Master thesis
Computer Science

Radboud University

List Theorem Solving:

A segmentation approach based on length and congruence closure

Author:
Jeroen Kool
s1064054

Co-supervisor/assessor:
Robbert Krebbers
`robbert@cs.ru.nl`

Co-supervisor:
Ike Mulder
`i.mulder@cs.ru.nl`

Second assessor:
Sebastian Junges
`sjunges@cs.ru.nl`

March 21, 2024

Abstract

In this master thesis, we present a solver for proving theorems about lists, formulated in the proof assistant Coq. The solver is a new Coq tactic that attempts to automatically prove these theorems. Theorems about lists may include operations such as length, concatenation, reverse, map, take, and drop. For this fragment of theory, little research has been done and, to our knowledge, it has only been (partially) investigated in [40]. Our Coq solver leverages on new ideas of the identity property of the reverse operator (whereby applying reverse twice returns the original list) and employ a segmentation approach based on the length of the lists, utilizing the hypothesis with a congruence closure algorithm. We provide the solver's theoretical framework, practical implementation via new Coq tactics by using Coq's Ltac, and a new OCaml plugin, which is a modification of the existing congruence closure plugin. Evaluation against benchmarks demonstrates our solver's effectiveness, comparing favorably with existing methods, and highlighting its contributions to automated theorem proving in Coq. This work not only presents a significant advancement in list theorem solving but also lays the groundwork for future exploration in automated proof generation for list data structures.

Contents

Chapter 1

Introduction

Lists are common in programming. Arrays and lists are supported in a wide range of programming languages (such as C++, C#, Java, JavaScript, Python, Swift [17]), and are common to use in the development of algorithms [21, 32]. In many contemporary applications, it is important that the software works correctly [31]. To achieve the correctnes of software, we can use several program verification tools, such as Viper, VeriFast, Why3, Iris and VST. Typically, these systems work with Hoare/separaton logic, where we give a program and specifications as input to the verification tools, and the tools will create side conditions that need to be checked for the correctness of the program. The side conditions include, for example, integer arithmetic, but as in [18, 14] conditions on lists can also appear. The correctness of the side conditions can, for example, be checked by using an SMT solver [10] or a prove assistant [37], which is used, depending on the verification tool.

Problem To get a good idea of which programs are correct, we need to be able to verify conditions on lists. To achieve this, we need a solver that is able to check a wide range of conditions on lists. These conditions about lists can contain the following operators: append, reverse, map, update, length, take, drop, sublist (as a combination of take and drop), nth element access, and list repetition.

Prior work Currently, there is a solver for automatically proving theorems about lists that works in Coq, which is part of VST. This solver attempts to prove the same kind of theorems as the solver we as we described in our problem praragraph. The solver that is part of VST is theoretically complete for a fragment of the theory about lists, but requires an external completion not included in Coq, specifically a complete base solver based on the LIA+UF theory. Furthermore, this solver struggles with the reverse operator, because the reverse operator is not included out of the box, but has to be defined manually later on. Besides VST's solver, there are theorems about lists that can be transformed into SMT string theory. Operators not supported by SMT string theory include reverse and map, making them suboptimal for the fragment we want to examine.

Solution In this paper, we present a new solver that is implemented in Coq. We implemented two innovative concepts aimed at enhancing the solubility of our approach beyond what is currently achievable in existing research. The first idea is that we make more use of the properties of the reverse operator in the proof, specifically that applying the reverse operator twice on a list returns the original list. The second idea is that we try to segment the lists by looking at the properties of the list lengths, trying to determine if we can gain more information from comparisons with list expressions that use concatenation. To make the most of the segmentation idea, we will use the congruence closure algorithm to see if we can obtain more comparisons to which we can apply the idea. To use the idea of segmentation, we use the following lemma, where length is the length operator and $++$ is the append operator:

$$\mathsf{length}\, l_1 = \mathsf{length}\, l_3 \to l_1 \mathbin{++} l_2 = l_3 \mathbin{++} l_4 \to (l_1 = l_3 \land l_2 = l_4).$$

Contributions

- We give new ideas on how we can prove theorems about lists (Chapter 3).
- We describe an inference system that can be applied on list problems (Chapter 4). This inference system will be a formal way to describe the (new) ideas.
- We will discuss a way of implementing this inference system in Coq's Ltac and a OCaml based plugin.
- We give a set of benchmarks on which list solvers can be applied and we compare our solver to the existing work.

Outline In Chapter 2 we start with giving a brief explanation about lists in Coq, giving an overview of the operators we use, and giving some example lemma about lists. Afterward, in Chapter 3 we will see which problems we encounter when solving lemmas about lists, and explain which ideas we use in our solver to overcome these problems. In Chapter 4 we present the grammar of the theorems that we will solve and give a more formal formulation of the ideas that we use in the solver. The formal formulations will be given as inference rules, which will form a (non-deterministic) inference system which can be applied on lists. We finish Chapter 4 with an order in which the inference rules can be applied to create a system that finds proofs. Subsequently, in Chapter 5, we will see how we have implemented the rules of Chapter 4 into Coq's Ltac and explain how we extended this with an OCaml-based plugin for Coq. In Chapter 6, we will see how this implementation performs, compared to the existing solvers. We exam this performance by looking at benchmarks that are obtained by looking for theorems in Coq's standard library and look at the goals the existing solver has to solve. Chapter 6 is followed by Chapter 7 in which we will give an overview of the existing solvers, explain more about SMT, and explain something about the SMT array theory. Finally, in Chapter 8, we will conclude this thesis with a conclusion and future work chapter, to summarize the results and give ideas for further research.

Chapter 2

Background

In this section, we will give an idea which lemmas we aim to solve with our solver. First in Section 2.1 we will give the definition of lists and see how this is defined in Coq. Then, in Section 2.2, we will show the operators that we use on lists that are predefined in Coq, followed by the operators that we can derive from the predefined operators in Section 2.3. Finally, there will be some sample lemmas in Section 2.4 that give an idea what kind of lemmas we aim to solve.

2.1 Definition of lists in Coq

In Coq, lists are defined as an inductive type as follows:

```
Inductive list (A:Type) : Type :=
| nil : list A
| cons : A → list A → list A.
```

Here, A represents the type of elements in the lists. The definition consists of two constructors, nil, and cons. Going forward, we will denote this inductive type as $list_\alpha$ where α signifies the type of elements in the list. While there could be a theory associated with the elements of this type, we opt to disregard such a theory and treat the list elements as abstract entities for the implementation of our solver. For example, the singleton list $[1+2]$, is different from the lists $[3]$ and we disregard the theory for natural numbers.

nil represents a list without any element, the empty lists. We also denote the empty list as $[\,]$. The cons constructor allows for the creation of lists by adding an element at the beginning of a list. By providing cons an element of an arbitrary type and a list where the elements have the same type, we generate an extended list. For example, we can express a list of natural numbers 1, 2, and 3, in that specific order as $\mathsf{cons}\,1\,(\mathsf{cons}\,2\,(\mathsf{cons}\,3\,\mathsf{nil}))$. We also adopt the infix notation :: for the `cons`. Utilizing this notation the list of numbers 1, 2 and 3 can be written as $1 :: 2 :: 3 :: \texttt{nil}$ or $1 :: 2 :: 3 :: [\,]$. For lists we also use a shorter notation, for $1 :: 2 :: 3 :: [\,]$ we can express it as $[1; 2; 3]$

2.2 Definition of list operators

We consider six operators that are predefined in Coq, each with its type, definition, and recursive definition. In subsequent chapters we normally omit the recursive definition unless we state otherwise. We provide the recursive definitions for completeness, as the definitions in Coq are similar to the recursive definitions.

length

The length operator has type $\mathit{list}_\alpha \to \mathbb{N}$ and returns the length of the list. It is defined as $\mathsf{length}\,[x_1; \ldots; x_n] := n$.

The length operators could also be recursively defined as

$$\mathsf{length}\,[\,] := 0$$
$$\mathsf{length}\,(a :: l) := 1 + \mathsf{length}\,l.$$

append

The append operator, *app*, concatenates two lists. It has type $\mathit{list}_\alpha \to \mathit{list}_\alpha \to \mathit{list}_\alpha$ and is defined as

$$\mathsf{app}\,[x_1; \ldots; x_n]\,[y_1; \ldots; y_n] := [x_1; \ldots x_n; y_1; \ldots; y_n].$$

For instance, given two lists of type $\mathit{list}_\mathbb{N}$, $[1; 2]$ and $[3; 4]$. The append operator allows us to concatenate these lists and $\mathsf{app}\,[1; 2]\,[3; 4]$ returns $[1; 2; 3; 4]$.

For the append operator, we often use the infix notation $++$. This means that the notation $\mathsf{app}\,[1; 2]\,[3; 4]$ means the same as $[1; 2] ++ [3; 4]$.

The append operator could also be recursively defined as

$$\mathsf{app}\,[\,]\,l' := l$$
$$\mathsf{app}\,(x :: l)\,l' := x :: (\mathsf{app}\,l\,l').$$

reverse

The reverse operator, rev, reverses the order of the list. The operator has type $\mathit{list}_\alpha \to \mathit{list}_\alpha$ and is defined as

$$\mathsf{rev}\,[x_1; \ldots; x_n] := [x_n; \ldots; x_1].$$

For example, $\mathsf{rev}\,[1; 2; 3] = [3; 2; 1]$.

The reverse operator could also be recursively defined as

$$\begin{aligned} \mathsf{rev}\,[\,] &:= [\,] \\ \mathsf{rev}\,(x :: l) &:= \mathsf{rev}\, l ++[x]. \end{aligned}$$

take

The take operator of type $\mathbb{N} \to \mathit{list}_\alpha \to \mathit{list}_\alpha$, is defined as

$$\mathsf{take}\, m\, [x_1; \ldots; x_n] := [x_1; \ldots; x_{\min(n,m)}].$$

For a given list l, $\mathsf{take}\, m\, l$ will return the first m number of elements or, if the list has less than or equal to m elements, the entire list. For example, $\mathsf{take}\, 2\, [1; 2; 3; 4] = [1; 2]$ and $\mathsf{take}\, 4\, 1; 2; 3; 4 = [1; 2; 3; 4]$.

The take operator could recursively be defined as:

$$\begin{aligned} \mathsf{take}\, 0\, l &:= [\,] \\ \mathsf{take}\, (S\, n)\, [\,] &:= [\,] \\ \mathsf{take}\, (S\, n)\, (x :: l) &:= x :: \mathsf{take}\, n\, l. \end{aligned}$$

drop

The drop operator also with type $\mathbb{N} \to \mathit{list}_\alpha \to \mathit{list}_\alpha$, is defined as

$$\mathsf{drop}\, m\, [x_1; \ldots; x_n] := [x_{m+1}; \ldots; x_n].$$

Therefore the operator drop removes the first m elements from a list. If a list has less than or equal to m elements, the operator returns the empty list. For instance, $\mathsf{drop}\, 2\, [1; 2; 3; 4] = [3; 4]$ and $\mathsf{drop}\, 4\, [1; 2; 3; 4] = [\,]$.

The drop operator could be recursively defined as

$$\begin{aligned} \mathsf{drop}\, 0\, l &:= l \\ \mathsf{drop}\, (S\, n)\, [\,] &:= [\,] \\ \mathsf{drop}\, (S\, n)\, (x :: l) &:= \mathsf{drop}\, n\, l. \end{aligned}$$

map

The map operator has type $(\alpha \to \beta) \to \mathit{list}_\alpha \to \mathit{list}_\beta$ and is defined as

$$\mathsf{map}\, f\, [x_1; \ldots; x_n] := [f(x_1); \ldots; f(x_n)],$$

Where f is a function that maps elements of type α to elements of type β. As we treat the elements of the list as abstract entities, we also tread the functions of elements as abstract entities, i.e. we will not evaluate the function. For example, $\mathsf{map}\, (\lambda\, x.\ x + 3)\, [0; 1; 2] = [(0 + 3); (1 + 3); (2 + 3)]$.

repeat

The repeat operator of type $\mathbb{N} \to \alpha \to \mathit{list}_\alpha$. $\mathsf{repeat}\, n\, v$ generates a list of length n, where all elements of the list are equal to v. The repeat operator is recusivly defined as

$$\begin{aligned} \mathsf{repeat}\, 0\, x &:= [\,] \\ \mathsf{repeat}\, (S\, n)\, x &:= x :: \mathsf{repeat}\, n\, x. \end{aligned}$$

For example, $\mathsf{repeat}\, 3\, 1 := [1; 1; 1]$

nth

The type of the *nth* operator is $\mathbb{N} \to \mathit{list}_\alpha \to \alpha$. $\mathsf{nth}\, i\, l$ returns the i -th element of the list l if i is in range, and returns a default element if not in range. The nth operator works only if there exists a default $\mathbb{d}$ for the type α, that is, its inhabited. We use type classes in Coq [19] to define the default element of each type. This gives us that the nth operator also can be seen as

$$\mathsf{nth}\, i\, [x_1; \ldots; x_n] := \begin{cases} x_{i-1} & \text{if } i \leq n \\ \mathbb{d} & \text{if } i > n. \end{cases}$$

The recursive definition of nth is

$$\begin{aligned} \forall n.\ \mathsf{nth}\, n\, [\,] &:= \mathbb{d} \\ \mathsf{nth}\, 0\, (x :: l) &:= x \\ \mathsf{nth}\, (S\, n)\, (x :: l) &:= \mathsf{nth}\, n\, l. \end{aligned}$$

2.3 Derived operators

By combining the operators from the preceding section in a specific sequence, we can derive new operators. For our solver, we have derived two new operators from the existing ones. By deriving operators from previous definitions, we do not have to consider rules or techniques to prove theorems with these derived operators. Instead of having rules or prove techniques for these derived operators, the solver can unfold the definitions of the derived operators and deal with the list operators of Section 2.2.

update

The update operator is of type $\mathbb{N} \to \mathit{list}_\alpha \to \alpha \to \mathit{list}_\alpha$. It is defined as follows:

$$\begin{aligned} \mathsf{update}\, i\, l\, v := (&\mathsf{take}\, (\mathsf{length}\, l) \\ &(\mathsf{take}\, i\, l) +\!+ [v] +\!+ (\mathsf{drop}\, (i + 1)\, l)). \end{aligned}$$

To update the i-th position of list l, the update operator performs the following steps:

1. Takes the first i elements from the list l.
2. Concatenates them with the singleton list consisting of v.
3. Appends the remaining elements of the list l starting from position $i+1$.
4. take the first $\mathsf{length}\, l$ elements, in case of when i is out of bound.

If the list v is empty, the i -th element of the list l is replaced by the default element $\mathbb{d}$.

For example, $\mathsf{update}\, 2\, [1;2;3;4;5]\, 0 = [1;2;0;4;5]$ and $\mathsf{update}\, 6\, 1;2;3;4;5 = [1;2;3;4;5]$.

This definition is equivalent to the recursive definition

$$\begin{aligned} \forall n.\ \mathsf{update}\, n\, [\,]\, v &:= [\,] \\ \mathsf{update}\, 0\, (x :: l)\, v &:= v :: l \\ \mathsf{update}\, (S\, n)\, (x :: l)\, v &:= x :: \mathsf{update}\, n\, l\, v. \end{aligned}$$

flip_ends

The $\mathsf{flip\_ends}$ operator has type $\mathbb{N} \to \mathbb{N} \to \mathit{list}_\alpha \to \mathit{list}_\alpha$. The operator is defined as

$$\begin{aligned} \mathsf{flip\_ends}\, lo\, hi\, l := {} & \mathsf{drop}\, 0\, (\mathsf{take}\, lo\, (\mathsf{rev}\, l)) \\ & ++\ \mathsf{drop}\, lo\, (\mathsf{take}\, hi\, l) \\ & ++\ \mathsf{drop}\, hi\, (\mathsf{take}\, (\mathsf{length}\, l)\, (\mathsf{rev}\, l)) \end{aligned}$$

For example,

$$\mathsf{flip\_ends}\, 2\, 6\, [1;2;3;4;5;6;7;8;9] = [9;8;3;4;5;6;3;2;1]$$

2.4 Sample lemmas

We will now present some simple sample lemmas about lists to illustrate what kind of lemmas we aim to solve and the manual approach we use to solve them. Many simple lemmas about lists will be proved by using induction on the list element. In Chapter 3 we will see more advanced lemmas that cannot be proven by simply using induction.

Example 2.4.1. We will show how to prove the associativity of the append operator, i.e.

$$l_1 ++ (l_2 ++ l_3) = (l_1 ++ l_2) ++ l_3.$$

Applying induction on l_1 we get two subgoals. The first one is

$$[\,] ++ (l_2 ++ l_3) = ([\,] ++ l_2) ++ l_3,$$

and can be solved by using simplification with using the recursive definition of the append operator.

The second goal is

$$\frac{\text{IHl}:\ l_1 +\!\!+ (l_2 +\!\!+ l_3) = (l_1 +\!\!+ l_2) +\!\!+ l_3}{a :: l_1 +\!\!+ (l_2 +\!\!+ l_3) = a :: (l_1 +\!\!+ l_2) +\!\!+ l_3.}$$

This goal can be proven by rewriting with the induction hypothesis.

Example 2.4.2. We will prove the lemma

$$\mathsf{rev}\,(l_1 +\!\!+ l_2) = \mathsf{rev}\,l_2 +\!\!+ \mathsf{rev}\,l_1.$$

In the first step of proving this, we generalize over l_2 by using the `revert` tactic. Afterwards, we can prove the lemma by using induction on l_1.

