# 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.

```rust
/// A module exposing an axiomitized generic linked list
pub mod linked_list {
  /// A generic linked list
  #[derive(Clone)]
  pub enum LinkedList<T> where T: Clone {
    /// 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>> },
  }

  /// Given two linked lists of similar type, return a new list
  /// containing the elements of the first (x) followed by the
  /// elements of the second (y).
  pub fn append<T>(
    x: LinkedList<T>,
    y: LinkedList<T>) -> LinkedList<T> where T: Clone {
    match x {
      LinkedList::Nil => y,
      LinkedList::Cons{head, tail} => LinkedList::Cons{
        head: head,
        tail: Box::new(append(
          (*tail).clone(), y
        ))
      }
    }
  }

  /// Returns whether or not two lists are identical (EG of the
  /// same length and containing corresponding items)
  pub fn eq<T>(x: &LinkedList<T>, y: &LinkedList<T>
    ) -> bool where T: Clone + std::cmp::PartialEq {
    match x {
      LinkedList::Nil => match y {
        LinkedList::Nil => true,
        _ => false
      },
      LinkedList::Cons{head: x_head, tail: x_tail} => match y {
        LinkedList::Nil => false,
        LinkedList::Cons{head: y_head, tail: y_tail} =>
```

```
          (x_head == y_head) && eq(x_tail, y_tail)
        }
      }
    }
}
```

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 {
    // ...
  }

  // ...
}
```

Our implementation is given below.

```
/*
; ./problems/list_append_inj_1.smt2
; 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)))))
*/

#[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

  // Wraps equality for nodes
  #[define]
  fn eq(a: &LinkedList<T>, b: &LinkedList<T>) -> bool {
    a == b
  }

  // Wrapper for nonempty list constructor
  #[declare]
  fn cons(head: T, tail: LinkedList<T>) -> LinkedList<T> {
    LinkedList::<T>::Cons{head: head, tail: Box::new(tail)}
  }

  // Gets the data from the given node (or null if nil)
  #[declare]
  fn data(cur: &LinkedList<T>) -> T {
    match cur {
      LinkedList::<T>::Cons{head, tail: _} => *head,
      _ => NULL
    }
  }

  #[assume]
  fn data_eq_pres() -> bool {
    forall(|a: LinkedList<T>, b: LinkedList<T>| {
      implies(eq(a, b), data(&a) == data(&b))
    })
  }

  // Gets the node after the given node (or the nil node)
  #[declare]
  fn next(cur: LinkedList<T>) -> LinkedList<T> {
    match cur {
      LinkedList::<T>::Cons{head: _, tail} => *tail,
      _ => NIL
    }
  }

  // Wrapper for appendation, called "++" in the problem
  // statement. Note: I've done some reformatting here
  #[define]
  #[recursive]
  fn append(x: LinkedList<T>, y: LinkedList<T>) -> LinkedList<T> {
    if eq(&x, &NIL) {
      y
    } else if eq(&y, &NIL) {
      x
    } else {
      cons(data(&x), append(next(x), y))
    }
  }

  /////////////////////////////////////////////////////////////
  // Axioms
  // Note: Many of these are likely redundant. Hopefully none
  // are cheating.

  // eq is total
  #[assume]
  #[for_inst(eq(a, b))]
  fn eq_totality() -> bool {
    forall(|a: LinkedList<T>, b: LinkedList<T>| {
      eq(a, b) || !eq(a, b)
    })
  }
```

```
// next is total
#[assume]
#[for_inst(next(a))]
fn next_totality() -> bool {
  forall(|a: LinkedList<T>| {
    let _ = next(a);
    true
  })
}

// data is total
#[assume]
#[for_inst(data(a))]
fn data_totality() -> bool {
  forall(|a: LinkedList<T>| {
    let _ = data(a);
    true
  })
}

#[assume]
#[for_inst(next(append(cons(a, NIL), y)))]
#[for_inst(data(append(cons(a, NIL), y)))]
fn simple_append() -> bool {
  forall(|a: T, y: LinkedList<T>| {
    eq(next(append(cons(a, NIL), y)), y) &&
    data(append(cons(a, NIL), y)) == a
  })
}

// Cons meaning
#[assume]
#[for_inst(eq(&x, &y))]
#[for_inst(eq(&cons(a, x), &cons(b, y)))]
fn cons_meaning() -> bool {
  forall(|a: T, b: T, x: LinkedList<T>, y: LinkedList<T>| {
    implies(
      a == b && eq(&x, &y),
      eq(&cons(a, x), &cons(b, y))
    )
  })
}

