Universiteit Utrecht
Faculteit Wijsbegeerte

On Formalizing Decreasing Proof Orders

A thesis for the Cognitive Artificial Intelligence master's program

Yannick Bitane
April 9th 2015

1st Supervisor: Vincent van Oostrom
2nd Supervisor: Albert Visser
Co-assessor: Gerard Vreeswijk
Credits: 60 ECTS

Contents

1 Introduction

In this chapter, the reader gets an impression of the subject matter of our research project, finds out about its relevance to the field of AI, and gets a taste of chapters to follow.

1.1 Background

The field of AI has traditionally been a two-fold endeavor. On the one hand, we attempt to simulate the mechanics of natural intelligence by which it solves a problem, in order to better understand its solution. For example, we simulate a neural network to understand how the brain operates. On the other hand, we construct artificially intelligent programs, attempting to solve problems as effectively as possible, to better understand the nature of the problem. For example, we build a chess computer to uncover the patterns and aspects involved in winning a game of chess. This research is geared to contribute to the latter endeavor. In particular, we have applied ourselves to the mathematical aspects of engineering.

One of the most appealing aspects of mathematics is the associated sense of absolute certainty. In an inherently chaotic universe, having algorithms that are guaranteed to return the correct output for *any* valid input certainly is a great asset for anyone seeking to solve problems effectively. There is no risk to an unfavorable outcome (false output) if there is only one outcome which is favorable (correct output). This greatly reduces the need to be anxious about decisions to be made. One does not have to doubt the reliability of information if it is guaranteed to be correct. To this end, an algorithm can have several desirable properties. For example, low *computational complexity* when a fast response time is crucial. But even when complexity is low, reliability of such a highly efficient algorithm is essentially dependent upon *termination*. If it is unclear if the algorithm will return the correct answer or keep us waiting forever, we might not want to bet our lives on it just yet. The presence or absence of such properties can be established by mathematical analysis.

A means of modelling an algorithm or set of algorithms for this purpose is called *term rewriting*.[d] A software program (or any algorithmic procedure for that matter) is really just a very specific sequence of simple instructions. Each instruction takes the provided input, modifies it in some way and passes the output onto the next instruction. In term rewriting, the instructions are modelled as *rewrite rules*, and the inputs as *terms*. A set of terms and rewrite rules over these terms together are called a *term rewriting system*.

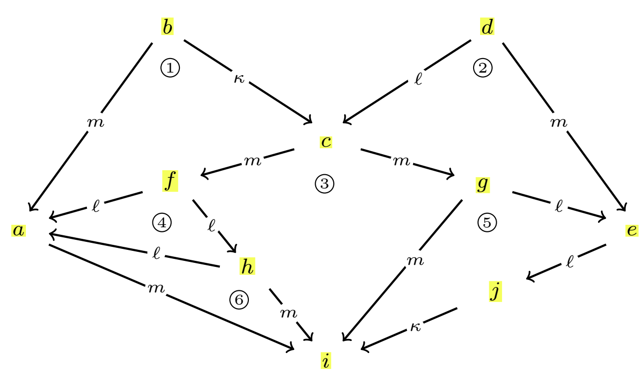


Figure 1: a decreasing diagram of conversions[a]

The *decreasing diagrams* technique is a versatile method for proving a desirable property of term rewriting systems, called *confluence*.[b] This holds that when the same rewrite rules can be applied in multiple sequences, we always get the same end result. Think of addition with numbers for example. We want 1 + 4 * 5 to have the same result as 5 * 4 + 1. In Figure 1 above, we see a graph where the nodes represent terms, and the arrows rewrite rules. [.....] Van Oostrom's article[a] on *decreasing proof orders* is a further refinement of the decreasing diagrams technique. The current thesis revolves around a partial formalization of this article. We have developed a correctness verification of this technique in an automated theorem prover called "Coq".

Why Coq?

Correctness verification is part of the field of mathematics. To verify correctness of a proof, it doesn't suffice merely to follow the steps and lines of reasoning as the authors present them on paper. This is because, in favor of readability, many steps are subtly skipped over in going from premise to conclusion. Abbreviations are used, and many details are left implicit. The intelligent human audience is expected to be able to see the validity of these jumps in reasoning. However, for our purposes, ie. correctness verification, we want a guarantee of every single step along the way being correct, skipping over none. This is where Coq comes into play.

Coq is an interactive proof verification tool. It allows the user to assert properties, and create "in dialogue" with the program a proof that is automatically verified for correctness. Developing a theorem proof in Coq is very much like developing software. The user defines functions, specifying the data types of input and output, and describing in detail how the one is converted into the other. This would be a very time-consuming task were it not for the vast libraries to draw upon, of statements already proven and verified, ready to be used. As such, Coq's automated approach to theorem proving makes it possible to simultaneously construct *and* verify theorem proofs, all with the conciseness and readability of an old-fashioned paper printed proof.

Intuitionistic logic

The formalism of Coq employs an intuitionistic approach to logic. In *classical logic*, (ie. regular logic) the *law of excluded middle* is in effect. That is, given a proposition P, either it holds or it doesn't: P or $\neg P$. A common way to prove a statement then, is to demonstrate that its negation leads to contradiction. From this, one then concludes that P must hold (as dictated by the law of excluded middle). In *intuitionistic logic*, this law of excluded middle is rejected. As such, proofs by contradiction are generally invalid. In an intuitionistic logic, truth equals constructability. To assert that an object with certain properties exists, is to claim that such an object can be *constructed*. So, to see if P is true, is to see what the *proofs* of P are (if any).

This different notion of truth leads to different results in terms of what is provable. Some things unprovable in classical logic are provable in intuitionistic logic and vice versa. What makes intuitionistic logic attractive for our purposes is that it enjoys a very powerful property called the *Curry-Howard correspondence*. Simply put, it's a one-to-one correspondence between systems of formal logic and computational calculi. Each natural-deduction proof in intuitionistic propositional logic can be associated with a term in the simply typed lambda calculus. Or, in layman terms, such mathematics can be directly applied in software engineering: correct programs can be *automatically* extracted from their proofs. Coq has built-in support for such automated extraction.

1.2 Contents

Before delving into details, we look at the goals and scope of our research, and at the way in which this document is structured in order to present its results.

Scope

The main two lemmas of the proof described in the article are implemented and proven using the interactive theorem prover Coq. We have written a framework in support of these two lemmas. As described in Chapter 2, the first lemma concerns the main data type and correctness of its implementation. The second lemma confirms the essential properties of the Decreasing Proof Order. We provide proof of the first part of this two-part lemma. This is described in Chapter 3.

Status quaestionis

An older proof of the decreasing diagrams technique has been formalized by Zankl using Isabelle.[e]

Assumed knowledge

Our intended audience is fellow students of the Cognitive Artificial Intelligence master's program. The reader is assumed to have a basic understanding of logic, formal notation and programming. For a more in-depth explanation of term rewriting, the reader is referred to Terese or Baaden-Nipkow.

Presentation

Creating a certified proof in Coq is usually a back and forth between polishing the statement to be proven, and readjusting the angle of approach towards proving it. Stating a theorem involves developing firstly the simpler statements and concepts from which it is composed. Once stated, proving the theorem requires development of simpler theorems (so-called lemmas) involving the theorem's various components. In an ideal world, this would be a one-pass process. However, in reality, multiple passes are needed: due to new insights gained during development, adaptations are made. Some of these consist of adapting the proof strategy used, and some of adapting the concepts from which the statement itself is constructed.

The cyclical nature of this process is reflected in the presentation of our findings. Moving from elementary to complex, and then from complex to elementary, we present the components from which statements are constructed, and proofs of these statements. The components are first defined mathematically, we then describe their implementation in Coq. Some detours to failed attempts are made for more insight. Our theorem proofs in Coq are presented as *proof state* sequences. These are simplified prints of the interaction between input and output to and from Coq, closely narrated to give the reader a clear idea of why they are structured the way they are. Some explanations of constructs are given in advance of the proof, some are deferred to and expanded upon after the proof.

How to read

Some final pointers before we get started. The reader is assumed to be viewing this document digitally. Many clickable cross-references are made throughout the document for optimal accessibility. Definitions of our own functions and lemmas are given once, and can thereafter be found via the index at the end of this document. Native Coq tactics and definitions are explained conceptually to greater or lesser degrees, but formal definitions are omitted. Those can be found in the Coq Reference Manual, at `https://coq.inria.fr/distrib/8.4pl3/refman/`. For some of those definitions, a hyperlink to the url of the corresponding sections in the reference manual is given at its first occurrence, denoted by underlining or by a raised reference marker: `example n`, example.[n] First occurrences of these are listed in the index. Several of our own key lemmas and/or functions were provided by Vincent van Oostrom. These are marked `example`[v] to indicate firstly their origin, and secondly that their proofs are beyond the scope of this document.

Requirements

This research endeavor has been very practical in nature. For optimal clarity, the reader is advised to follow the proofs using CoqIDE in a synchronous fashion. CoqIDE can be downloaded here: `http://coq.inria.fr/download`. This project was developed in Coq version 8.4pl3.

The source code of this project can be found in plain text at `http://igitur.uu.nl/bitane/thesiscode.zip`. For ease of reference, an HTML rendering of it can be viewed at `http://igitur.uu.nl/bitane/thesiscode.html`. In addition to CoqIDE, the following two external libraries are required.

- `CoLoR`: `http://color.inria.fr/`, `https://gforge.inria.fr/scm/viewvc.php/trunk/?root=color&pathrev=2043`
  (we have used SVN revision 2043, released december 3rd, 2013)
- Cantor: `http://www.lix.polytechnique.fr/coq/pylons/coq/pylons/contribs/view/Cantor/v8.4`
  (the version we used is dated januari 10th, 2013)

Haskell extraction

One of the more interesting features of this project is that it's fully *constructive* (see page 1). The interested reader can add the following lines at the end of the source code to automatically extract any `Definition`, `Fixpoint` or `Lemma` of the form `forall` ... `exists` ... to Haskell format.[2]

```
Extraction Language Haskell.
Extraction "filename.hs" name_of_construct.
```

Excerpt #1

2 Representation Proof

In this chapter, the decreasing proof order framework is formalized. This framework is to be used for developing a proof of the main theorem, as will be described in Chapter 3. We will present the framework in Section 2.1, and then prove its correctness in Section 2.2.

2.1 Presenting the framework

Let us start off by having a look at all the components that are involved in formalizing the decreasing proof order framework. In describing these, the following format is used: each component is defined first in mathematical terms, and then a description is given of how it is implemented in our framework. When appropriate, examples are given. We'll describe *French strings*, *French terms*, how to construct the latter from the former by means of *stratification*, and how to reconstruct the former from the latter by means of *flattening*.

2.1.1 French strings

Every occurrence of our main data type has both a string form and a tree form, which are semantically equivalent and (thus) interconvertible. We first consider here the string form, a data type called *French string*.

Definition 1. Let L be an alphabet. Then for any letter $\ell \in L$, accenting it acute ($\acute{\ell}$) or grave ($\grave{\ell}$) creates a *French letter*. Denote a French letter with its accentuation type abstracted $\hat{\ell}$. Let $\widehat{L}$ denote the set of *French strings*, ie. the set of finite strings of French letters over L.

Example. Suppose we have $L = \{\ell, m, k\}$. Then $\{\acute{\ell}, \grave{\ell}, \acute{m}, \grave{m}, \acute{k}, \grave{k}\}$ would be the corresponding set of French letters. Some examples of French strings would be $\acute{m}\grave{\kappa}\acute{\ell}\grave{m}$ and $\acute{\ell}\grave{\ell}\grave{m}\acute{m}\grave{\ell}$.

Implementation

Analogous to Definition 1, an `fletter` consists of a `letter` and an `accent`. An `fstring` then, is a (finite) list thereof.

```
Inductive accent : Type :=
  acute | grave.

Inductive letter : Type :=
  m | k | l.

Inductive fletter : Type :=
  fletter_cons : letter → accent → fletter.

Definition fstring : Type :=
  list fletter.
```

Excerpt #2

Example. $\acute{m}$ would be represented as `(fletter_cons m acute)`, with `fletter_cons` being the constructor function, and $\acute{m}\grave{\kappa}\acute{\ell}\grave{m}$ as `fletter_cons m acute :: fletter_cons k grave :: fletter_cons l acute :: fletter_cons m grave :: nil`.

2.1.2 French terms

Every French string has a *French term* counterpart, which can be conceptualized as a tree. Before getting to a formal description in Definition 4, we consider two concepts involved, that of the *Hoare order* and *having arity*. The former is an order on French strings lifting an order on letters, the latter is a property of terms, describing the ratio between a function symbol's length and its number of arguments.

Definition 2. Let L be an alphabet, $\succ$ an order on L, and $\widehat{L}$ the set of French strings over L. Given s, $r \in \widehat{L}$, we say that s relates to r in the *Hoare order* induced by L and $\succ$ if, for each French letter $\hat{x}$ occurring in s, there exists a French letter $\hat{y}$ in r such that $y \succ x$.