The first case, after simplifying it, is

$$\mathsf{rev}\,l_2 = \mathsf{rev}\,l_2 +\!\!+ [\,].$$

To prove this case, we need the fact that $\forall\, l.\ l = l +\!\!+ [\,]$, which can be proven by induction on l.

The second case,

$$\frac{\text{IHl}:\ \forall l_2,\ \mathsf{rev}\,l_1 +\!\!+ l_2 = \mathsf{rev}\,l_2 +\!\!+ \mathsf{rev}\,l_1}{\mathsf{rev}\,l_1 +\!\!+ (a :: l_2) = \mathsf{rev}\,(a :: l_2) +\!\!+ \mathsf{rev}\,l_1,}$$

can be proven by using the recursive definition of the reverse operator, $\mathsf{rev}\,(a :: l_1) = \mathsf{rev}\,l_1 +\!\!+ [a]$, by rewriting with the induction hypothesis, and using the lemma from the previous example. We also have to notice that $a :: l_1 = [a] +\!\!+ l_1$.

Example 2.4.3. The lemma

$$\mathsf{rev}\,(\mathsf{rev}\,l) = l$$

can be proven by induction in l.

The first case,

$$\mathsf{rev}\,(\mathsf{rev}\,[\,]) = [\,],$$

is proved by simplification.

The second case,

$$\frac{\text{IHl}:\ \mathsf{rev}\,(\mathsf{rev}\,l) = l}{\mathsf{rev}\,(\mathsf{rev}\,(a :: l)) = a :: l,}$$

can be proven by using the lemma of the previous example and by using the induction hypothesis.

Example 2.4.4. We will show how we can prove the lemma

$$\mathsf{take}\, n\, (l_1 \mathbin{++} l_2) = (\mathsf{take}\, n\, l_1) \mathbin{++} (\mathsf{take}\, (n - \mathsf{length}\, l_1)\, l_2).$$

We first generalize over n, followed by an induction over l_1.

The first case can be proven by showing that $\mathsf{take}\, n\, [\,] = [\,]$, which can be done by destructing n.

The second case,

$$\frac{\mathrm{IHl}_1 \ :\ \forall n,\ \mathsf{take}\, n\, (l_1 \mathbin{++} l_2) = (\mathsf{take}\, n\, l_1) \mathbin{++} (\mathsf{take}\, (n - \mathsf{length}\, l_1)\, l_2)}{\mathsf{take}\, n\, ((a :: l_1) \mathbin{++} l_2) = (\mathsf{take}\, n\, (a :: l_1)) \mathbin{++} (\mathsf{take}\, (n - \mathsf{length}\, (a :: l_1))\, l_2),}$$

can be proven by destructing n and rewriting with the induction hypothesis.

Chapter 3

Key ideas

Not all lemmas can be easily proven with simple induction. As seen before in Section 2.4, sometimes we need to generalize, use the definitions of the operators, or we need previous proven lemmas to finalize the proof. Therefore, our solver employs alternative methods to prove the lemmas. In this chapter, we demonstrate the problems we encounter using examples. In each section, we introduce a new key idea utilized in the solver to prove the lemmas.

This chapter is structured as follows:

1. We give an example
2. We explain why the example can not be proven through application of the earlier mentioned methods
3. We will explain what the new key idea is
4. We demonstrate how the key idea can be utilized to prove the example

3.1 Operator rearrangement lemmas

Example 3.1.1.

$$\mathsf{rev}\,(\mathsf{rev}\,l \mathbin{++} [x]) = [x] \mathbin{++} l$$

In this example, induction on l cannot be utilized to prove this example. If we apply induction on l we get in the inductive case $l = [y] \mathbin{++} l'$. Thereby we see that our goal become

$$\mathsf{rev}\,(\mathsf{rev}\,l \mathbin{++} [y] \mathbin{++} [x]) = [x] \mathbin{++} [y] \mathbin{++} l.$$

On the right-hand side of our new goal, we get a singleton list $[y]$ between $[x]$ and l of our induction hypothesis, whereby we can see that we will not be able to apply to use the induction hypothesis to prove this goal.

The key idea is to use rewrite rules for the operators, to put the operators in a specific order. By putting the operators in a specific order, we obtain a normal form for the list expressions. By rewriting list expressions in a normal form, we can see if two expressions of the goal are equal, as we will see at the end of this section in Example 3.1.1. Furthermore, by putting the hypothesis in a normal form, we will be able to apply the other key ideas, which we will see in the later sections of this chapter.

In the following arrangement, we show how we ordered the operators and the `nil` element.

$$[\,] > \mathsf{app},\ {+\!+} > \mathsf{repeat} > \mathsf{map} > \mathsf{rev} > \mathsf{drop} > \mathsf{take} > \mathsf{nth}$$

While operators later in the ordering can be present in the arguments of previous operators, the reverse is not true.

For example, $\mathsf{rev}\, l_1 \mathbin{++} [\,]$ is not in normal form, since $\mathbin{++}$ and $[\,]$ precede rev in the list and are in the argument of rev. On the other hand $\mathsf{rev}\, l_1 \mathbin{++} \mathsf{drop}\, n\, l_2$, is considered to be in normal form.

In Figure 3.1 and Figure 3.2, we outline the rewrite rules used to rewrite an expression in a normal form. We only need to provide rewriting rules for the element or operator that is higher in order. Furthermore, all the rules will only be applied from left to right.

In our solver, we do not consider the constructor cons; instead, we represent $a :: l$ as $[a] \mathbin{++} l$. The list constructor will become the append of a singleton. Consequently, Figure 3.1 and Figure 3.2 do not have rules for cons. Note that we do not have a box for repeat in the figure, because the repeat operator does not take a list as an argument.

Upon examining Example 3.1.1, we observe that we can prove the lemma by first rewriting the left-hand side with the rule:

$$\mathsf{rev}\,(l_1 \mathbin{++} l_2) = \mathsf{rev}\, l_2 \mathbin{++} \mathsf{rev}\, l_1.$$

Thereby we get the new goal

$$\mathsf{rev}\,[x] \mathbin{++} \mathsf{rev}\,(\mathsf{rev}\, l) = [x] \mathbin{++} l.$$

Subsequently we can use the rules $\mathsf{rev}\,(\mathsf{rev}\, l) = l$ and $\mathsf{rev}\,[x] = [x]$ to obtain

$$[x] \mathbin{++} l = [x] \mathbin{++} l,$$

whereby we can conclude the proof.

$$\underline{\mathsf{app}}$$

$$\begin{aligned}
[\,] \mathbin{+\!+} l &= l \\
l \mathbin{+\!+} [\,] &= l
\end{aligned}$$

$$\underline{\mathsf{rev}}$$

$$\begin{aligned}
\mathsf{rev}\,[\,] &= [\,] \\
\mathsf{rev}\,(l_1 \mathbin{+\!+} l_2) &= \mathsf{rev}\,l_2 \mathbin{+\!+} \mathsf{rev}\,l_1 \\
\mathsf{rev}\,(\mathsf{repeat}\,n\,v) &= \mathsf{repeat}\,n\,v \\
\mathsf{rev}\,(\mathsf{map}\,f\,l) &= \mathsf{map}\,f\,(\mathsf{rev}\,l)
\end{aligned}$$

$$\underline{\mathsf{map}}$$

$$\begin{aligned}
\mathsf{map}\,f\,[\,] &= [\,] \quad \text{note that the list on the left- and right-hand side have different types} \\
\mathsf{map}\,f\,(l_1 \mathbin{+\!+} l_2) &= (\mathsf{map}\,f\,l_1) \mathbin{+\!+} (\mathsf{map}\,f\,l_2) \\
\mathsf{map}\,f\,(\mathsf{repeat}\,n\,v) &= \mathsf{repeat}\,n\,(f\ v)
\end{aligned}$$

$$\underline{\mathsf{drop}}$$

$$\begin{aligned}
\mathsf{drop}\,n\,l &= l \quad \text{if } n = 0 \\
\mathsf{drop}\,n\,l &= [\,] \quad \text{if } n \geq \mathsf{length}\,l \\
\mathsf{drop}\,n\,(l_1 \mathbin{+\!+} l_2) &= \mathsf{drop}\,n\,l_1 \mathbin{+\!+} \mathsf{drop}\,(n - \mathsf{length}\,l_1)\,l_2 \\
\mathsf{drop}\,n\,(\mathsf{repeat}\,m\,v) &= \mathsf{repeat}\,(m-n)\,v \\
\mathsf{drop}\,n\,(\mathsf{map}\,f\,l) &= \mathsf{map}\,f\,(\mathsf{drop}\,n\,l) \\
\mathsf{drop}\,n\,(\mathsf{rev}\,l) &= \mathsf{rev}\,(\mathsf{take}\,(\mathsf{length}\,l - n)\,l)
\end{aligned}$$

$$\underline{\mathsf{take}}$$

$$\begin{aligned}
\mathsf{take}\,n\,l &= [\,] \quad \text{if } n = 0 \\
\mathsf{take}\,n\,l &= l \quad \text{if } n \geq \mathsf{length}\,l \\
\mathsf{take}\,n\,(l_1 \mathbin{+\!+} l_2) &= \mathsf{take}\,n\,l_1 \mathbin{+\!+} \mathsf{take}\,(n - \mathsf{length}\,l_1)\,l_2 \\
\mathsf{take}\,n\,(\mathsf{repeat}\,m\,v) &= \mathsf{repeat}\,(\min(n,m))\,v \\
\mathsf{take}\,n\,(\mathsf{map}\,f\,l) &= \mathsf{map}\,f\,(\mathsf{take}\,n\,l) \\
\mathsf{take}\,n\,(\mathsf{rev}\,l) &= \mathsf{rev}\,(\mathsf{drop}\,(\mathsf{length}\,l - n)\,l) \\
\mathsf{take}\,n\,(\mathsf{drop}\,m\,l) &= \mathsf{drop}\,m\,(\mathsf{take}\,(n+m)\,l)
\end{aligned}$$

Figure 3.1: Rewrite rules for rearranging operators #1

nth

$$\mathsf{nth}\,i\,[\,] = \mathbb{d}$$

$$\mathsf{nth}\,i\,(l_1 \mathbin{+\!\!+} l_2) = \begin{cases} \mathsf{nth}\,i\,l_1 & \text{if } i < \mathsf{length}\,l_1 \\ \mathsf{nth}\,(i - \mathsf{length}\,l_1)\,l_2 & i \geq \mathsf{length}\,l_1 \end{cases}$$

$$\mathsf{nth}\,i\,(\mathsf{repeat}\,n\,v) = \begin{cases} v & \text{if } i < n \\ \mathbb{d} & \text{if } i \geq n \end{cases}$$

$$\mathsf{nth}\,i\,(\mathsf{map}\,f\,l) = \begin{cases} f(\mathsf{nth}\,i\,l) & \text{if } i < \mathsf{length}\,l \\ \mathbb{d} & \text{if } i \geq \mathsf{length}\,l \end{cases}$$

$$\mathsf{nth}\,i\,(\mathsf{rev}\,l) = \begin{cases} \mathsf{nth}\,(\mathsf{length}\,l - i - 1)\,l & \text{if } i < \mathsf{length}\,l \\ \mathbb{d} & \text{if } i \geq \mathsf{length}\,l \end{cases}$$

$$\mathsf{nth}\,i\,(\mathsf{drop}\,j\,l) = \mathsf{nth}\,(i + j)\,l$$

$$\mathsf{nth}\,i\,(\mathsf{take}\,j\,l) = \begin{cases} \mathsf{nth}\,i\,l & \text{if } i < j \\ \mathbb{d} & \text{if } i \geq j \end{cases}$$

Special rules

$$(l_1 \mathbin{+\!\!+} l_2) \mathbin{+\!\!+} l_3 = l_1 \mathbin{+\!\!+} (l_2 \mathbin{+\!\!+} l_3)$$

$$\mathsf{repeat}\,n\,v \mathbin{+\!\!+} \mathsf{repeat}\,m\,v = \mathsf{repeat}\,(n + m)\,v$$

$$\mathsf{repeat}\,1\,v = [v]$$

$$\mathsf{rev}\,[x] = [x]$$

$$\mathsf{rev}\,(\mathsf{rev}\,l) = l$$

$$\mathsf{map}\,f\,[x] = [f(x)]$$

$$\mathsf{drop}\,n\,(\mathsf{drop}\,m\,l) = \mathsf{drop}\,(n + m)\,l$$

$$\mathsf{take}\,n\,(\mathsf{take}\,m\,l) = \mathsf{take}\,(\min(n, m))\,l$$

$$[\mathsf{nth}\,i\,l] = \mathsf{take}\,(\mathsf{length}\,l)\,(\mathsf{drop}\,i\,l \mathbin{+\!\!+} \mathbb{d})$$

$$\min(n, m) = \begin{cases} n & \text{if } n \leq m \\ m & \text{if } m < n \end{cases}$$

$$\mathsf{update}\,i\,(l_1 \mathbin{+\!\!+} l_2)\,v = \begin{cases} \mathsf{update}\,i\,l_1\,v \mathbin{+\!\!+} l_2 & \text{if } i < \mathsf{length}\,l_1 \\ l_1 \mathbin{+\!\!+} \mathsf{update}\,(i - \mathsf{length}\,l_1)\,l_2\,v & \text{if } i \geq \mathsf{length}\,l_1 \end{cases}$$

Figure 3.2: Rewrite rules for rearranging operators #2

3.2 Reverse list assumption

Example 3.2.1.

$$l_1 \mathbin{++} l_2 = \mathsf{rev}\, k \to \mathsf{rev}\, l_1 \mathbin{++} \mathsf{rev}\, l_2 = k$$

Applying induction will give us the same problem as we have seen in Example 3.1.1, that is, we get a singleton in the middle of the append, and thereby the induction hypothesis will become useless. Furthermore, none of the operator rearrangement rules we have seen in Section 3.1 can be applied to the goal.

We establish the proof based on the fact that $l = k \to \mathsf{rev}\, l = \mathsf{rev}\, k$. Even when induction is performed on l_1, we require this property in the base case. The operator rearrangement lemmas introduced earlier do not provide enough information, because we really need the information that $l_1 \mathbin{++} l_2 = \mathsf{rev}\, k$.

The key idea here is to employ the lemma $l = k \to \mathsf{rev}\, l = \mathsf{rev}\, k$ for every hypothesis in the context that constitutes an equation with lists. Subsequently, we rewrite the newly obtained hypothesis by applying the operator rearrangement rules from Section 3.1. If this yields a new hypothesis, it is incorporated into the context.

By examining Example 3.2.1, we have the hypothesis $l_1 \mathbin{++} l_2 = \mathsf{rev}\, k$. When we apply the lemma $l = k \to \mathsf{rev}\, l = \mathsf{rev}\, k$ to this hypothesis, we obtain a new hypothesis $\mathsf{rev}\,(l_1 \mathbin{++} l_2) = \mathsf{rev}\,(\mathsf{rev}\, k)$. Subsequently, by applying the rewrite rules

$$\mathsf{rev}\,(l_1 \mathbin{++} l_2) = \mathsf{rev}\, l_2 \mathbin{++} \mathsf{rev}\, l_1$$

and

$$\mathsf{rev}\,(\mathsf{rev}\, l) = l,$$

we arrive at the hypothesis $\mathsf{rev}\, l_2 \mathbin{++} \mathsf{rev}\, l_1 = k$, consequently allowing us to conclude the proof of the lemma.

3.3 List segmentation

Example 3.3.1.

$$\begin{aligned}
&l_1 \mathbin{++} l_2 = l_3 \mathbin{++} l_4 \mathbin{++} l5 \to \\
&\mathsf{length}\, l_1 = \mathsf{length}\, l_3 \to \\
&l_4 \mathbin{++} l_5 = l_6 \mathbin{++} l_7 \\
&l_2 = l_6 \mathbin{++} l_7
\end{aligned}$$

When applying induction, we already get stuck in the base case, where the list on which we apply induction will become the empty list. All expressions are already in a normal form, and therefore the operation rearrangement from Section 3.1 cannot be applied. Finally, the goal and hypothesis do not contain the reverse operator and, therefore, reversing the list assumptions, the idea of Section 3.2 will not provide us with useful information.

In this example, the information we need to prove the lemma is contained in the hypothesis with an append operator of the lists. To validate that both sides of the append operator are equal, we require additional information about (the length of) the lists, which is contained in the hypothesis $\mathsf{length}\, l_1 = \mathsf{length}\, l_3$.

When we have an equation about lists in the context of our goal, we search for a prefix of the left- and right-hand sides that has the same length. Identifying such a prefix, we establish that the prefix and suffix have to be equal. Consequently, we are using the lemma $\mathsf{length}\, l_{11} = \mathsf{length}\, l_{21} \to l_{11} \mathbin{++} l_{12} = l_{21} \mathbin{++} l_{22} \to l_{11} = l_{21} \wedge l_{12} = l_{22}$.

In the case of Example 3.3.1, we know that l_1 and l_3 have the same length. Therefore, we can apply the lemma $\mathsf{length}\, l_{11} = \mathsf{length}\, l_{21} \to l_{11} \mathbin{++} l_{12} = l_{21} \mathbin{++} l_{22} \to l_{11} = l_{21} \wedge l_{12} = l_{22}$. This lemma enables us to deduce that $l_2 = l_4 \mathbin{++} l_5$. Consequently, we can conclude the lemma by rewriting with the hypothesis $l_4 \mathbin{++} l_5 = l_6 \mathbin{++} l_7$.

