Mechanically Proving the Equivalence of Unzip Functions

[redacted]

December 14, 2020

1 Introduction

An association-list (or alist) is a list in which each element is a pair; in ACL2s, the recognizer `alistp` is implemented to check that each element of the list is itself a cons. We give some alist examples below.

```
'((a . 1) (b . 2) (c . 3))
'((0) (0) (0))
'((a b c) (1 2 3) )
```

Given an alist, a natural operation is to split apart each of the pairs, separating the car of each pair from the cdr and building up two separate lists, one containing all of the cars and the other containing all of the cdrs. The terminology we use to describe this operation is "unzipping" the alist. Unzipping the alist examples from above results in:

```
'((a b c) . (1 2 3))
'((0 0 0) . (nil nil nil))
'((a 1) . ((b c) (2 3)))
```

Notice that taking the car of the unzipped list returns a list containing all of the alist's cars, whereas taking the cdr of the unzipped list returns a list containing all of the alist's cdrs. Implementing the unzip operation can be done in a variety of ways. We have given two such implementations below:

```
; Given an association list, recursively splits the cars
; and cdrs of each pair and returns as a list.
(definec unzip-lists (al :alist) :tl
  (cond
   ((endp al) (cons nil nil))
   (t (let ((res (unzip-lists (cdr al))))
        (cons (cons (car (car al)) (car res))
              (cons (cdr (car al)) (cdr res)))))))
```

```
; Recursively accumulates the list of cars in acc1 and the list
; of cdrs in acc2.
(definec unzip-lists2-acc (al :alist acc1 :tl acc2 :tl) :tl
  (cond
   ((endp al) (cons acc1 acc2))
   (t (unzip-lists2-acc (cdr al)
                        (app2 acc1 (list (car (car al))))
                        (app2 acc2 (list (cdr (car al))))))))

; Starts the accumulators in unzip-lists2-acc as empty lists
(definec unzip-lists2 (al :alist) :tl
  (unzip-lists2-acc al '() '()))
```

`unzip-lists` recursively walks down the alist and builds two parallel lists. At each step, the function breaks apart the current pair of the alist and attaches the two elements of the pair to the correct sub-list.

`unzip-lists-2` also walks down the alist, but instead, builds two accumulators, one to hold all of the cars, and one to hold all of the cdrs. For each pair in the alist, the function will break it apart and append each element to the correct accumulator list. (In addition to the definitions above, we have separately defined `app2` to append two lists together. This implementation has been listed in the Appendix).

Our goal is to prove the equivalence of `unzip-lists` and `unzip-lists2`. The formal conjecture is:

```
Conjecture 1.
(implies (alistp a)
         (equal (unzip-lists a) (unzip-lists2 a)))
```

2 Initial Proof Structure

As its definition suggests, the unzip operation can really be reduced to two separate operations - collecting all of the cars of each pair into one list, and the cdrs of each pair into another. These can be implemented as two functions `get-cars` and `get-cdrs`. As their names suggest, `(get-cars a)` will return all of the cars of each pair in the alist a, whereas `(get-cdrs a)` will return all of the cdrs of each pair in the alist a.

Below are the initial recursive implementations of `get-cars` and `get-cdrs`.

```
; Recursively builds a list of all cars in the alist
(definec get-cars (a :alist) :tl
  (cond
   ((endp a) nil)
   (t (cons (car (car a)) (get-cars (cdr a))))))

; Recursively builds a list of all cdrs in the alist.
```

```
(definec get-cdrs (a :alist) :tl
  (cond
   ((endp a) nil)
   (t (cons (cdr (car a)) (get-cdrs (cdr a))))))
```

If we show that both `unzip-lists` and `unzip-lists2` can be written in terms of the stripped cars and cdrs (Lemmas 1 and 2), then it must be the case that both versions are equivalent functions by the transitive property, thereby proving Conjecture 1.

```
Lemma 1.
(implies (alistp a)
         (equal (unzip-lists a)
                (cons (get-cars a)
                      (get-cdrs a)))))

Lemma 2.
(implies (alistp a)
         (equal (unzip-lists2 a)
                (cons (get-cars a)
                      (get-cdrs a)))))
```

3 Proving Conjecture 1

ACL2s can automatically prove Lemma 1, owing to the similar recursive scheme between `unzip-lists`, `get-cars`, and `get-cdrs`. However, it cannot automatically prove Lemma 2.