Example. Suppose we have an alphabet $L = \{m, \ell, \kappa\}$, and an order $\succ$ such that $m \succ \kappa, \ell$. Then $\acute{\ell}\grave{\kappa}$ relates to $\grave{m}$ in the Hoare order induced by L and $\succ$, but not to ε or $\grave{\ell}$. Likewise, $\grave{m}\acute{m}$ does not relate to any string over $\widehat{L}$ in this order, and ε relates to any non-empty string in this order.

Implementation

```
Inductive hoare_lt (x y : list fletter) : Prop :=
  hoare_lt_cons : y ≠ nil →
                (forall  e1, In e1 x →
                 exists2 e2, In e2 y & fl_Lt e1 e2)
              → hoare_lt x y.
```

Excerpt #3

Example. Analogously to the example of Definition 2, `hoare_lt (í :: k̀ :: nil) (m̀ :: nil)` holds, because it holds that `forall e1, In e1 (í :: k̀ :: nil) → exists2 e2, In e2 (m̀ :: nil) & fl_Lt e1 e2`.

We have used some shorthands in this example to benefit readability.

```
Definition ḿ : fletter := fletter_cons m acute.
Definition m̀ : fletter := fletter_cons m grave.
...
Definition ∅ : term := Fun fs_Sig nil nil. (* the empty term *)
```

Excerpt #4

Definition 3. Let L be an alphabet, Σ a signature over strings of L, and t a term over Σ. We say of t that it *has arity* if, for each function symbol f in t, if $f = \varepsilon$ the number of arguments for f equals 0, and otherwise the number of arguments for f equals f's length + 1.

Example. Let t be a term that has arity. If $\grave{m}\acute{m}$ is a function symbol in t, it has 3 arguments.

Implementation

In Coq, this ratio between a function symbol's length and the number of arguments is implemented via `ar`.

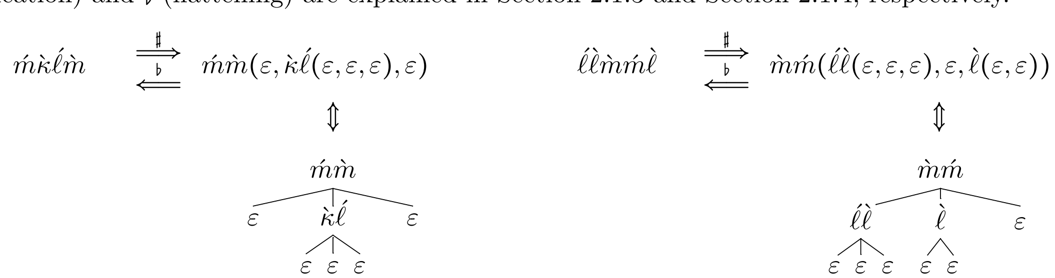
```
Definition ar (fs : fstring) : nat :=
  match fs with
  | nil ⇒ 0
  | _   ⇒ 1 + length fs
  end.
```

Excerpt #5

Example. `ar (m̀ :: ḿ :: nil) = 3`

Definition 4. Let L be an alphabet, $\succ$ an order on L, and $\widehat{L}$ the set of French strings over L. The *French term signature* over L, denoted $L^{\sharp}_{\succ}$, consists of all strings in $\widehat{L}$ composed of letters mutually $\succ$-incomparable. In vein of the tree metaphor, these function symbols are also called (*node*) *labels*. Let t be a term over $L^{\sharp}_{\succ}$. We say that t is a *French term* if it has arity, and each node in t has a label that is related to its ancestor node's label by the Hoare order.

Example. Suppose we have an order $\succ$ such that $m \succ \kappa, \ell$. To give an idea of how a French string corresponds to a French term, consider the below two figures. Note that, all node labels consist of letters that are mutually $\succ$-incomparable, for each node, the branching factor of that node is either zero or its label's length plus one, and every node's label relates to the ancestor node's label in the Hoare order induced by L and $\succ$. The operations $\sharp$ (stratification) and $\flat$ (flattening) are explained in Section 2.1.3 and Section 2.1.4, respectively.



Implementation

Suppose we have `L` = `{m, l, k}`. Let `fl_Lt` be an order on `L` such that `strict_order fl_Lt`. Now, to implement *French terms*, we use the `term` data type, from the external library `CoLoR`, as shown in Excerpt #6 (implicit parameters added in gray). This is a *dependent data type* that, given a signature `Sig`, can construct a tree with nodes of type `Sig`. It also has a constructor for variables, but we won't be using those for now.

```
Inductive term (Sig : Signature) : Type :=
  | Var : nat → term Sig
  | Fun : forall f : Sig, list (term Sig) → term Sig.
```

Excerpt #6 — CoLoR.Term.Varyadic.VTerm

A `vterm` then is a `term` instantiated with `fs_Sig`, a signature over `fstring`.

```
Definition vterm : Type := term fs_Sig.
```

Excerpt #7

Example. The term $\acute{m}\grave{m}(\varepsilon, \grave{k}\acute{\ell}(\varepsilon,\varepsilon,\varepsilon),\varepsilon)$ would be represented as

```
Fun fs_Sig (ḿ :: m̀ :: nil) (∅ :: (Fun fs_Sig (k̀ :: ĺ :: nil) (∅ :: ∅ :: ∅ :: nil)) :: ∅ :: nil).
```

`CoLoR`'s `term` data type is expressive enough for our purposes but not strict enough, as neither arity nor the Hoare order are imposed. These additional constraints are implemented via `vt_wellformed`. In our framework, a French term `ft` is a `vterm` such that `vt_wellformed ft` is `True`, as we'll see in Excerpt #9.

```
Fixpoint vt_wellformed (vt : vterm) : Prop :=
  match vt with
①  | Var _      ⇒ False                      ②
   | Fun f args ⇒ vterm_empty vt ∨ ( vterm_not_empty vt ∧
                                      ar f = length args ∧      ③
                                      fs_incomparable f ∧       ④
           Forall (fun x ⇒ hoare_lt (vt_head x) f) args ∧       ⑤
                               lforall vt_wellformed args )     ⑥
  end.
```

Excerpt #8

Let's have a closer look at the components of `vt_wellformed` so we can see how it helps to implement Definition 4. It takes a `vterm` named `vt` and returns a `Prop`. ① `vt` is pattern-matched against the constructor functions of its data type (see Excerpt #6). By definition, any `vterm` matching the `Var` constructor is not well-formed. When `vt` is matched against the constructor `Fun`, its head is bound to the variable `f`, and its arguments to `args`. ② A case distinction is made on whether or not `vt` is empty. This disjunction would follow independently from item ③ but is made explicit for ease of use. The empty case suffices to qualify `vt` as being well-formed. The non-empty case is more complex, being embedded in a conjunction. Let's consider the other conjuncts.

Recall from Definition 4 that, for a French term, it holds that **1.** *all node labels consist of letters that are mutually* $\succ$*-incomparable.* This is expressed as `fs_incomparable f` (item ④, see Excerpt #10). **2.** *it has arity.* This is expressed as `ar f = length args` (item ③, see Excerpt #5). **3.** *every node's label relates to the ancestor node's label in the Hoare order induced by* L *and* $\succ$. This is expressed as `Forall (fun x ⇒ hoare_lt (vt_head x) f) args` (item ⑤, see Excerpt #3). Finally, all arguments of `vt` are well-formed themselves. This is expressed as `lforall vt_wellformed args` (item ⑥). The constraints of Definition 4 are thus fully covered by `vt_wellformed`.

```
Definition fterm : Type :=
  {vt : vterm & vt_wellformed vt}.
```

Excerpt #9

A *French term* then is a `vterm` such that `vt_wellformed` holds. This is captured in the type `fterm`, a pair consisting of a `vterm` and a proof its well-formedness. In Coq, such pairing of a data type with proof of some property is called a *sigma type.*[18]

Epilogue

Here we display `fs_incomparable`, which was skipped over in explaining our implementation of French terms.

```
Inductive fl_comparable : relation fletter :=
  | fl_comparable_cons1 l1 l2 : fl_Lt l1 l2 → fl_comparable l1 l2
  | fl_comparable_cons2 l1 l2 : fl_Lt l2 l1 → fl_comparable l1 l2.

Inductive fl_incomparable (f : fletter) (L : list fletter) : Prop :=
  fl_incomparable_cons : (forall e, In e L → ~ fl_comparable e f) → fl_incomparable f L.

Inductive fs_incomparable (fs : fstring) : Prop :=
  fs_incomparable_cons: (forall e, In e fs → fl_incomparable e fs) → fs_incomparable fs.
```

Excerpt #10

An `fstring` is `fs_incomparable` if all its letters are `fl_incomparable`, which holds if they are all not `fl_comparable` to any letter in `fs`. In other words, if `∀xy∈fs : ¬ fl_Lt x y` and `¬ fl_Lt y x`.

This concludes our description of French strings and French terms. We can get a clearer understanding of an `fterm`'s structure by looking at how it is created, as is done in the next section.

2.1.3 Stratification

The French string form and French term form are interconvertible. This section is about the operation converting a French string to its corresponding French term, called *stratify*. The operation inverse to this, called *flattening*, will be the subject of Section 2.1.4. We begin this section by considering the concept of a *scattered substring*. We then proceed to examine stratification in a fashion similar to that of the previous two sections on *French strings* and *French terms*, going back and forth between mathematical description, example and Coq implementation.

A scattered substring of a string f is any string made from f by omitting letters from it. Or, more formally,

Definition 5. Let L^* be the set of strings over some alphabet L, and let $f, g \in L^*$. We say that f is a *scattered substring* of g if $f = a_1 \dots a_n$ and $g = x_0 a_1 x_1 \dots a_n x_n$ for some $a_i \in L$ and $x_j \in L^*$.

Example. $\grave{m}\acute{m}$ is a scattered substring of $\acute{\ell}\grave{\ell}\grave{m}\acute{m}\grave{\ell}$, as are $\grave{\ell}\grave{m}\acute{m}\grave{\ell}$ and $\acute{\ell}\grave{\ell}\grave{\ell}$. Also, any string is a substring of itself, since the empty string ε is also in L^* ($f = g$ if all the x_j in the above definition are empty).

Implementation

In Coq we have implemented this as a relation between `fstring` called `sublist`.

```
Inductive sublist : list fletter → list fletter → Prop :=
| sublist_cons1 : sublist nil nil
| sublist_cons2 : forall L1 L2 f, sublist L1 L2 → sublist      L1  (f :: L2)
| sublist_cons3 : forall L1 L2 f, sublist L1 L2 → sublist (f :: L1) (f :: L2).
```

Excerpt #11

Any two `fstring` relating in `sublist` to one another can be constructed by taking two empty lists `L1` and `L2`, and repeatedly adding a single letter `f` either to both `L1` and `L2`, or only to `L2`.

Example. In the case of `sublist (m̀ :: ḿ :: nil) (ĺ :: l̀ :: m̀ :: ḿ :: l̀ :: nil)`: this holds, because we can construct it by, starting from `sublist_cons1`, consecutively applying `sublist_cons2` with `f := l̀`, `cons3` with `ḿ`, `cons3` with `m̀`, `cons2` with `l̀`, and `cons2` with `ĺ`.

A French string f is converted to a French term t by means of stratification. This is a recursive process, where the scattered substring containing all letters maximal in f is assigned to be the function symbol, and the stratification of each omitted fragment as arguments. More specifically,

Definition 6. *Stratification*, denoted $\sharp$, maps each French string to its corresponding French term. This is defined inductively as follows: $\varepsilon^\sharp = \varepsilon$, and $(s_0\hat{\ell}_1 \ldots \hat{\ell}_n s_n)^\sharp = \hat{\ell}_1 \ldots \hat{\ell}_n(s_0^\sharp, \ldots, s_n^\sharp)$, with $\hat{\ell}_i$ all occurrences of $\succ$-maximal letters in the string, s_i the substrings around each $\hat{\ell}_i$, and $n > 0$.

Example. Again, suppose we have an order $\succ$ such that $m \succ \kappa, \ell$. Then

$$(\acute{\ell}\grave{\ell}\grave{m}\acute{m}\grave{\ell})^\sharp = \grave{m}\acute{m}(\acute{\ell}\grave{\ell}^\sharp, \varepsilon^\sharp, \grave{\ell}^\sharp) = \grave{m}\acute{m}(\acute{\ell}\grave{\ell}(\varepsilon^\sharp, \varepsilon^\sharp, \varepsilon^\sharp), \varepsilon, \grave{\ell}(\varepsilon^\sharp, \varepsilon^\sharp)) = \grave{m}\acute{m}(\acute{\ell}\grave{\ell}(\varepsilon, \varepsilon, \varepsilon), \varepsilon, \grave{\ell}(\varepsilon, \varepsilon)).$$

Implementation

To better understand our present implementation of `stratify` in Coq, consider first an early attempt at implementing stratification and why it was not satisfactory. To explain this we distinguish between *standard* and *non-standard* recursion in Coq.

