}, \mathtt{0}\colon u,\ldots\} \wedge \phi \mid \mathtt{x}^{\{'b\}} \overset{'a}{\longmapsto} T(\{\delta\colon \mathsf{V_0}, \mathtt{0}\colon \hat{y},\ldots\},\hat{x}) * \mathtt{y} \overset{'b}{\longmapsto} T_y(u,\hat{y}) * \ldots * P\right\}\ c_0\ \{Q\} \\ \left\{v = \{\delta\colon \mathsf{V_1}, \mathtt{0}\colon w,\ldots\} \wedge \phi \mid \mathtt{x}^{\{'b\}} \overset{'a}{\longmapsto} T(\{\delta\colon \mathsf{V_1}, \mathtt{0}\colon \hat{z},\ldots\},\hat{x}) * \mathtt{z} \overset{'b}{\longmapsto} T_z(w,\hat{z}) * \ldots * P\right\}\ c_1\ \{Q\} \qquad \ldots \end{matrix}}{\left\{\phi \mid \mathtt{x} \overset{'a}{\longmapsto} T(v,\hat{x}) * P\right\}\ \mathbf{let}\ (\mathsf{r},\ldots) = \mathbf{match}\ \mathtt{x}\ \left\{\begin{matrix}\mathsf{V_0}(\mathsf{y},\ldots) \Rightarrow c_0, \\ \mathsf{V_1}(\mathsf{z},\ldots) \Rightarrow c_1,\ \ldots\end{matrix}\right\}\ \{Q\}}$$

$$\textsc{ConstrBorrow}\quad \frac{T = (T_0 \times \ldots)^{\mathsf{V_0}} + (T_1 \times \ldots)^{\mathsf{V_1}} + \ldots \qquad \theta = \beta \cup \ldots}{\left\{\mathtt{x}^{\{'b\}} \overset{'a}{\longmapsto} T(\{\delta\colon \mathsf{V_n}, \mathtt{0}\colon \hat{y},\ldots\},\hat{x}) * \mathtt{y}^{\beta} \overset{'b}{\longmapsto} \mathsf{T_n}(v,\hat{y}) * \ldots\right\}\ \mathtt{drop!}(\mathtt{y});\ldots\ \left\{\mathtt{x}^{\theta} \mapsto T(\{\delta\colon \mathsf{V_n}, \mathtt{0}\colon v,\ldots\},\hat{x})\right\}}$$

$$\textsc{BorrowIn}\quad \frac{}{\left\{\mathtt{x} \overset{'a}{\longmapsto}_{\mu} \&\mathtt{mut}\ \mathtt{T}((v,\hat{a}),\hat{x})\right\}\ \mathbf{let}\ \mathtt{y} = \&\mu\ {*}{*}\mathtt{x}\ \left\{\mathtt{x}^{\{'b\}} \overset{'a}{\longmapsto}_{\mu} \&\mathtt{mut}\ \mathtt{T}((\hat{y},\hat{a}),\hat{x}) * \mathtt{y} \overset{'b}{\longmapsto}_{\mu} T(v,\hat{y})\right\}}$$

$$\textsc{BorrowOut}\quad \frac{}{\left\{\mathtt{x}^{\{'b\}} \overset{'a}{\longmapsto} \&\mathtt{mut}\ \mathtt{T}((\hat{y},\hat{a}),\hat{x}) * \mathtt{y}^{\theta} \overset{'b}{\longmapsto} T(v,\hat{y})\right\}\ \mathtt{drop!}(\mathtt{y})\ \left\{\mathtt{x}^{\theta} \overset{'a}{\longmapsto} \&\mathtt{mut}\ \mathtt{T}((v,\hat{a}),\hat{x})\right\}}$$

Fig. 2. Borrowing Rules.

$$\textsc{Frame}\quad \frac{\{\phi \mid P\}\ c\ \{\psi \mid Q\}}{\{\phi \mid P * R\}\ c\ \{\psi \mid Q * R\}} \qquad \textsc{Consequence}\quad \frac{\phi \Rightarrow \phi' \qquad \psi' \Rightarrow \psi \qquad \{\phi' \mid P\}\ c\ \{\psi' \mid Q\}}{\{\phi \mid P\}\ c\ \{\psi \mid Q\}}$$

$$\textsc{Seq}\quad \frac{\{\phi \mid P\}\ c_0\ \{\psi \mid Q\} \qquad \{\psi \mid Q\}\ c_1\ \{\chi \mid R\}}{\{\phi \mid P\}\ c_0; c_1\ \{\chi \mid R\}}$$

$$\textsc{Emp}\quad \frac{\phi \Longrightarrow \psi}{\{\phi \mid \mathsf{emp}\}\ c\ \{\psi \mid \mathsf{emp}\}} \qquad \textsc{Call}\quad \frac{\phi \Longrightarrow \phi' \qquad \{\phi' \mid \mathsf{a}\colon T_a(v) * \ldots\}\ \mathtt{foo}\ \{\psi \mid \mathsf{result}\colon T(w)\}}{\{\phi \mid \mathsf{a}\colon T_a(v) * \ldots * P\}\ c; \mathbf{let}\ \mathsf{r} = \mathtt{foo}(\mathsf{a},\ldots)\ \{\phi \wedge \psi \mid \mathsf{r}\colon T(w) * P\}}$$

Fig. 3. Structural rules.