Our revised strategy is to write new accumulator versions of `get-cars` and `get-cdrs` that closely mirror the recursive scheme of `unzip-lists2` and use those when reasoning with `unzip-lists2`. Below are the implementations of the new `get-cars2` and `get-cdrs2` functions.

```
; Accumulates a list of all cars in the alist
(definec get-cars2-acc (a :alist acc :tl) :tl
  (cond
   ((endp a) acc)
   (t (get-cars2-acc (cdr a) (app2 acc (list (car (car a))))))))

; Starts the accumulator in get-cars2-acc
(definec get-cars2 (a :alist) :tl
  (get-cars2-acc a '()))

; Accumulates a list of all cdrs in the alist.
(definec get-cdrs2-acc (a :alist acc :tl) :tl
  (cond
   ((endp a) acc)
```

```
   (t (get-cdrs2-acc (cdr a) (app2 acc (list (cdr (car a))))))))

; Starts the accumulator in get-cdrs2-acc
(definec get-cdrs2 (a :alist) :tl
  (get-cdrs2-acc a '()))
```

Using these new definitions, we can rewrite Lemma 2 as follows:

```
Lemma 2a.
(implies (alistp a)
         (equal (unzip-lists2 a)
                (cons (get-cars2 a)
                      (get-cdrs2 a)))))
```

Now, if we can prove that `get-cars2` and `get-cdrs2` are equivalent to their non-accumulator counterparts, we can use Lemma 2a and Lemma 2 interchangeably. Thus, we want to be able to prove the following:

```
Lemma 3.
(implies (alistp a)
         (equal (get-cars a) (get-cars2 a)))

Lemma 4.
(implies (alistp a)
         (equal (get-cdrs a) (get-cdrs2 a)))
```

ACL2s cannot prove these on its own, so we must specify a general theorem that relates the accumulator function `get-cars2-acc` to `get-cars`. To that end, we introduce the following lemmas:

```
Lemma 5.
(implies (and (alistp a)
              (tlp acc))
         (equal (get-cars2-acc a acc) (app2 acc (get-cars a)))))

Lemma 6.
(implies (and (alistp a)
              (tlp acc))
         (equal (get-cdrs2-acc a acc) (app2 acc (get-cdrs a)))))
```

Knowing that `get-cars2` and `get-cdrs2` merely initialize their respective accumulators as nil, it is clear by the substitution `((acc nil))` that `(app2 acc (get-cars a))` will return `(get-cars a)`. Therefore, Lemmas 5 and 6 are sufficient to show that `get-cars2` and `get-cdrs2` are equivalent to `get-cars` and `get-cdrs`, respectively. We confirm this in ACL2s; including these two lemmas as `defthms` (which are automatically provable) sets up the appropriate rewrite-rules that allow ACL2s to also prove Lemma 3 and Lemma 4.

We have shown that Lemma 2a and Lemma 2 are equivalent; all that remains is to prove Lemma 2a itself, which ACL2s cannot do just yet.

Lemma 2a tries to relate `unzip-lists2` to `get-cars2` and `get-cdrs2`. However, `unzip-lists2` only sets up an accumulator for the real work-horse function `unzip-lists2-acc`, which proves to be a challenge for ACL2s to reason with. As is standard when reasoning about accumulators, we have to set up a lemma that directly relates `unzip-lists2-acc` to `get-cars2` and `get-cdrs2`, as follows:

```
Lemma 7.
(implies (and (alistp a)
              (tlp acc1)
              (tlp acc2))
         (equal (unzip-lists2-acc a acc1 acc2)
                (cons (app2 acc1 (get-cars2 a))
                      (app2 acc2 (get-cdrs2 a))))))
```

ACL2s manages to automatically prove Lemma 7 with some help from the rewrite-rules given by Lemmas 5 and 6, which allow statements of the form `(get-cars2-acc a acc)` to be written as `(app2 acc (get-cars a))`.

The rewrite rule obtained from proving Lemma 7 allows ACL2s to prove Lemma 2a under the substitutions `((acc1 nil) (acc2 nil))`, completing the final missing link of the proof. Since we have shown equivalence between `get-cars` and `get-cars2` (resp. cdrs) in Lemmas 3 and 4, we now can prove Lemma 2.