Standard recursion

A recursive process terminates if it doesn't contain infinite chains of nested recursive calls. The absence of infinite chains is established in Coq by assigning to the recursion a *decreasing argument*[1]. This argument can be of any inductively defined data type. To rule out non-terminating recursion, Coq wants to see this argument differ by at least one constructor between iterations. For example, consider the function `finite_nat` below.

```
Fixpoint finite_nat (n : nat) : Prop :=
  match n with
  | 0   ⇒ True
  | S x ⇒ trivial x
```

Excerpt #12

Here, the decreasing argument is `n` of type `nat`. The second clause is recursive. How do we know if this recursion will terminate? Well, if `n` matches `S x`, then `x`, the argument used in the next iteration, has decreased by one constructor relative to `n` (by the successor function `S`). As objects of type `nat` are inductively defined, this peeling off of constructors will continue until the core of the construct, `0` in the case of `nat`, is inevitably reached.

Similarly, objects of type `list` are inductively defined. However, for our purposes, this does not suffice to serve as a decreasing argument. To see why, consider an earlier attempt at implementing stratification in Coq. In Excerpt #13 below, we see an implementation of `stratify` using standard recursion on lists.

```
Fixpoint stratify_rec (vt : vterm) (lfs : list fstring) : vterm :=
  match lfs with
  | nil          ⇒ vt
  | head :: args ⇒ stratify_rec (@Fun fs_Sig head
                                    (map (fun x ⇒ stratify_rec vt x) (map stratify_sub' args))
                               ) nil
  end.

Definition stratify (fs : fstring) : vterm :=
  stratify_rec empty_vterm (stratify_sub fs).
```

Excerpt #13 example_1_stratifail.v

Shown is part of the code. For the full example see `example_1_stratifail.v` The definition of `stratify_sub` (of type `fstring → list fstring`) can be found there as well, but is unimportant to understand our issue here with standard recursion. Having said that, let's look at the data flow chart of this algorithm, as displayed in Figure 2 on the next page.

The process begins with `stratify_sub fs`. This returns a list, `MAX :: l0 :: l1 :: ...`, which is preformatted for `stratify_rec`. The head, here denoted `MAX`, is the list of all letters that are maximal in `fs`. The tail are all the other letters (non-maximal in `fs`), grouped as they occur in between the maximal letters. This list is input for `stratify_rec`, which builds a `term` using `MAX` as its head, and `l0`, `l1`, `l2`, `...`, `ln` as arguments. These arguments are then recursed upon using `map stratify_sub`.

As an algorithm to build terms, this setup should work. However the problem here is that Coq cannot tell if this recursion will ever stop, as `stratify_sub` is external to `stratify_rec`. From `stratify_rec`'s point of view, it's unclear how the current iteration's main argument relates to that of the next iteration, because this relation is obscured by the intervention of `stratify_sub`. As far as `stratify_rec` can tell, `stratify_sub` could be spawning child nodes *larger* than `stratify_rec`'s current input. Using those in its next iteration would make the recursion loop indefinitely.

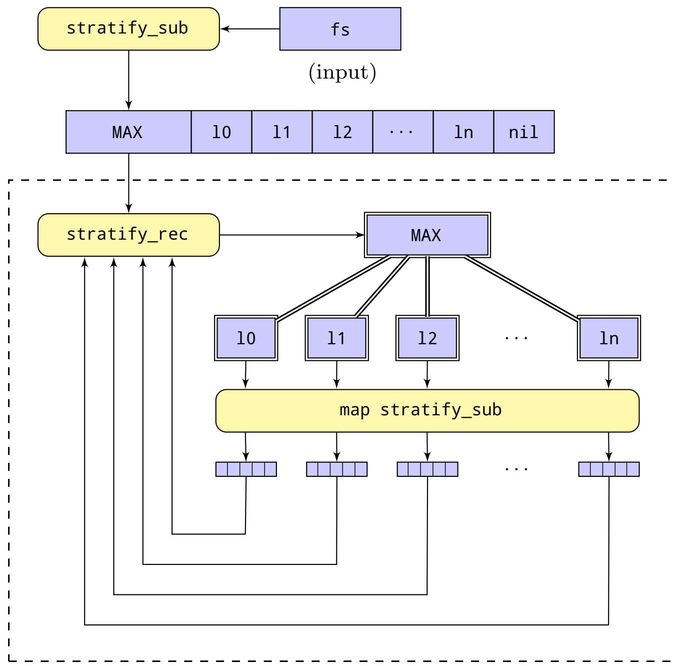


Figure 1: non-terminating recursion in `stratify`

Non-standard recursion

So what does a terminating version of `stratify` look like? Well, Coq allows the user to select *any* inductively defined object to recurse upon, instead of the standard objects (`list`, `nat`, et cetera).[1] Intuitively, it's obvious that our recursive process would end at some point: we are working with finite lists, so repeatedly splitting such a list cannot continue indefinitely. Each subsequent list is smaller in length than its predecessor.

To capture this fact in an inductively defined object, we use the accessibility constructor `Acc` applied to the relation `lengthOrder` to serve as our base for terminating recursion. `lengthOrder` is a simple less-than relation on string lengths.

```
Definition lengthOrder (x y : fstring) :=
  length x < length y.
```

Excerpt #14

`Acc` is a relation property with respect to a specific domain. Consider `Acc`'s definition below. Given a data type `A`, a relation `R` between instances of `A`, and `x`, an instance of `A`, `Acc R x` is an object if, for any `y` such that `R y x`, `Acc R y` is also an object. Let's apply this to our relation `lengthOrder`. `Acc lengthOrder fs` is an object if, for all `fs'` such that `length fs' < length fs`, `Acc lengthOrder fs'` is also an object.

```
Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
  Acc_intro : (forall y:A, R y x → Acc R y) → Acc R x.
```

Excerpt #15 Coq.Init.Wf

This construction chain must end at some point, because there can only be so many lengths smaller than the previous before `length 0` is included (more specifically, `lengthOrder` is well-founded). As such, structural recursion can be established. Whereas in Excerpt #12 the successor function constructor was peeled off, here we peel off the accessibility constructor. And so if we take the object `Acc lengthOrder fs` to be our decreasing argument we can establish terminating recursion. Let's see how this is implemented, first schematically and then specifically.

Sketch

In the previous setting (see Figure 2), we saw that Coq could not establish termination of the recursive process. In each iteration, the main argument actually *was* decreasing, but this was obscured by the intervention of `map` and `stratify_sub`. We prevent this here by literally keeping a side note of the measure of decreasingness.

A paired argument structure is used where the string `fs` is held in the first part, and its measure of decreasingness in the second part, `Acc lengthOrder fs`.

Let's have a quick step by step through the recursion as outlined schematically in Figure 3. Firstly, `stratify` is merely a wrapper function, initiating the main function, `stratify'`. This function then constructs a term where `fl_max` of `fs` is the head, and `split_fstring_sigT fs` are the arguments (red arrows) These arguments are then processed recursively by `stratify'` (blue arrows), until a full term is constructed. When these recursive calls are made, the decreasing argument has become smaller by at least one constructor of `Acc`.

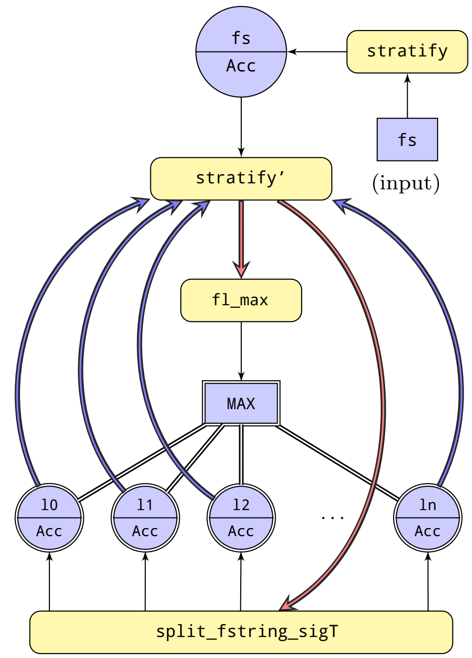


Figure 2: terminating recursion in `stratify`

Specifics

For a more technical description of this, consider the following. As the definition of `stratify'` shows in Excerpt #16 below, `stratify'` builds a `vterm` by taking two arguments, a French string `fs` and some proof `a` of `Acc lengthOrder fs`. It then calls the constructor `@Fun` with three arguments. Firstly, `fs_Sig`, the signature over French strings. Secondly, `fl_max fs`, the maximal elements in `fs`. This will be the term's head. Thirdly, the function `stratify'` mapped over the term's arguments, as computed by `split_fstring_sigT`.

```
Fixpoint stratify' (fs : fstring) (a  : Acc lengthOrder fs)
                       {struct a} : vterm :=
  @Fun fs_Sig (fl_max fs)
                (map (fun e ⇒ stratify' (projT1 e) (Acc_inv a (projT2 e)))
                     (split_fstring_sigT fs)
                ).
```

Excerpt #16

Let's zoom into this third argument a little. `split_fstring_sigT fs` returns a list of non-maximal substrings $\texttt{fs}_i$ of `fs`, each paired with a proof that `lengthOrder` $\texttt{fs}_i$ `fs`. Now, `map` takes each of these pairs and applies `fun e ⇒ stratify' (projT1 e) (Acc_inv a (projT2 e))` to them. In each of these applications of `stratify'`, it is thus given two arguments. Firstly, `projT1` $\texttt{fs}_i$: the first projection of $\texttt{fs}_i$. Secondly, `Acc_inv a (projT2` $\texttt{fs}_i$`)`: its second projection with `Acc_inv a` applied to it. This is a lemma that derives precisely the required decreasingness measure, `Acc lengthOrder` $\texttt{fs}_i$, given a proof "a" that `Acc lengthOrder fs`, and a proof that `lengthOrder` $\texttt{fs}_i$ `fs`.

```
Lemma Acc_inv :
  forall (A : Type) (R : A → A → Prop) (x : A),
    Acc R x → forall y : A, R y x → Acc R y.
```

Excerpt #17 Coq.Init.Wf

`R` would bind here to `lengthOrder`, `y` to $\texttt{fs}_i$ and `x` to `fs`, resulting in the conclusion that `Acc lengthOrder` $\texttt{fs}_i$. And so the next iteration of `stratify'` has a substring to process and an anchor point for its terminating recursion.

The main function, `stratify`, is thus defined as follows.

```
Definition stratify (fs : fstring) : vterm:=
  stratify' fs (lengthOrder_wf fs).
```

Excerpt #18

Given `fs`, `stratify` calls `stratify' fs lengthOrder_wf`, where `lengthOrder_wf` is a proof that `Acc lengthOrder fs`. More specifically, `lengthOrder_wf` is a proof that `well_founded lengthOrder`. Given some relation `R` on instances of `A`, `well_founded R` means that *`forall`* `a : A, Acc R a`.

This concludes our description of `stratify`. Before we move on to Section 2.1.4 about flattening, some definitions are expanded upon in the epilogue to this section, that would have distracted from the main subject should we have done so at their first mention.

Epilogue

In this epilogue to Section 2.1.3, we expand upon these definitions used in `stratify'`: `fl_max`, `split_fstring'`, `split_fstring_sigT'`.

`fl_max`

To determine what letters to select for a term's head, we use `fl_max`.

```
Definition not_below f L :=
  Forall (fun x ⇒ ~ fl_Lt f x) L.
```

Excerpt #19

We define a letter `f` to be *maximal* in `L` if there is no letter `x` in `L` such that `fl_Lt f x`. That is, if `not_below f L`.

```
Fixpoint fl_max' (rest static : fstring) : fstring :=
  match rest with
  | x :: xs ⇒ match (not_below_dec x static) with
              | left  _ ⇒ x :: fl_max' xs static
              | right _ ⇒      fl_max' xs static
              end
  | nil     ⇒ nil
  end.

Definition fl_max (L : fstring) : fstring :=
  fl_max' L L.
```

Excerpt #20

Given an `fstring L`, `fl_max` returns the list of all French letters *maximal* in `L`. Internally, `fl_max` is merely an interface for `fl_max'`, calling it with the same `fstring` twice.

Given two French strings `rest` and `static`, `fl_max'` computes the list of all letters maximal in `L` by taking each letter `x` of `rest`, and adding it to the end result if `not_below_dec x static`. Otherwise, `x` is discarded and `fl_max'` continues with the remainder of `rest`. Here, `not_below_dec` is a lemma that proves decidability of `not_below`. Recall from Section 1.1 that decidability is not inherent to our logic.