We will elaborate on the idea of list segmentation in the following subsections. Firstly, in Section 3.3.1 we will explore the interaction of the length operator with other operators. Subsequently, in Section 3.3.2 we will demonstrate how we can utilize the information in the context, by extracting the information of length from the context and use the newly obtained information with linear integer arithmetic. Finally, in Section 3.3.3 we reveal how we can utilize the information in the context to identify more equalities to which we can apply the list segmentation idea by using a congruence closure algorithm.

3.3.1 The length operator

Example 3.3.2.

$$\begin{aligned}&\mathsf{rev}\, l_1 \mathbin{++} l_2 = l_3 \mathbin{++} \mathsf{rev}\, l_4 \to\\ &\mathsf{length}\, l_1 = \mathsf{length}\, l_3 \to\\ &\mathsf{rev}\, l_2 = l_4\end{aligned}$$

In order to apply the list segmentation in this example, we need to know that $\mathsf{length}\,(\mathsf{rev}\, l_1) = \mathsf{length}\, l_3$. To obtain such information, we need to know how the length operator interacts with the rev operator.

For every operator, we have a rewrite rule that is given in Figure 3.3. We apply these rules to rewrite the expressions from the left-hand side of the equality to the right-hand side of the equality.

$$\begin{aligned}
\mathsf{length}\,[\,] &= 0\\
\mathsf{length}\,[x] &= 1\\
\mathsf{length}\,(l_1 \mathbin{++} l_2) &= \mathsf{length}\,l_1 + \mathsf{length}\,l_2\\
\mathsf{length}\,(\mathsf{rev}\,l) &= \mathsf{length}\,l\\
\mathsf{length}\,(\mathsf{map}\,f\,l) &= \mathsf{length}\,l\\
\mathsf{length}\,(\mathsf{take}\,n\,l) &= \min(n, \mathsf{length}\,l)\\
\mathsf{length}\,(\mathsf{drop}\,n\,l) &= \mathsf{length}\,l - n\\
\mathsf{length}\,(\mathsf{repeat}\,n\,v) &= n
\end{aligned}$$

Figure 3.3: Rewrite rules for the length

As the take and drop operators have comparable ideas, when deriving the length for the drop operator, we can also say that $\mathsf{length}\,(\mathsf{drop}\,n\,l) = \max(0, \mathsf{length}\,l - n)$. Because we operate with natural numbers, each number is greater than or equal to 0. This will lead to the result that $\max(0, \mathsf{length}\,l - n) = \mathsf{length}\,l - n$, and therefore we can omit the max operator in the rewrite rule.

Consequently, we can obtain the rules for the derived operators as shown in Figure 3.4.

$$\begin{aligned}
\mathsf{length}\,(\mathsf{update}\,i\,l\,v) &= \mathsf{length}\,l\\
\mathsf{length}\,(\mathsf{flip\_ends}\,lo\,hi\,l) &= \mathsf{length}\,l + \mathsf{length}\,(\mathsf{take}\,lo\,l) - hi
\end{aligned}$$

Figure 3.4: Rewrite rules for the length of derived operators

In case of Example 3.3.2 we now see that $\mathsf{rev}\,l_1$ and l_3 have the same length by applying the given rules and use the hypothesis. Therefore, we obtain $l_2 = \mathsf{rev}\,l_4$, and with the reverse list assumption idea of Section 3.2 we conclude that $\mathsf{rev}\,l_2 = l_4$.

3.3.2 Length of lists assumptions with lia

Example 3.3.3.

$$\begin{aligned}
&l_1 \mathbin{++} l_2 \mathbin{++} l_3 = l_4 \mathbin{++} l_5 \mathbin{++} l_6 \to\\
&l_2 = l_5 \to\\
&\mathsf{length}\, l_3 = \mathsf{length}\, l_6 \to\\
&l_1 = l_4
\end{aligned}$$

In the given example, the context involves an equation with concatenated lists on both sides: $l_1 \mathbin{++} l_2 \mathbin{++} l_3 = l_4 \mathbin{++} l_5 \mathbin{++} l_6$. To apply the list segmentation technique, we need to ensure that the lengths of l_1 and l_4 are equal, because this condition is associated with list segmentation. The information about the equal lengths can be obtained from other hypotheses.

The lemma $l = l' \to \mathsf{length}\, l = \mathsf{length}\, l'$ is valuable in situations where the left- and right-hand sides of an equation in the context involve list terms. This lemma establishes a connection between the equality of lists and the equality of their lengths.

After applying this lemma to relevant hypotheses, the next step involves leveraging linear integer arithmetic [15], which is facilitated by the 'lia' tactic that is available in the Coq standard library. It is particularly useful for reasoning about integer properties, such as list lengths.

In Example 3.3.3 we derive equations such as

$$\mathsf{length}\, l_1 + \mathsf{length}\, l_2 + \mathsf{length}\, l_3 = \mathsf{length}\, l_4 + \mathsf{length}\, l_5 + \mathsf{length}\, l_6$$

and

$$\mathsf{length}\, l_2 = \mathsf{length}\, l_5,$$

by utilizing the lemma $l = l' \to \mathsf{length}\, l = \mathsf{length}\, l'$. Additionally, we have the hypothesis $\mathsf{length}\, l_3 = \mathsf{length}\, l_6$, already present in the context. Applying arithmetic reasoning, we deduce that $\mathsf{length}\, l_1 = \mathsf{length}\, l_4$ enabling us to successfully prove the lemma using the list segmentation technique.

3.3.3 Congruence

Example 3.3.4.

$$\begin{aligned}
&l_{23} = l_c \mathbin{++} l_d \to\\
&l_{12} = l_1 \mathbin{++} l_2 \to\\
&l_c \mathbin{++} l_d = l_e \mathbin{++} l_f \to\\
&l_e \mathbin{++} l_f = l_2 \mathbin{++} l_3' \to\\
&l_{12} \mathbin{++} l_3 = l_1 \mathbin{++} l_{23} \to\\
&l_3 = l_3'
\end{aligned}$$

In this example, substitution does not provide the required information. Instead, we need a clever approach to rewriting the hypothesis to recognize its applicability to the list segmentation idea to prove this lemma.

By introducing the associativity of the append operator to the context, we can leverage the congruence closure algorithm [20, 35, 25] to establish $l_1 \mathbin{++} l_2 \mathbin{++} l_3 = l_1 \mathbin{++} l_2 \mathbin{++} l'_3$, a custom proof of this can be found below this paragraph. Subsequently, with the idea of list segmentation and adding the length of the lists from Section 3.3.2, we can then finalize the proof. The congruence closure algorithm internally constructs a tree of provable equations. By traversing this tree, we identify equations suitable for applying the list segmentation idea.

$$\dfrac{\dfrac{\dfrac{\dfrac{l_c \mathbin{++} l_d = l_e \mathbin{++} l_f \qquad l_{23} = l_c \mathbin{++} l_d}{l_{23} = l_e \mathbin{++} l_f} \qquad l_e \mathbin{++} l_f = l_2 \mathbin{++} l'_3}{l_{23} = l_2 \mathbin{++} l'_3}}{} \qquad l_1 \mathbin{++} l_2 \mathbin{++} l_3 = l_1 \mathbin{++} l_{23}}{l_1 \mathbin{++} l_2 \mathbin{++} l_3 = l_1 \mathbin{++} l_2 \mathbin{++} l'_3}$$

3.4 Take and drop

Example 3.4.1.

$$\begin{aligned}
&n < m < \mathsf{length}\, l \rightarrow \\
&\mathsf{take}\, n\, l \mathbin{++} \mathsf{drop}\, n\, (\mathsf{take}\, m\, l) \mathbin{++} \mathsf{drop}\, m\, l = l
\end{aligned}$$

To address a specific lemma that cannot be readily solved based on the application of previous methods and ideas, we introduce a novel approach centered on the take and drop operators. The key idea involves the substitution and careful handling of length information, which is based on Lemma 1, given below.

Lemma 1.

$$\begin{aligned}
&l = lt \mathbin{++} ld \rightarrow \\
&\mathsf{length}\, lt = \min(n, \mathsf{length}\, l \rightarrow \\
&\mathsf{length}\, ld = \mathsf{length}\, l - n \rightarrow \\
&lt = \mathsf{take}\, n\, l \wedge ld = \mathsf{drop}\, n\, l
\end{aligned}$$

Essentially, when encountering take and drop we substitute these with new variables, say lt and ld, respectively. Simultaneously, we rewrite all occurrences of l using $l = lt \mathbin{++} ld$, supported by the lemma

$$l = \mathsf{take}\, n\, l \mathbin{++} \mathsf{drop}\, n\, l.$$

Additional context includes the preservation of length information via $\mathsf{length}\, lt = \min(n, \mathsf{length}\, l)$ and $\mathsf{length}\, ld = \mathsf{length}\, l - n$.

In Example 3.4.1, we first substitute $\mathsf{take}\, n\, l$ with lt, $\mathsf{drop}\, n\, l$ with ld and then rewrite $l = lt \mathbin{++} ld$. After this substitution and rewriting, we get the new goal.

$$\frac{\begin{array}{c}\mathrm{H} : n < m < \mathsf{length}\, lt + \mathsf{length}\, ld \\ \mathrm{Hlt} :\ \mathsf{length}\, lt = n \\ \mathrm{Hld} :\ \mathsf{length}\, ld = \mathsf{length}\, lt + \mathsf{length}\, ld - n\end{array}}{lt \mathbin{++} \mathsf{drop}\, n\, (\mathsf{take}\, m\, lt \mathbin{++} ld) \mathbin{++} \mathsf{drop}\, m\, lt \mathbin{++} ld = lt \mathbin{++} ld}$$

If we now apply the operator rearrangement lemmas and simplify the length, we get the new goal:

$$\frac{\begin{array}{c}\mathrm{H} : n < m < \mathsf{length}\, lt + \mathsf{length}\, ld \\ \mathrm{Hlt} :\ \mathsf{length}\, lt = n \\ \mathrm{Hld} :\ \mathsf{length}\, ld = \mathsf{length}\, lt + \mathsf{length}\, ld - n\end{array}}{\begin{array}{c}lt \mathbin{++} \mathsf{drop}\, n\, (\mathsf{take}\, m\, lt) \mathbin{++} \\ \mathsf{drop}\, (n - \mathsf{length}\, (\mathsf{take}\, m\, lt))\, (\mathsf{take}\, (m - \mathsf{length}\, lt)\, ld) \\ \mathbin{++} \mathsf{drop}\, m\, lt \mathbin{++} \mathsf{drop}\, (m - \mathsf{length}\, lt)\, ld = lt \mathbin{++} ld.\end{array}}$$

Using the rewrite rules from Figure 3.1 and linear arithmetic, we can see that

$$\begin{aligned}\mathsf{drop}\, n\, \mathsf{take}\, m\, lt &= [\,] \\ \mathsf{drop}\, m\, lt &= [\,]\end{aligned}$$

and

$$\begin{aligned}&\mathsf{drop}\, (n - \mathsf{length}\, (\mathsf{take}\, m\, lt))\, (\mathsf{take}\, (m - \mathsf{length}\, lt)\, ld) \\ &= \mathsf{take}\, (m - \mathsf{length}\, lt)\, ld.\end{aligned}$$

Therefore, we end up with the goal:

$$\frac{\begin{array}{c}\mathrm{H} : n < m < \mathsf{length}\, lt + \mathsf{length}\, ld \\ \mathrm{Hlt} :\ \mathsf{length}\, lt = n \\ \mathrm{Hld} :\ \mathsf{length}\, ld = \mathsf{length}\, lt + \mathsf{length}\, ld - n\end{array}}{lt \mathbin{++} \mathsf{take}\, (m - \mathsf{length}\, lt)\, ld \mathbin{++} \mathsf{drop}\, (m - \mathsf{length}\, lt)\, ld = lt \mathbin{++} ld.}$$

In this goal, we can again apply the a substituting for take and drop. We substitute $\mathsf{take}\, (m - \mathsf{length}\, lt)\, ld$ with lt_0 and $\mathsf{drop}\, (m - \mathsf{length}\, lt)\, ld$ with ld_0. When doing this in the goal and rewriting $ld = lt_0 \mathbin{++} ld_0$, we get the goal $lt \mathbin{++} lt_0 \mathbin{++} ld_0 = lt \mathbin{++} lt_0 \mathbin{++} ld_0$ and see that we can prove the lemma in this way.

Chapter 4

Formal presentation of the system

In this chapter, we will provide the fragment of the theory of lists on which we would like to prove propositions automatically with our solver. This fragment of the theory is discussed in Section 4.1. In Section 4.2, 4.3, 4.4 and 4.5, we give a more formal definition of the ideas from the previous chapter, which will lead to the inference system that is given in Figure 4.1. Subsequently, in Section 4.6, we provide an overview of how the ideas work together and in which order the solver will perform the rules.

4.1 Formal theory

We consider the following grammar, which defines the lists and propositions that we aim to prove with our solver.

Definition 1 (Goal Grammar)**.**

$$\begin{aligned}
t_{nat} \in \mathit{Term}_{nat} &::= m \mid v \mid t_{nat} + t_{nat} \mid \mathsf{length}\; t_{list_\alpha} \\
&\quad \mathit{where}\ m \in \mathit{Con}_{nat}\ \&\ v \in \mathit{Var}_{nat} \\
t_\alpha \in \mathit{Term}_\alpha &::= x \mid \mathsf{nth}\, t_{nat}\, t_{list_\alpha}\ \ \mathit{where}\ x \in \mathit{Var}_\alpha \\
t_{list_\alpha} \in \mathit{Term}_{list_\alpha} &::= [\,] \mid [t_\alpha] \mid t_{list_\alpha} \mathbin{++} t_{list_\alpha} \mid \mathsf{rev}\; t_{list_\alpha} \mid \mathsf{map}\, f\; t_{list_\alpha} \\
&\quad \mid \mathsf{drop}\, t_{nat}\; t_{list_\alpha} \mid \mathsf{take}\, t_{nat}\; t_{list_\alpha} \mid \mathsf{repeat}\, t_{nat}\, t_\alpha \mid l \\
&\quad \mathit{where}\ f \in (\mathit{Var}_\alpha \rightarrow \mathit{Var}_\alpha) \\
&\quad \mathit{and}\ l \in \mathit{Var}_{list_\alpha} \\
A \in \mathit{Atom} &::= t_{nat} = t_{nat} \mid t_{list_\alpha} = t_{list_\alpha} \\
\Delta \in \mathit{Ctx} &::= \ \mid \Delta, A \\
G \in \mathit{Goal} &::= A \mid A \rightarrow G
\end{aligned}$$

The notation Con_{nat} means constant naturals, so Con_{nat} is the set that contains $0, 1, 2, \ldots$.

The propositional goals given by us consist of implications involving equalities. The equalities may involve terms with natural numbers as their type or terms with a list type.

In our grammar of Definition 1, we have three types of variables: natural numbers (Var_{nat}), lists (Var_{list_α}) and elements of an abstract type (x in $[x]$). When formulating a proposition within this grammar, the variables within the proposition are implicitly universally quantified.

Example 4.1.1 (Bound variables)**.** If we write

$$l_1 = l_2 \to [x] \mathbin{++} \mathsf{rev}\, l_1 = [x] \mathbin{++} \mathsf{rev}\, l_2,$$

we mean the proposition

$$\forall\ (x\ :\ A), (l_1, l_1\ :\ list_\alpha).\ l_1 = l_2 \to [x] \mathbin{++} \mathsf{rev}\, l_1 = [x] \mathbin{++} \mathsf{rev}\, l_2.$$

4.1.1 Proof judgement and inference rules

To analyze the goals outlined in Definition 1 and illustrate the steps in the proof, we will use proof judgments and inference rules. The collective set of inference rules constitutes an inference system.

In a proof judgement, we differentiate between the context and the proposition that we aim to prove. The context consists of axioms and hypotheses assumed to be true. The grammar of a context is also defined in 1 as *Ctx*.

We denote the judgement as $\Delta \vdash Q$, where Δ represents the context, while Q is the proposition we aim to prove in this context. We also refer to Q as the goal.

In the judgments, we apply inference rules as depicted in Figure 4.1. These rules are used to prove various theorems and lemmas concerning lists, which we will discuss in the following sections.

Explanation of the inference rules

In general, an inference rule,

$$\frac{A_1\ A_2\ \cdots\ A_n}{\mathrm{B}}\ \text{name},$$

indicates that to prove of satisfy the goal B, it is sufficient to prove or satisfy $A_1 A_2$, up to A_n.