With Lemmas 1 through 4, we show that `unzip-lists` and `unzip-lists2` both reduce to `get-cars` and `get-cdrs`. From here, ACL2s can easily prove Conjecture 1 and conclude that `unzip-lists` and `unzip-lists2` are equivalent functions.

4 Introducing Another Version of unzip-lists

For further enrichment, we used a similar strategy to show that the following implementation of unzip is equivalent to the ones we already have.

```
; Accumulates caars and cdars in acc1 and acc2 respectively, but
; in reverse order
(definec unzip-lists3-acc (al :alist acc1 :tl acc2 :tl) :tl
  (cond
   ((endp al) (cons (rev2 acc1) (rev2 acc2)))
   (t (unzip-lists3-acc (cdr al)
                        (cons (car (car al)) acc1)
                        (cons (cdr (car al)) acc2)))))

; Calls the accumulator version of the function with empty
; accumulators
```

```
(definec unzip-lists3 (al :alist) :tl
  (unzip-lists3-acc al nil nil))
```

This implementation intentionally accumulates the cars and cdrs of a given alist in reverse order, but remedies this in the base case of the recursion by calling `rev2` on each of the accumulators to correct the ordering (see Appendix for the implementation of rev2). The following conjecture is our new goal.

```
Conjecture 2.
(implies (alistp a)
         (equal (unzip-lists a) (unzip-lists3 a)))
```

To adhere to the same strategy used to prove Conjecture 1, we want to show that `unzip-lists3` reduces to some form of `get-cars` and `get-cdrs`. Specifically, we will show that `unzip-lists3` reduces to `get-cars2` and `get-cdrs2`.

```
Lemma 8.
(implies (alistp a)
         (equal (unzip-lists3 a)
                (cons (get-cars2 a)
                      (get-cdrs2 a)))))
```

We can follow the same methodology as in Section 3 to define a new set of `get-cars` and `get-cdrs` functions with the same recursive structure of `unzip-lists3`, and reason about these new functions instead. The implementations of `get-cars3` and `get-cdrs3` are as follows:

```
; Accumulates all the caars of the alist in acc, but in reverse order
(definec get-cars3-acc (al :alist acc :tl) :tl
  (cond
   ((endp al) (rev2 acc))
   (t (get-cars3-acc (cdr al) (cons (car (car al)) acc)))))

; Calls get-cars3-acc with an empty accumulator
(definec get-cars3 (al :alist) :tl
  (get-cars3-acc al '()))

; Accumulates all the cdars of the alist in acc, but in reverse order
(definec get-cdrs3-acc (al :alist acc :tl) :tl
  (cond
   ((endp al) (rev2 acc))
   (t (get-cdrs3-acc (cdr al) (cons (cdr (car al)) acc)))))

; Calls get-cdrs3-acc with an empty accumulator
(definec get-cdrs3 (al :alist) :tl
  (get-cdrs3-acc al '()))
```

Proving the following sub-goals will therefore be sufficient to prove Lemma 8. If we can show Lemma 8, then Conjecture 2 trivially follows by our previous work relating `unzip-lists` to `get-cars2` and `get-cdrs2`.

```
Lemma 9.
(implies (alistp a)
         (equal (unzip-lists3 a)
                (cons (get-cars3 a)
                      (get-cdrs3 a)))))
Lemma 10.
(implies (alistp a)
         (equal (get-cars3 a) (get-cars2 a))))

Lemma 11.
(implies (alistp a)
         (equal (get-cdrs3 a) (get-cdrs2 a))))
```

We begin the proof by attempting to prove Lemmas 10 and 11. Since both versions 2 and 3 of `get-cars` and `get-cdrs` rely on accumulator functions, the first step is to set up a rewrite rule relating these accumulators (resp cdrs).

```
Lemma 12.
(implies (and (alistp a)
              (tlp acc))
         (equal (get-cars3-acc a (rev2 acc))
                (get-cars2-acc a acc))))

Lemma 13.
(implies (and (alistp a)
              (tlp acc))
         (equal (get-cdrs3-acc a (rev2 acc))
                (get-cdrs2-acc a acc)))
```

In Section 3, we set up the similar Lemmas 5 and 6 to get ACL2s to define rewrite-rules relating `get-cars` and `get-cars2`. ACL2s proved Lemmas 5 and 6 without any additional assistance, but it could not do the same for Lemmas 12 and 13. After inspecting the failed proof attempts, we realized that including a call to `rev2` in the goal "distracted" ACL2s - it did not have any information about `rev2` and `app2` outside of their definition and contract rules, and it got stuck trying to prove statements about `rev2` and `app2` inside of the larger proof structure.