Example. Suppose again that we have an order $>$ such that $m > \kappa, \ell$. Then

```
 fl_max' (m̀ :: l̀ :: ḱ :: ḿ :: nil) (m̀ :: l̀ :: ḱ :: ḿ :: nil)          =
m̀ :: (fl_max' (l̀ :: ḱ :: ḿ :: nil) (m̀ :: l̀ :: ḱ :: ḿ :: nil)) :: nil   =
m̀ :: (fl_max'      (ḱ :: ḿ :: nil) (m̀ :: l̀ :: ḱ :: ḿ :: nil)) :: nil   =
m̀ :: (fl_max'           ḿ :: nil) (m̀ :: l̀ :: ḱ :: ḿ :: nil)) :: nil   =  m̀ :: ḿ ::  nil.
```

split_fstring'

For any `fstring L`, the complement of `(fl_max L)` in `L` is `(split_fstring' L)`. This will become more apparent in Section 2.2.2, where we prove that inverting this decomposition results in the original string. Whereas `(fl_max' L)` returns the head of `stratify L`, `(split_fstring' L)` computes the list of scattered substrings that will be recursed upon to form the arguments of `stratify L`. This corresponds to $\bigcup_i^n \{s_i\}$, with s_i from Definition 6.

```
Fixpoint split_fstring' (fs lx max : fstring) : list fstring :=
  match max with
  | nil    ⇒ (lx ++ fs) :: nil
  | mi :: M ⇒ match fs with
              | nil    ⇒ nil
              | li :: L ⇒ match (eq_fletter_dec li mi) with
                          | left  _ ⇒ lx :: split_fstring' L nil M
                          | right _ ⇒      split_fstring' L (lx ++ (li :: nil)) (mi :: M)
                          end
              end
  end.
```

Excerpt #21

Let's see how this works. `split_fstring'` takes three arguments. The main argument is `fs`. Then we have `lx`, which is the substring currently being constructed. Finally, `max` is the list of letters maximal in `fs`.

Example. Take `m` and `l` with `m` maximal. Suppose we have

```
max  =        m̀  m̀
fs   =  ĺ  l̀  m̀  ḿ  ĺ
```

Here, `max` has the letters maximal in `fs` with their order of appearance preserved. Now, for each letter `mi` in `max`, `split_fstring'` keeps comparing it to the next letter `li` in `fs`. So, for clarity, during this whole recursive process, `mi` is always the next maximal letter in (what's left of) `fs`. When comparing `mi` to `li`, the next letter in `fs`, we can thus infer that if they are *not* the same, `li` is not maximal (since `mi` is the next maximal letter in `fs`). In that case, `li` is added to `lx`. If they *are* the same, then there are no more non-maximal letters before `mi` in `fs`. The list `lx` is added to the result, and the process is continued with the next letter in `max`, until `max` is empty. At that point, the rest of `fs`, which is all non-maximal, is added to the result, and the recursion is complete.

In this example, the result would be the list `(ĺ :: l̀ :: nil) :: nil :: (ĺ :: nil) :: nil`.

split_fstring_sigT'

How does this relate to `split_fstring_sigT'`?

```
Definition split_fstring_sigT'
  (fs : fstring) (input  : list {x : fstring & In x (split_fstring fs)})
    : list {x : fstring & lengthOrder x fs}.
```

Excerpt #22

Basically, `split_fstring_sigT'` is a version of `split_fstring'`, adapted for `stratify'`. It takes the result of `split_fstring fs` and pairs each element `e` with a proof that `lengthOrder e fs`. This paired data type allows for `stratify` to establish a terminating recursive process, as discussed in Section 2.1.3 above.

This concludes the epilogue to Section 2.1.3. In Section 2.1.4, we describe the inverse to stratification, called *flattening*.

2.1.4 Flattening

This section is about the operation inverse to stratification, called *flattening*. Its explanation is more succinct compared to that of stratification, because the operation itself is simpler, both to define and to implement. Flattening is to the head of a term and `interleave` what stratification is to `fl_max` of a string and `split_fstring`. More specifically,

Definition 7. *Flattening*, denoted $\flat$, maps each French term to its corresponding French string. This is defined inductively as follows: $\varepsilon^\flat = \varepsilon$, and $(\hat{\ell}_1 \ldots \hat{\ell}_n(t_0, \ldots, t_n))^\flat = t_0^\flat \hat{\ell}_1 \ldots \hat{\ell}_n t_n^\flat$.

Example. Suppose we have an order $\succ$ such that $m \succ \kappa, \ell$. Then

$$(\grave{m}\acute{m}(\acute{\ell}\grave{\ell}(\varepsilon,\varepsilon,\varepsilon),\varepsilon,\grave{\ell}(\varepsilon,\varepsilon)))^\flat = (\acute{\ell}\grave{\ell}(\varepsilon,\varepsilon,\varepsilon))^\flat \grave{m}\varepsilon^\flat \acute{m}(\grave{\ell}(\varepsilon,\varepsilon))^\flat = \varepsilon^\flat \acute{\ell}\varepsilon^\flat \grave{\ell}\varepsilon^\flat \grave{m}\acute{m}\varepsilon^\flat \grave{\ell}\varepsilon^\flat = \acute{\ell}\grave{\ell}\grave{m}\acute{m}\grave{\ell}.$$

Implementation

Flattening a term in Coq is done as follows. Given a French term `vt`, `flatten` takes the `head` and list of arguments `args`, and uses `interleave` to put the first element of `args` before the first letter of `head`, then put the second element before the second letter, et cetera. `flatten` is then recursively applied to each element of `args`.

```
Fixpoint interleave ( h : fstring ) ( t : list fstring ) : fstring :=
  match h, t with
  | head, nil        ⇒ head
  | nil, _           ⇒ fold_right (fun x y ⇒ app x y) nil t
  | x :: xs, l :: ll ⇒ l ++ x :: (interleave xs ll)
  end.
...
Fixpoint flatten (vt : vterm) : fstring :=
  match vt with
  | Var _          ⇒ empty_fstring
  | Fun head args ⇒ interleave head (map flatten args)
  end.
```

Excerpt #23

This concludes the description of our framework. Next, in Section 2.2, we prove this framework to be correct.

2.2 Proving the framework to be correct

In this section we go over a multi-part proof of correctness for our framework. To be correct, within the framework every French string should uniquely correspond to a single French term and every French term should uniquely correspond to a single French string. This correspondence is established via a bijection between the domain of French strings and the domain of French terms. That is, a one-on-one relation between French strings and French terms. This is captured in Lemma 12, displayed below.[1]

Lemma 12. *The functions* $\sharp$ *and* $\flat$ *are each other's inverse. In other words,* $\flat \circ \sharp$ *and* $\sharp \circ \flat$ *are the identity.*

Our proof of Lemma 12 in Coq will be threefold. Each segment of our threefold proof is described in its own subsection, as sketched out in Figure 4.

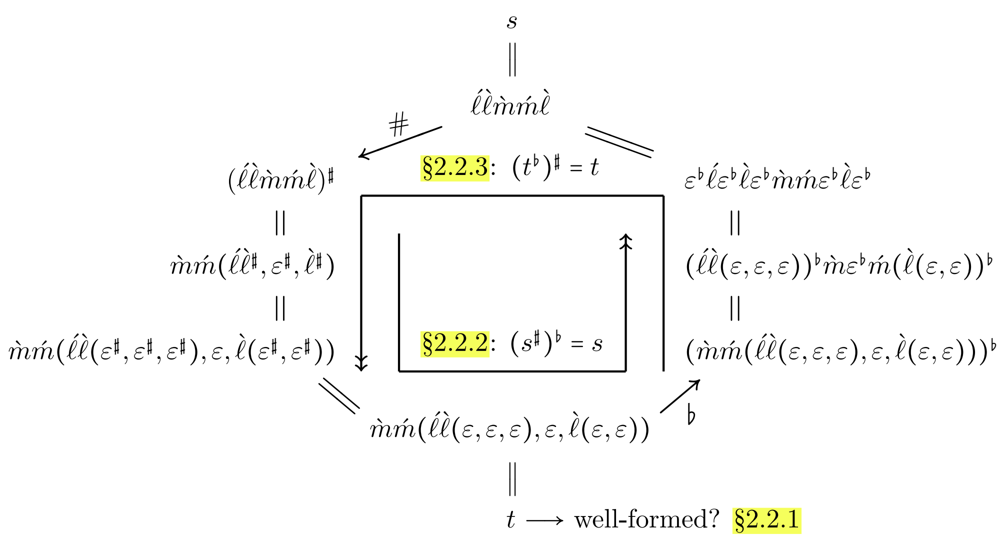


Figure 4: contents of Section 2.2

In Section 2.2.1, we prove that `vt_wellformed t` holds for any term `t` generated by `stratify`. That is, French terms generated by our stratification function are well-formed: it holds that, `forall fs, vt_wellformed (stratify fs)`. Next, in Section 2.2.2, we prove that $\flat \circ \sharp$ (flatten after stratify) is equivalent to the identity function: it holds that, `forall fs, flatten (stratify fs) = fs`. Finally, in Section 2.2.3 we prove that $\sharp \circ \flat$ (stratify after flatten) is equivalent to the identity function.

[1] the name of this lemma, "Lemma 12", was adopted directly from Van Oostrom's article[a] for sake of clarity

2.2.1 Well-formedness of stratify

In our framework, the properties described in Definition 4 should hold for any term `t` created by `stratify`. That is, it should hold that, `forall fs, vt_wellformed (stratify fs)`. Otherwise, we might be working with terms that are not French terms. Recall from Section 2.1.2 that *"`CoLoR`'s `term` data type is expressive enough for our purposes but not strict enough, as neither arity nor the Hoare order are imposed. These additional constraints are implemented via `vt_wellformed`"*.

We will first examine a construct used in proving this, called *unfold once*, then break down the proof itself, and then expand upon several auxiliary lemmas used in the proof.

Prologue

In this prologue to Section 2.2.1, we explain *unfold once* in preparation for the proof of `vt_wellformed_stratify`.

unfold once

S24 The tactic used to replace a function with its definition is `unfold`. However, what happens if we apply this to a function that is recursively defined? Applying `unfold` to `stratify` here gives us the `(fix ...)` format as displayed in State S25, which is highly undesirable.

```
1 subgoals                                                          S24
x : fstring
IH : forall y, lengthOrder y x → vt_wellformed (stratify y)
_____________________________________________(1/1)
vt_wellformed (stratify x)
```

`intros x IH. unfold stratify. unfold stratify'.`

S25 This happens because Coq is unclear as to what the end result of the *recursive* unfolding will be. It keeps the 'result so far' embedded inside of a new, locally defined function, because the end result could be anything as far as Coq is concerned. This is inconvenient, because no tactics can be applied to embedded components of such a partially unfolded function.

```
1 subgoals                                                          S25
x : fstring
IH : forall y, lengthOrder y x → vt_wellformed (stratify y)
_____________________________________________(1/1)
vt_wellformed
  ((fix stratify' (fs : fstring) (a : Acc lengthOrder fs)
                                         {struct a} : vterm :=
   Fun (fl_max fs)
     (map
       (fun e0 ⇒ stratify' (projT1 e0) (Acc_inv a (projT2 e0)))
       (split_fstring_sigT fs))) x (lengthOrder_wf x))
```

So we want to get rid of this embedding. This is done by use of a lemma dedicated to this, `unfold_1_stratify'`. Rather than attempting to lay bare the full definition, it reveals the non-recursive part and leaves the recursive part folded. Throughout the project, we have labelled such lemmas `unfold_1_f`, with `f` being the function unfolded one layer deep, in this case `stratify'`.

```
Lemma unfold_1_stratify' :
 forall (x : fstring) (a : Acc lengthOrder x),
  stratify' x a = @Fun fs_Sig (fl_max x)
                              (map (fun e ⇒ stratify' (projT1 e) (Acc_inv a (projT2 e)))
                                   (split_fstring_sigT x)
                              ).
```

Excerpt #26

S27 applying `unfold_1_stratify'` rather than `unfold stratify'` we arrive at the following proof state (compare State S25). The head is not embedded inside of a locally defined function, making it readily accessible to our tactics.

```
1 subgoals                                                          S27
x : fstring
IH : forall y, lengthOrder y x → vt_wellformed (stratify y)
_____________________________________________(1/1)
vt_wellformed
  (Fun (fl_max x)
     (map
        (fun e0 : {x0 : fstring & lengthOrder x0 x} ⇒
         stratify' (projT1 e0)
                   (Acc_inv (lengthOrder_wf x) (projT2 e0)))
        (split_fstring_sigT x)))
```

Proof

Now onto the actual proof itself. Our proof of the statement shown below in Excerpt #28 will follow the structure of `vt_wellformed`'s definition as indicated by the circled numbers in Excerpt #8.

```
Lemma vt_wellformed_stratify :
  forall fs,
    vt_wellformed (stratify fs).
```

Excerpt #28

S29 We do well-founded induction using `lengthOrder`, as structural recursion in `stratify'` is based on `lengthOrder` (see Section 2.1.3).

```
1 subgoals                                              S29
______________________________________(1/1)
forall fs : fstring, vt_wellformed (stratify fs)
```

`apply well_founded_ind with lengthOrder; [apply lengthOrder_wf|idtac].` (2997)

S30 Given that, for all `fstring` `y` *shorter* than `x`, `stratify y` is well-formed (that is, given the induction hypothesis), we now have to show that `stratify x` too is well-formed. We introduce the induction hypothesis as `IH`, and begin to unfold `stratify` (*once*).

```
1 subgoals                                              S30
______________________________________(1/1)
forall x,
  (forall y, lengthOrder y x → vt_wellformed (stratify y))
  → vt_wellformed (stratify x)
```

`intros x IH. unfold stratify. rewrite unfold_1_stratify'.` (2998)

S31 We'd also like to unfold `vt_wellformed` once. We can dismiss the `Var _` case[11] (item 1 in Excerpt #8): as revealed by `unfold_1_stratify'`, the result of `stratify` begins with "`@Fun fs_Sig (fl_max..`", and such a term is not a variable.

