

# Proving properties of lazy functional programs with SPARKLE

## Answers of Assignments: FIRST DRAFT (summer version)

Maarten de Mol, Marko van Eekelen and Rinus Plasmeijer

{maartenm, marko, rinus}@cs.ru.nl  
Institute for Computing and Information Sciences  
Radboud University Nijmegen, The Netherlands

### Assignments / Answers

**Assignment 1:** (*loading a program into SPARKLE automatically*)

- (a) Open the CLEAN-project `primes.prj` in the `Examples\CEFP` folder.
- (b) Examine the code of the main module (`primes.icl`) and attempt to predict the behavior of the program. Then, compile and run the program.
- (c) Find the `Theorem Prover Project` option and use it to launch SPARKLE.

**Answer 1:** (*loading a program into SPARKLE automatically*)

- (b) Program computes the lazy, infinite list all all prime numbers. The first 100 elements of this list are displayed.

**Assignment 2:** (*browsing through the program structure*)

- (a) Find the window that displays the list of modules that are currently loaded. In this list, find the `primes` module and open it.
- (b) The opened window actually filters *all* available definitions with the formula ‘functions from the `primes` module’. Change the filter to find all functions in `StdList` and `StdFunc` that begin with the letter ‘s’.

**Answer 2:** (*browsing through the program structure*)

- (b) 10 in total: `scan`, `seq`, `seqList`, `shift`, `span`, `splitAt`, `stolacc`, `subscript_error`, `sum`, `sum2`

**Assignment 3:** (*browsing through the program components*)

- (a) Open the definition of the function `isPrime` in the `primes` module.
- (b) Follow the internal link to the `canBeDividedByAny` function.
- (c) Follow the internal link to the predefined `rem` function.

**Answer 3:** (*browsing through the program components*)

- (b) In module `primes`.
- (c) In module `StdInt`; defined by means of ABC-code.

**Assignment 4:** *(effect of the optional display options)*

- (a) Open the function definitions `isPrime` and `canBeDividedByAny` from the `primes` module and `span` from the `StdList` module.
- (b) Toggle the display options `Pattern Matching` and `Case/Let vs #/!`. The 'real' CORE-CLEAN program is displayed when the options are toggled off.
- (c) There is one difference between the internal version of `isPrime` and the CLEAN version that cannot be hidden. What is this difference?

**Answer 4:** *(effect of the optional display options)*

- (b) Before: pattern-matching and `|`. After: case distinctions.
- (c) The CLEAN-version contains a dot-dot-expression, namely `[2..n-1]`.  
The CORE-version contains a function call, namely `_from_to 2 (n - 1)`.

**Assignment 5:** (*partial undefinedness in practice*)

- (a) Open the `undefined` project with the IDE. Run and compile it.
- (b) Replace the body of `my_undefined` with another computation that also terminates erroneously.
- (c) Cycle through the available `Start` bodies and examine the run-time results.

**Answer 5:** (*partial undefinedness in practice*)

- (a) Produces 'Error!'. Semantically, produces  $\perp$ .
- (b) For example: `hd [], 5/0, let x = x in x`.
- (c) Second program does not compile; replace `12` by `[12]`.  
Correct second program produces the result `[12]`.  
Third program produces `[1,2,3:⊥` (partially defined result).

**Assignment 6:** (*undefinedness cannot be detected*)

- (a) What famous (unsolvable) problem would be solved if it was possible to detect undefinedness within a `CLEAN` program?

**Answer 6:** (*undefinedness cannot be detected*)

- (a) The halting problem. (insert a program, produce `True` when it is defined and `False` when it is undefined)

**Assignment 7:** (*validity of the example properties*)

- (a) Of the six example properties, only five are true, and one is in fact false (it needs an additional precondition). Which one is false?  
(*Hint: lists may be infinite in CLEAN*)
- (b) What happens to the sixth property if either  $i$  or  $j$  is undefined?

**Answer 7:** (*validity of the example properties*)

- (a) True: 1,2,3,5,6. False: 4.  
Reason: `reverse (reverse ones) = ⊥ ≠ ones`.
- (b) Then the condition  $i > j$ , which is an abbreviation for  $i > j = \text{True}$ , becomes false, because  $i > j = \perp$  and  $\perp \neq \text{True}$ . Because the condition is false, the proposition as a whole becomes true.