For instance, the introduction rule is expressed as

$$\frac{\Delta, A \vdash Q}{\Delta \vdash A \to Q}\ \text{intro}$$

$$\frac{\overline{e_1} = \overline{e_2}, \psi_1, \Delta \vdash Q \qquad \cdots \qquad \overline{e_1} = \overline{e_2}, \psi_n, \Delta \vdash Q \qquad \psi_1 \lor \cdots \lor \psi_n}{e_1 = e_2, \Delta \vdash Q} \; \textit{list\_normal\_form}_{\textit{hyps}}$$

$$\frac{\psi_1, \Delta \vdash \overline{e_1} = \overline{e_2} \qquad \cdots \qquad \psi_n, \Delta \vdash \overline{e_1} = \overline{e_2} \qquad \psi_1 \lor \cdots \lor \psi_n}{\Delta \vdash e_1 = e_2} \; \textit{list\_normal\_form}_{\textit{goal}}$$

$$\frac{|e_1|_{\textit{nf}} = |e_2|_{\textit{nf}}, \Delta \vdash Q}{e_1 = e_2, \Delta \vdash Q} \; \textit{length\_normal\_form}_{\textit{hyps}}$$

$$\frac{\Delta \vdash |e_1|_{\textit{nf}} = |e_2|_{\textit{nf}}}{\Delta \vdash e_1 = e_2} \; \textit{list\_normal\_form}_{\textit{goal}}$$

$$\frac{\begin{array}{c} |e_1|_{\textit{nf}} = |e_2|_{\textit{nf}} \notin \Delta \\ |e_1|_{\textit{nf}} = |e_2|_{\textit{nf}}, e_1 = e_2, \Delta \vdash Q \end{array}}{e_1 = e_2, \Delta \vdash Q} \; \textit{add\_length\_hyps}$$

$$\frac{\begin{array}{c} \overline{\mathsf{rev}\, e_1} = \overline{\mathsf{rev}\, e_2} \notin \Delta \\ \overline{\mathsf{rev}\, e_1} = \overline{\mathsf{rev}\, e_2}, \overline{e_1} = \overline{e_2}, \Delta \vdash Q \end{array}}{\overline{e_1} = \overline{e_2}, \Delta \vdash Q} \; \textit{add\_rev\_hyps}$$

$$\frac{\begin{array}{c} \Delta \vdash e_1 \mathbin{++} e_2 \approx e_3 \mathbin{++} e_4 \qquad \Delta \vdash |e_1|_{\textit{nf}} = |e_3|_{\textit{nf}} \\ e_1 = e_3, e_2 = e_4, \Delta \setminus e_1 \mathbin{++} e_2 = e_3 \mathbin{++} e_4 \vdash Q \end{array}}{\Delta \vdash Q} \; \textit{list\_segmentation}$$

Where $\approx$ defines a congruence equivalence relation
(i.e. we can prove it with the `congruence` tactic).

$$\frac{\Delta \vdash |\mathsf{length}\, l|_{\textit{nf}} = 0 \qquad \Delta[l := \mathsf{nil}] \vdash Q[l := \mathsf{nil}]}{l, \Delta \vdash Q} \; \textit{length\_zero}$$

$$\frac{\begin{array}{c} \Delta \vdash |\mathsf{length}\, l|_{\textit{nf}} < 2 \qquad \exists \varphi,\ \varphi(\mathsf{rev}\, l) \in \Delta \cup Q \\ \Delta[\mathsf{rev}\, l := l] \vdash Q[\mathsf{rev}\, l := l] \end{array}}{\Delta \vdash Q} \; \text{rev\_lt\_2}$$

$$\frac{\Delta \vdash e_1 = e_3 \qquad \Delta \vdash e_2 = e_4}{\Delta \vdash e_1 \mathbin{++} e_2 =_{\textit{assoc}} e_3 \mathbin{++} e_4} \; \text{goal\_segmentation}$$

$$\frac{\begin{array}{c} \Big(\mathsf{length}\, lt = |\mathsf{take}\, n\, l|_{\textit{nf}}, \mathsf{length}\, ld = |\mathsf{drop}\, n\, l|_{\textit{nf}}, \Delta \vdash Q\Big) \begin{bmatrix} \mathsf{take}\, n\, l := lt \\ \mathsf{drop}\, n\, l := ld \\ l := lt \mathbin{++} ld \end{bmatrix} \\ \text{Where } \mathsf{take}\, n\, l \text{ or } \mathsf{drop}\, n\, l \text{ is occurring (as a subterm) in } \Delta \text{ or } Q \end{array}}{\Delta \vdash Q} \; \textit{take\_drop\_subst}$$

$$\frac{\begin{array}{c} \Delta \vdash n = m \qquad (\Delta \vdash Q)[n := m] \\ \text{Where } \mathsf{take}\, n\, l \text{ and } \mathsf{take}\, m\, l \text{ or} \\ \mathsf{drop}\, n\, l \text{ and } \mathsf{take}\, m\, l \text{ or } \mathsf{drop}\, n\, l \text{ and } \mathsf{drop}\, m\, l \\ \text{are occurring (as a subterm) in } \Delta \text{ or } Q \end{array}}{\Delta \vdash Q} \; \textit{eq\_take\_drop\_indices}$$

Figure 4.1: Inference rules

This rule indicates that in order to prove $A \to Q$ within the context Δ, it is sufficient to prove that Q holds in the context Δ, A.

4.2 List normal form

By rewriting the rules from Figure 3.1 and Figure 3.2 from left to right, we can formally define a conditional term rewrite system [9], which from now on we will simply refer to as a (conditional) rewrite system.

The rewrite system defined in this way is always terminating[1], which means that by continuously applying the rewrite system we will eventually reach a normal form, i.e. an expression to which we cannot apply any rewrite rule.

Because we deal with a conditional rewrite system, a normal form depends on the context. For example, the term $\mathsf{take}\, n\, l$ in the judgment $\vdash \mathsf{take}\, n\, l = l'$ is a normal form, but in the judgement $n = 0 \vdash \mathsf{take}\, n\, l = l'$ it is not a normal form, because in this context we can rewrite $\mathsf{take}\, n\, l$ to $[\,]$. Given a context Δ of a proof judgement and a expression e, we write $\Delta \vdash e \rightsquigarrow \overline{e}$ to mean that $\overline{e}$ is a normal form of e in context Δ. To simplify the notation of the coming inference rules, when we write one of the following judgments in the inference rules

$$\overline{e} = x, \Delta \vdash Q$$

$$\Delta \vdash \overline{e} = x,$$

we will mean the following

$$\overline{e} = x, \Delta \vdash Q \wedge \Delta \vdash e \rightsquigarrow \overline{e}$$

$$\Delta \vdash \overline{e} = x \wedge \Delta \vdash e \rightsquigarrow \overline{e}.$$

We conjecture that the normal form is unique, given a context, but proving this is future work. Since this has not been proven, we talk about 'a' normal form, instead of 'the' normal form.

In Figure 3.2 we have multiple piecewise-defined rewrite rules, which are all of the form

$$a = \begin{cases} b & \text{if } \psi \\ c & \text{if } \neg\psi. \end{cases}$$

In the case that we are rewriting a and we can not determine that ψ of $\neg\psi$ holds in the context, we will do a case disinction. That is, we will split the goal into two subgoals, adding ψ to the context of the first subgoal, and $\neg\psi$ to the context of the second subgoal.

For our inference system, we utilize two inference rules that transform expressions into a normal form; one for expressions in the context and another

[1] This is proven by the program AProVE [28], and the proof can be found as part of our supplementary material [29]

for expressions in the goal. When rewriting, it can be the case that we have to deal multiple times with piecewise-defined rewrite rules in one expression. We capture this in one inference rule where we can get multiple subgoals and use the notation ψ_i to describe the conjunction of the conditions that we added to the context to rewrite the expression in the new form. If we did not apply piecewise-defined rewrite rules, we will omit ψ_i. The two inference rules are as follows.

$$\frac{\overline{e_1} = \overline{e_2}, \psi_1, \Delta \vdash Q \qquad \cdots \qquad \overline{e_1} = \overline{e_2}, \psi_n, \Delta \vdash Q \qquad \psi_1 \vee \cdots \vee \psi_n}{e_1 = e_2, \Delta \vdash Q}\ \mathit{list\_normal\_form_{hyps}}$$

$$\frac{\psi_1, \Delta \vdash \overline{e_1} = \overline{e_2} \qquad \cdots \qquad \psi_n, \Delta \vdash \overline{e_1} = \overline{e_2} \qquad \psi_1 \vee \cdots \vee \psi_n}{\Delta \vdash e_1 = e_2}\ \mathit{list\_normal\_form_{goal}}$$

4.3 Reverse list assumptions

For the key idea of Section 3.2 we add the following inference rule.

$$\frac{\begin{array}{c}\overline{\mathsf{rev}\, e_1} = \overline{\mathsf{rev}\, e_2} \notin \Delta \\ \overline{\mathsf{rev}\, e_1} = \overline{\mathsf{rev}\, e_2}, \overline{e_1} = \overline{e_2}, \Delta \vdash Q\end{array}}{\overline{e_1} = \overline{e_2}, \Delta \vdash Q}\ \mathit{add\_rev\_hyps}$$

We introduce a normal form of the reverse if this is not already in context. Checking if the normal form is already present in the context, consequently causing this procedure to terminate if we repeatedly apply it, due to the fact that $\mathsf{rev}\,(\mathsf{rev}\, l) = l$.

4.4 List segmentation

For list segmentation, we define the following inference rule.

$$\frac{\begin{array}{c}\Delta \vdash e_1 +\!\!+\, e_2 \approx e_3 +\!\!+\, e_4 \qquad \Delta \vdash |e_1|_{nf} = |e_3|_{nf} \\ e_1 = e_3, e_2 = e_4, \Delta \setminus e_1 +\!\!+\, e_2 = e_3 +\!\!+\, e_4 \vdash Q\end{array}}{\Delta \vdash Q}\ \mathit{list\_segmentation}$$

In this inference rule $\approx$ defines a congruence equivalence relation (i.e. we can prove it with the `congruence` tactic). In Section 4.4.3 we take a closer look at how the congruence closure algorithm works.

In the condition $\Delta \vdash |e_1|_{nf} = |e_3|_{nf}$ of the inference rule, we verify if the length of the prefix of the list we found by the congruence closure algorithm is equal. We put the length in a normal form as we will explain in Section 4.4.1. In Section 4.4.2 we will explore how we can extend the context with assumptions about lengths, in order to prove additional length equalities.

4.4.1 Length normal form

$$\frac{|e_1|_{nf} = |e_2|_{nf}, \Delta \vdash Q}{e_1 = e_2, \Delta \vdash Q}\ \mathit{length\_normal\_form}_{hyps}$$

$$\frac{\Delta \vdash |e_1|_{nf} = |e_2|_{nf}}{\Delta \vdash e_1 = e_2}\ \mathit{list\_normal\_form}_{goal}$$

With the rewrite rules for the length from Section 3.3.1 we can formally define a terminating[2] rewrite system for a normal form of the length of a lists. With the same idea as in Section 4.2 we can create two inference rules to rewrite the expressions in the context and the goal in a normal form. In case of the normal form of the length, we adopt the notation $|e|_{nf}$, for a normal form of expression e.

4.4.2 Length of lists assumptions and lia

$$\frac{\begin{array}{c}|e_1|_{nf} = |e_2|_{nf} \notin \Delta \\ |e_1|_{nf} = |e_2|_{nf}, e_1 = e_2, \Delta \vdash Q\end{array}}{e_1 = e_2, \Delta \vdash Q}\ \mathit{add\_length\_hyps}$$

As presented in Section 3.3.2, if we know that two lists are equal, we also know that the lists have the same length. Because we need information about the length of the list to apply list segmentation, we will integrate this information to our context. This provides information that will allow `lia` to prove that the prefixed of a list are of equal length.

In the inference rule, we verify whether the equality is not already present in the context. This test is implemented in order to ensure termination when repeatedly applying the inference rule.

4.4.3 Congruence

We present the inference rules utilized in the congruence closure algorithm in Figure 4.2. Congruence closure establishes an equivalence relation by enforcing symmetry, transitivity, and reflexivity through application of the first three rules. Additionally, it utlizes the property that functions map elements from the domain to unique elements in the codomain. In our case, this property applies to the functions `rev` and $++$. The congruence relation initially incorporates the equalities in the context, leading to rule 6 in the congruence closure inference system outlined in Figure 4.2. The last rule is specific to lists and results from the E-match [23] algorithm implemented in the Coq proof assistant. We include this rule as the E-match algorithm in Coq attempts to generate new terms by utilizing universally quantified

[2] This is also proven by the program AProVE [30]

$$1.\quad \frac{}{\Delta \vdash e_1 \approx e_1}$$

$$2.\quad \frac{\Delta \vdash e_1 \approx e_2 \qquad \Delta \vdash e_2 \approx e_3}{\Delta \vdash e_1 \approx e_3}$$

$$3.\quad \frac{\Delta \vdash e_1 \approx e_1}{\Delta \vdash e_2 \approx e_1}$$

$$4.\quad \frac{\Delta \vdash e_1 \approx e_2}{\Delta \vdash \mathsf{rev}\, e_1 \approx \mathsf{rev}\, e_2}$$

$$5.\quad \frac{\Delta \vdash e_1 \approx e_3 \qquad \Delta \vdash e_2 \approx e_4}{\Delta \vdash e_1 +\!\!+\; e_2 \approx e_3 +\!\!+\; e_4}$$

$$6.\quad \frac{}{\Delta, e_1 = e_2 \vdash e_1 \approx e_2}$$

$$7.\quad \frac{}{\Delta \vdash (e_1 +\!\!+\; e_2) +\!\!+\; e_3 \approx e_1 +\!\!+\; (e_2 +\!\!+\; e_3)}$$

Figure 4.2: Inference rules for congruence

hypotheses. Introducing the associative property of append, i.e., $\forall l_1, l_2, l_3 : \mathit{list}_\alpha.\ (l_1 +\!\!+\; l_2) +\!\!+\; l_3 = l_1 +\!\!+\; (l_2 +\!\!+\; l_3)$, provides our congruence system with this unique rule.

4.5 Take and drop

For the substitution of take and drop as outlined in Section 3.4, we require substitution. We notate $(\Delta \vdash Q)[x := y]$ in order to express that we will replace all the occurrences of x in Δ and Q by y. The inference rule is presented below. In the inference rule, lt and ld are fresh variables, meaning that they may not occur in the context.

$$\frac{\begin{array}{c}\left(\mathsf{length}\, lt = |\mathsf{take}\, n\, l|_{nf}, \mathsf{length}\, ld = |\mathsf{drop}\, n\, l|_{nf}, \Delta \vdash Q\right)\begin{bmatrix}\mathsf{take}\, n\, l := lt \\ \mathsf{drop}\, n\, l := ld \\ l := lt +\!\!+\; ld\end{bmatrix} \\ \text{Where } \mathsf{take}\, n\, l \text{ or } \mathsf{drop}\, n\, l \text{ is occurring (as a subterm) in } \Delta \text{ or } Q\end{array}}{\Delta \vdash Q}\ \mathit{take\_drop\_subst}$$

4.6 All together

In addition to the inference rules for the key ideas, we also have some less complex ideas for which we will present the inference rules in Section 4.6.1. In Section 4.6.2 we will explain and motivate in which order the ideas are applied to form the solver.

4.6.1 Some other inference rules

Equal take & drop indices

To decrease the number of times we have to apply the method from Section 4.5, we can look if there are indices of the take and drop operators that are equal to Presburger arithmetic, by using the ‘lia’ tactic. We need inference rules for every combination of the operator take and drop as shown below.

$$\dfrac{\begin{array}{c}\Delta \vdash n = m \qquad (\Delta \vdash Q)[n := m] \\ \text{Where } \mathsf{take}\, n\, l \text{ and } \mathsf{take}\, m\, l \text{ or} \\ \mathsf{drop}\, n\, l \text{ and } \mathsf{take}\, m\, l \text{ or } \mathsf{drop}\, n\, l \text{ and } \mathsf{drop}\, m\, l \\ \text{are occurring (as a subterm) in } \Delta \text{ or } Q\end{array}}{\Delta \vdash Q}\ \mathit{eq\_take\_drop\_indices}$$

List with length zero

When acquiring information about the length of the lists, it is possible that we obtain that the length of certain list expressions is zero. In such cases we can conclude that this expression has to be equal to $[\,]$, therefore we use the following inference rule.

$$\dfrac{\Delta \vdash |\mathsf{length}\, l|_{nf} = 0 \qquad (\Delta \vdash Q)[l := \mathsf{nil}]}{l, \Delta \vdash Q}\ \mathit{length\_zero}$$

Revert of singleton and empty list

The revert operator has a special property on a singleton list. We observe that $\mathsf{rev}\,[x] = [x]$. We combine this with the rewrite rule for an empty list, namely that $\mathsf{rev}\,[\,] = [\,]$ and we get the following inference rule.

$$\dfrac{\begin{array}{c}\Delta \vdash |\mathsf{length}\, l|_{nf} < 2 \qquad \exists \varphi,\ \varphi(\mathsf{rev}\, l) \in \Delta \cup Q \\ \Delta[\mathsf{rev}\, l := l] \vdash Q[\mathsf{rev}\, l := l]\end{array}}{\Delta \vdash Q}\ \mathit{rev\_lt\_2}$$

Goal segmentation

To finalize the proofs, we use a segmentation technique for the goal. The idea is $l_1 = l_3 \wedge l_2 = l_4 \to l_1 \mathbin{++} l_2 = l_3 \mathbin{++} l_4$. We capture this in the following inference rule.

$$\dfrac{\Delta \vdash e_1 = e_3 \qquad \Delta \vdash e_2 = e_4}{\Delta \vdash e_1 \mathbin{++} e_2 =_{assoc} e_3 \mathbin{++} e_4}\ \mathit{goal\_segmentation}$$

4.6.2 Order of the operations

In the preceding sections, we have presented methods with their inference rules to ultimately obtain a proof. The inference system constructed by the inference rules is non-deterministic. To reduce the search domain and improve efficiency, we introduce an order in which the rules are applied. The order in which we apply the methods is the following:

1. Equal `take` & `drop` indices (Section 4.6.1)
2. Substitute `take` & `drop` (Section 4.5)
3. Length of list assumptions (Section 4.4.2)
4. List with length zero (Section 4.6.1)
5. Repeat segmentation and reverse list assumptions (Section 4.3 and Section 4.4)
6. Revert of singleton and empty list (Section 4.6.1)
7. Goal segmentation (Section 4.6.1)

An important note here is that all the methods are applied as much as possible, and before every method is applied we verify that everything is in list normal form and length normal form.