// Data meaning
#[assume]
```

```
#[for_inst(data(cons(a, x)))]
fn data_meaning() -> bool {
  forall(|a: T, x: LinkedList<T>| {
    data(cons(a, x)) == a
  })
}

// Next meaning
#[assume]
#[for_inst(eq(next(cons(a, x)), x))]
#[for_inst(eq(cons(a, x), y))]
#[for_inst(eq(x, next(y)))]
#[for_inst(data(&y))]
fn next_meaning() -> bool {
  forall(|a: T, x: LinkedList<T>| {
    eq(next(cons(a, x)), x)
  }) && forall(|a: T, x: LinkedList<T>, y: LinkedList<T>| {
    implies(
      eq(cons(a, x), y),
      a == data(&y) && eq(x, next(y))
    ) && implies(
      a == data(&y) && eq(x, next(y)),
      eq(cons(a, x), y)
    )
  })
}

// Axiom: (t, x) == (u, y) iff t == u and x == y
#[assume]
#[for_inst(data(&x))]
#[for_inst(data(&y))]
#[for_inst(next(x))]
#[for_inst(next(y))]
fn equality_meaning() -> bool {
  forall(|x: LinkedList<T>, y: LinkedList<T>| {
    implies(
      eq(x, y),
      data(&x) == data(&y) && eq(next(x), next(y))
    ) && implies(
      data(&x) == data(&y) && eq(next(x), next(y)),
      eq(x, y)
    )
  })
}

// Axiom: appending nothing does not change a list
```

```
#[assume]
#[for_inst(append(x, NIL))]
fn append_nil() -> bool {
  forall(|x: LinkedList<T>| {
    eq(append(x, NIL), x)
  })
}

#[assume]
#[for_inst(cons(a, x))]
#[for_inst(cons(b, x))]
fn prepend() -> bool {
  forall(|x: LinkedList<T>, a: T, b: T| {
    implies(
      eq(
        cons(a, x),
        cons(b, x)
      ),
      a == b
    )
  })
}

// Axiom: The special NIL object is the end of lists
#[assume]
#[for_inst(next(NIL))]
fn empty_is_empty() -> bool {
  eq(next(NIL), NIL)
}

#[assume]
#[for_inst(append(x, cons(t, y)))]
#[for_inst(append(append(x, cons(t, NIL)), y))]
fn append_item() -> bool {
  forall(|x: LinkedList<T>, y: LinkedList<T>, t: T| {
    eq(
      append(x, cons(t, y)),
      append(append(x, cons(t, NIL)), y)
    )
  })
}

#[assume]
#[for_inst(eq(l, r))]
#[for_inst(eq(append(l, z), append(r, z)))]
fn appendation_equality() -> bool {
    forall(|l: LinkedList<T>, r: LinkedList<T>, t: T| {
      let z = cons(t, NIL);
      implies(
        eq(l, r),
        eq(append(l, z), append(r, z))
      ) && implies(
        eq(append(l, z), append(r, z)),
        eq(l, r)
      )
    })
  }

  ////////////////////////////////////////////////////////////////
  // The property we want to prove (injectivity)

  #[verify]
  fn injectivity_of_append() -> bool {
    forall(|xs: LinkedList<T>,
            ys: LinkedList<T>,
            zs: LinkedList<T>| {
      implies(eq(append(xs, zs), append(ys, zs)), eq(xs, ys))
    })
  }
}
```

## Verification

The following statement was found by the model checker to be SAT.

```
(exists
  (
    (x_xs UI_LinkedList__UI_T__)
    (x_ys UI_LinkedList__UI_T__)
    (x_zs UI_LinkedList__UI_T__)
  )
  (exists
    (
      (xn_17
        UI_LinkedList__UI_T__
      )
    )
    (exists
      (
        (xn_18
          UI_LinkedList__UI_T__
        )
```

```
   )
   (and
    (and
     (and
      (not
       (F_lists_are_eq__UI_T__
        x_xs
        x_ys
       )
      )
      (F_lists_are_eq__UI_T__
       xn_18
       xn_17
      )
     )
     (F_append_lists__UI_T__
      x_ys
      x_zs
      xn_17
     )
    )
    (F_append_lists__UI_T__
     x_xs
     x_zs
     xn_18
    )
   )
  )
 )
)
```