**Assignment 8:** (specify the example properties (1))

- (a) Use **New Theorem** to manually enter all six example properties.  
(Hint: in case of failure, attempt to add brackets)

**Answer 8:** (specify the example properties (1))

- (a) [P] [Q] (P /\ Q) <-> (Q /\ P)  
17 > 12 = True  
[f] [xs] [ys] map f (xs ++ ys) = map f xs ++ map f ys  
[xs] reverse (reverse xs) = xs  
[n] [xs] ~(n=\_|\_) -> ~(xs=\_|\_) -> length (take n xs) = n  
[i] [j] (i>j=True /\ j>0=True)  
-> (primes !! i) > (primes !! j) = True

**Assignment 9:** (specify properties with overloading)

The manual specification of types is essential when making use of overloading:

- (a) Without explicit types, attempt to specify  $\forall_x \forall_y. x + y = y + x$ .  
(b) Use explicit types ( $x :: \text{Int}, y :: \text{Int}$ ) to help SPARKLE solve the overloading in  $\forall_x \forall_y. x + y = y + x$ .

**Answer 9:** (specify properties with overloading)

- (a) Produces 'Internal Error: no suitable instance found.'  
Reason: SPARKLE does not know which instance of + has to be used.  
(b) Type: [x::Int] [y::Int] x + y = y + x.  
Alternative solution: [x] [y] x +\_int y = y +\_int x.  
This forces SPARKLE to choose the specialization of + on Int's.

**Assignment 10:** (specify the example properties (2))

- (a) Specify the example properties again, using the features described above.  
Do *not* quit SPARKLE afterwards.

**Answer 10:** (specify the example properties (2))

- (a) (P /\ Q) <-> (Q /\ P)  
17 > 12  
map f (xs ++ ys) = map f xs ++ map f ys  
reverse (reverse xs) = xs  
~(n=\_|\_) -> ~(xs=\_|\_) -> length (take n xs) = n  
(i>j /\ j>0) -> ((primes !! i) > (primes !! j))  
(last example: requires additional brackets due to parsing problems)

**Assignment 11:** *(save properties into sections)*

- (a) Create a new section with the name `temp`.
- (b) Open both the `main` section and the `temp` section.
- (c) Move the example properties from the `main` section into the `temp` section.
- (d) Save the `temp` section and quit SPARKLE.

**Answer 11:** *(save properties into sections)*

- (a) On the Theorem Info window, use the 'label' icon.
- (b) On the Section Center Window, use the 'disc' icon.

**Assignment 12:** *(load sections into memory)*

- (a) Start SPARKLE manually (directly and not from within the IDE).
- (b) Attempt to load the predefined section `lists`.
- (c) Open the `primes` project from within SPARKLE.
- (d) Load the predefined section `lists`.
- (e) Load the section `temp` of the previous assignment.

**Answer 12:** *(load sections into memory)*

- (b) Gives 'dependency check failures'. Best to abort completely.  
If not aborted, then at the end remove the list module from memory, using the X-icon in the Section Center Window.

**Assignment 13:** *(examples of (in)equality)*

- (a) Are `'ones'` and `'let x = [1:x] in x'` equal? If so, argue; if not, give the program that distinguishes between them.
- (b) Same question for `'ones'` and `'ones ++ ones'`.
- (c) Same question for `'ones'` and `'[2] ++ ones'`.
- (d) Same question for `'ones'` and `'ones ++ [2]'`.
- (e) Same question for `'⊥'` and `'[1:⊥]'`.
- (f) Same question for `'⊥'` and `'λx.⊥'`.  
*(Hint: make use of explicit strictness)*
- (g) Same question for `'⊥'` and `'let x = x in x'`.  
*(Hint: only basic values and constructors are meaningful output)*

**Answer 13:** *(examples of (in)equality)*

- (a) Yes, both produce the infinite list of ones.
- (b) Yes, both produce the infinite list of ones. Note that `CLEAN` will never reach the part behind the `++`; it is therefore not used to compute the end result and is of no consequence to the meaning of the expression as a whole.
- (c) No, the program `'hd ●'` distinguishes between them.
- (d) Yes, both produce the infinite list of ones. (see (b))
- (e) No, the program `'hd ●'` distinguishes between them.
- (f) No, the program `'F ●'` (`F :: !a -> Bool; F x = True`) distinguishes.
- (g) Yes, both produce `⊥`.

**Assignment 14:** (*backwards proving*)

(a) Why is SPARKLE's reasoning style sometimes also called *backwards proving*?

**Answer 14:** (*backwards proving*)

(a) Because you start at the conclusion(end,bottom) and end with the assumptions(beginning,top).

**Assignment 15:** (*decompose the property*)

The proof states in Fig. 3 and Fig. 4 are taken from an actual proof.

(a) Which property corresponds to the current goal in Fig. 3?

(b) Which property was the starting point of the proof?