To minimize the number of times that we have to apply a substitution in step 2, we start by applying step 1, checking if the indices of `take` and `drop`. So step 1 have to be applied before step 2.

The information regarding the length of a list being zero, which we need in step 4, can be deduced from getting the length of the list assumptions that we get with the method from Section 4.4.2 that is applied in step 3. So step 3 have to be applied before step 4.

Because the substitution of `take` and `drop` can create variables with length zero, we check this afterward and apply step 4 after step 2. The $[\,]$ will be removed in an append operator when writing it in normal form, this will simplify the expressions. Therefore, we have chosen to apply step 4 as soon as possible. We have seen that step 1 has to be applied before step 2 and step 2 and 3 have to be applied before step 4. Therefore, this is as soon as possible for step 4.

It is important that we have added the length assumptions before we apply the segmentation, because the information of lengths is a fundamental property for applying the segmentation as mentioned in Section 3.3.2 and Section 4.4.2. So step 3 have to be applied before step 5.

It is also required to apply the `take` and `drop` substitutions (step 2) before applying segmentation (step 5). This is because after applying the segmentation we have general list variables, instead of the take and drop operators, on which we can apply the segmentation.

The list segmentation and adding the reverse assumptions are repeatedly applied after each other, until no new information is provided. This is because applying list segmentation gives us a new hypothesis from which we can possibly adopt a new reverse assumption, and with new reverse assumptions, the congruence closure algorithm may be able to find a new equation.

The revert singleton method from Section 4.6.1 in step 6 is applied after we are done with reverting assumptions in step 5, because then there will not appear new reverse operators.

The goal segmentation is the last step and is used to finalize the proof. The goal segmentation and the revert singleton method are interchangeable. Although there are interchangeable in the inference system, it is more easy to implement it in this order as we will see in 5.

There is no argument for applying step 3, after step 1 and 2. Instead of using order 1, 2, 3 of the steps, we could also apply step 3 earlier and use ordering 1, 3, 2 or 3, 1, 2 of the steps. We have chosen the ordering as we did here to continue with in the implementation, but we have experimented with another ordering in the implementation and evaluated this with the method we use in 6, and did not notice any significant differences. The only differences we saw were small differences in computation time.

We conjecture that, besides the segmentation, everything preserves completeness. The segmentation will not preserve completeness, because it takes a depth argument in the implementation, which restricts the number of times that we can apply the seventh rule of Figure 4.2. The question of whether we can make this complete is future work. Furthermore, to preserve completeness, we require the normal form to be unique, which is, as mentioned before, also is future work.

Chapter 5

Implementation

In this chapter, we will demonstrate how we implement the list solver. We provide the implementation sequence of the key ideas and inference rules we have seen in the previous chapters, and additionally we introduce new operations that the solver applies, in order to obtain a more precise overview of the solvers implementation.

In Figure 5.1 flowchart illustrates the process of our solver. States with the same color will be covered in the same section, as indicated at the bottom of the figure.

At the beginning of the flowchart we observe the states `intro` and `subst` without any color. Our solver starts by applying the `intros` and `subst` tactics that are implemented in Coq by default [39]. These two steps are not discussed in this chapter as they are standard procedure in developing Coq proofs.

The flowchart follows a sequence similar to the one presented in Section 4.6.2. The box marked with the ⟳ symbol at the top-left corner of a box indicates a repeat loop. In the final box, we seek to determine the appropriate segmentation for our goal using backtracking.

We start this chapter with Section 5.1 by discussing the implementation of the normal form, which encompasses both the list normal form and the length normal form. This corresponds to the key idea of Section 3.1 and Section 3.3.1 and the formal presentation of Section 4.2 and Section 4.4.1.

Subsequently, in Section 5.2 we discuss several smaller implementations. We explain how we implemented the new operation to destruct `min` and `max` and how we unfolded the derived operators. Furthermore, we discuss the implementation of two key ideas, adding the length of the list assumptions (from Section 3.3.2 and Section 4.4.2) and the reverse list assumptions (from Sec-

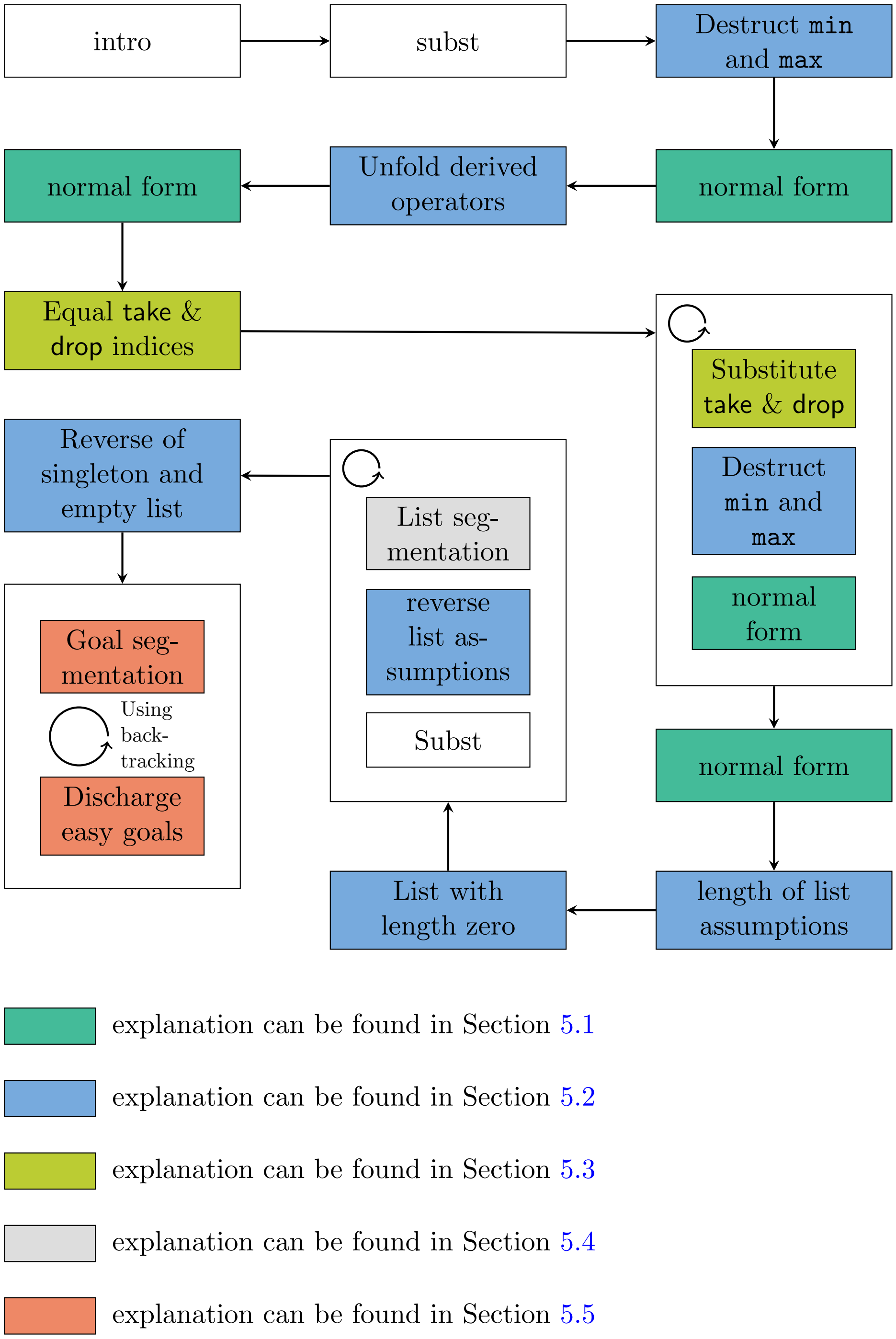
intro
subst
Destruct min and max
normal form
Unfold derived operators
normal form
Equal take & drop indices
Substitute take & drop
Destruct min and max
normal form
normal form
length of list assumptions
List with length zero
List segmentation
reverse list assumptions
Subst
Reverse of singleton and empty list
Goal segmentation
Using back-tracking
Discharge easy goals
explanation can be found in Section 5.1
explanation can be found in Section 5.2
explanation can be found in Section 5.3
explanation can be found in Section 5.4
explanation can be found in Section 5.5

Figure 5.1: Flowchart of our list solver

tion 3.2 and Section 4.3). We examine the implementation of two inference rules from Section 4.6.1, lists with length zero, and the reverse of a singleton list. Finally, we will also look at how we implemented ‘$\not\in \Delta$’ that we have seen in several inference rules of Chapter 4.

In Section 5.3 we cover how we implemented the substitution of take and drop, as outlined in Section 3.4. Additionally, we explore the process of identifying equal indices of take and drop, a concept discussed in Section 4.6.1.

Moving on to section Section 5.4 we address the list segmentation of Section 3.3. This involves modifying Coqs congruence closure plugin to facilitate list segmentation.

Finally, in Section 5.5, we discuss the application of segmentation to the goals as discussed in Section 4.6.1, subsequently demonstrating how to resolve simpler goals after segmentation.

5.1 Normal form

In this section, we present our implementation of the normal form, consisting of both list expressions and expressions pertaining to the length of lists. The implementation approach for the two normal forms is based on the same idea. In both instances, the implementation method consists of two parts. First, we apply simple rewrite rules (without conditions), using Coq’s `rewrite` tactic. Subsequently, in Section 5.1.1 we may require case splitting for piecewise-defined rewrite rules. Although it is possible to employ a more efficient rewrite system than Coq’s `rewrite` tactic, by for example using generalized rewriting [38], this is considered to be future work.

In Section 5.1.1 we reveal how we implemented the length normal form in Ltac. It will become clear where we utilize rewrite rules and where we use case splitting. Subsequently, in Section 5.1.2 we demonstrate our case splitting approach and how we utilize it to achieve efficiency. In Section 5.1.3 we highlight the additional features we have incorporated into our `list_normal_form` tactic.

5.1.1 Length normal form

The length normal form is implemented as follows:

```
Ltac length_normal_form :=
  repeat (rewrite nil_length in *
        || rewrite rev_length in *
        || rewrite app_length in *
        || rewrite drop_length in *
        || rewrite update_length in *
        || rewrite repeat_length in *
        || rewrite map_length in *
        || rewrite singleton_length in *
        || rewrite flip_ends_length in *
        || (rewrite take_length_minimum in *; minimum_destruct)
        ).
```

We start with the rewrite rules of Figure 3.3. After rewriting the length of the take operator, we apply the `minimum_destruct` tactic, which we will discuss in Section 5.1.2, and may require case splitting. We use a newly defined `minimum` operator, instead of the standard `min` operator of Coq. The semantic definitions of the `min` and `minimum` operators are identical, and in Section 5.2.2 we provide justification for this decision.

5.1.2 Careful case splitting

We apply careful case splitting on every piecewise-defined rewrite rule outlined in Figure 3.1 and Figure 3.2. Additionally, we introduce two rewrite rules that are distinct from Figure 3.1. We use the following two rules:

$$\mathsf{take}\,n\,(l_1 \mathbin{++} l_2) = \begin{cases} \mathsf{take}\,n\,l_1 & \text{if } n \leq \mathsf{length}\,l_1 \\ l_1 \mathbin{++} \mathsf{take}\,(n - \mathsf{length}\,l_1)\,l_2 & \text{if } \mathsf{length}\,l_1 < n < \mathsf{length}\,l_1 \mathbin{++} l_2 \\ l_1 \mathbin{++} l_2 & \text{if } \mathsf{length}\,l_1 + \mathsf{length}\,l_2 \leq n \end{cases}$$

$$\mathsf{drop}\,n\,(l_1 \mathbin{++} l_2) = \begin{cases} \mathsf{drop}\,n\,l_1 \mathbin{++} l_2 & \text{if } n \leq \mathsf{length}\,l_1 \\ \mathsf{drop}\,(n - \mathsf{length}\,l_1)\,l_2 & \text{if } \mathsf{length}\,l_1 < n < \mathsf{length}\,l_1 \mathbin{++} l_2 \\ [\,] & \text{if } \mathsf{length}\,l_1 \mathbin{++} l_2 \leq n \end{cases}$$

We formulated these rules differently, for the sake of efficiency. However, the result remains consistent with the rewrite rules delineated in Section 3.1, along with the rules:

$$\begin{aligned} \mathsf{drop}\,n\,l &= l \text{ if } n = 0 \\ \mathsf{drop}\,n\,l &= [\,] \text{ if } n \geq \mathsf{length}\,l \\ \mathsf{take}\,n\,l &= [\,] \text{ if } n = 0 \\ \mathsf{take}\,n\,l &= l \text{ if } n \geq \mathsf{length}\,l \end{aligned}$$

In the rest of this subsection we will describe the exact implementation of the `minimum_destruct` tactic, and afterwards we will explore the implementation of the rewrite rule for $\mathsf{take}\,n\,(l_1 \mathbin{++} l_2)$.

The `minimum_destruct` tactic is defined as follows:

```
Lemma minimum_case1 n m :
  m ≤ n → minimum n m = m.
```

```
Lemma minimum_case2 n m :
  n ≤ m → minimum n m = n.

Lemma minimum_split n m :
  (n ≤ m ∧ minimum n m = n)
  ∨ (m < n ∧ minimum n m = m).

Ltac minimum_destruct_helper n m :=
  let H' := fresh "H'" in
  (assert (m ≤ n) as H' by lia; apply minimum_case1 in H'; try rewrite H' in ∗; clear H')
  ||  (assert (n ≤ m ) as H' by lia; apply minimum_case2 in H'; try rewrite H' in ∗; clear H')
  ||  (destruct (minimum_split n m) as [[? H']|[? H']];   try rewrite H' in ∗; clear H').

Ltac minimum_destruct :=
  repeat(
  match goal with
    | _ :  context [ minimum ?n ?m ] |− _ ⇒ minimum_destruct_helper n m
    | |− context [ minimum ?n ?m ] ⇒ minimum_destruct_helper n m
  end).
```

We verify the existence of n and m such that `minimum n m` is present in a hypothesis or in the goal. If this is the case, we attempt to determine if we can replace the minimum with n or m by using the lemmas `minimum_case1` and `minimum_case2`. If neither lemma applies, we will create two subgoals by destructing the lemma `minimum_split`, and rewrite the minimum operator in both subgoals.

First, we ascertain the particular case we are dealing with, because case splitting is a time intensive operation. Initial case identification minimizes the need for subsequent case splits, consequently reducing computational overhead. After we have applied a case split, we have two related subgoals. In order to optimize for efficiency, we aim to avoid the application of all subsequent steps of the list solver to both subgoals.

For take n $(l_1 \mathbin{++} l_2)$ we use a similar approach, and this is implemented as follows:

```
Ltac split_take_app_tac_helper n l1 l2 :=
  let H' := fresh "H'" in
  (    (assert (n ≤ length l1) as H' by (length_normal_form_goal; lia);
       apply (take_app_case1 n l1 l2) in H'; try rewrite H' in ∗; clear H')
  ||   (assert (length l1 < n < length l1 + length l2) as H' by (length_normal_form_goal; lia);
       apply (take_app_case2 n l1 l2) in H'; try rewrite H' in ∗; clear H')
  ||   (assert (length l1 + length l2 ≤ n) as H' by (length_normal_form_goal; lia);
       apply (take_app_case3 n l1 l2) in H'; try rewrite H' in ∗; clear H')
  ||   (pose proof (split_take_app n l1 l2) as [[? H']|[[? H']|[?   H']]];
       try rewrite H' in ∗; clear H')
  ).
```

```
Ltac split_take_app_tac :=
  repeat match goal with
  | |− context [ take ?n (?l1 ++ ?l2) ] ⇒ split_take_app_tac_helper n l1 l2
  | _ : context [ take ?n (?l1 ++ ?l2) ] |− _ ⇒ split_take_app_tac_helper n l1 l2
  end.
```

In this example, although we have three cases, the idea remains the same: We attempt to determine the case we are dealing with and apply the corresponding rewrite rule. If we cannot determine this, we differentiate the goal into several subgoals.

Note that we match the goal on two branches, one for the expressions in the hypothesis and one for the expressions in the goal.

5.1.3 List normal form

In addition to the rewrite rules, the normal form for list expressions includes two additional features. We have implemented a simplification for natural numbers, which has led to increased performance, and we implemented the rewriting of $x :: l$ to $[x] ++ l$.

We first look at the simplification for natural numbers, we call this tactic `nat_simp`.

```
Tactic Notation "nat_simp" :=
  repeat match goal with
  | |− context [ ?n − ?n ] ⇒ replace (n − n) with (0) by lia
  | _ : context [ ?n − ?n ] |− _ ⇒ replace (n − n) with (0) by lia
  | |− context [ 0 + ?n ] ⇒ replace (0 + n) with (n) by lia
  | _ : context [ 0 + ?n ] |− _ ⇒ replace (0 + n) with (n) by lia
  | |− context [ ?n + 0 ] ⇒ replace (n + 0) with (n) by lia
  | _ : context [ ?n + 0 ] |− _ ⇒ replace (n + 0) with (n) by lia
  | |− context [ 0 − ?n ] ⇒ replace (0 − n) with (n) by lia
  | _ : context [ 0 − ?n ] |− _ ⇒ replace (0 − n) with (n) by lia
  | |− context [ ?n − 0 ] ⇒ replace (n − 0) with (n) by lia
  | _ : context [ ?n − 0 ] |− _ ⇒ replace (n − 0) with (n) by lia
  end.
```

In this tactic, we aim to replace every addition and subtraction of 0 and if we have some occurrence of $n - n$ for some n, we replace it by 0. This is done to improve efficiency.