So the first real hurdle will be item 2 : is it the case that after `stratify x` either `vterm_empty` holds or `vterm_not_empty`? We begin to answer this by doing case analysis on `x`.[16]

```
1 subgoals                                              S31
x : fstring
IH : forall y, lengthOrder y x → vt_wellformed (stratify y)
______________________________________(1/1)
vt_wellformed
  (Fun (fl_max x)
     (map
        (fun e0 : {x0 : fstring & lengthOrder x0 x} ⇒
         stratify' (projT1 e0)
                   (Acc_inv (lengthOrder_wf x) (projT2 e0)))
        (split_fstring_sigT x)))
```

`case_eq x; intros.` (2999)
`rewrite unfold_1_vt_wellformed. simpl.` (3000)

Case `x = nil`

S32 This gives us two subgoals. The first subgoal, where `x = nil`, corresponds to the left-hand side of item 2 in Excerpt #8.

Firstly, `fl_max nil = nil` holds by definition. Secondly, `split_fstring_sigT nil = nil`, the other half of `vterm_is_empty (stratify nil)`, is already simplified here to `nil = nil`, which is trivial (`split_fstring_sigT` maps over `split_fstring`, and `map f nil = nil` for any function `f`).

```
2 subgoal                                               S32
IH : forall y, lengthOrder y nil → vt_wellformed (stratify y)
______________________________________(1/2)
(fl_max nil = nil ∧ nil = nil) ∨
(fl_max nil ≠ nil ∧ nil ≠ nil
  ∧ 0 = 0 ∧ fs_incomparable (fl_max nil)
  ∧ Forall (fun x0 ⇒ hoare_lt (vt_head x0) (fl_max nil)) nil
  ∧ True)
______________________________________(2/2)
...
```

`left. split; trivial.` (3000)

Case `x ≠ nil`

S33 We are then left to prove the second subgoal, in which `x = f::l`, ie. `x` not empty.

Let's unfold `vt_wellformed` once, and break down what this well-formedness implies into seperate subgoals by repeatedly applying `split` to the conjunction on the right-hand side of item (2) in Excerpt #8.

```
1 subgoals                                                   S33
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
                  vt_wellformed (stratify y)
________________________________________________(1/1)
vt_wellformed
  (Fun (fl_max (f::l))
    (map
      (fun e0 : {x0 : fstring & lengthOrder x0 (f::l)} ⇒
        stratify' (projT1 e0)
          (Acc_inv (lengthOrder_wf (f::l)) (projT2 e0)))
      (split_fstring_sigT (f::l))))
```

```
rewrite unfold_1_vt_wellformed. right. (3001)
split. Focus 2. split. Focus 2. (...) Unfocus. Unfocus. (3002)
```

S34 This brings us to State S34. `X` and `Y` are edited in for reduction of clutter. Note that subgoals 2 to 5 here correspond to items (3) to (6) in Excerpt #8. Let's run by each subgoal, starting with subgoal 1, in which we are to prove that `vterm_not_empty (Fun X Y)`. That is, neither the head of `Fun X Y` nor its list of arguments is empty (item (3)).

```
5 subgoal                                                    S34
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
                  vt_wellformed (stratify y)
X := fl_max (f::l)
Y := (map (fun e0 : {x0 : fstring & lengthOrder x0 (f::l)} ⇒
      stratify' (projT1 e0) (Acc_inv (lengthOrder_wf (f::l))
      (projT2 e0))) (split_fstring_sigT (f::l)))
________________________________________________(1/5)
vterm_not_empty (Fun X Y)
________________________________________________(2/5)
ar X = length Y
________________________________________________(3/5)
fs_incomparable X
________________________________________________(4/5)
Forall (fun x0 : vterm ⇒ hoare_lt (vt_head x0) X) Y
________________________________________________(5/5)
lforall vt_wellformed Y
```

```
Focus 1. unfold vterm_not_empty. split. (3003)
```

State S34 *subgoal 1*

S35 Firstly, we prove that `X`, `fl_max (f::l)` is not empty, as `f::l` is not empty. This is proven by `fl_max_neq_nil` :

`forall fs, fs ≠ nil → fl_max fs ≠ nil.`

Intuitively, this already makes sense: any non-empty list should have at least one element maximal for that list Next, `neq_nil` proves that `f::l` is a non-empty list.

```
2 subgoal                                                    S35
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
                  vt_wellformed (stratify y)
________________________________________________(1/2)
fl_max (f::l) ≠ nil
________________________________________________(2/2)
map (fun e0 : {x : fstring & lengthOrder x (f::l)} ⇒
   stratify' (projT1 e0) (Acc_inv (lengthOrder_wf (f::l))
   (projT2 e0))) (split_fstring_sigT (f::l)) ≠ nil
```

```
apply fl_max_neq_nil. apply neq_nil. (3004)
```

S36 Secondly, we prove that `Y` is not empty. Will this list of arguments, after each element having `stratify'` applied to it recursively, be non-empty? Well, suppose it were empty. Then, from `map_eq_nil` (`∀f: map f nil = nil`) it would follow that `split_fstring_sigT (f::l)` were also empty. But `split_fs_sigT_neq_nil` proves this to be false: `forall fs,`

`fs ≠ nil → split_fstring_sigT fs ≠ nil.`

And so we dismiss the subgoal by contradiction.

```
1 subgoals                                                   S36
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
                        vt_wellformed (stratify y)
________________________________________________(1/1)
map
  (fun e0 : {x : fstring & lengthOrder x (f::l)} ⇒
   stratify' (projT1 e0) (Acc_inv (lengthOrder_wf (f::l))
   (projT2 e0))) (split_fstring_sigT (f::l)) ≠ nil
```

```
intro. apply map_eq_nil in H. apply split_fs_sigT_neq_nil in H. (3005)
assumption. apply neq_nil. (3006)
```

State S34 subgoal 2

S37 Next, we want to establish that the term *has arity* (item 2 in Excerpt #8) This property is captured in `ar`. By `map_length` we know that `length (map f L)` equals `length L`.

We want to make a case distinction on `fl_max (f::l)`. This way we can split our subgoal according to `ar`'s definition, generating two new subgoals. One for which `fl_max (f::l)` is `nil`, and one for which it is not.

```
4 subgoal                                                      S37
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
               vt_wellformed (stratify y)
______________________________________________(1/4)
ar (fl_max (f::l)) =
length (map
    (fun e0 : {x : fstring & lengthOrder x (f::l)} ⇒
     stratify' (projT1 e0) (Acc_inv (lengthOrder_wf (f::l))
     (projT2 e0))) (split_fstring_sigT (f::l)))
```

```
rewrite map_length. unfold ar. (3007)
case_eq (fl_max (f::l)); intros;[idtac|rewrite ← H]. (3008)
```

Case `fl_max (f::l) = nil`

S38 The first case has a false assumption, `H`. As we saw in State S35, `fl_max L` is not empty for any non-empty list `L`. By this contradictory assumption we can dismiss the first case.

```
5 subgoal                                                      S38
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
              vt_wellformed (stratify y)
H : fl_max (f::l) = nil
______________________________________________(1/5)
0 = length (split_fstring_sigT (f::l))
______________________________________________(2/5)
1 + length (fl_max (f::l)) =
    length (split_fstring_sigT (f::l))
```

```
apply fl_max_neq_nil in H;[idtac|apply neq_nil]. contradiction H. (3009)
```

Case `fl_max (f::l) ≠ nil`

S39 `split_fstring_sigT` is a function mapped over `split_fstring` to convert a list of sublists of `L` into a list of pairs `(s,p)`, where `s` is a sublist of `L` and `p` a proof of this fact. We see in the current subgoal a comparison of lengths. The conversion made by `split_fstring_sigT` is length preserving, so we can strip this layer, effectively replacing `split_fstring_sigT` with `split_fstring`.

```
4 subgoal                                                      S39
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
              vt_wellformed (stratify y)
______________________________________________(1/4)
1 + length (fl_max (f::l)) =
    length (split_fstring_sigT (f::l))
```

```
unfold split_fstring_sigT. unfold split_fstring_sigT'. (3010)
rewrite map_length. rewrite from_list_length. (3011)
```

We have captured the ratio between `vt_wellformed` and `split_fstring` in the lemma `ar_holds`, which states:

```
Lemma ar_holds :
  forall fs,
      (fs =  nil ∧                            length (split_fstring fs) = 0)
    ∨ (fs ≠  nil ∧  1 + length (fl_max fs) = length (split_fstring fs)).
```

Excerpt #40

S41 So we address the current subgoal by adding to the list of hypotheses `ar_holds` applied to `f::l`, and splitting its disjunction, leading to State S42 and State S43.

```
4 subgoal                                                      S41
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
               vt_wellformed (stratify y)
______________________________________________(1/4)
1 + length (fl_max (f::l)) = length (split_fstring (f::l))
```

```
cut (f::l = nil ∧ length (split_fstring (f::l)) = 0 ∨ ...);
    [intro|apply ar_holds]. do 2 destruct H0. (3016)
```

S42 The left disjunct of `ar_holds` represents the case where `fs = nil`, which is here false, so by this false assumption, the current subgoal can be dismissed.

```
5 subgoal                                                    S42
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
         vt_wellformed (stratify y)
H0 : f::l = nil
H1 : length (split_fstring (f::l)) = 0
______________________________________(1/5)
1 + length (fl_max (f::l)) = length (split_fstring (f::l))
```

`inversion H0.` (3016)

S43 The right disjunct of `ar_holds`, the case where `fs ≠ nil`, precisely matches the current subgoal, so we close it by this assumption.

```
4 subgoal                                                    S43
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
         vt_wellformed (stratify y)
H0 : f::l ≠ nil
H1 : 1 + length (fl_max (f::l)) =
       length (split_fstring (f::l))
______________________________________(1/4)
1 + length (fl_max (f::l)) = length (split_fstring (f::l))
```

`assumption.` (3016)

State S34 subgoal 3

S44 The stratified term's head is an `fstring` that should consist of mutually incomparable letters (item 4 in Excerpt #8). This is proven directly by lemma `fl_max_incomparable` (for its proof see Excerpt #72), which states:

`forall L, fs_incomparable (fl_max L).`

```
3 subgoal                                                    S44
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
         vt_wellformed (stratify y)
______________________________________(1/3)
fs_incomparable (fl_max (f::l))
```

`apply fl_max_incomparable` (3017)

State S34 subgoal 4

S45 Next, we show that `hoare_lt` holds between the head of a stratified term and the heads of its arguments. This corresponds to item 5 in Excerpt #8, which states:

`Forall (fun x ⇒ hoare_lt (vt_head x) f) args.`

Let's first decompose `Forall`. By `Forall_forall`:

```
forall P l,
    Forall P l ↔ (forall e, In e l → P e),
```

item 5 would state

`forall x, In x args → hoare_lt (vt_head x) f.`

We apply `Forall_forall` to our current subgoal, adding the premises (`x` and `In x args`) to our list of hypotheses using `intros`.

```
2 subgoal                                                    S45
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
         vt_wellformed (stratify y)
______________________________________(1/2)
Forall (fun x ⇒ hoare_lt (vt_head x) (fl_max (f::l)))
  (map
    (fun e0 : {x : fstring & lengthOrder x (f::l)} ⇒
     stratify' (projT1 e0) (Acc_inv (lengthOrder_wf (f::l))
     (projT2 e0))) (split_fstring_sigT (f::l)))
```

`apply Forall_forall. intros.` (3018)

S46 We apply `in_map_iff`, which states

```
forall f l y,
  In y (map f l) ↔ (exists x, f x = y ∧ In x l),
```

and thus we separate `y`'s membership of `l` from its having `f` applied to it.

```
2 subgoal                                                    S46
...
x : vterm
H : In x (map
        (fun e : {x : fstring & lengthOrder x (f::l)} ⇒
          stratify' (projT1 e)
                (Acc_inv (lengthOrder_wf (f::l))
                     (projT2 e)
                )
        )
        (split_fstring_sigT (f::l))
      )
______________________________________(1/2)
hoare_lt (vt_head x) (fl_max (f::l))
```