**Answer 15:** (*decompose the property*)

(a)  $\forall n. \neg(n = \perp) \rightarrow (\mathbf{take\ } n\ \perp) ++ (\mathbf{drop\ } n\ \perp) = \perp$

(b)  $\forall_{xs} \forall n. \neg(n = \perp) \rightarrow (\mathbf{take\ } n\ xs) ++ (\mathbf{drop\ } n\ xs) = xs$

**Assignment 16:** (*specification of a property of map*)

(a) Open SPARKLE from scratch, then load the standard environment (Ctrl-E).

(b) Create a new section with the name `map_section`.

(c) In `map_section`, create a new theorem named `map_property`, stating:

$$\forall f \forall_{xs} \forall_{ys}. \mathbf{map\ } f\ (xs ++ ys) = \mathbf{map\ } f\ xs ++ \mathbf{map\ } f\ ys$$

(d) Open the proof window (Ctrl-P) that corresponds to the created theorem.

**Assignment 17:** (*proving the map property with the hint mechanism*)

(a) Open the Tactic Suggestions Window (Ctrl-H) and set the threshold to 1.

(b) Set the threshold back to 101. Why is this necessary prior to (c)?

(c) Enter `◀Restart.▶` at the command-line interface.

(*From now on, `◀cmd.▶` will be used to denote textual input to the command-line. For reasons of parsing, these commands have to end with a closing '.', otherwise SPARKLE will not be able to recognize them.*)

(d) Redo the proof by applying suggestions manually with the hot-keys.

**Answer 17:** (*proving the map property with the hint mechanism*)

(a) The proof is completed automatically.

(b) Otherwise, immediately after entering the `◀Restart.▶` command, the proof is performed automatically again.

(d) Repeatedly pressing F1 will complete the same proof again.

**Assignment 18:** (*browsing through the proof*)

- (a) Open the Theorem Info Window of the completed proof.
- (b) Click ‘browse’ after the first tactic and then browse through the proof using the ‘previous’ and ‘next’ buttons.
- (c) Undo the first application of `Reflexive` only.
- (d) Click on the brown star to return to the Proof Window.
- (e) Use a different tactic to prove the goal.

**Answer 18:** (*proving the map property with the hint mechanism*)

- (d) `◀Definedness.▶` does the trick as well.

**Assignment 19:** (*injectivity and strictness*)

- (a) Why does injectivity not hold for strict data constructors?

**Answer 19:** (*injectivity and strictness*)

- (a) Suppose `SCons :: !a !(SList a) -> (List a)` is a constructor. Then, due to the strictness, `SCons ⊥ [] = ⊥ = SCons 5 ⊥` holds. However, the implied injective property `⊥ = 5 ∧ [] = ⊥` does not hold.

**Assignment 20:** (*manual proof of the map property*)

- (a) Prove the `map` property again, using the tactic dialogs only.
- (b) Prove the `map` property again, using the command interface only.  
(*Hint: `◀Reduce.▶` abbreviates `◀Reduce NF All.▶`, and `◀Intros.▶` is a variant of introduction that comes up with suitable names on its own)*
- (c) The automatic proof consists of the application of 13 tactics. It is possible to prove the property in less steps (our shortest proof consists of 8 steps). Try to shorten the proof yourself.

**Answer 20:** (*manual proof of the map property*)

- (b) 1: `◀Induction xs.▶`;
  - 1.1: `◀Intros.▶`; `◀Reduce.▶`; `◀Reflexive.▶`;
  - 1.2: `◀Intros.▶`; `◀Reduce.▶`; `◀Reflexive.▶`;
  - 1.3: `◀Intros.▶`; `◀Reduce.▶`; `◀Injective.▶`; `◀Split.▶`;
  - 1.3.1: `◀Reflexive.▶`;
  - 1.3.2: `◀Apply IH.▶`.
- (c) 1: `◀Induction xs.▶`;
  - 1.1: `◀Reduce.▶`; `◀Reflexive.▶`;
  - 1.2: `◀Reduce.▶`; `◀Reflexive.▶`;
  - 1.3: `◀Intros.▶`; `◀Reduce.▶`; `◀Rewrite IH.▶`; `◀Reflexive.▶`.(Note: 9 steps, incorrectly stated as 8 in the assignment)

**Assignment 21:** *(more small proofs)*

- (a) Manually prove  $\forall_{xs} \forall_{ys} \forall_{zs}. xs ++ (ys ++ zs) = (xs ++ ys) ++ zs$ .
- (b) Manually prove  $\forall_{xs}. \neg(xs = \perp) \rightarrow \neg(xs = []) \rightarrow [\text{hd } xs : \text{tl } xs] = xs$ .
- (c) Manually prove  $\forall_n \forall_{xs}. \neg(n = \perp) \rightarrow \text{take } n \text{ } xs ++ \text{drop } n \text{ } xs = xs$ .
- (d) Manually prove  $\forall_{xs}. \text{finite } xs \rightarrow \text{finite } (\text{reverse } xs)$ .
- (e) Manually prove  $\forall_P \forall_Q. (\neg P \leftrightarrow Q) \leftrightarrow (P \leftrightarrow \neg Q)$ .