For replacing cons with ++ , we require the following lemma:

```
Lemma cons_app {A} (v : A) (l : list A):
  v :: l = [v] ++ l.
```

Repeatedly rewriting with this lemma will not terminate. The reason for this is because the end of a list is marked with $\mathsf{cons}(\_, \mathsf{nil})$, consequently $[v]$

will be rewritten as [v] ++ [ ]. Therefore, we create the following Ltac in which we rewrite carefully and do not apply the lemma to the end of a list. This implementation is as follows:

```
Ltac cons_app_rewrite_helper x xs :=
  lazymatch xs with
  | nil ⇒ fail
  | _ ⇒ replace (x::xs) with ([x] ++ xs) by by rewrite cons_app
end.

Ltac cons_app_rewrite :=
  repeat match goal with
  | |− context [?x::?xs] ⇒ cons_app_rewrite_helper x xs
  | _ : context [?x::?xs] |− _ ⇒ cons_app_rewrite_helper x xs
end.
```

5.2 Small Ltac implementations

In this section, we go over some small Ltac implementations.

5.2.1 Unfold derived operators

Unfolding of the derived operators `update` and `flip_ends` is done by Coqs `unfold` tactic [39]. In order to enhance performance, we intersperse this unfolding with rewriting everything in normal form.

```
Ltac list_solver_unfold :=
  unfold flip_ends in ∗;
  list_normal_form;
  unfold update in ∗.
```

5.2.2 Destructing min and max

During the development of our list solver, we have observed that the operators `min` and `max` can significantly slow down the `lia` tactic. Through experimentation, we have discovered that it is more efficient to apply case analysis, and we hope that we can determine in which case we are, by using a careful case splitting from Section 5.1.2. The tactic is as follows.

```
Ltac min_max_destruct :=
  try rewrite ← min_minimum in ∗;
  try rewrite ← max_maximum in ∗;
  minimum_destruct;
  maximum_destruct.
```

We start by rewriting the operator `min` and `max` with the semantically equivalent operators `minimum` and `maximum` and subsequently employ careful case splitting in order to manage the rewritten operators.

In order to avoid interference from lia in determining cases, we rewrite the operators. The following example illustrates the necessity:

Suppose that we want to prove $a \leq \mathtt{min}(a, b) \to a \leq b$. If we do not modify the operator, lia will know that $a \leq b$ and will rewrite $\min(a, b)$ with a, leading to the new goal $a \leq a \to a \leq b$, which is unprovable. Hence we introduce a new operator that lia will not take into account.

Although none of our rewrite rules generate a max operator, we include the operator for consistency. In particular, this is useful if a user places a max in their lemma, ensuring uniformity.

5.2.3 The clean_hyps tactic

We implement '$\notin \Delta$' utilizing the clean_hyps tactic in conjunction with the repeat tactic, that we have seen in several inference rules of Chapter 4. The clean_hyps tactic removes all duplicate hypotheses, ensuring a hypothesis appears only once. The repeat tactic repeats a tactic until it does not change the proof context (or until the tactic fails).

We often use it in the following order

```
repeat (some_tactic; clean_hyps).
```

Consequently, the tactic some_tactic, will only be executed when it changes the goal or introduces new information to the context. If it adds information that is already present in the hypothesis, it will be removed by clean_hyps, and the repeat tactic will break the loop because the goal we have at the end of the loop is the same as we had at the beginning of the loop, so we do not make progress.

The clean_hyps tactic also removes the reflexive hypothesis to increase the solver performance. The tactic is implemented as follows.

```
Ltac clear_dup :=
  match goal with
    | [ H : ?X |- _ ] =>
      match goal with
        | [ H' : ?Y |- _ ] =>
          match H with
            | H' => fail 2
            | _ => unify X Y ; (clear H' || clear H)
          end
      end
  end.

Ltac clear_refl :=
  match goal with
  | [ H : ?a = ?a |- _ ] =>
```

```
      clear H
  end.

Ltac clean_hyps :=
  repeat clear_refl;
  repeat clear_dup.
```

Here `clear_dub` removes double hypotheses in the context and `clear_refl` will remove the reflexive hypothesis.

5.2.4 Length of list assumptions

For all the hypotheses about lists, we want to add the length as presented in Section 3.3.2 and Section 4.4.2. We implemented this by checking if an hypothesis is an equation with list expressions at both sides and if so, we integrate the corresponding equation about the length. To ensure termination of this process, we revert the hypothesis, which relocates the hypothesis from the list of hypotheses and adds it as an implication to the goal, i.e., we apply the following inference rule.

$$\frac{\Delta \vdash A \to B}{\Delta, A \vdash B}$$

The `length_lists` tactic is used in combination with the `clean_hyps` tactic, to only add a hypothesis that provides new information.

```
Ltac length_lists :=
  repeat match goal with
  | H : ?l1 = ?l2 |- _ =>
        assert (length l1 = length l2) as ?H1;
        [by rewrite H|]; reverse H;
        length_normal_form;
        reverse H1
  end;
  intros.
```

5.2.5 Reverse list assumptions

In the implementation of the reverse list assumptions, we use a definition called `eq_fold`. The `eq_fold` is used to rewrite an equality, so that we can track which equalities we have processed. For every hypothesis, we assert the reverse, and both the original and new hypotheses will be folded in the `eq_fold` definition. When there are no hypotheses left of which we can take the reverse, we unfold all the `eq_fold` definitions, to return the equations in the context. In this way, the names of the hypotheses will remain the same. The `rev_hyp` tactic is also used in combination with the `clean_hyps` tactic, and is implemented as follows:

```
Definition eq_fold (p : Prop) := p.

Lemma eq_fold_rewrite (p : Prop) :
  eq_fold p ↔ p.

Ltac rev_hyp :=
  repeat (let Hrev := fresh "Hrev" in
  match goal with
    | H : ?l1 = ?l2 |- _ ⇒ assert (rev l1 = rev l2) as ?Hrev; [by rewrite H|];
        list_normal_form;
        apply eq_fold_rewrite in H;
        apply eq_fold_rewrite in Hrev
  end); unfold eq_fold in *.
```

5.2.6 List with length zero

In a goal, we can have two types of lists that may have a length of zero. The first is a list variable, and the second is the repeat operator. The implementation is as follows:

```
Ltac repeat_zero_length_then_nil_helper v n :=
  let H' := fresh "H'" in
  assert (length (repeat v n) = 0) as H' by (rewrite repeat_length; lia);
  apply nil_length_inv in H'; rewrite H' in *; clear H'.

Ltac zero_length_then_nil :=
  repeat (
  let H' := fresh "H'" in
  match goal with
    | l : (list _) |- _ ⇒
      assert (length l = 0) as H' by lia; apply nil_length_inv in H'; subst
    | |- context [repeat ?v ?n] ⇒
      repeat_zero_length_then_nil_helper v n
    | _ : context [repeat ?v ?n] |- _ ⇒
      repeat_zero_length_then_nil_helper v n
  end).
```

When constructing a proof with Coq, we have all the variables that we consider in the context. For every list variable, we check if we can prove that the length is equal to zero by using the `lia` tactic. If we can obtain this proof, we will rewrite the list as [ ].

The occurrence of the repeat operator has to be checked in the context and the goal. We try to prove that some occurrence of repeat $n\,v$ has length zero, by rewriting it with the rewrite rules for the length. Subsequently, we only need to prove that n is equal to zero, which we can do using `lia`. If we can prove $n = 0$, we will rewrite repeat $n\,v$ as nil.

5.2.7 Reverse of singleton and empty list

If we take the reverse of some list, we will check if we can omit the rev operator, by checking if the length of the list on which the rev operator applies is less than or equal to 2. We can justify this with the fact that the reverse of the empty list is the empty list, and the reverse of a singleton list is the lists itself. The implementation is:

```
Lemma len_le_2_rev {A} (l : list A) :
  length l < 2 →
  rev l = l.

Ltac len_le_2_rev_tac_helper l :=
  let H' := fresh "H'" in
  assert (length l < 2) as H' by (length_normal_form_goal; lia);
  apply len_le_2_rev in H'; rewrite H' in ∗; clear H'.

Ltac len_le_2_rev_tac :=
  match goal with
  | |- context [rev ?l] ⇒ len_le_2_rev_tac_helper l
  | _ : context [rev ?l] |- _ ⇒ len_le_2_rev_tac_helper l
  end.
```

To be able to apply the lemma `len_le_2_rev`, we assert the length of the expression in the helper tactic and put the expression in normal form of the length. We try to finish the proof using `lia`.

5.3 Substitute take and drop

In this section, we will explain how we have implemented the substitution of take and drop. Before applying the substitution, we verify whether there are equal indices of the take and drop operators, so that we can rewrite this with the same expression. This is done to reduce the workload in the later substitution because we get fewer equalities in the context. An additional benefit of having few equations in our context is that not only `lia` becomes faster, but we know that we can prevent `lia` from raising a stack overflow. We noticed that if too many comparisons are included in the context, lia can raise a stack overflow.

We determine if the indices are the same using `lia`. We only have to verify if the indices are syntactically the same, if the indices are not semantically the same, for instance, we do have to check that $i = i + 0$, but $i = i$ does not have to be checked because both sides of the equal sign are semantically the same. We accomplish this using a `lazymatch`. We have to compare every take and drop in the hypothesis and goal. Here, we give a fragment of the implementation of how we compare the indices of the take operators that are in the goal.

```
Ltac lia_eq_take_l :=
  let H := fresh "H" in
  multimatch goal with
    | |- context [take ?i ?l] ⇒
        multimatch goal with
          | |- context [take ?j l] ⇒ assert (i = j) as H by lia;
                lazymatch goal with
                  | H : (i = i) |- _ ⇒ fail
                  | H : _ |- _ ⇒
                      assert (take i l = take j l) as ← by (by rewrite H);
                      clear H
                end
        end
    ...
  end.
```

After we have checked all the indices of the take and drop operators, we will substitute the operators. We first match the goal with the occurrences of take and drop by using the following Ltac code:

```
Ltac take_drop_generalize :=
  match goal with
    | |- context [ take ?n ?l ] ⇒ take_drop_subst_n n l
    | _ : context [ take ?n ?l ] |- _ ⇒ take_drop_subst_n n l
    | |- context [ drop ?n ?l ] ⇒ take_drop_subst_n n l
    | _ : context [ drop ?n ?l ] |- _ ⇒ take_drop_subst_n n l
  end.
```

After identifying any take $n\, l$ or drop $n\, l$ expressions in the hypothesis or goal, we apply the real substitution tactic. We replace all instances of take $n\, l$ with a new variable `lt` and we will replace all representations of drop $n\, l$ by a new variable `ld`. Additionally, we store the information about the length of the lists in the context. We do this with the following Ltac code:

```
Lemma take_drop_segment {A} n (l : list A) (P : list A →list A →Prop) :
  (forall lt ld,
    length lt = minimum n (length l) →
    length ld = length l − n →
    l = lt ++ ld →
    P lt ld) →
  P (take n l) (drop n l).

Ltac take_drop_subst_n n l :=
let H' := fresh "H'" in
let lt := fresh "lt" in
let ld := fresh "ld" in
generalize dependent l;
intros l;
pattern (take n l);
pattern (drop n l);
apply (take_drop_segment n l);
intros lt ld ?Hlt ?Hld H';
intros;
rewrite → H' in *;
clear H'.
```

We start this process by choosing new names for variables `lt` and `ld`. Subsequently, with revert every hypothesis that contains variable `l` by using `generalize dependent l`. Using the `pattern` tactic, we perform a β-expansion with the terms $\mathsf{take}\, n\, l$ and $\mathsf{drop}\, n\, l$. On this β-expansion, we can apply the `take_drop_segment` lemma, to replace take and drop with variables and store the information about the lengths. Finally, we only have to introduce the new and reverted information into our context and rewrite all the occurrences of `l` with `lt` ++ `ld`.

5.4 List segmentation

In this section, we explain how we implemented the list segmentation. Recall that the idea of the segmentation idea is that we try to find an equation $l_{11} \mathbin{++} l_{12} = l_{21} \mathbin{++} l_{22}$ by using the congruence closure so we can apply the following lemma:

$$\mathsf{length}\, l_{11} + \mathsf{length}\, l_{21} \to l_{11} \mathbin{++} l_{12} = l_{21} \mathbin{++} l_{22} \to l_{11} \mathbin{++} l_{21} \land l_{12} = l_{22}.$$

We modified Coq's congruence closure plugin to create a new tactic, which is implemented in the OCaml programming language [39]. We will start by providing a brief overview of how the congruence closure algorithm works, followed by how we implemented our plugin and how this plugin works with the Ltac code to apply segmentation.

5.4.1 The congruence closure algorithm

The congruence closure algorithm [20, 25, 35] constructs a directed graph where expressions of the graph are represented as nodes, i.e., if we have an equation $a = b$ in the context, then a and b will become nodes in the graph. It also takes into account the arguments of the function application. If $\mathsf{rev}\, a = \mathsf{rev}\, b$ is in the context, then $\mathsf{rev}\, a$, $\mathsf{rev}\, b$, rev, a, and b will be nodes.

The vertices of the graph signify the arguments of function applications. If $\mathsf{rev}\, a$ is a node, then there are vertices from $\mathsf{rev}\, a$ to the nodes rev and a.

After the graph is constructed, the algorithm strives to realize equivalence classes that are as elaborate as possible. These classes facilitate the identification of expressions that are equivalent up to congruence, i.e. two expressions in the same equivalence class are equal up to congruence.

5.4.2 New tactic and Ltac

For all the hypotheses, we check if we can apply a segmentation on something that is equivalent up to congruence. If we have an expression $a = b$, we check which expressions are equivalent to a and which are equivalent to b, both up to congruence. We hope to find some c, such that $c \approx a$ and some d such that $d \approx b$, so that we can apply segmentation on $c = d$.

Given a hypothesis, we first make a list of equivalent equations. We do this using `+` inside our tactic `or_congruence_eq`. In this way, we create a point for backtracking, so we try equalities until we succeed to apply segmentation.

In the `or_congruence_eq` tactic we have five steps.

1. Given a hypothesis $l_1 = l_2$ we have to process this to get l_1 and l_2 separately.
2. We create a tree for the congruence closure algorithm. We first put l_1 and l_2 in it, then we modify the function that creates the tree so that $l_1 = l_2$ will never occur in it. This is because we want to observe which terms are equal to l_1 and which terms are equal to l_2, that is, we want to create two distinct equivalence classes.
3. We store which nodes in the tree correspond to l_1 and l_2.
4. We execute the congruence closure algorithm on the tree. We modified the algorithm, so it returns the tree. By doing this, we can extract the equivalence classes from the tree.
5. We transform the equivalence classes into sets. For the two sets that correspond to the equivalence classes of l_1 and l_2 we take the Cartesian product, so we get a set with all possible pairs. These pairs will be transformed into new equations in which we will try to apply the segmentation lemma $\mathsf{length}\, l_{11} = \mathsf{length}\, l_{21} \to l_{11} \mathbin{++} l_{12} = l_{21} \mathbin{++} l_{22} \to l_{11} = l_{21} \land l_{12} = l_{22}$.

A pair to which we will apply segmentation looks like

$$l_1 \mathbin{++} l_2 = l_1' \mathbin{++} l_2'$$

, which have to be in normal form. We determine whether $\mathsf{length}\, l_1 = \mathsf{length}\, l_1'$ is provable using the normal form of the length and utilizing `lia`. If so, we know that $l_1 = l_1'$ and $l_2 = l_2'$. Otherwise, it can be the case that l_2 or l_2' can be written as

$$\begin{gathered} l_2 = l_3 \mathbin{++} l_4 \\ \text{or } l_2' = l_3' \mathbin{++} l_4'. \end{gathered}$$

Notice that this is not the case for l_1 and l_1', because the expressions are in normal form.

If $l_2 = l_3 ++ l_4$, we choose $l_1 = l_1 ++ l_3$ and $l_2 = l_4$. If this is not the case, but $l'_2 = l'_3 ++ l'_4$, we choose $l'_1 = l'_1 ++ l'_3$ and $l'_2 = l'_4$.

We have implemented this by the following recursive Ltac code:

```
Ltac recursive_segmentation llh llt lrh lrt :=
  first [apply_segmentation_lemma|
  (match lrt with
    | ?lrh' ++ ?lrt' ⇒ recursive_segmentation llh llt (lrh ++ lrh') lrt'
  end) |
  (match llt with
    | ?llh' ++ ?llt' ⇒ recursive_segmentation (llh ++ llh') llt' lrh lrt
  end)].
```

5.5 Goal segmentation and discharge easy goals

To discharge the easy goals we use

1. congruence;
2. f_equal in conjunction with done and lia;
3. some method for nth, explained in Section 5.5.1.