To remedy this, we introduced the following lemma (inspired by one of the HW proofs) to provide ACL2s with a catch-all rewrite-rule relating nested `rev2` and `app2` calls.

```
Lemma 14.
(implies (and (tlp a)
```

```
      (tlp b))
 (equal (rev2 (app2 a b)) (app2 (rev2 b) (rev2 a))))
```

With Lemma 14 in its back pocket, ACL2s has all it needs to prove Lemmas 12 and 13. From there, it can also show Lemmas 10 and 11 by the substitution `((acc nil))` into Lemmas 12 and 13.

Thus, all that is left is to show Lemma 9, which relates `unzip-lists3` to `get-cars3` and `get-cdrs3`. As with Lemma 2a in Section 3, ACL2s can't directly show Lemma 9 on its own, so we can introduce another lemma to relate the accumulator function unzip-lists3-acc with `get-cars3` and `get-cdrs3` instead.

```
Lemma 15.
(implies (and (alistp a)
              (tlp acc1)
              (tlp acc2))
         (equal (unzip-lists3-acc a acc1 acc2)
                (cons (app2 (rev2 acc1) (get-cars3 a))
                      (app2 (rev2 acc2) (get-cdrs3 a)))))
```

ACL2s can prove Lemma 15 without any further assistance, and it is clear by the substitutions `((acc1 nil) (acc2 nil))` we can now prove Lemma 9 to arrive at the conclusion that `unzip-lists3` reduces to `get-cars3` and `get-cdrs3`.

Since we have shown Lemmas 9, 10, and 11, ACL2s now has everything it needs to prove Lemma 8. By our work in proving Conjecture 1, we know that `unzip-lists2` also reduces to `get-cars2` and `get-cdrs2` and thus can prove Conjecture 2, showing that all three implementations of `unzip-lists` are equivalent functions.

5 Personal Progress

Initially, we attempted to prove the equivalence of `unzip-lists` and `unzip-lists2` outright, without resorting to the reduction to `get-cars` and `get-cdrs`. We immediately recognized that we had to reason about `unzip-lists2`'s accumulator rather than `unzip-lists2` itself - a pretty standard approach when reasoning about functions with accumulators and a strategy we had seen on other assignments.

Naturally, one of the first things we attempted was to get ACL2s to relate the `unzip-lists` function directly to `unzip-lists2-acc`, in a similar fashion to how we related `get-cars` to `get-cars2` in Lemmas 5 and 6. We mostly tried to push the lemma through by brute force - when the proof inevitably got stuck on some subgoal, we would make that subgoal into another lemma, try to push the top-level lemma through again, and repeat the cycle. After a few hours of this, it was clear we weren't making much progress - we knew we had to relate `unzip-lists2-acc` to `unzip-lists` somehow, but we didn't have a set path to get there.

We hypothesized that this initial failure stemmed from not giving ACL2s the right tools to reason about the recursion in `unzip-lists`. Specifically, we weren't sure if ACL2s could recognize that `(car (unzip-lists al))` returned all the cars in the alist `al` (resp. cdrs). This fact is especially important to the definition of `unzip-lists`, which relies on breaking apart the recursive call to `(unzip-lists (cdr al))` using the `car` and `cdr` destructors to access the appropriate sub-lists.

At this point, we recognized that we could think of unzipping an alist as two distinct operations - collecting all of the cars in the alist, and collecting all of the cdrs - and reason about each operation separately. Using this realization, we changed our proof structure, the idea being that if we could show that any implementation of `unzip-lists` reduces to collecting the cars and collecting the cdrs, then the `unzip-lists` implementations must be equivalent.