**Answer 21:** *(more small proofs)*

- (a) See proof file: 21a\_associative\_++.
- (b) See proof file: 21b\_hd\_tl.
- (c) See proof file: 21c\_take\_drop.
- (d) See proof file: 21d\_finite\_++ and 21d\_finite\_reverse.
- (e) See proof file: 21e\_iff.

**Assignment 22:** *(Explore the effect of sharing during reduction in proofs)*

- (a) Consider in SPARKLE the following trivial theorem `let n = 1 + 2 + 3 in n + n = 12`. Prove it using `◀Reduce NF All.▶`, followed by `◀Reflexive.▶`.
- (b) Undo the proof using Ctrl-z and now prove it replacing normal form reduction by reduction for which the number of steps is given: `◀Reduce 4.▶`.
- (c) Undo the proof again and try performing it using stepwise reduction: `◀Reduce 1.▶`. This will take 6 steps. Look carefully at the intermediate goals and explain why more steps are needed.

**Answer 22:** *(Explore the effect of sharing during reduction in proofs)*

- (a) `(let n = 1+2+3 in n+n) = 12;`
- (b) Ctrl-Z does not work; use `◀Restart.▶` instead.
- (c) Sharing has been broken after the first reduction step, whereas in the previous cases it was only broken after the last reduction step.

**Assignment 23:** *(Case distinction on lazy lists)*

- (a) Define the following theorem:  $\forall_{xs \in [a]} [\text{length } xs > 0 \Rightarrow (xs = [])]$ .
- (b) First apply the tactic `◀Introduce.▶` and then use case distinction on `xs`. Study carefully the goals that are created and finish the proof.

**Answer 23:** *(Case distinction on lazy lists)*

- (a) Should read: `length xs > 0 -> ~(xs = [])` (the `~` was left out)
- (b) See proof file: 23a\_length\_nonempty.

**Assignment 24:** *(Extensionality)*

- (a) Prove using extensionality that `sum o (map (const 1)) = length` holds.

**Answer 24:** *(Extensionality)*

- (a) See proof file: 24a\_sum\_map\_const\_1.

**Assignment 25:** (*Induction on lazy lists*)

For each of the theorems below: prove it or show that it is not admissible.

Apart from `finite`, which is defined in `StdSparkle`, all the functions used in this assignment are standard functions available in the standard `CLEAN` library `StdEnv`.

- (a)  $\forall xs \in [a] \ [\text{finite } xs \Rightarrow \text{take } (\text{length } xs) \ xs = xs]$ .
- (b)  $\forall xs \in [a] \ [xs = \text{ones} \Rightarrow \text{hd } xs = 1]$ .
- (c)  $\forall xs \in [a] \ [xs = \text{ones} \Rightarrow \text{finite } xs]$ .
- (d)  $\forall xs \in [a] \ \forall f \in a \rightarrow b \ \forall p \in b \rightarrow \text{Bool} \ [\text{all } p \ (\text{map } f \ xs) \Leftrightarrow \text{all } (p \circ f) \ xs]$ .
- (e)  $\forall xs \in [a] \ \forall ys \in [a] \ [xs = ys \Rightarrow xs == ys]$ .

**Answer 25:** (*Induction on lazy lists*)

- (a) Admissible. (`finite xs = True` is on negative position, but has finite type)  
See also proof file: `25a_length_positive` and `25a_take_length_identity`.
- (b) Not admissible. (`xs = ones` is on negative position, and has non-finite type)  
Note: this statement holds and can be proved without induction.  
See also proof file: `25b_hd_ones`.
- (c) Not admissible. (`xs = ones` is on negative position, and has non-finite type)  
See also proof file: `25c_finite_ones`.
- (d) Admissible. (all expressions have finite type)  
See also proof file: `25d_all_map`.
- (e) Not admissible. (`xs = ys` is on negative position, and has non-finite type)  
(*Note:* only makes sense when the type of `xs` has been specified explicitly, i.e. by means of `[x :: [Int]]`.)  
See also proof file: `25e_==`.