This in combination with segmenting the goal is implemented the following way:

```
segment_goal;
  (congruence ||
  (f_equal; (done|| lia)) ||
  (apply nth_l_overflow_eq;
  (length_normal_form; lia)) ||
  ((apply nth_l_eq; [lia|list_solver]))).

Ltac segment_goal :=
  idtac +
  ((match goal with
    | |- ?llh ++ ?llt = ?lrh ++ ?lrt ⇒
        app_segment_goal llh llt lrh lrt
  end);segment_goal).

Ltac app_segment_goal llh llt lrh lrt :=
  (apply (split_list_app llh llt lrh lrt); list_normal_form) +
  (match lrt with
    | ?lrh' ++ ?lrt' ⇒ app_segment_goal llh llt (lrh ++ lrh') lrt'
  end) +
  (match llt with
    | ?llh' ++ ?llt' ⇒ app_segment_goal (llh ++ llh') llt' lrh lrt
  end).
```

```
Lemma split_list_app {A} (l1 l2 l3 l4 : list A):
  l1 = l3 → l2 = l4 → l1 ++ l2 = l3 ++ l4.
```

We start with the segment_goal tactic, and in this tactic we start with idtac, this is to check if the goal on which the tactic is applied can be solved directly by one of the three methods. Notice that the segment_goal tactic is recursive if we apply a segmentation, this is why we need the '+' for backtracking inbetween the idtac and the match in the tactic. If it is not possible to discharge the goal for which the segment_goal tactic is applied directly with the idtac, we will try to segment the goal and create subgoals. The main idea of the goal segmentation is to apply the following lemma to the goal:

$$l1 = l3 \rightarrow l2 = l4 \rightarrow l1 + +l2 = l3 + +l4.$$

In the subgoals we create by segmenting the segment_goal tactic, we will call again the segment_goal tactic, because of its recursive call. By implementing the segment_goal tactic with backtracking, by using '+', in combination with recusion, we try every possible segmentation, until we can discharge all of the segmentation by using one of the 3 methods (congruence, f_equal, or the method for nth)

5.5.1 Solving nth_l

The definition of nth_l is Coqs nth definition with the type class, i.e.

```
Definition nth_l {A} `{Inhabitant A} (i : nat) (l : list A) :=
    nth i l default.
```

If we end up with an equality that is not about lists, but the nth-element of a list, we have two ways of solving this. The first way of solving equations of nth-element is by looking if both sides are an overflow, i.e. we can prove the following with the length normal form and lia:

```
Lemma nth_l_overflow_eq {A} {d : Inhabitant A} (i j : nat) (l l' : list A):
  i ≥ length l →
  j ≥ length l' →
  nth_l i l = nth_l j l'.
```

To implement this, we try to apply the following after we applied segment_goal:

```
apply nth_l_overflow_eq; (length_normal_form; lia)
```

If this is not the case, we will check if the indices are the same, by lia, and establish if the lists are the same, by using our list solver, so we check this by the following lemma:

```
Lemma nth_l_eq {A} {d : Inhabitant A} (i j : nat) (l l' : list A):
  i = j → l = l' →
  nth_l i l = nth_l j l'.
```

We implent this to apply the folling after the `segment_goal` as follows:

```
apply nth_l_eq; [lia|list_solver]
```

In the case where $i \neq j$ and $l = l'$ in when checking if nth_lil = nth_ljl', it we have not implemented a way to check whether this is equal or not. The check for this is out of scope of our research, and we consider this to be future work.

Chapter 6

Evaluation

To evaluate the performance of our solver, we gathered some benchmarks. We solved these benchmarks with three solvers, 1) our solver, 2) the solver that is implemented in VST, and 3) an SMT solver, CVC5. The technical background of the solver implemented in VST can be found in [40]. The solver from VST does not support the `rev` operator out of the box, but, as discussed in [2], we can add the operator manually to the solver. We added the `rev` operator before applying the solver on the benchmarks. All the benchmarks are lemmas that we want to prove. We will look if the solver is able to provide a proof for the lemma, and if so, how long it will take to provide a proof.

With our solver and VST's solver, the proof is directly verified by Coq. The CVC5 solver can only provide a proof in Alethe format [5]. A proof in Alethe formal can theoretically be verified by Coq, but we did not do this verification, because this requires to implement a small checker for sting theory that can be handled by SMTCoq. An more detailed explanation of the Alethe format and verifying the proof in Coq will be discussed in Chapter 7. Because we did not verify the proofs that were produced by CVC5 in Coq, we are not able to compare the running time of the SMT solver with the other solvers.

An overview of the names of the lemmas and the time it took to find a proof can be found in Table 6.1 and Table 6.2. First, in Section 6.1 we will explain how we obtained the benchmarks and what the source of our benchmarks are. Subsequently, in Section 6.2 we will evaluate the result of the benchmarks.

6.1 Source of the benchmarks

The first 23 lemmas, from `update_update` up to and including `own_test_9` were created by us. The lemmas were made during the development of our solver for testing specific features of our solver, or testing ideas. Most lemmas involve the `rev` operator or are used to test the segmentation idea of our solver.

The subsequent 10 lemmas, from `app_eq_nil` up to and including `rev_nth`, are obtained from the Coq standard library and the Iris extended standard library 'coq-std++'. We specifically searched for lemmas that fit the grammar of our solver and put all of them in our set of benchmarks, ensuring suitability for performing solver tests. Some of the lemmas require us to unfold a definition to be part of our grammar.

All lemmas listed in Table 6.2 are obtained from VST [1]. VST contains a tactic for solving lemmas about lists; this is the tactic mentioned in Table 6.1 and Table 6.2.

When building VST, the list solver is applied on some side conditions that arise in the verification of C programs. To capture these side conditions, we modified the list solver in the source of VST. With the modification, the list solver prints the (sub)goals on which it is applied. However, a significant amount of the printed goals pertained to the length of lists, which we omitted because most of them could be solved by `lia` or were very straightforward. There are thus often no explicit lemmas statements about lists, and a lot of the lemmas were prove judgements in separation logic. We transformed as much as possible prove judgment in separation logic, to equalities, i.e. if the goal was to prove that two expressions evaluated to the same value in the separation logic judgement, we tried to prove that the two expressions where equivalent. We named the goals that we obtained by modifying VST with the prefix 'VST #' in Table 6.2.

The last four lemmas of Table 6.2 were also discovered within the VST library. While reading the VST source code, to get a better understanding of the lemmas that we obtained by modifying the list solver in VST and building VST, we found these lemmas by coincidence, which were involving list equations and fit the grammar.

The exact formulation of the benchmarks can be found in our supplementary material [29].

6.2 Benchmark results

Examining Table 6.1 and Table 6.2, it becomes apparent that our solver outperforms others by providing the most proofs for the benchmarks. However, there is one specific lemmas that can only be solved by SMT, namely ex_suff_of_take.This is due to the fact that the grammar of SMT is different from the Coq implemented solvers. The SMT solver is able to handle the existence quantifier, enabling it to solve the ex_suff_of_take lemma, which is not possible for the two Coq based solvers we present here. The ex_suff_of_take lemma is as follows:

```
Lemma ex_suff_of_take {A} {_ : Inhabitant A} (l l' : list A) (n : nat):
  l = take n l' →
  exists x, l ++ x = l'.
```

Additionally, the fact that certain operators like reverse and map is not currently supported in the SMT String Theory, and cannot be obtained by concatenating current supported operators, limits the range of lemmas that can be solved.

Our solver demonstrates superior performance solving a strict superset of the lemmas solved by the VST's list solver. This superiority is expressed especially in two types of lemmas in which our solver outperforms the VST's list solver in terms of solubility. The first type of lemmas are the lemmas with the rev operator, and the second type of lemmas are the lemmas where the segmentation idea is used in the proof.

For lemmas pertaining to the rev operator such as rev_injective, rev_eq_app and rev_nth, as given below, our solver outperforms the VST's list solver. This advantage is realized based on the implementation of the rev operator as a primitive operator, whereas in the VST's list solver the rev operator can be added later.

```
Lemma rev_injective {A} {_ : Inhabitant A} (l1 l2 : list A):
  rev l1 = rev l2 → l1 = l2.

Lemma rev_eq_app {A} {_ : Inhabitant A} (l l1 l2 : list A):
  rev l = l1 ++ l2 → l = rev l2 ++ rev l1.

Lemma rev_nth {A} {_ : Inhabitant A} (l : list A) (n : nat):
    n < length l → nth n (rev l) = nth (length l − (n + 1)) l.
```

Additionally, our solver excels in processing lemmas by using the segmentation method, for example, in Section 3.3 and Section 4.4, are some of the own_test lemmas, and the suffix_snoc_inv_1 lemma. The lemma own_test_5 and suffix_snoc_inv_1 are given below to give an idea of how these lemmas may look like. Interestingly, for lemma own_test_7, the VST's list solver did not even terminate within 30 minutes, revealing a significant performance discrepancy in favor of our solver.

```
Lemma own_test_5 {A} {_ : Inhabitant A} (l12 l1 l2 l23 l3 l3' : list A):
  l12 = l1 ++ l2 →
  l23 = l2 ++ l3' →
  l12 ++ l3 = l1 ++ l23 →
  l3 = l3'.

Definition suffix {A} (l1 l2 : list A):=
   exists k : list A, l2 = k ++ l1.

Lemma suffix_snoc_inv_1
  {A} {_ : Inhabitant A} (x y : A) (l1 l2 : list A):
    l1 ++ [x] `suffix_of` l2 ++ [y] →x = y.
Proof.
  unfold suffix in *;
  intros. destruct H as [k H];
  list_solver.
Qed.
```

Although our solver was able to solve the most lemmas, it is worth noting that it performed slow in solving lemmas that utilized the take and drop operators. This is primarily due to the rewriting of the expressions required for the application of list segmentation, which consequently results in a complication of the indices for the take and drop operators, translating to an increased computational complexity and corresponding processing time for the `lia` tactic.

Lemma	SMT	VST	Our work	operators	# op
update_update	0.056	0.097	13.243	update	3
sublist_lem	29.921	N/A	14.614	take, drop, ++	12
nth_l_app_eq	0.560	0.053	0.954	nth, ++	3
concat_fix_tail	0.078	1.688	1.427	++ , length	8
double_rev_id	N/A	0.073	0.268	rev	2
rev_cons_append	N/A	0.021	0.321	rev, ++	4
rev_app_commute	N/A	0.231	0.327	rev, ++	4
app_eq_tail_cancel	0.015	2.065	0.852	++	4
concat_assoc	0.011	0.205	0.366	++	4
rev_preserves_eq	N/A	0.115	0.258	rev	2
rev_injective	N/A	N/A	0.631	rev	2
left_app_cancel	0.015	0.9	0.67	++	2
right_app_cancel	0.011	0.773	0.695	++	2
ex_suff_of_take	0.275	N/A	N/A	take, ++ , ∃	3
own_test_1	N/A	0.367	00607	rev, ++	7
own_test_2	N/A	0.38	0.7	rev, ++	7
own_test_3	N/A	15.155	0.813	rev, ++	16
own_test_4	0.048	1.749	1.111	length, ++	7
own_test_5	0.031	N/A	0.99	++	4
own_test_6	0.022	0.036	3.569	++	6
own_test_7	0.125	N/A*	5.809	++	18
own_test_8	0.051	N/A	2.498	++	6
own_test_9	N/A	N/A	2.741	++	10
app_eq_nil	0.016	0.758	0.397	++	1
rev_unit	N/A	0.203	0.356	rev, ++	4
rev_eq_app	N/A	N/A	0.839	rev, ++	5
suffix_snoc_inv_1*	0.175	N/A	1.818	app	3
app_cons_not_nil	0.052	N/A	0.812	app, ¬	2
app_singleton	0.043	N/A	0.908	app	1
rev_involutive	N/A	0.073	0.269	rev	2
rev_length	N/A	0.009	0.042	rev, length	3
sublist_rev	N/A	N/A	0.889	take, drop, rev, length	8
rev_nth	N/A	N/A	0.673	rev, nth, length	5

Table 6.1: Running time (in seconds) for lemmas in solvers

Lemma	SMT	VST	Our work	operators	# op
VST #1	0.023	0.107	0.341	take, drop, length, app	6
VST #2	0.081	0.101	0.326	take, drop, length, app	4
VST #3	3.355	0.36	4.405	take, drop, length, app	13
VST #4	1.167	0.062	0.842	nth, length	3
VST #5	1.199	0.081	9.837	nth, length	4
VST #6	N/A	0.347	2.943	take, drop, rev, length, app	17
VST #7	N/A	3.714	727.46	take, drop, length, flip_ends, app, update, map, nth	68
VST #8	N/A	0.209	1.608	map, flip_ends, length	4
VST #9	N/A	1.617	1325.028	length, update, nth, map, flip_ends	19
VST #10	N/A	1.742	7.056	flip_ends, length, map, rev	8
VST #11	N/A	0.338	9.222	map, update, nth	6
VST #12	N/A	0.098	1.207	length, map, repeat, ++	18
VST #13	N/A	0.084	1.173	length, map, repeat, ++	16
VST #14	N/A	1.584	3.558	map, ++ , update, repeat, length	13
`flip_fact_1`	N/A	0.32	5.952	flip_ends, rev, length	3
`flip_fact_3`	N/A	1.696	90.215	take, drop, flip_ends, ++	20
`flip_ends_map`	N/A	N/A	43.56	flip_ends, map	4
`flip_fact_2`	N/A	0.327	5.199	length, nth, flip_ends	4

Table 6.2: Running time (in seconds) for lemmas from VST in solvers

Chapter 7

Related work

In this chapter, we will discuss solvers that are related to our list solver. Firstly, in Section 7.1 we will discuss how we can prove lemmas with SMT and delineate two types of theories implemented in SMT in Section 7.1.1 and Section 7.1.2 that can solve some lemmas involving lists. Subsequently, in Section 7.2 we provide the grammar for the solver that is implemented in VST and discuss some properties of the solver. In Table 7.1 we will give a breef overview of the related work.

7.1 SMT

The SMT theory is used to determine whether a statement is satisfiable. This is different from proving a lemma, where we determine whether a statement is valid. Despite the fact that the two things determine different properties, there is a link between them. If we have a proposition, the proposition is valid if it holds for all possible assignments of the variables. We know that a proposition is valid if and only if the negation of the proposition is unsatisfiable, i.e., for some proposition $\mathbb{P}$,

$$\forall x.\, \mathbb{P}(x) \Leftrightarrow \neg\left(\exists x.\, \neg\mathbb{P}(x)\right).$$

The satisfiability problems that an SMT solver will solve are typically described in the file with SMT-LIB 2.0 standard. In this file, we describe the logic that is utilized and a description of the problem that we intend to solve is provided. The name of the logic tells us which theories are used by the logic, for example the logic named ‘UFSLIA’ supports the theory of uninterpreted functions (UF), strings (S), and linear integer arithmetic (LIA). The ability to combine different theories is a advantage over the solver in Coq, this allows the SMT theory to also look at the underlying theory of the elements of the list, instead of seeing the elements as abstract entities, an example will follow in Section 7.1.1. In Section 7.1.1 and Section 7.1.2

	Fragment	Complete	Language	No. of lemmas	solved
Array	arrays without concatenation and lengths, but support quantifiers	Yes*	SMT-LIB 2.0	1	1
String	strings with concatenation and lengths, does not support quantifiers	Yes*	SMT-LIB 2.0	21	21
VST	Implications with equations about naturals and lists, accepting the operators: `length, append, update, sublist, nth, repeat` and `map`	Theoretically, for tangle-free fragment	Coq	48	37
Own solver	Implications with equations about naturals and lists, accepting the operators: `length, append, reverse, take, drop, nth, repeat` and `map`	Unknown	Coq	49	49

*The technique DPLL, that is used to solve this quantifier-free theory, is complete, but the implementation of the solver is not necessarily complete [16]

Table 7.1: Overview of related work

we discuss two of the theories, but there are many more theories, which are not relevant as related work.

The file in the SMT-LIB 2.0 standard can be given to an SMT solver, which will try to find a satisfiable assingment for the variables, or show that there does not exists any satisfiable assignment, i.e. the problem is unsatisfiable. Besides satisfiable or unsatisfiable, the solver can return 'unknown' if it can not determine whether the statement is satisfiable of unsatisfiable. It depends on the solver which logics can be used. Some examples of solvers are Z3 [34], CVC4 [6], CVC5 [4], and Z3str4 [33]. For example, the 'UFSLIA' logic is not supported by Z3, but is supported by CVC5. Because we work with the SMT-LIB standard and not all the solvers support all the logics from the SMT-LIB standard, it is hard to extend the SMT solves with new SMT theories of expand the existing theories, in that case first the SMT-LIB standard have to be updated and subsequently the solver need an implementation for the logics that use that theory.

The main approach for solving quantifier-free SMT theories is called DPLL, which is theoretically complete [36, 16, 13, 11]. Although the DPLL procedure is complete, it is not necessarily the case that the solvers are complete and sound, i.e. it can be the case that the implementation of DPLL contains bugs [16]. When a logic involves quantifiers, SMT solving will be incomplete [24], but even it is incomplete, that some logics supports quantifiers is an advantage compared to the solvers in Coq.

Due to the fact that SMT solvers can be unsound, the verification method is less reliable. To overcome this problem there are some SMT solvers that support to provide a prove in Alethe format [5] of unsatisfiable statements. As far as we know at the moment only VeriT [12] and CVC5 support the Alethe format. The Alethe format can be used to verify in a proof assistant that an statement is truly unsatisfiable. For Coq there exists the SMTCoq plugin [22, 3, 27]. The SMTCoq plugin can is not able to verify the proves of every theory [27]. Internally SMTCoq uses some small checkers to support the theories. SMTCoq can be extended with new checkers, to support more theories [26].