`apply in_map_iff in H.` (3018)

S47 Now that we have reformulated `H`, let's create a witness and isolate its properties.

```
2 subgoal                                                    S47
...
x : vterm
H : exists x0 : {x : fstring & lengthOrder x (f::l)},
            stratify' (projT1 x0)
                      (Acc_inv (lengthOrder_wf (f::l))
                               (projT2 x0)
                      ) = x ∧
            In x0 (split_fstring_sigT (f::l))
________________________________________(1/2)
hoare_lt (vt_head x) (fl_max (f::l))
```

`do 2 destruct H. destruct x0.` (3018)

S48 The hypothesis `l0` then allows us, by the induction hypothesis `IH`, to add to our list of hypotheses that `vt_wellformed (stratify x0)`: since our newly created witness `x0` relates to `f::l` in `lengthOrder` (by `l0`) we know by `IH` that `vt_wellformed (stratify x0)` must hold.

```
2 subgoal                                                    S48
...
IH : forall y, lengthOrder y (f::l) →
                 vt_wellformed (stratify y)
x0 : fstring
l0 : lengthOrder x0 (f::l)
H :  stratify' (projT1 x0)
           (Acc_inv (lengthOrder_wf (f::l))
                    (projT2 x0)
           ) = x
H0 : In (existT x0 l0) (split_fstring_sigT (f::l))
________________________________________(1/2)
hoare_lt (vt_head x) (fl_max (f::l))
```

`cut (vt_wellformed (stratify x0));[intro|apply IH; assumption].` (3019)

S49 We then generalize `H0`. A sublist `s`, paired with proof `p` of its being a sublist, only occurs in `split_fstring_sigT` if `s` is in `split_fstring`. This fact is captured in `split_fs_sigT_drop_proofs`:

```
forall fs li p,
    In (existT li p) (split_fstring_sigT fs) →
        In li (split_fstring fs).
```

And `split_fstring_sigT` in `H0` is thus replaced by `split_fstring`. Next, let's add `stratify x0 = x` to our hypotheses.

```
2 subgoal                                                    S49
...
x : vterm
x0 : fstring
l0 : lengthOrder x0 (f::l)
H :  stratify' (projT1 x0)
           (Acc_inv (lengthOrder_wf (f::l))
                    (projT2 x0)
           ) = x
H0 : In (existT x0 l0) (split_fstring_sigT (f::l))
H1 : vt_wellformed (stratify x0)
________________________________________(1/2)
hoare_lt (vt_head x) (fl_max (f::l))
```

`apply split_fs_sigT_drop_proofs in H0.` (3020)
`cut (stratify x0 = x);[intro|idtac]. Focus 2.` (3021)

justification for `stratify x0 = x`

S50 Proof for this is provided by `H`: the specific proof of `Acc lengthOrder x` given to `stratify'` is irrelevant (as captured in `proof_irrelevance`), so we can generalize `H` to `stratify x0 = x`. Recall from Excerpt #18 that `stratify fs` is merely a shorthand for `stratify' fs (Acc lengthOrder fs)`.

See Excerpt #61 for more on `proof_irrelevance`.

```
1 subgoals                                                   S50
...
H :  stratify' (projT1 x0)
           (Acc_inv (lengthOrder_wf (f::l))
                    (projT2 x0)
           ) = x
H0 : In x0 (split_fstring (f::l))
________________________________________(1/1)
stratify x0 = x
```

`rewrite ← H. apply proof_irrelevance.` (3022)
`rewrite ← H2. clear H2.` (3023)

$\boxed{S_{51}}$ Back to the subgoal at hand. We will be using the lemma `hoare_Lt_split_max : forall L, L ≠ nil → hoare_Lt (split_fstring L) (fl_max L)` (see Excerpt #68 for its proof). `hoare_Lt` simply raises `hoare_lt`, from comparing one list to one list, to comparing multiple lists to one list. In other words, `hoare_Lt_split_max` states that `hoare_lt` holds between the head of an `fterm` and each of its (unstratified) arguments.

However, our current subgoal is about `hoare_lt` between the head of `f::l` and the *head* of one of its arguments (stratified). So let us first transform this by means of `hoare_lt_sublist_congr1`:

```
forall M' L M,
  sublist M M' → hoare_lt M' L → hoare_lt M L.
```

```
2 subgoal                                              S51
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
           vt_wellformed (stratify y)
x : vterm
x0 : fstring
l0 : lengthOrder x0 (f::l)
H :  stratify' (projT1 x0)
        (Acc_inv (lengthOrder_wf (f::l))
                (projT2 x0)
        ) = x
H0 : In x0 (split_fstring (f::l))
H1 : vt_wellformed (stratify x0)
_________________________________________(1/2)
hoare_lt (vt_head (stratify x0)) (fl_max (f::l))
```

`apply hoare_lt_sublist_congr1 with x0.` (3024)

$\boxed{S_{52}}$ We have replaced `vt_head (stratify x0)` with `x0` in our subgoal. `sublist_head_fstring` provides proof that the former is a `sublist` of the latter.

```
3 subgoal                                              S52
...
_________________________________________(1/3)
sublist (vt_head (stratify x0)) x0
_________________________________________(2/3)
hoare_lt x0 (fl_max (f::l))
```

`apply sublist_head_fstring.` (3025)

$\boxed{S_{53}}$ Next, let's apply `hoare_Lt_split_max` in order to add to our hypotheses that:
`hoare_Lt (split_fstring (f::l)) (fl_max (f::l)).`

```
2 subgoal                                              S53
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
           vt_wellformed (stratify y)
x : vterm
x0 : fstring
l0 : lengthOrder x0 (f::l)
H :  stratify' (projT1 x0)
        (Acc_inv (lengthOrder_wf (f::l))
                (projT2 x0)
        ) = x
H0 : In x0 (split_fstring (f::l))
H1 : vt_wellformed (stratify x0)
_________________________________________(1/2)
hoare_lt x0 (fl_max (f::l))
```

`cut (hoare_Lt (split_fstring (f::l)) (fl_max (f::l)));`
`[intro|apply hoare_Lt_split_max; apply neq_nil].` (3027)

$\boxed{S_{54}}$ As explained above at State S_{51}, given some `list fstring LL` and an `fstring fs`, to say that `hoare_Lt LL fs`, is to say that for any member `m` of `LL`, `hoare_lt m fs` holds. We rewrite `H2` in accordance to this.

```
2 subgoal                                              S54
...
H0 : In x0 (split_fstring (f::l))
H2 : hoare_Lt (split_fstring (f::l)) (fl_max (f::l))
_________________________________________(1/2)
hoare_lt x0 (fl_max (f::l))
```

`unfold hoare_Lt in H2. rewrite Forall_forall in H2.` (3028)

$\boxed{S_{55}}$ Well, we have by `H0` that `x0` is a member of `split_fstring (f::l)`. So from this it follows that `hoare_lt x0 (fl_max (f::l))`, concluding this subgoal (ie. subgoal 4 of State S_{34}).

```
2 subgoal                                              S55
...
H0 : In x0 (split_fstring (f::l))
H2 : forall x, In x (split_fstring (f::l)) →
          hoare_lt x (fl_max (f::l))
_________________________________________(1/2)
hoare_lt x0 (fl_max (f::l))
```

`apply H2. assumption.` (3029)

State S_{34} subgoal 5

$\boxed{S_{56}}$ Last but not least, we proceed to show that the arguments of a stratified term are also well-formed, ie. item 6 in Excerpt #8.

The approach we will employ is very similar to that of subgoal 4: move properties of the list under investigation (`map ...`) to the list of hypotheses, split properties, and derive the subgoal from `IH` using the isolated properties. Let's begin by splitting `lforall` using `lforall_intro`:

```
forall P l,
   (forall x : A, In x l → P x) → lforall P l.
```

```
1 subgoals                                                    S56
f : fletter
l : list fletter
IH : forall y, lengthOrder y (f::l) →
             vt_wellformed (stratify y)
______________________________________(1/1)
lforall vt_wellformed
  (map
    (fun e0 : {x : fstring & lengthOrder x (f::l)} ⇒
     stratify' (projT1 e0) (Acc_inv (lengthOrder_wf (f::l))
     (projT2 e0))) (split_fstring_sigT (f::l)))
```

`apply lforall_intro. intros.` (3030)

$\boxed{S_{57}}$ Again, similar to what we did at State S_{47}, we break down `H` into its constituent parts using `in_map_iff`.

```
1 subgoals                                                    S57
...
x : vterm
H : In x (map
     (fun e : {x : fstring & lengthOrder x (f::l)} ⇒
      stratify' (projT1 e) (Acc_inv (lengthOrder_wf (f::l))
      (projT2 e))) (split_fstring_sigT (f::l)))
______________________________________(1/1)
vt_wellformed x
```

`apply in_map_iff in H. do 2 destruct H. destruct x0.` (3030)

$\boxed{S_{58}}$ We then add `vt_wellformed (stratify x0)` to our hypotheses by virtue of `IH` and `l0`: we can use the induction hypothesis because `x0` is smaller than `f::l`, same as State S_{48}.

```
1 subgoals                                                    S58
...
IH : forall y, lengthOrder y (f::l) →
             vt_wellformed (stratify y)
x : vterm
x0 : fstring
l0 : lengthOrder x0 (f::l)
H :  stratify' (projT1 (existT x0 l0))
          (Acc_inv (lengthOrder_wf (f::l))
                 (projT2 (existT x0 l0))
          ) = x
H0 : In (existT x0 l0) (split_fstring_sigT (f::l))
______________________________________(1/1)
vt_wellformed x
```

`cut (vt_wellformed (stratify x0));[intro|apply IH; assumption].` (3031)

$\boxed{S_{59}}$ We infer from `H` that `stratify x0 = x`, similar to what we did at State S_{49}.

```
1 subgoals                                                    S59
...
H :  stratify' (projT1 (existT x0 l0))
          (Acc_inv (lengthOrder_wf (f::l))
                 (projT2 (existT x0 l0))
          ) = x
H1 : vt_wellformed (stratify x0)
______________________________________(1/1)
vt_wellformed x
```

`cut (stratify x0 = x);[intro|rewrite ← H; apply proof_irrelevance].` (3032)

$\boxed{S_{60}}$ Which in turn enables us to conclude that `vt_wellformed x` for an arbitrary argument `x` of `stratify f::l`, thus concluding the fifth and final subgoal of State S_{34}. □

```
1 subgoals                                                    S60
...
H1 : vt_wellformed (stratify x0)
H2 : stratify x0 = x
______________________________________(1/1)
vt_wellformed x
```

`rewrite ← H2. assumption.` (3033)

This concludes our proof of `vt_wellformed_stratify`.

Epilogue

In this epilogue to Section 2.2.1, we explain the following auxiliary lemmas: `proof_irrelevance`, `hoare_Lt_split_max`, `fl_max_incomparable`.

`proof_irrelevance`

The stratification of a French term does not depend on the particular instance of its decreasingness measure.

```
Lemma proof_irrelevance :
  forall (x0 : fstring) (p1 p2 : Acc lengthOrder x0),
    stratify' x0 p1 = stratify' x0 p2.
```

Excerpt #61

S62 This is captured in the above lemma, `proof_irrelevance`. We begin its proof by doing induction on the length of `x0`.

```
1 subgoals                                              S62
_______________________________________(1/1)
forall (x0 : fstring) (p1 p2 : Acc lengthOrder x0),
stratify' x0 p1 = stratify' x0 p2
```

```
intro. apply well_founded_ind with
(P := fun x0 ⇒ forall p1 p2, stratify' x0 p1 = stratify' x0 p2)
            (R := lengthOrder);[apply lengthOrder_wf|idtac];
                               intro; intro IH; intros.
```
(2822)

S63 Our induction hypothesis `IH` now tells us that the particular decreasingness measure for `stratify'` of any `y` smaller than `x` is irrelevant. In other words, our subgoal already holds for the arguments of `stratify' x _`. Let's unfold `stratify'` once and include these in our subgoal.

```
1 subgoals                                              S63
x0 : fstring
x : fstring
IH : forall y, lengthOrder y x →
       forall p1 p2, stratify' y p1 = stratify' y p2
p1 : Acc lengthOrder x
p2 : Acc lengthOrder x
_______________________________________(1/1)
stratify' x p1 = stratify' x p2
```

```
do 2 rewrite unfold_1_stratify'.
```
(2823)

S64 This reveals that the terms we are comparing are indeed very similar. The only difference between them is the base of their arguments' decreasingness measure (`p1` or `p2`). So we simplify this subgoal by removing the heads, which are equal.

```
1 subgoals                                              S64
...
_______________________________________(1/1)
Fun (fl_max x)
    (map (fun e0 ⇒ stratify' (projT1 e0)
                             (Acc_inv p1 (projT2 e0))
         ) (split_fstring_sigT x)) =
Fun (fl_max x)
    (map (fun e0 ⇒ stratify' (projT1 e0)
                             (Acc_inv p2 (projT2 e0))
         ) (split_fstring_sigT x))
```

```
apply f_equal.
```
(2824)

