# Assignment 1: Get at least 1 working benchmark done

Jordan Dehmel

Due no later than 11:59 PM on Friday, January 16, 2026

## Choice of first benchmark

From a cursory glance, problem 8 (AKA `list_append_inj_1.smt2`) seems like one of the more simple ones. The following is the SMT2 code defining the problem.

```
; Injectivity of append
(declare-datatype
  list (par (a) ((nil) (cons (head a) (tail (list a))))))
(define-fun-rec
  ++
  (par (a) (((x (list a)) (y (list a))) (list a)))
  (match x
    ((nil y)
     ((cons z xs) (cons z (++ xs y))))))
(prove
  (par (a)
    (forall ((xs (list a)) (ys (list a)) (zs (list a)))
      (=> (= (++ xs zs) (++ ys zs)) (= xs ys)))))
```

## Understanding the benchmark in `SMT-LIB`

We will take this line-by-line according to the SMT2 documentation.

### Line 1: `declare-datatype`

Line 1 is

```
(declare-datatype list
  (par (a) (
    (nil)
    (cons
      (head a)
      (tail
        (list a)
      )
    )
  ))
)
```

The relevant information comes from `smt-lib` reference, page 61.

This means that `(declare-datatype list ...)` means "create an algebraic datatype `list` with the following constructors". The `(par (a) BODY)` syntax allows parametric types with parameter `a` (Ibid, page 29).

Therefore, line 1 creates a parametric algebraic datatype called `(list a)` (over parameter `a`) which is instantiable via either a `nil` (empty list / list termination) or a `cons` (non-empty list with two data members) constructor.

### Line 2: `define-fun-rec`

Line 2 is

```
(define-fun-rec
  ++
  (par (a) (
    (
      (x (list a))
      (y (list a))
    )
    (list a)
  ))
  (match x
    (
      (nil y)
      (
        (cons z xs)
        (cons z (++ xs y))
      )
    )
  )
)
```

The manual deals with `define-fun-rec` on page 63. This command takes the form `(define-fun-rec f ((x1 s1) ... (xn sn)) s t)`, where `f` is the name of the function (of sort `s`), `(xi si)` is the `i`-th argument named `xi` of sort `si`, and `t` is the function body. Also see page 29 for match statements.

```
(define-fun-rec              ; Define a recursive function
  ++                         ; Named "++"
  (par (a) (                 ; With signature parametrized by a
    (
      (x (list a))           ; Arg 1 named x of sort (list a)
      (y (list a))           ; Arg 2 named y of sort (list a)
    )
    (list a)                 ; Return sort (list a)
  ))
  (match x                   ; Begin function body: Match on x
    (
```

```
      (nil y)                 ; If x was nil, return y
      (
        (cons z xs)           ; Else, head=z tail=xs
        (cons z (++ xs y))    ; Return new list w head=z,
                              ; tail=(++ xs y)
      )
    )
  )
)
```

In effect, this creates a function `++` which takes in two lists `a, b` and returns a new list containing all the elements of `a` followed by all the elements of `b`.

#### Line 3: `prove`

Line 3 is

```
(prove
  (par (a)
    (forall
      (
        (xs (list a))
        (ys (list a))
        (zs (list a))
      )
      (=>
        (= (++ xs zs) (++ ys zs))
        (= xs ys)
      )
    )
  )
)
```

In effect, this says that:

> Given some type parameter `a`, for all `xs, ys, zs` of sort `(list a)`, if the appendation of `zs` onto the end of `xs` is equal to the appendation of `zs` onto the end of `ys`, it must be true that `xs` are equal to `ys`.

## Translating `(list a)` to rust

We need to have the following:

- A generic `LinkedList<T>` structure (or enumeration) with
  1. A `nil` variant constructor to signify an empty list
  2. A head/tail variant constructor to signify a nonempty list
- A function taking in two like-typed lists `a, b` and returning a new list containing all items of `a` followed by all items of `b`
  - **We will call this `append(a, b)`**

From these, we need to prove that: - For any generic type: - For any three like-typed lists `xs, ys, zs`: - `append(xs, zs) == append(ys, zs)` implies that `xs == ys`

A corresponding `rust` snippet is given below.

```
/// A module exposing an axiomitized generic linked list
pub mod linked_list {
  /// A generic linked list
  #[derive(Clone)]
  #[derive(PartialEq)]
  pub enum LinkedList<T: Clone + PartialEq + fmt::Display> {
    /// The end-of-list / empty-list constructor
    Nil,
    /// A non-empty constructor which points to memory on the
    /// heap
    Cons { head: T, tail: Box<LinkedList<T>> },
  }
}
```

The functions operating on this struct will be axiomitized later. Now we just need to figure out how to encode our "prove" statement in `ravencheck` syntax.