As a proof of concept, we defined `get-cars` and `get-cdrs` recursively (i.e. without accumulators, so the recursive structure was similar to `unzip-lists`), and asked ACL2s to prove the following:

```
(implies (alistp a)
     (equal (unzip-lists a)
            (cons (get-cars a)
                  (get-cdrs a)))))
```

In our overall proof, this statement became Lemma 1, and, as we mentioned in Section 3, ACL2s proved it without any assistance. We also tried to prove Lemma 2, which failed, but we defined new versions of `get-cars` and `get-cdrs` which made reasoning about `unzip-lists2` easier. The rest of the proof followed as described in Section 3; after we redefined our overall proof strategy to show how the `unzip-lists` versions we give all reduce to `get-cars` and `get-cdrs`, the mechanics of the proof quickly fell into place.

Overall, we had one key breakthrough on the journey to successfully proving Conjecture 1. We realized that `unzip-lists` could be redefined in terms of `get-cars` and `get-cdrs`, which led to a fundamental shift in our overall proof strategy. We used this strategy to prove Conjecture 2 as well, which involved a more complicated implementation of `unzip-lists`.

6 Conclusion

In this paper, we introduced three distinct functions that can unzip the pairs of a given alist. We proved equivalence between all three implementations by expressing the fundamental behavior of unzip as the pairing of a list containing all the cars of the alist with another containing all the cdrs.

For each unzip version, we wrote two functions, `get-cars` and `get-cdrs`, to extract these car and cdr lists from the original alist, and we proved that all versions of `get-cars` and `get-cdrs` were equivalent. Then, by proving that each version of unzip could be reduced to the pairing of its corresponding versions

of `get-cars` and `get-cdrs`, we could prove that the two versions of unzip were equivalent:

```
Conjecture 1.
(implies (alistp a)
         (equal (unzip-lists a) (unzip-lists2 a)))
```

We applied this strategy to show that another implementation of unzip, using `rev2`, was equivalent to the two original implementations. In other words, we also showed the following alternate conjecture:

```
Conjecture 2.
(implies (alistp a)
         (equal (unzip-lists a) (unzip-lists3 a)))
```

Many of the definitions and lemmas necessary to show Conjecture 2 had parallels to the definitions and lemmas necessary to show Conjecture 1.

Since both of our proofs are based on abstracting the general unzip operation as the pairing of two smaller operations, it also provides an intuitive explanation for why these unzip versions are equivalent. Our proof does more than just mechanically show that the different functions will always return the same result; it shows that the functions are the same because they perform the same fundamental tasks involved in unzipping an alist.

7 Appendix

7.1 Conjectures

```
Conjecture 1.
(thm (implies (alistp a)
              (equal (unzip-lists a) (unzip-lists2 a))))

Conjecture 2.
(thm (implies (alistp a)
              (equal (unzip-lists3 a) (unzip-lists a))))
```

7.2 Function Definitions

```
Definition app2
(definec app2 (x :tl y :tl) :tl
  (if (endp x)
    y
    (cons (first x) (app2 (rest x) y))))

Definition rev2
(definec rev2 (x :tl) :tl
  (if (endp x)
    x
    (app2 (rev2 (cdr x)) (list (car x)))))

Definition unzip-lists
(definec unzip-lists (al :alist) :tl
  (cond
   ((endp al) (cons nil nil))
   (t (let ((res (unzip-lists (cdr al))))
        (cons (cons (car (car al)) (car res))
              (cons (cdr (car al)) (cdr res)))))))

Definition unzip-lists2-acc
(definec unzip-lists2-acc (al :alist acc1 :tl acc2 :tl) :tl
  (cond
   ((endp al) (cons acc1 acc2))
   (t (unzip-lists2-acc (cdr al)
                        (app2 acc1 (list (car (car al))))
                        (app2 acc2 (list (cdr (car al))))))))

Definition unzip-lists2
(definec unzip-lists2 (al :alist) :tl
  (unzip-lists2-acc al '() '()))
```

```
Definition unzip-lists3-acc
(definec unzip-lists3-acc (al :alist acc1 :tl acc2 :tl) :tl
  (cond
   ((endp al) (cons (rev2 acc1) (rev2 acc2)))
   (t (unzip-lists3-acc (cdr al)
                        (cons (car (car al)) acc1)
                        (cons (cdr (car al)) acc2)))))

Definition unzip-lists3
(definec unzip-lists3 (al :alist) :tl
  (unzip-lists3-acc al nil nil))

Definition get-cars
(definec get-cars (a :alist) :tl
  (cond
   ((endp a) nil)
   (t (cons (car (car a)) (get-cars (cdr a))))))

Definition get-cars2-acc
(definec get-cars2-acc (a :alist acc :tl) :tl
  (cond
   ((endp a) acc)
   (t (get-cars2-acc (cdr a) (app2 acc (list (car (car a))))))))

Definition get-cars2
(definec get-cars2 (a :alist) :tl
  (get-cars2-acc a '()))

Definition get-cdrs
(definec get-cdrs (a :alist) :tl
  (cond
   ((endp a) nil)
   (t (cons (cdr (car a)) (get-cdrs (cdr a))))))

Definition get-cdrs2-acc
(definec get-cdrs2-acc (a :alist acc :tl) :tl
  (cond
   ((endp a) acc)
   (t (get-cdrs2-acc (cdr a) (app2 acc (list (cdr (car a))))))))

Definition get-cdrs2
(definec get-cdrs2 (a :alist) :tl
  (get-cdrs2-acc a '()))

Definition get-cars3-acc
(definec get-cars3-acc (al :alist acc :tl) :tl
```

```
  (cond
   ((endp al) (rev2 acc))
   (t (get-cars3-acc (cdr al) (cons (car (car al)) acc)))))

Definition get-cars3
(definec get-cars3 (al :alist) :tl
  (get-cars3-acc al '()))

Definition get-cdrs3-acc
(definec get-cdrs3-acc (al :alist acc :tl) :tl
  (cond
   ((endp al) (rev2 acc))
   (t (get-cdrs3-acc (cdr al) (cons (cdr (car al)) acc)))))

Definition get-cdrs3
(definec get-cdrs3 (al :alist) :tl
  (get-cdrs3-acc al '()))
```