**Assignment 26:** (*proving in the context of strictness*)

- (a) Define the example above with the strictness information in a `CLEAN` program and try to perform with `SPARKLE` a proof the property  $\forall x [\text{f } x = 5]$ .  
Of course this fails. Study the situation where you got stuck.
- (b) Prove with `SPARKLE`  $\forall x [x \neq \perp \rightarrow \text{f } x = 5]$ .

**Answer 26:** (*proving in the context of strictness*)

- (a) `SPARKLE` does not allow the expression `f x` to be reduced.
- (b) 1: `◀Intros.▶`; `◀Reduce.▶`; `◀Reflexive.▶`.

**Assignment 27:** (*proofs using eval*)

Use the function `eval` from `StdSparkle` to prove the following properties.

- (a)  $\forall n[\text{eval } n \rightarrow n < n = \text{False}]$ .
- (b)  $\forall n, xs[\text{eval } n \rightarrow \text{take } n \text{ } xs ++ \text{drop } n \text{ } xs = xs]$
- (c)  $\forall p, xs[\text{eval } (\text{map } p \text{ } xs) \rightarrow \text{takeWhile } p \text{ } xs ++ \text{dropWhile } p \text{ } xs = xs]$ .
- (d)  $\forall x, p, xs[\text{eval } x \rightarrow \text{eval } xs \rightarrow \text{eval } (\text{map } p \text{ } xs) \rightarrow$   
 $\text{isMember } x (\text{filter } p \text{ } xs) = \text{isMember } x \text{ } xs \ \&\& \ p \ x] .$

**Answer 27:** (*proofs using eval*)

- (a) See proof file: `27a_n_<_n`.
- (b) See proof file: `21c_take_drop`.
- (c) See proof file: `27c_takeWhile_dropWhile`.
- (d) See proof file: `27d_eval_isMember_int`, `27d_split_eval_list_bool`,  
`27d_split_eval_list_int`, `27d_isMember_filter`.

**Assignment 28:** (*advanced use of eval*)

- (a) Prove  $\forall f, g, xs[\forall x \forall xs[\text{isMember } x \text{ } xs \rightarrow \text{eval}(g \ x)] \rightarrow \text{map } (f \circ g) \text{ } xs =$   
 $\text{map } f (\text{map } g \text{ } xs)]$

**Answer 28:** (*advanced use of eval*)

- (a) See proof file: `28_isMember_map_o`.  
(Note: you need to explicitly give `g` type `Int → Int`)  
(Erratum add an explicit  $\forall xs$  in the condition)

**Assignment 29:** (*use of evalSpine*)

- (a) Prove  $\forall xs[\text{eval } (\text{length } xs) \rightarrow \text{evalSpine } xs]$ .
- (b) Prove  $\forall xs[\text{evalSpine } xs \rightarrow \text{evalSpine } (\text{reverse } xs)]$ .
- (c) The `reverse` example of page 26 can now be reformulated as:  
 $\forall xs[\text{evalSpine } xs \rightarrow \text{reverse } (\text{reverse } xs) = xs]$ . Prove it.

**Answer 29:** (*use of evalSpine*)

- (a) See proof file `29a_length_evalSpine`.  
(Note 1: specialize to `Ints`.)  
(Note 2: `finite` was used instead of `evalSpine`)
- (b) See proof file: `29b_finite_++`.  
(Note: `finite` was used instead of `evalSpine`)
- (c) See proof file: `29c_reverse_++` and `29c_reverse_reverse`. (Note: `finite`  
was used instead of `evalSpine`)

**Assignment 30:** (*properties of eval*)

(a) Prove  $\forall x[\text{eval } x \rightarrow x \neq \perp]$ .

(b) Prove  $\forall x[\text{eval } x \neq \text{False}]$ .

**Answer 30:** (*properties of eval*)

(a) See proof file 30a\_eval.

(Note: specialize to Ints.)

(b) See proof file: 30b\_eval\_False.

(Note: specialize to Ints.)

## 1 SPARKLE proofs

THEOREM:

20c\_map\_++

DEPENDS:

9 3

PROOF:

Induction xs.

1. Reduce NF All ( ).

Definedness.

2. Reduce NF All ( ).

Reflexive.

3. Introduce x xs IH f ys.

Reduce NF All ( ).

Rewrite -> All IH.

Reflexive.

THEOREM:

21a\_associative\_++

DEPENDS:

3

PROOF:

Induction xs.

1. Reduce NF All ( ).

Reflexive.

2. Reduce NF All ( ).

Reflexive.

3. Introduce x xs IH ys zs.

Reduce NF All ( ).

Rewrite -> All IH.

Reflexive.