### Expressing in `ravencheck`

The first hurdle is "universally quantifying" (not really) over type parameters. My main resource here is `ravencheck`'s own parametrized set example, an excerpt of which is shown below.

```
#[ravencheck::check_module]
#[declare_types(u32, HashSet<_>)]
#[allow(dead_code)]
mod my_mod {
  use std::collections::HashSet;
  use std::hash::Hash;

  // ...

  #[declare]
  fn empty_poly<E: Eq + Hash>() -> HashSet<E> {
    // ...
  }

  // ...

  #[assume]
  #[for_type(HashSet<E> => <E>)]
  fn empty_no_member<E: Eq + Hash>() -> bool {
```

```
    // ...
  }

  // ...
}
```

A simpler solution also presented in the documentation is to simply alias `u32` (or something else) to a "generic" type, in which case `ravencheck` will leave it uninterpreted. We will do this latter method. Our implementation is given below.

```
#[ravencheck::check_module]
#[declare_types(LinkedList<_>, u32)]
#[allow(dead_code)]
mod p8 {
  // Import the enum we are examining
  use crate::list::linked_list::LinkedList;

  // Make an UNINTERPRETED datatype
  #[declare]
  type T = u32;

  // Returned when we try to access a null cons's data
  #[declare]
  const NULL: T = 0;

  #[declare]
  const NIL: LinkedList<T> = LinkedList::<T>::Nil{};

  ////////////////////////////////////////////////////////////
  // Relations

  #[define]
  #[total]
  fn eq(a: &LinkedList<T>, b: &LinkedList<T>) -> bool {
    a == b
  }

  // This is safe from sort cycles
  #[declare]
  #[total]
  fn data(cur: &LinkedList<T>) -> T {
    match cur {
      LinkedList::<T>::Cons{head, tail: _} => *head,
      _ => NULL
    }
  }
```

```
#[declare]
#[total]
fn is_next(cur: &LinkedList<T>,
           candidate: LinkedList<T>) -> bool {
  match cur {
    LinkedList::<T>::Cons{head: _, tail} =>
      eq(&*tail, &candidate),
    _ => eq(&NIL, &candidate)
  }
}

//////////////////////////////////////////////////////////////
// Appendation

#[declare]
fn is_appendation(x: LinkedList<T>, y: LinkedList<T>,
                  z: LinkedList<T>) -> bool {
  if data(&x) != data(&z) {
    false
  } else {
    match x {
      LinkedList::<T>::Cons{tail: x_next, ..} => match z {
        LinkedList::<T>::Cons{tail: z_next, ..} =>
          is_appendation(
            *x_next, y, *z_next
          ),
        _ => false
      },
      _ => eq(&z, &NIL)
    }
  }
}

// (a ++ b == d) and (a ++ c == d) iff b == c
#[assume]
fn appendation_axiom() -> bool {
  forall(|a: LinkedList<T>, b: LinkedList<T>,
          c: LinkedList<T>, d: LinkedList<T>| {
    implies(
      is_appendation(a, b, d) && is_appendation(a, c, d),
      b == c
    ) && implies(
      b == c,
      is_appendation(a, b, d) && is_appendation(a, c, d)
    )
```

```
    })
  }

  /////////////////////////////////////////////////////////////////
  // WTS

  #[annotate_multi]
  #[for_values(xs: LinkedList<T>, ys: LinkedList<T>,
               zs: LinkedList<T>, a: LinkedList<T>,
               b: LinkedList<T>)]
  fn injectivity_of_append() -> bool {
    implies(
      is_appendation(xs, zs, a) &&
      is_appendation(ys, zs, b) &&
      eq(a, b),
      eq(xs, ys)
    )
  }
}
```

`is_next` and `is_appendation` needed to be formulated as relations explicitly in order to avoid sort cycles: Much debugging time was spent figuring this out. Once they were correctly formulated, only a single appendation axiom was needed.

## Verification

After relation-izing the operations and adding our axiom, problem 8 verified in 0.05 seconds.

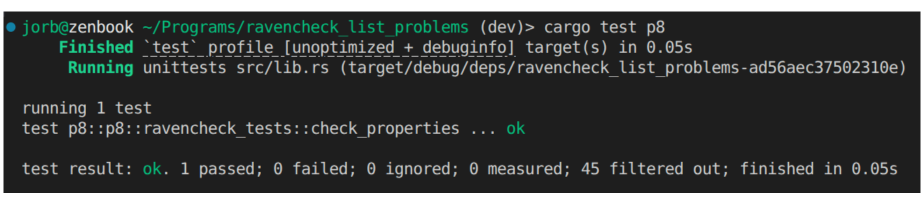


Figure 1: `p8` verified

Since problem 9 is nearly identical to problem 8, I copied over the axioms and added the corresponding `verify` function.

```
// ... same as p8

#[annotate_multi]
#[for_values(xs: LinkedList<T>, ys: LinkedList<T>,
             zs: LinkedList<T>, a: LinkedList<T>,
```

```
            b: LinkedList<T>)]
fn injectivity_of_append_2() -> bool {
  implies(
    is_appendation(xs, ys, a) &&
    is_appendation(xs, zs, b) &&
    eq(a, b),
    eq(ys, zs)
  )
}

// ...
```

Problem 9 unexpectedly also verified with no additional work!

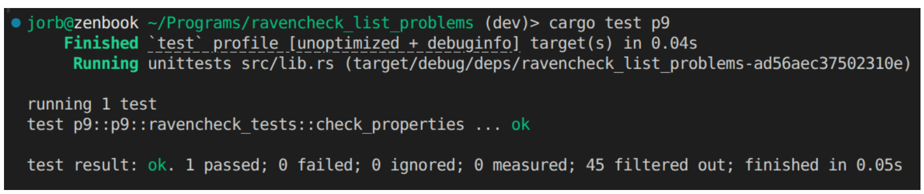


Figure 2: In solving `p8`, we got `p9` for free!

We now have 2 of the 46 problems done.