7.1.1 Array Theory

Array theory, also known as the extensional theory of arrays, in SMT referred to as the “ArraysEx” theory [7, 8]. SMT logics supporting the array theory, contain the letter “A”; for example, the logic“AUFNIRA” encompasses the array theory. The grammar of the array theory can be represented as follows:

$$\begin{aligned}
F &::= \mathit{Atom} \mid F \wedge F \mid F \vee F \mid \neg F \\
\mathit{Atom} &::= t_I = t_I \mid t_S = t_S \mid t_{ArrayIS} = t_{ArrayIS} \\
t_{ArrayIS} &::= \mathit{store}\ t_{ArrayIS}, t_I, t_S \mid \mathit{const}\ (\mathit{ArrayIS}), t_S \\
t_S &::= \mathit{select}\ t_{ArrayIS}, t_I \mid m \mid v \text{ where } m \in \mathit{Cons}_S\ \&\ v \in \mathit{Vars}_S \\
t_I &::= i \mid j \text{ where } i \in \mathit{Con}_I\ \&\ \mathit{Var}_I
\end{aligned}$$

In this theory, arrays have indices of the sort I and the elements of the array are of the sort S, denoted as $ArrayIS$. In SMT,constants or variables can be elected for the types I and S.

This theory has three functions: *const*, *store* and *select*.

The $const : I \rightarrow S \rightarrow t_I \rightarrow t_S \rightarrow t_{ArrayIS}$ operator will create a basic array of the type given as the first argument, with all elements of the list being the second argument.

The $store : t_{ArrayIS} \rightarrow t_I \rightarrow t_S \rightarrow t_{ArrayIS}$ operator will change a single element of the list that is given as the first argument. It change the element in the list that is in the position of the list given in the second argument. The element changes to the element that is given in the third argument.

The $select : t_{ArrayIS} \rightarrow t_I \rightarrow t_S$ operator returns an element of the list given in the first argument. It will return the element that is in the position of the list given in the second argument.

The ArraysEx theory does not support lengths of arrays and concatenation of arrays. Therefore, the only benchmark to which we can apply the theory is update_update.

As an illustration. we demonstrate how update_update can be written in SMT-LIB 2.0 format. When we solve it with CVC5, it will return `unsat`.

```
(set-logic AUFNIRA)
(declare-const i Int)
(declare-const x Int)
(declare-const y Int)
(declare-const l (Array Int Int))
(assert (not (= (store (store l i y) i x) (store l i x))))
(check-sat)
(exit)
```

7.1.2 String Theory

String Theory in SMT, consists of various functions and operators for manipulating strings. Below a part of the grammar of the string theory is provided: [33]

$$
\begin{aligned}
F &::= Atom \mid F \wedge F \mid F \vee F \mid \neg F \\
Atom &::= t_{str} = t_{str} \mid A_{int} \mid A_{ext} \mid A_{re} \\
A_{re} &::= t_{str} \in RE \\
A_{int} &::= t_{int} = t_{int} \mid t_{int} \leq t_{int} \\
A_{ext} &::= contains\ t_{str}, t_{str} \mid prefix\ t_{str}, t_{str} \mid suffix\ t_{str}, t_{str} \\
t_{int} &::= m \mid v \mid length\ t_{str} \mid t_{int} + t_{int} \mid m \cdot t_{int} \mid indexof\ t_{str}, t_{str}, t_{int} \mid \\
&\qquad str.to\_int\ t_{str} \text{ where } m \in Con_{nat} \ \&\ v \in Var_{nat} \\
t_{str} &::= s \mid v \mid t_{str} \cdot t_{str} \mid str.from\_int\ t_{int} \mid replace\ t_{str}, t_{str}, t_{str} \mid \\
&\qquad charAt\ t_{str}, t_{int} \mid \\
&\qquad substr\ t_{str}, t_{int}, t_{int} \text{ where } s \in Con_{str} \ \&\ v \in Var_{str}
\end{aligned}
$$

This grammar omits the operators for regular expressions, which is also included in the SMT string theory [7, 8]. We omit these operators, because this is not related to theorems about lists.

The function $contains\colon String \to String \to Bool$ returns if a string contains a certain substring. The $prefix\colon String \to String \to Bool$ check if a string is a prefix of a substring, and $suffix\colon String \to String \to Bool$ check if a string is the suffix of a string.

The function $\mathsf{length}\colon String \to Integer$ returns the length of a sting. The $indexof\colon String \to String \to Integer$ returns the index of the first occurrence of the second string in the first one starting at the position specified by the third argument, it returns -1 if there is no occurrence of the second string.

The $str.to\_int\colon String \to Integer$ and $str.from\_int\colon Integer \to String$ convert stings into integers and the other way around. The to_int function returns -1 if the string cannot be represented by an integer.

The $replace\colon String \to String \to String \to String$ function will replace the first occurrence of the second argument in the string of the first argument by the third argument.

The $substr\colon String \to Integer \to Integer \to String$ will return the longest substring of the first argument with at most the length of the third argument, starting at the position of the second argument.

The infix operator "$\cdot$" will concatenate two strings or multiply two integers.

With this theory, we can solve some lemmas about lists. The strings consist of a number of characters, as the lists consists of a number of elements. In both strings and lists, the order of the characters and elements is important and we can visualize strings as a list where the characters are the elements. The fact that we see the elements of lists as abstract entities, allows us to translate list theorems to string theorems We see lists as strings, where we use arbitrary characters as abstract list entities. The Table 7.2 below displays how the operators in the grammar of the string theory corresponds with the operators in our solver.

String theory	Own work
$length$	length
$\cdot$	$++$
$replace$	update
$charAt$	nth
$substr(l, 0, n)$	$\mathsf{take}\, n\, l$
$substr(l, (n+1), \mathsf{length}\, l)$	$\mathsf{drop}\, n\, l$

Table 7.2: Relation between operators in string theory and own solver

Although string theory offers functionality related to certain list operations, its applicability to solving list lemmas is limited. For example, it can handle operations such as concatenation and substring extraction, which correspond to list concatenation and take/drop operations, respectively. However, it lacks support for operations such as list reversal and mapping, which are crucial for solving a broader range of list lemmas, as we have seen in Chapter 6.

7.2 VST list solver

As mentioned earlier, VST contains a solver that can solve theorems about lists. In this section, we will give some more background information of the solver. The grammar of the solver is similar to our grammar and can be described as follows:

$$
\begin{aligned}
t_{nat} \in \mathit{Term}_{nat} &::= m \mid v \mid t_{nat} + t_{nat} \mid \mathsf{length}\ t_{list_\alpha} \\
&\qquad \text{where } m \in \mathit{Con}_{nat} \ \&\ v \in \mathit{Var}_{nat} \\
t_\alpha \in \mathit{Term}_\alpha &::= x \mid \mathit{nth}\ t_{nat}\ t_{list_\alpha} \text{ where } x \in \mathit{Var}_\alpha \\
t_{list_\alpha} \in \mathit{Term}_{list_\alpha} &::= [\,] \mid [t_\alpha] \mid t_{list_\alpha} \mathbin{++} t_{list_\alpha} \mid \mathit{update}\ t_{nat}\ t_\alpha\ t_{list_\alpha} \\
&\quad \mid \mathit{map}\ f\ t_{list_\alpha} \mid \mathit{sublist}\ t_{nat}\ t_{nat}]\ t_{list_\alpha} \mid \mathit{repeat}\ t_{nat}\ t_\alpha \mid l \\
&\qquad \text{where } f \in (\mathit{Var}_\alpha \rightarrow \mathit{Var}_\alpha) \\
&\qquad \text{and } l \in \mathit{Var}_{list_\alpha} \\
A \in \mathit{Atom} &::= t_{nat} = t_{nat} \mid t_{list_\alpha} = t_{list_\alpha} \mid t_{nat} \leq t_{nat} \\
\Delta \in \mathit{Ctx} &::= \ \mid \Delta, A \\
G \in \mathit{Goal} &::= A \mid A \rightarrow G
\end{aligned}
$$

This grammar uses *sublist*, instead of take and drop. We can define *sublist* with take and drop and the other way around,

$$\mathit{sublist}\ n\ m\ l = \mathsf{drop}\, n\, (\mathsf{take}\, m\, l).$$

The solver is proven to be theoretically complete on a ”tangle-free” fragment. This proof can be found in [40], and needs a complete base solver to hold. This base solver needs to combine the theory of linear arithmetic and uninterpreted functions, and this solver is not present in Coq.

If a lemma does not contain index shifting, we know that it is tangle-free. Formally, the entanglement of a lemma, i.e. if it is tangle-free or not, is decided by a computation with an index propagation graph. An index propogation graph (IPG) is a directed multigraph with integer terms on each edge. The integer term on the edged represents index shifting. If the IPG has a cycle with nonzero accumulated weight, the lemma is entangled, and otherwise it is tangle free. More information on IPG and how to determine a cycle with nonzero accumulated weight can be found in [40].

A advantage of this solver is its extensibility. The solver can be extended with new operators. For instance, the solver does not support the `rev` operator out of the box, but this can added later on; this also clarifies the difference in the benchmark results between the solvers in VST and our solver.

To extend the solver with a new operator, a rule for the length of the new operator and a rule for the nth of the new operator have to be provided. For example the following two rules have to be as hints to a database in Coq:

$$\begin{gathered}\mathsf{length}\,\mathsf{rev}\,l = \mathsf{length}\,l\\ \forall(i:\mathbb{Z})(l:\mathit{list}_\alpha),\, 0 \leq i < \mathsf{length}\,l \rightarrow\\ \mathsf{nth}\,i\,(\mathsf{rev}\,l) = \mathsf{nth}\,(\mathsf{length}\,l - i - 1)\,l.\end{gathered}$$

In this rules we use $\mathbb{Z}$, instead of $\mathbb{N}$, because the solver from VST uses integers instead of naturals.

Chapter 8

Conclusion and future work

We created a solver in Coq for theorems about lists, which automatically generates a proof. Our solver utilizes new techniques, most notably segmentation, and took the reverse operator and employed it as a primitive operator. In Chapter 6 we have seen that it outperforms the existing solver in Coq in terms of solubility, but at a cost of time efficiency. Furthermore, with having the map and reverse operator in our grammar, we have a different scope than the SMT string theory.

Since we provided an inference system, one would be able to implement differently while utilizing the same ideas. In this way, it would be possible in future work to also create a solver for program verification software that relays other verification tools than Coq, such as SMT.

For future work, it would be interesting to investigate a completeness argument for the underlying theory of the solver. In Chapter 5, we posit that, except for segmentation, everything preserves completeness. The initial step to validating this conjecture would be to verify if the normal form is unique. The segmentation will not preserve completeness because it takes a depth argument in the implementation; in future work it would be necessary to check if there is a bypass for this, to establish a completeness argument.

As demonstrated in Chapter 6, the solver may require a considerable amount of time to compute a proof. Therefore, it would be necessary to optimize the implementation in future work, to reduce the computation time for the solver. As we have seen in Chapter 5, this can, for example, be done by implementing the rewrite system differently, for example using generalized rewriting instead of only using Coq's `rewrite` tactic.

We did not encounter lemmas with conjunctions, disjunctions, and negations in the hypothesis, or conjunctions and negations in the goal for which our solver was unable to solve them. This is interesting because our solver was not specifically designed for these types of lemmas. Conjunctions and disjunctions in the hypothesis and conjunctions in the goal can be handled by the `intuition` tactic which will transform them into (sub)goals without conjunctions and disjunctions in the hypothesis and conjunctions in the goal. When there is a negation in the goal, one can unfold the negation, introduce the implication that follows, and rely on `lia` to discharge the goal after applying our solver. A nice feature to explore would be to check if we can extend the grammar with disjunction in the goals, which could be possible with the `sauto` tactic, because the `sauto` tactic is a proof search tactic, with the option to finalize the poofs with a given tactic (so for example our solver). It would also be nice to check if the solver can be modified so that it is able to handle lemmas with the existential quantifier.

Finally, as explained at the end of Section 5.5.1, the solver cannot handle all the goals that have to be finalized with a prove involving `nth` at both sides of the equation. The solubility of the solver would be extended if we had a solver that could handle this goal, and therefore we suggest to investigate if such a solver exists or could be created.

Bibliography

[1] Andrew W. Appel. *VST*. https://github.com/PrincetonUniversity/VST. 2023.

[2] Andrew W. Apple. *Verifiable C*. 2020. URL: https://vst.cs.princeton.edu/download/VC.pdf.

[3] Michaël Armand et al. "A modular integration of SAT/SMT solvers to Coq through proof witnesses". In: *International Conference on Certified Programs and Proofs*. Springer. 2011, pp. 135–150.

[4] Haniel Barbosa et al. "cvc5: A versatile and industrial-strength SMT solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2022, pp. 415–442.

[5] Haniel Barbosa et al. "The Alethe Proof Format". In: ().

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. "The CVC4 SMT Solver". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. Vol. 6806. 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14.

[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.

[8] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The smt-lib standard: Version 2.0". In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*. Vol. 13. 2010, p. 14.

[9] Marc Bezem, Jan Willem Klop, and Roel de Vrijer. *Term rewriting systems*. Cambridge University Press, 2003.

[10] Nikolaj Bjørner and Leonardo de Moura. "Applications of SMT solvers to program verification". In: *Notes for the Summer School on Formal Techniques* (2014).

[11] Miquel Bofill et al. "A write-based solver for SAT modulo the theory of arrays". In: *2008 Formal Methods in Computer-Aided Design*. IEEE. 2008, pp. 1–8.

[12] Thomas Bouton et al. "veriT: an open, trustable and efficient SMT-solver". In: *International Conference on Automated Deduction*. Springer. 2009, pp. 151–156.

[13] Aaron R Bradley, Zohar Manna, and Henny B Sipma. "What's decidable about arrays?" In: *Verification, Model Checking, and Abstract In-*

terpretation: 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006. Proceedings 7. Springer. 2006, pp. 427–442.

[14] Julien Braine, Laure Gonnord, and David Monniaux. “Verifying Programs with Arrays and Lists”. PhD thesis. ENS Lyon, 2016.

[15] Martin Bromberger, Thomas Sturm, and Christoph Weidenbach. “Linear integer arithmetic revisited”. In: *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*. Springer. 2015, pp. 623–637.

[16] Alexandra Bugariu and Peter Müller. “Automatically testing string solvers”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 1459–1470.

[17] Kenneth Leroy Busbee and Dave Braunschweig. *Programming Fundamentals: A Modular Structured Approach*. 2018.

[18] Qinxiang Cao et al. “VST-Floyd: A separation logic tool to verify correctness of C programs”. In: *Journal of Automated Reasoning* 61 (2018), pp. 367–422.

[19] Pierre Castéran and Matthieu Sozeau. *A gentle introduction to type classes and relations in Coq*. Tech. rep. Technical Report hal-00702455, version 1, 2012.

[20] Pierre Corbineau. “Deciding equality in the constructor theory”. In: *International Workshop on Types for Proofs and Programs*. Springer. 2006, pp. 78–92.

[21] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022.

[22] *cvc5 Documentation*. Version 1.0.0. URL: `https://cvc5.github.io/docs/cvc5-1.0.0/proofs/output_alethe.html`.

[23] Leonardo De Moura and Nikolaj Bjørner. “Efficient E-matching for SMT solvers”. In: *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21*. Springer. 2007, pp. 183–198.

[24] David Déharbe, Pascal Fontaine, and Bruno Woltzenlogel Paleo. “Quantifier inference rules for SMT proofs”. In: *First International Workshop on Proof eXchange for Theorem Proving-PxTP 2011*. 2011.

[25] Peter J Downey, Ravi Sethi, and Robert Endre Tarjan. “Variations on the common subexpression problem”. In: *Journal of the ACM (JACM)* 27.4 (1980), pp. 758–771.

[26] Burak Ekici et al. “Extending SMTCoq, a certified checker for SMT”. In: *arXiv preprint arXiv:1606.05947* (2016).

[27] Burak Ekici et al. “SMTCoq: A plug-in for integrating SMT solvers into Coq”. In: *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30*. Springer. 2017, pp. 126–133.

[28] N. Hirokawa et al. *AProVe*. https://github.com/aprove-developers/aprove-releases/releases/. 2024.

[29] Jeroen Kool. *coq-list-solver*. https://github.com/Jeroen74/coq-list-solver/. 2024.

[30] Cynthia Kop. "Higher order termination". In: *Faculty of Sciences, Department of Computer Science, VUA* 14 (2012), pp. 2012–15.

[31] David Chenho Kung and Hong Zhu. *Software Verification and Validation.* 2008.

[32] Allen Weiss Mark. *Data structures and algorithm analysis in C*. 1992.

[33] Federico Mora et al. "Z3str4: A multi-armed string solver". In: *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24*. Springer. 2021, pp. 389–406.

[34] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 4963. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[35] Greg Nelson and Derek C Oppen. "Fast decision procedures based on congruence closure". In: *Journal of the ACM (JACM)* 27.2 (1980), pp. 356–364.

[36] Umut Oztok and Adnan Darwiche. "An exhaustive DPLL algorithm for model counting". In: *Journal of Artificial Intelligence Research* 62 (2018), pp. 1–32.

[37] Christine Paulin-Mohring. "Introduction to the Coq proof-assistant for practical software verification". In: *LASER Summer School on Software Engineering*. Springer, 2011, pp. 45–95.

[38] Matthieu Sozeau. *The Coq Reference Manual Release 8.12.1*. 2020. URL: http://coq.inria.fr.

[39] The Coq development team. *The Coq proof assistant reference manual version 8.16.1*. 2002. URL: http://coq.inria.fr.

[40] Qinshi Wang and Andrew W Appel. "A Solver for Arrays with Concatenation". In: *Journal of Automated Reasoning* 67.1 (2023), p. 4.