THEOREM:

21b\_hd\_tl

DEPENDS:

25 35

PROOF:  
 Introduce xs H1 H2.  
 Cases xs.  
 1. Definedness.  
 2. Contradiction H2.  
 Reflexive.  
 3. Reduce NF All ( ).  
 Reflexive.

THEOREM:  
 21c\_take\_drop

DEPENDS:  
 3 19 20

PROOF:  
 Induction xs.  
 1. Definedness.  
 2. Reduce NF All ( ).  
 Reflexive.  
 3. Introduce x xs IH n H1.  
 Reduce NF All ( ).  
 SplitCase 1.  
 1. Definedness.  
 2. Reduce NF All ( ).  
 Rewrite -> All IH.  
 1. Reflexive.  
 2. Definedness.  
 3. Reduce NF All ( ).  
 Reflexive.

THEOREM:  
 21d\_finite\_++

DEPENDS:  
 0 3

PROOF:  
 Induction xs.  
 1. Definedness.  
 2. Reduce NF All ( ).  
 Introduce ys H1 H2.  
 Exact H2.  
 3. Reduce NF All ( ).  
 Introduce x xs IH ys H1 H2.  
 Apply IH.  
 Split Deep.  
 1. Exact H1.  
 2. Exact H2.

THEOREM:  
 21d\_finite\_reverse

DEPENDS:  
 0 1

PROOF:

Induction xs.  
1. Definedness.  
2. Reduce NF All ( ).  
Reflexive.  
3. Reduce NF All ( ).  
Introduce x xs IH H1.  
Apply "21d\_finite\_++".  
Split Shallow.  
1. Apply IH.  
Exact H1.  
2. Reduce NF All ( ).  
Reflexive.

THEOREM:

21e\_iff

DEPENDS:

PROOF:

Introduce P Q.  
SplitIff.  
1. Introduce H1.  
Rewrite <- All H1.  
SplitIff.  
1. Introduce H2.  
Contradiction.  
Absurd H3 H2.  
2. Introduce H2.  
Contradiction.  
Absurd H3 H2.  
2. Introduce H1.  
Rewrite -> All H1.  
SplitIff.  
1. Introduce H2.  
Contradiction.  
Absurd H3 H2.  
2. Introduce H2.  
Contradiction.  
Absurd H3 H2.

THEOREM:

23a\_length\_nonempty

DEPENDS:

32 33 34 21 4

PROOF:

Introduce xs H1.  
Cases xs.  
1. Definedness.  
2. ReduceH NF All in H1 ( ).  
AbsurdEqualityH H1.

3. Contradiction.  
AbsurdEqualityH H2.

THEOREM:

24a\_sum\_map\_const\_1

DEPENDS:

10 27 28 26 29 30 9 31 4

PROOF:

Extensionality xs.

Reduce NF All ( ).

Induction xs.

1. Reduce NF All ( ).

Definedness.

2. Reduce NF All ( ).

Reflexive.

3. Introduce x xs IH.

Reduce NF All ( ).

IntArith All.

Rewrite -> All IH.

Reflexive.

THEOREM:

25a\_length\_positive

DEPENDS:

26 4 0 21

PROOF:

Induction xs.

1. Definedness.

2. Reduce NF All ( ).

Reflexive.

3. Reduce NF All ( ).

IntArith All.

Introduce x xs IH H1.

Apply "transitive\_<".

WitnessE (@ 26 +\_int (@ 4 length\_list xs) (INT 1)).

Split Shallow.

1. Apply IH.

Exact H1.

2. Rewrite -> All ("subtract\_from\_<" E (INT 1)).

1. IntArith All.

Apply "x\_<\_succ\_x".

Contradiction.

Rewrite -> All H2 in IH.

Apply IH to H1.

Definedness.

2. Definedness.

THEOREM:

25a\_take\_length\_identity

DEPENDS:

```

21 26 4 0 19
PROOF:
  Induction xs.
  1. Definedness.
  2. Reduce NF All ( ).
     Reflexive.
  3. Reduce NF All ( ).
     IntArith All.
     Introduce x xs IH H1.
     Assume (= (@ 21 <_int (INT 0) (@ 26 +_int
      (@ 4 length_list xs) (INT 1))) (BOOL True)).
  1. Rewrite -> All H2.
     Reduce NF All ( ).
     Rewrite -> All IH.
     1. Reflexive.
     2. Exact H1.
  2. Apply "25a_length_positive".
     Exact H1.

```

```

THEOREM:
  25b_hd_ones

```

```

DEPENDS:
  24 25

```

```

PROOF:
  Introduce xs H1.
  Rewrite -> All H1.
  Reduce NF All ( ).
  Reflexive.