S65 We then arrive at an equation between the arguments of these stratifications. That is, between two mappings of `stratify'`. These take the same arguments except for `p1` and `p2`. We exploit functional extensionality to prove that these mappings are equal (see Excerpt #83 for a definition of `map_ext`).

```
1 subgoals                                              S65
...
_______________________________________(1/1)
map (fun e0 ⇒ stratify' (projT1 e0)
                        (Acc_inv p1 (projT2 e0))
    ) (split_fstring_sigT x) =
map (fun e0 ⇒ stratify' (projT1 e0)
                        (Acc_inv p2 (projT2 e0))
    ) (split_fstring_sigT x)
```

```
apply map_ext with (g :=
  (fun e0 ⇒ stratify' (projT1 e0) (Acc_inv p2 (projT2 e0)))).
```
(2825)

S66 Which leaves us with the burden to prove that the functions mapped were in fact equal. This is provided by the induction hypothesis `IH`. From the type of `a0` it follows that the premise `lengthOrder _ x` is fulfilled. This makes sense because these are the arguments of the original stratification. Recall from `split_fstring_sigT'` that they are smaller in `lengthOrder`. □

```
1 subgoals                                                S66
...
IH : forall y, lengthOrder y x →
       forall p1 p2, stratify' y p1 = stratify' y p2
________________________________________________(1/1)
forall a0 : {x1 : fstring & lengthOrder x1 x},
stratify' (projT1 a0) (Acc_inv p1 (projT2 a0)) =
stratify' (projT1 a0) (Acc_inv p2 (projT2 a0))
```

`intro. apply IH. destruct a0. simpl. assumption.` (2826)

hoare_Lt_split_max

For any French string `L`, the Hoare order holds between each element of `split_fstring L` and `fl_max L`. To appreciate our proof of the lemma corresponding to this statement, consider first `hoare_Lt`'s definition, which is simply a lifting of `hoare_lt`, from `(fstring → fstring → Prop)` to `(list fstring → fstring → Prop)`.

```
Definition hoare_Lt (LL : list fstring) (L : fstring) :=
  Forall (fun x ⇒ hoare_lt x L) LL. (*V*)
```

Excerpt #67

The lemma then reads as follows.

```
Lemma hoare_Lt_split_max :
  forall L, L ≠ nil →
    hoare_Lt (split_fstring L) (fl_max L).
```

Excerpt #68

S69 For any two lists `M` and `L`, if `hoare_Lt M L` then `hoare_Lt M (fl_max L)`. This is captured in `hoare_Lt_then_also_fl_max`, the proof of which is touched upon below.

```
1 subgoals                                                S69
________________________________________________(1/1)
forall L, L ≠ nil → hoare_Lt (split_fstring L) (fl_max L)
```

`intros. apply hoare_Lt_then_also_fl_max.` (2634)

S70 The proof state we are left with is proven by `split_fstring_hoare_Lt`.[V] □

```
1 subgoals                                                S70
L : list fletter
H : L ≠ nil
________________________________________________(1/1)
hoare_Lt (split_fstring L) L
```

`apply split_fstring_hoare_Lt. assumption.` (2634)

```
Lemma hoare_Lt_then_also_fl_max :
  forall M L,
    hoare_Lt M L → hoare_Lt M (fl_max L).
```

Excerpt #71

If `hoare_Lt` holds between `M` and `L`, it should also hold between `M` and `fl_max L`. In other words, if for each `fstring` `m` in `M` it holds that `hoare_lt m L`, then it should also hold for each `fstring` `m` in `M` that `hoare_lt m (fl_max L)`. Let `m` be an arbitrary string in `M`, and $\mathtt{m}_i$ an arbitrary letter in `m`. Then by `hoare_Lt M L` we know that there is some letter $\mathtt{l}_j$ in `L` such that `fl_Lt` $\mathtt{m}_i$ $\mathtt{l}_j$. Since `fl_max L` contains all letters maximal in `L`, either $\mathtt{l}_j$ itself is in `fl_max L`, or there is a letter $\mathtt{l}_k$ in `fl_max L` such that `fl_Lt` $\mathtt{l}_j$ $\mathtt{l}_k$. By transitivity then, `fl_Lt` $\mathtt{m}_i$ $\mathtt{l}_k$. □

fl_max_incomparable

According to Definition 4, *the French term signature over L consists of all strings in $\widehat{L}$ composed of letters mutually $>$-incomparable.* For our stratification process, this means that each head should by construction consist of only letters that are mutually $>$-incomparable, which is expressed in the following lemma.

```
Lemma fl_max_incomparable :
  forall L,
    fs_incomparable (fl_max L).
```

Excerpt #72

This lemma however is merely a wrapper for the next lemma, which we will proceed to prove below. Recall from Section 2.1.3 that *"internally, `fl_max` is merely an interface for `fl_max'`, calling it with the same `fstring` twice"*.

```
Lemma fl_max'_incomparable :
  forall M L, sublist M L →
    fs_incomparable (fl_max' M L).
```

Excerpt #73

S74 An `fstring L` is only `fs_incomparable` if for any `e` in `L` it holds that `not_below e L`. This is expressed in `all_not_below_incomparable`, which states: `forall L,`

`(forall e, In e L → not_below e L) →`
`fs_incomparable L.`

```
1 subgoals                                          S74
M : list fletter
L : list fletter
H : sublist M L
______________________________________________(1/1)
fs_incomparable (fl_max' M L)
```

`apply all_not_below_incomparable. intros.` (1467)

S75 For a letter `e` to be in `fl_max' M L` is to be `not_below e L`. By `fl_max_not_below`, which states `forall M e L, In e (fl_max' M L) → not_below e L`, we can thus replace `H0` by `not_below e0 L`.

```
1 subgoals                                          S75
...
H : sublist M L
e0 : fletter
H0 : In e0 (fl_max' M L)
______________________________________________(1/1)
not_below e0 (fl_max' M L)
```

`apply fl_max_not_below in H0.` (1468)

S76 If `not_below e X`, then also `not_below e S` for any sublist `S` of `X`. By `sublist_not_below`, a lemma capturing this, we thus replace the current subgoal by `not_below e0 L` (ie. `H0`) and `sublist (fl_max' M L) L`.

```
1 subgoals                                          S76
...
H : sublist M L
H0 : not_below e0 L
______________________________________________(1/1)
not_below e0 (fl_max' M L)
```

`apply sublist_not_below with L;[idtac|assumption].` (1469)

S77 The maximal elements of a list are a sublist of that list. □

```
1 subgoals                                          S77
...
H : sublist M L
______________________________________________(1/1)
sublist (fl_max' M L) L
```

`apply fl_max'_L_sublist_L; assumption.` (1470)

This concludes Section 2.2.1. In the next section we prove that `flatten` is the operation inverse to `stratify`.

2.2.2 Flatten after stratify

In this section, we prove that for any French string s it holds that $(s^\sharp)^\flat = s$. That is, we prove that the function $\flat \circ \sharp$ (flatten after stratify) is equivalent to the identity function in our framework. This is the first half of our proof of Lemma 12 (recall its statement from our introduction to Section 2.2). The other half of Lemma 12, proof that for any French term t, it holds that $(t^\flat)^\sharp = t$, is the topic of the next section, §2.2.3.

Below in Figure 5, the seemless transition from French string s via recursive application of stratification to French term and then via recursive application of flattening back to the original string s, is outlined schematically. After we have presented the proof, in the epilogue to this section we will expand upon a lemma called `interleave_split_id`.

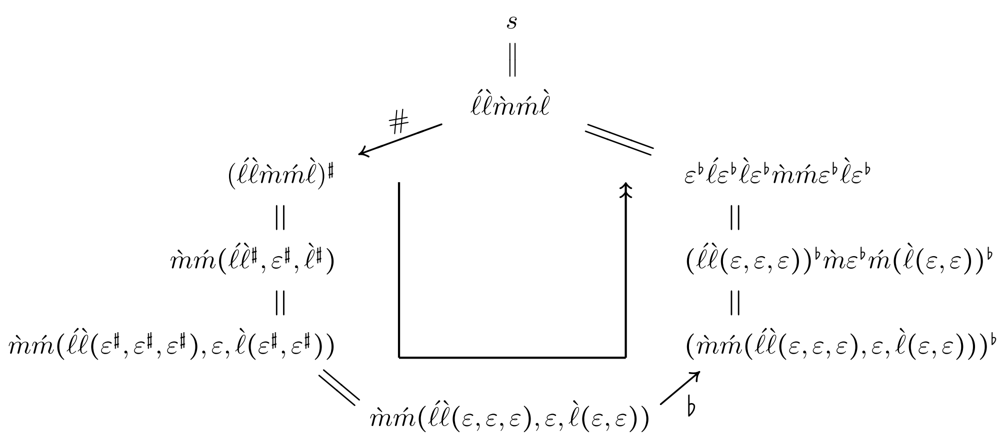


Figure 5: $\flat \circ \sharp$ is equivalent to the identity function

Proof

For any `fstring fs`, applying `flatten` to the result of `stratify fs` should result in `fs` again. That is,

```
Lemma flatten_after_stratify_id :
  forall fs,
    flatten (stratify fs) = fs.
```

Excerpt #78

$\boxed{S_{79}}$ We're doing well-founded induction using `lengthOrder`, again because the recursion of `stratify'` is based on just that (see Excerpt #16). Proof of its well-foundedness is provided by `lengthOrder_wf`. We will begin this proof by unfold the top layers of this statement. Let's start by unfolding `stratify` *once*.

```
1 subgoals                                              S79
______________________________________(1/1)
forall fs : fstring, flatten (stratify fs) = fs
```

`apply well_founded_ind with lengthOrder;`
`[apply lengthOrder_wf|idtac]; intros.` (2853)
`unfold stratify. rewrite unfold_1_stratify'.` (2854)

$\boxed{S_{80}}$ `split_fstring_sigT` is a wrapper function for `split_fstring_sigT'`, let's unfold it.

```
1 subgoals                                              S80
x : fstring
H : forall y, lengthOrder y x → flatten (stratify y) = y
______________________________________(1/1)
flatten (
  Fun (fl_max x)
      (map (fun e0 ⇒
             stratify' (projT1 e0)
                       (Acc_inv (lengthOrder_wf x)(projT2 e0)))
             (split_fstring_sigT x))
) = x
```

`unfold split_fstring_sigT.` (2855)

$\boxed{S_{81}}$ To flatten a term is to interleave its head with its arguments, and flatten each of those arguments recursively (see also Section 2.1.4). Let's unfold `flatten` once.

```
1 subgoals                                                         S81
x : fstring
H : forall y, lengthOrder y x → flatten (stratify y) = y
______________________________________________(1/1)
flatten (
  Fun (fl_max x)
      (map (fun e0 ⇒
            stratify' (projT1 e0)
                        (Acc_inv (lengthOrder_wf x)(projT2 e0)))
            (split_fstring_sigT' x
                     (from_list (split_fstring x))
            )
) = x
```

`rewrite unfold_1_flatten.` (2855)

$\boxed{S_{82}}$ After unfolding `flatten` once, we see that `map` is applied to the result of another `map`. We can simplify this using `map_map` :

```
forall f g l,
  map g (map f l) = map (fun x ⇒ g (f x)) l.
```

```
1 subgoals                                                         S82
x : fstring
H : forall y, lengthOrder y x → flatten (stratify y) = y
______________________________________________(1/1)
interleave (fl_max x)
  (map flatten
      (map (fun e0 ⇒
            stratify' (projT1 e0)
                        (Acc_inv (lengthOrder_wf x)(projT2 e0)))
            (split_fstring_sigT' x
                     (from_list (split_fstring x))
            )
  ) = x
```

`rewrite map_map.` (2856)

$\boxed{S_{83}}$ After rewriting `map flatten (map stratify _)` to `map (fun x ⇒ flatten (stratify x)) _`, we want to simplify this compounded function being mapped. We use functional extensionality, which is captured in `map_ext` :

```
forall f g,
  (forall a, f a = g a) →
  forall l, map f l = map g l.
```

We replace the function `fun x0 ⇒ flatten ( stratify' (projT1 x0) (Acc_inv (lengthOrder_wf x)(projT2 x0)))` by `fun x0 ⇒ flatten (stratify (projT1 x0))`, effectively folding `stratify'` to `stratify`.

```
1 subgoals                                                         S83
x : fstring
H : forall y, lengthOrder y x → flatten (stratify y) = y
______________________________________________(1/1)
interleave (fl_max x)
  (map (fun x0 ⇒
           flatten (stratify'
                        (projT1 x0)
                        (Acc_inv (lengthOrder_wf x)(projT2 x0))
       )
       (split_fstring_sigT' x
                (from_list (split_fstring x))
       )
  ) = x
```