B NOTABLE SYNTHESIS RESULTS

B.1 Synthesis from Types Alone

Below is a sample of interesting examples from the "non-primitive" part of the 100-Crates test suite; these programs are synthesized just from their type signatures.

```
// h2/src/frame/data.rs:28 (identical to actual impl)
fn new(stream_id: StreamId, payload: T) -> Data<T> {
    let flags = DataFlags(0);
    Data { stream_id, data: payload, flags, pad_len: None }
}

// nom/src/internal.rs:348 (close to actual impl, but Ok case uses closure)
fn parse(&mut self, i: I) -> Result<(I, O2), Err<E>> {
    let result = self.f.parse(i);
    match result {
    // Should be:
    // Result::Ok((i, o)) => Result::Ok((i, (self.g)(o))),
    Result::Ok(_) => {
        let _0 = Err::Incomplete(Needed::Unknown);
        Result::Err(_0)
    }
    Result::Err(_0) => Result::Err(_0),
    }
}

// either/src/lib.rs:832 (identical to actual impl)
fn factor_first(self) -> (T, Either<A, B>) {
    match self {
        Either::Left(_0) => {
            let _1 = Either::Left(_0.1);
            (_0.0, _1)
        }
        Either::Right(_0) => {
            let _1 = Either::Right(_0.1);
            (_0.0, _1)
        }
    }
}

// syn/src/punctuated.rs:984 (identical to actual impl)
fn into_tuple(self) -> (T, Option<P>) {
    match self {
        Pair::Punctuated(_0, _1) => {
            let _1 = Some(_1);
            (_0, _1)
        }
        Pair::End(_0) => (_0, None),
    }
}
```

B.2 Complex Reasoning about References

The next function for a linked list iterator, which operates on the `Iter` datatype with a reference-typed field.

```
// SLL Tutorial definition, rather than our simplified definition
struct List<T> { elem: T, next: Option<Box<List<T>>> }
pub struct Iter<'a, T> { next: Option<&'a List<T>> }
```

```
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;

    #[ensures(match self.len() {
        0 => (^self).len() == 0 && matches!(result, None),
        v => (^self).len() == v-1 && matches!(result, Some(_)),
    })]
    fn next(&mut self) -> Option<Self::Item> {
    let (new, result) = match &mut self.next {
        None => (None, None),
        Some(_0) => match &_0.next {
            None => {
                let result = Some(&_0.elem);
                (None, result)
            }
            Some(_0_next_de) => {
                let result = Some(&_0.elem);
                let new = Some(&**_0_next_de);
                (new, result)
            }
        },
    };
    self.next = new;
    result
    }
}
```

B.3 Complex Recursion

The *non-destructive append* benchmark takes as input immutable references to two linked lists and creates a new list containing elements of both. The main function, `sll_append_copy`, recursively traverses one of the lists, and upon reaching the end, it calls the auxiliary `sll_append_copy_7`, whose task is to recursively copy the other list. This auxiliary is not provided by the user, but is automatically discovered by RusSOL.

```
// Simple linked list
enum List<T> { Nil, Cons(Box<(T, List<T>)>) }

#[ensures(result.len() == x1.len() + x2.len())]
fn sll_append_copy<T: Copy>(x1: &List<T>, x2: &List<T>) -> List<T> {
    match x2 {
        List::Nil => sll_append_copy_7(x1),
        List::Cons { elem, next } => {
            let de = *elem;
            let result = sll_append_copy(x1, &**next);
            let next = Box::new(result);
            List::Cons { elem: de, next }
        }
    }
}
fn sll_append_copy_7<T: Copy>(x: &List<T>) -> List<T> {
    match x {
        List::Nil => List::Nil,
        List::Cons { elem, next } => {
            let de = *elem;
            let result = sll_append_copy_7(next);
            let next = Box::new(result);
            List::Cons { elem: de, next }
        }
    }
}
```

The *rose tree copy* benchmark takes as input an immutable reference to a rose tree and creates a new tree with the same set of elements. A rose tree is a variable-arity tree, whose children are stored in a linked list. To copy a rose tree, the synthesizer needs to create two mutually recursive functions: copy copies a tree and calls copy_8 to copy its children; copy_8 copies a list of trees and calls copy to copy each of them.

```
    // A rose tree is either empty or a node with a value and a linked list of subtrees
    enum Tree<T> { Nil, Cons { elem: T, next: List<Tree<T>> } }
    // Simple linked list
    enum List<T> { Nil, Cons(Box<(T, List<T>)>) }

    impl<T> Tree<T> {
        #[ensures(result.elems() == self.elems())]
        fn copy(&self) -> Tree<T> {
            match self {
                Tree::Nil => Tree::Nil,
                Tree::Cons { elem, next } => {
                    let de = *elem;
                    Self::copy_8(de, elem, next) // mutual recursion
                }
            }
        }
        fn copy_8(de: T, elem_self: &T, next: &List<Tree<T>>) -> Tree<T> {
            match next {
                List::Nil => Tree::Cons { elem: de, next: List::Nil },
                List::Cons(_0) => {
                    let result = _0.0.copy();  // mutual recursion
                    match result {
                        Tree::Nil => Self::copy_8(de, elem_self, &_0.1),
                        Tree::Cons { elem: elem_result, next } => {
                            let result = Self::copy_8(de, elem_self, &_0.1);
                            let bx = (result, next);
                            let _0 = Box::new(bx);
                            let next = List::Cons(_0);
                            Tree::Cons { elem: elem_result, next }
                        }
                    }
                }
            }
        }
    }
```