7.3 Lemmata

```
Lemma 1.
(implies (alistp a)
         (equal (unzip-lists a)
                (cons (get-cars a)
                      (get-cdrs a)))))
Lemma 2.
(implies (alistp a)
         (equal (unzip-lists2 a)
                (cons (get-cars a)
                      (get-cdrs a)))))

Lemma 2a.
(implies (alistp a)
         (equal (unzip-lists2 a)
                (cons (get-cars2 a)
                      (get-cdrs2 a)))))
Lemma 3.
(implies (alistp a)
         (equal (get-cars a) (get-cars2 a)))

Lemma 4.
(implies (alistp a)
         (equal (get-cdrs a) (get-cdrs2 a)))

Lemma 5.
(implies (and (alistp a)
```

```
             (tlp acc))
         (equal (get-cars2-acc a acc) (app2 acc (get-cars a)))))

Lemma 6.
(implies (and (alistp a)
              (tlp acc))
         (equal (get-cdrs2-acc a acc) (app2 acc (get-cdrs a)))))

Lemma 7.
(implies (and (alistp a)
              (tlp acc1)
              (tlp acc2))
         (equal (unzip-lists2-acc a acc1 acc2)
                (cons (app2 acc1 (get-cars2 a))
                      (app2 acc2 (get-cdrs2 a))))))

Lemma 8.
(thm (implies (alistp a)
              (equal (unzip-lists3 a)
                     (cons (get-cars2 a)
                           (get-cdrs2 a)))))

Lemma 9.
(implies (alistp a)
         (equal (unzip-lists3 a)
                (cons (get-cars3 a)
                      (get-cdrs3 a)))))

Lemma 10.
(implies (alistp a)
         (equal (get-cars3 a) (get-cars2 a))))

Lemma 11.
(implies (alistp a)
         (equal (get-cdrs3 a) (get-cdrs2 a))))


Lemma 12.
(implies (and (alistp a)
              (tlp acc))
         (equal (get-cars3-acc a (rev2 acc))
                (get-cars2-acc a acc))))

Lemma 13.
(implies (and (alistp a)
              (tlp acc))
```

```
         (equal (get-cdrs3-acc a (rev2 acc))
                (get-cdrs2-acc a acc)))

Lemma 14.
(implies (and (tlp a)
              (tlp b))
           (equal (rev2 (app2 a b)) (app2 (rev2 b) (rev2 a))))

Lemma 15.
(implies (and (alistp a)
                (tlp acc1)
                (tlp acc2))
           (equal (unzip-lists3-acc a acc1 acc2)
                  (cons (app2 (rev2 acc1) (get-cars3 a))
                        (app2 (rev2 acc2) (get-cdrs3 a)))))
```