# Assignment 1: Get at least 1 working benchmark done

Jordan Dehmel

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

**Note:** This was updated Jan 22, 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`

## Expressing in `ravencheck`

The following expresses the `LinkedList` type, **without** a type parameter. It does include the `append` function. Note that all these function signatures are perfectly fine with respect to sort cycles: They would need to be quantified for that to be an issue.

```
/// A module exposing an axiomitized generic linked list
#[ravencheck::export_module]
#[allow(dead_code)]
pub mod linked_list {
  // Make an UNINTERPRETED datatype
  #[declare]
  type T = u32;

  /// A generic linked list
  #[define]
  pub enum LinkedList {
    /// The end-of-list / empty-list constructor
    Nil,
    /// A non-empty constructor which points to memory on the
    /// heap
    Cons(T, Box<LinkedList>),
  }

  #[define]
  #[recursive]
  fn interleave(x: LinkedList, y: LinkedList) -> LinkedList {
    match x {
      LinkedList::Nil => y,
      LinkedList::Cons(z, xs) =>
        LinkedList::Cons(z, Box::new(interleave(y, *xs)))
    }
  }

  #[define]
  #[recursive]
  fn append(x: LinkedList, y: LinkedList) -> LinkedList {
    match x {
```

```
      LinkedList::Nil => y,
      LinkedList::Cons(z, xs) => LinkedList::Cons(z, Box::new(append(*xs, y)))
    }
  }
}
```

## Verification

Problem 8 ended up requiring further axiomitization: However, problem 9 (`list_append_inj_2`) was immediately verifiable.

```
// /*
// ; 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 ys) (++ xs zs)) (= ys zs)))))
// */

#[ravencheck::check_module]
#[allow(dead_code)]
#[allow(unused_imports)]
mod p9 {
  #[import]
  use crate::list::linked_list::*;

  #[annotate_multi]
  #[for_values(xs: LinkedList, ys: LinkedList, zs: LinkedList)]
  #[for_call(append(xs, ys) => a)]
  #[for_call(append(xs, zs) => b)]
  fn injectivity_of_append_2() -> bool {
    implies(
      a == b,
      ys == zs
    )
  }
}
```

Note that **all** function calls must be given in `for_call` annotations, and that the

return values of these statements do **not** need to be quantified via `for_values`.