```

```

THEOREM:
  25c_finite_ones

```

```

DEPENDS:
  24 0

```

```

PROOF:
  *

```

```

THEOREM:
  25d_all_map

```

```

DEPENDS:
  23 9 10

```

```

PROOF:
  Induction xs.
  1. Reduce NF All ( ).
     Reflexive.
  2. Reduce NF All ( ).
     Reflexive.
  3. Introduce x xs IH p f.
     Reduce NF All ( ).
     Rewrite -> All IH.
     Reflexive.

```

THEOREM:  
25e\_=\_=  
DEPENDS:  
22 7 8  
PROOF:  
\*

THEOREM:  
27a\_n<\_n  
DEPENDS:  
2 21  
PROOF:  
IntCompare.

THEOREM:  
27b\_take\_drop  
DEPENDS:  
18 2 3 19 20  
PROOF:  
Induction xs.  
1. Definedness.  
2. Reduce NF All ( ).  
Reflexive.  
3. Introduce x xs IH n H1.  
Reduce NF All ( ).  
SplitCase 1.  
1. Definedness.  
2. Reduce NF All ( ).  
Rewrite -> All IH.  
1. Reflexive.  
2. Assume ~(= (@ 18 -\_int n (INT 1)) BOTTOM).  
1. Reduce NF All ( ).  
Reflexive.  
2. Definedness.  
3. Reduce NF All ( ).  
Reflexive.

THEOREM:  
27c\_takeWhile\_dropWhile  
DEPENDS:  
11 12 13 9 3 16 17  
PROOF:  
Induction xs.  
1. Definedness.  
2. Reduce NF All ( ).  
Reflexive.  
3. Reduce NF All ( ).  
Introduce x xs IH p.  
SplitCase 2.

1. Rewrite  $\rightarrow$  All H1.  
Reduce NF All ( ).  
Definedness.
2. Introduce H2.  
ReduceH NF All in H2 ( ).  
Reduce NF All ( ).  
Rewrite  $\rightarrow$  All IH.  
  1. Reflexive.
  2. Exact H2.
3. Rewrite  $\rightarrow$  All H1.  
Reduce NF All ( ).  
Reflexive.

THEOREM:

27d\_eval\_isMember\_int

DEPENDS:

2 11 12 13 5 6 7 8

PROOF:

Induction xs.

1. Definedness.
2. Reduce NF All ( ).  
Reflexive.
3. Introduce x xs IH y H1 H2.  
Rewrite  $\rightarrow$  All "27d\_split\_eval\_list\_int" in H2.  
Split Shallow H2.  
Reduce NF All ( ).  
SplitCase 1.  
  1. Definedness.
  2. Reduce NF All ( ).  
Reflexive.
  3. Apply IH.  
Split Deep.  
    1. Exact H1.
    2. Exact H3.

THEOREM:

27d\_isMember\_filter

DEPENDS:

5 6 7 8 13 2 11 12 9 14 15

PROOF:

Induction xs.

1. Definedness.
2. Reduce NF All ( ).  
Reflexive.
3. Introduce y ys IH p x H1 H2 H3.  
Rewrite  $\rightarrow$  All "27d\_split\_eval\_list\_int" in H2.  
Split Shallow H2.  
ReduceH NF (map 1 NO) in H3 ( ).  
Rewrite  $\rightarrow$  All "27d\_split\_eval\_list\_bool" in H3.  
Split Shallow H3.

```

Reduce NF All ( ).
SplitCase 1.
1. Definedness.
2. Reduce NF All ( ).
  SplitCase 1.
  1. Definedness.
  2. Reduce NF All ( ).
    Rewrite -> All "int_==" in H9.
    1. Rewrite -> All H9 in H8.
      Symmetric.
      Exact H8.
    2. Definedness.
    3. Definedness.
  3. Apply IH.
    Split Deep.
    1. Exact H1.
    2. Exact H5.
    3. Exact H7.
3. SplitCase 1.
  1. Definedness.
  2. Rewrite -> All "int_==" in H9.
    1. Rewrite <- All H9.
      Rewrite -> All H8.
      Reduce NF All ( ).
      Rewrite -> All IH.
    1. Rewrite -> All H8.
      Assume (= (@ 13 eval_bool
                 (@ 5 isMember
                   (@ 6 _create_dictionary_Eq;
                     (@ 7 _create_dictionary_==;
                       (@ 8 ==_int))) y ys)) (BOOL True)).
    1. Cases (@ 5 isMember
              (@ 6 _create_dictionary_Eq;
                (@ 7 _create_dictionary_==;
                  (@ 8 ==_int))) y ys).
      1. Definedness.
      2. Reduce NF All ( ).
        Reflexive.
      3. Reduce NF All ( ).
        Reflexive.
    2. Apply "27d_eval_isMember_int".
      Split Shallow.
      1. Exact H4.
      2. Exact H5.
    2. Exact H4.
    3. Exact H5.
    4. Exact H7.
  2. Definedness.
  3. Definedness.
3. Apply IH.

