A SYNTHESES 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.
\begin{align*}
\text{Drop} & \quad T \text{ not ref} \quad \{x: T(v)\} \; \text{drop!}(x) \{\text{emp}\} \\
\text{Rename} & \quad T \text{ not ref} \quad \{x: T(v)\} \; \text{let} \; y = x \{y: T(v)\} \\
\text{Unreachable} & \quad \{\text{false} \mid P\} \; \text{unreachable!}() \{Q\}
\end{align*}

\begin{align*}
\text{DROP} & \quad T \text{ not ref} \quad \{x: T(v)\} \; \text{drop!}(x) \{\text{emp}\} \\
\text{RENAME} & \quad T \text{ not ref} \quad \{x: T(v)\} \; \text{let} \; y = x \{y: T(v)\} \\
\text{IMPORT} & \quad \{\text{false} \mid P\} \; \text{unreachable!}() \{Q\}
\end{align*}

\begin{align*}
\text{PRIMITIVE} & \quad T_a, \ldots, T_x \text{ are primitive} \\
& \quad \{\phi \mid a: T_a(v) \star \ldots \star \phi \mid \{x: T_x(w) * \star a: T_a(v) \star \ldots \}\}
\end{align*}

\begin{align*}
\text{CONSTR} & \quad T = (T_0 \times \ldots)^{V_0} + \ldots + (T_n \times \ldots)^{V_n} + \ldots \\
& \quad \{y: T_n(v) \star \ldots\} \; \text{let} \; x = V_n(y, \ldots) \{x: T(\{\delta: V_n, y: v, \ldots\})\}
\end{align*}

Fig. 1. Ownership Rules.

**Borrowing References.** The borrowing rules were discussed in detail in Section 3, apart from \textsc{BorrowIn} and \textsc{BorrowOut}. These rules are analogous to \textsc{DestrBorrow} and \textsc{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 \textsc{Frame}, \textsc{Consequence}, and \textsc{Seq} are familiar from SL and Hoare logic. One difference about our \textsc{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 \textsc{Call} is similar to the one in SuSLik, except that it is applied backwards.
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.

---

**Fig. 2. Borrowing Rules.**

---

**Fig. 3. Structural rules.**

---

**B. NOTABLE SYNTHESIS RESULTS**
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.

```rust
// 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,
}

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.

```rust
enum Tree<T> { Nil, Cons { elem: T, next: List<T> } }
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 bx = (result, next);
                        let _0 = Box::new(bx);
                        let _0 = List::Cons(_0);
                        Tree::Cons { elem: elem_result, next }
                    }
                }
            }
        }
    }
}
```

// A rose tree is either empty or a node with a value and a linked list of subtrees
// Simple linked list