```

Split Deep.  
1. Exact H1.  
2. Exact H5.  
3. Exact H7.

THEOREM:

27d\_split\_eval\_list\_bool

DEPENDS:

11 12 13

PROOF:

Introduce ys y.  
Reduce NF All ( ).  
SplitCase 1.  
1. SplitIff.  
1. Definedness.  
2. Definedness.  
2. SplitIff.  
1. Introduce H2.  
Split Deep.  
1. Exact H1.  
2. Exact H2.  
2. Introduce H2.  
Split Deep H2.  
Exact H3.  
3. SplitIff.  
1. AbsurdEquality.  
2. Introduce H2.  
Split Deep H2.  
Rewrite -> All H2 in H1.  
AbsurdEqualityH H1.

THEOREM:

27d\_split\_eval\_list\_int

DEPENDS:

11 12 2

PROOF:

Introduce ys y.  
Reduce NF All ( ).  
SplitCase 1.  
1. SplitIff.  
1. Definedness.  
2. Definedness.  
2. SplitIff.  
1. Introduce H2.  
Split Deep.  
1. Exact H1.  
2. Exact H2.  
2. Introduce H2.  
Split Deep H2.  
Exact H3.

3. SplitIff.
  1. AbsurdEquality.
  2. Introduce H2.
    - Split Deep H2.
    - Rewrite -> All H2 in H1.
    - AbsurdEqualityH H1.

THEOREM:

28\_isMember\_map\_o

DEPENDS:

5 6 7 8 2 9 10

PROOF:

Induction xs.

1. Reduce NF All ( ).
  - Reflexive.
2. Reduce NF All ( ).
  - Reflexive.
3. Introduce y ys IH f g H1.
  - Reduce NF All ( ).
    - Injective.
    - Split Shallow.
      1. Reflexive.
      2. Apply IH.
        - Introduce z zs H2.
        - Apply H1.
        - WitnessE zs.
        - Exact H2.

THEOREM:

29a\_length\_evalSpine

DEPENDS:

4 2 0

PROOF:

Induction xs.

1. Definedness.
2. Reduce NF All ( ).
  - Reflexive.
3. Introduce x xs IH H1.
  - ReduceH NF All in H1 ( ).
    - Reduce NF All ( ).
      - Apply IH.
      - Assume ~(= (@ 4 length\_list xs) BOTTOM).
      - 1. Reduce NF All ( ).
        - Reflexive.
      - 2. Definedness.

THEOREM:

29b\_finite\_++

DEPENDS:

0 3

PROOF:

- Induction xs.
1. Definedness.
  2. Reduce NF All ( ).  
Introduce ys H1 H2.  
Exact H2.
  3. Introduce x xs IH ys H1 H2.  
ReduceH NF All in H1 ( ).  
Reduce NF All ( ).  
Apply IH.  
Split Shallow.
    1. Exact H1.
    2. Exact H2.

THEOREM:

29c\_reverse\_++

DEPENDS:

0 1 3

PROOF:

- Induction xs.
1. Definedness.
  2. Reduce NF All ( ).  
Reflexive.
  3. Introduce x xs IH y H1.  
ReduceH NF All in H1 ( ).  
Reduce NF All ( ).  
Rewrite -> All IH.
    1. Reduce NF All ( ).  
Reflexive.
    2. Exact H1.

THEOREM:

29c\_reverse\_reverse

DEPENDS:

0 1

PROOF:

- Induction xs.
1. Definedness.
  2. Reduce NF All ( ).  
Reflexive.
  3. Introduce x xs IH H1.  
ReduceH NF All in H1 ( ).  
Reduce NF All ( ).  
Rewrite -> All "29c\_reverse\_++".
    1. Rewrite -> All IH.
      1. Reflexive.
      2. Exact H1.
    2. Apply "21d\_finite\_reverse".  
Exact H1.

THEOREM:  
  30a\_eval  
DEPENDS:  
  2  
PROOF:  
  Definedness.

THEOREM:  
  30b\_eval\_False  
DEPENDS:  
  2  
PROOF:  
  Introduce x.  
  Cut ("tertium" P (= x BOTTOM)).  
  Introduce H1.  
  Case Shallow H1.  
  1. Definedness.  
  2. Reduce NF All ( ).  
    Contradiction.  
    AbsurdEqualityH H2.