`rewrite map_ext with (g := (fun x0 ⇒ flatten (stratify (projT1 x0))))`. (2857)

`Focus 2.` (2858)

map_ext justification

$\boxed{S_{84}}$ To justify this maneuver, we have to prove the premise of `map_ext`. Something of the form `flatten A = flatten B` can be rewritten by means of <u>`f_equal`</u> to `A = B`. We unfold `stratify` to further homogenize the equation.

```
1 subgoals                                                         S84
x : fstring
H : forall y, lengthOrder y x → flatten (stratify y) = y
______________________________________________(1/1)
forall a0 : {x0 : fstring & lengthOrder x0 x},
flatten (stratify' (projT1 a0)
                     (Acc_inv (lengthOrder_wf x) (projT2 a0)))
= flatten (stratify (projT1 a0))
```

`intro. f_equal. unfold stratify.` (2858)

S85 This brings us to an equation between two instances of `stratify'`, that are identical except for their proof of `lengthOrder a0`. This scenario is covered by `proof_irrelevance`, see Excerpt #61.

```
1 subgoals                                                           S85
x : fstring
H : forall y, lengthOrder y x → flatten (stratify y) = y
a0 : {x0 : fstring & lengthOrder x0 x}
______________________________________(1/1)
stratify' (projT1 a0) (Acc_inv (lengthOrder_wf x) (projT2 a0))
= stratify' (projT1 a0) (lengthOrder_wf (projT1 a0))
```

`apply proof_irrelevance.` (2858)

main proof, continued

S86 `fun x0 ⇒ flatten (stratify (projT1 x0))` can then be replaced by `fun x0 ⇒ projT1 x0`, again using `map_ext`.

```
1 subgoals                                                           S86
x : fstring
H : forall y, lengthOrder y x → flatten (stratify y) = y
______________________________________(1/2)
interleave (fl_max x)
  (map (fun x0 ⇒
            flatten (stratify (projT1 x0)))
       (split_fstring_sigT' x
              (from_list (split_fstring x))
       )
  ) = x
```

`rewrite map_ext with (g := (fun x0 ⇒ projT1 x0)). Focus 2.` (2859)

`map_ext` justification

S87 First we introduce `a0` and dissect it.

```
1 subgoals                                                           S87
x : fstring
H : forall y, lengthOrder y x → flatten (stratify y) = y
______________________________________(1/1)
forall a0 : {x0 : fstring & lengthOrder x0 x},
flatten (stratify (projT1 a0)) = projT1 a0
```

`intro. destruct a0.` (2860)

S88 This then leads to a proof state where `projT1 (existT x0 l)` occurs. That is, the first projection of the pair `(x0,l)`. We simplify and replace this by `x0`.

```
1 subgoals                                                           S88
x : fstring
H : forall y, lengthOrder y x → flatten (stratify y) = y
x0 : fstring
l : lengthOrder x0 x
______________________________________(1/1)
flatten (stratify (projT1 (existT x0 l))) =
                    projT1 (existT x0 l)
```

`simpl.` (2860)

S89 The resulting subgoal is matched by our induction hypothesis, `H`. The premise of its application, `lengthOrder x0 x`, is covered by `l`.

```
1 subgoals                                                           S89
x : fstring
H : forall y, lengthOrder y x → flatten (stratify y) = y
x0 : fstring
l : lengthOrder x0 x
______________________________________(1/1)
flatten (stratify x0) = x0
```

`apply H. simpl. assumption.` (2860)

main proof, continued

S90 We clean up the proof paired formatting using `split_fstring_sigT_inversion`, stating:

```
forall fs L,
  map (fun x ⇒ projT1 x) (split_fstring_sigT fs)
  = split_fstring fs.
```

```
1 subgoals                                              S90
x : fstring
H : forall y, lengthOrder y x → flatten (stratify y) = y
______________________________________________(1/1)
interleave (fl_max x)
  (map (fun x0 ⇒ projT1 x0)
       (split_fstring_sigT' x (from_list (split_fstring x)))
  ) = x
```

```
rewrite split_fstring_sigT_inversion;
[idtac|exact (from_list (split_fstring x))].
```
(2861)

S91 The rest is handled by `interleave_split_id`, which is elaborated upon next. □

```
1 subgoals                                              S91
x : fstring
H : forall y, lengthOrder y x → flatten (stratify y) = y
______________________________________________(1/1)
interleave (fl_max x) (split_fstring x) = x
```

```
apply interleave_split_id.
```
(2862)

Epilogue

In this epilogue to Section 2.2.2, we address a key auxiliary lemma, called `interleave_split_id`.

interleaving `fl_max` and `split_fstring`

```
Lemma interleave_split_id :
  forall L,
    interleave (fl_max L) (split_fstring L) = L.
```

Excerpt #92

To reduce the complexity of this lemma's proof, define another auxiliary lemma `interleave_split'_id`, where we generalize over `fl_max L`. This way we don't have to be concerned with the mechanics of `fl_max`.

```
Lemma interleave_split'_id :
  forall L l0 max, sublist max L →
    interleave max (split_fstring' L l0 max) = l0 ++ L.
```

Excerpt #93

Before looking at its proof, consider why this lemma would hold. `interleave` and `split_fstring` essentially invert each other. `split_fstring` separates a string into substrings, dropping elements of `max`, which function to mark the border between one substring and the next. `interleave` concatenates a list of substrings into a single string, adding an element of `max` in between each sublist.

S94 So let's begin by doing induction on the length of `L`.

```
1 subgoals                                              S94
______________________________________________(1/1)
forall L l0 max, sublist max L →
  interleave max (split_fstring' L l0 max) = l0 ++ L
```

```
intro. induction L; intros.
```
(2301)

induction base

S95 The induction base, where `L`'s length is zero, is trivial. As `max` is a `sublist` of `nil`, its value must be `nil`. We substitute `nil` for `max` and simplify to make the triviality of this proof state more visible.

```
2 subgoal                                               S95
l0 : fstring
max : list fletter
H : sublist max nil
______________________________________________(1/2)
interleave max (split_fstring' nil l0 max) = l0 ++ nil
```

```
inversion H. simpl.
```
(2302)

$\boxed{S_{96}}$ This results in the following equation. Adding an empty list to any list will not alter that list. The equation thus holds.

```
2 subgoal                                              S96
...
H0 : nil = max
_______________________________________(1/2)
(l0 ++ nil) ++ nil = l0 ++ nil
```

`rewrite ← app_assoc. trivial.` (2302)

induction step

$\boxed{S_{97}}$ Next, the induction step. For this, we delve into the definition of `split_fstring'`.

```
1 subgoals                                             S97
a0 : fletter
L : list fletter
IHL : forall l1 max, sublist max L →
        interleave max (split_fstring' L l1 max) = l1 ++ L
l0 : fstring
max : list fletter
H : sublist max (a0::L)
_______________________________________(1/1)
interleave max (split_fstring' (a0::L) l0 max) = l0 ++ a0::L
```

`unfold split_fstring'.` (2303)

$\boxed{S_{98}}$ Based on this definition, we will make some case distinctions. For starters on `max`, and shortly on `fletter_eq_dec a0 mi` (as we will see at State S_{101}).

```
1 subgoals                                             S98
...
_______________________________________(1/1)
interleave max (match max with
                 | nil ⇒ (l0 ++ a0::L) :: nil
                 | mi :: M ⇒ if (fletter_eq_dec a0 mi) then ...
                                                     else ...)
```

`case_eq max; intros; fold split_fstring'.` (2304)

induction step, case max = nil

$\boxed{S_{99}}$ The case distinction on `max` reveals the subgoal to be trivial for `max = nil`, as we can see after simplification.

```
2 subgoal                                              S99
...
H0 : max = nil
_______________________________________(1/2)
interleave nil ((l0 ++ a0::L) :: nil) = l0 ++ a0::L
```

`simpl.` (2305)

$\boxed{S_{100}}$ This is similar to State S_{96}. We eliminate parenthesis and declare the subgoal trivial.

```
2 subgoal                                              S100
...
H0 : max = nil
_______________________________________(1/2)
(l0 ++ a0::L) ++ nil = l0 ++ a0::L
```

`rewrite app_nil_r. trivial.` (2305)

induction step, case max ≠ nil

$\boxed{S_{101}}$ Next, we want to split our subgoal on `fletter_eq_dec a0 f` by making a case distinction on it. That is, we do case analysis on whether or not `a0` equals `f`, splitting our current subgoal into one where they are equal, and one where they are not.

```
1 subgoals                                             S101
...
f : fletter
l : list fletter
H0 : max = f::l
_______________________________________(1/1)
interleave (f::l)
       (if (fletter_eq_dec a0 f) then
           l0 :: split_fstring' L nil l
                                 else
           split_fstring' L (l0 ++ a0 :: nil) (f::l)) =
            l0 ++ a0::L
```

`case (fletter_eq_dec a0 f); intro; subst.` (2306)

induction step, case max ≠ nil, a0 = f

S102 This gives us the current subgoal, where `f` is substituted for any occurrence of `a0`.

```
2 subgoal                                                      S102
...
IHL : forall l1 max, sublist max L →
        interleave max (split_fstring' L l1 max) = l1 ++ L
H : sublist (f::l) (f::L)
________________________________________________(1/2)
interleave (f::l) (l0 :: split_fstring' L nil l)
= l0 ++ f::L
```

`simpl.` (2307)

S103 Simplification completes the first step of the recursive interweaving process, bringing `l0` and `f` to the surface of `interleave`. Now we can see both ends of the equation are lists with the same prefix. Whether or not the equation holds will thus depend on the rest of it. We remove `l0 ++ f` with `app_eq` and `cons_eq`.

```
2 subgoal                                                      S103
...
________________________________________________(1/2)
  l0 ++ f :: interleave l (split_fstring' L nil l)
= l0 ++ f :: L
```

`apply app_eq;[trivial|idtac]. apply cons_eq;[trivial|idtac].` (2308)

S104 The remainder of this subgoal is matched by our induction hypothesis `IHL` (let `l1 := nil`, and `max := l`). The premise of `IHL`, `sublist l L`, is covered by `H`, after we have dropped `f` by means of `sublist_incl2`.

```
2 subgoal                                                      S104
...
IHL : forall l1 max, sublist max L →
        interleave max (split_fstring' L l1 max) = l1 ++ L
H : sublist (f::l) (f::L)
________________________________________________(1/2)
interleave l (split_fstring' L nil l) = L
```

`apply IHL. apply sublist_incl2 in H. assumption.` (2309)

induction step, case max ≠ nil, a0 ≠ f

S105 Again we arrive at a subgoal where we can apply our induction hypothesis (with `l1 := l0 ++ a0 :: nil)`), and `max := f::l`.

```
1 subgoals                                                     S105
...
IHL : forall l1 max, sublist max L →
        interleave max (split_fstring' L l1 max) = l1 ++ L
n : a0 ≠ f
H : sublist (f::l) (a0::L)
________________________________________________(1/1)
interleave (f::l) (split_fstring' L (l0 ++ a0::nil) (f::l)) =
l0 ++ a0::L
```

`rewrite IHL.` (2310)

S106 We then arrive at a trivial subgoal. The premise of `IHL`, here subgoal 2, is covered by `H`. By `sublist_incl3`, `a0` can be dropped from `H` if `f` and `a0` are unequal. □

```
2 subgoal                                                      S106
...
IHL : forall l1 max, sublist max L →
        interleave max (split_fstring' L l1 max) = l1 ++ L
n : a0 ≠ f
H : sublist (f :: l) (a0 :: L)
________________________________________________(1/2)
(l0 ++ a0 :: nil) ++ L = l0 ++ a0 :: L
________________________________________________(2/2)
sublist (f :: l) L
```

`rewrite app_assoc_reverse. apply app_eq;[trivial|idtac].`
`simpl. trivial.` (2311)
`apply neq_sym in n. apply sublist_incl3 in H; assumption.` (2312)

This concludes Section 2.2.2. In the next section we prove that `stratify` is the operation inverse to `flatten`.

2.2.3 Stratify after flatten

In this section we prove that for any French term t it holds that $(t^\flat)^\sharp = t$. That is, we prove that the function $\sharp \circ \flat$ (stratify after flatten) is equivalent to the identity function in our framework, ie. the second half of our proof of Lemma 12 (recall its statement from our introduction to Section 2.2). The other half of Lemma 12, proof that for any French string s it holds that $(s^\sharp)^\flat = s$, was the topic of the previous section, §2.2.2.

Below in Figure 6, the seemless transition from French string t via recursive application of flattening to French string and then via recursive application of stratification back to the original term t, is outlined schematically. Before presenting the proof, we briefly touch upon an adaption of `flatten` called `flatten_cert`, and *term induction* versus the default induction predicate generated by Coq. After we have presented the proof, in the epilogue to this section we will expand upon a lemma called `move_stratify_inward`.

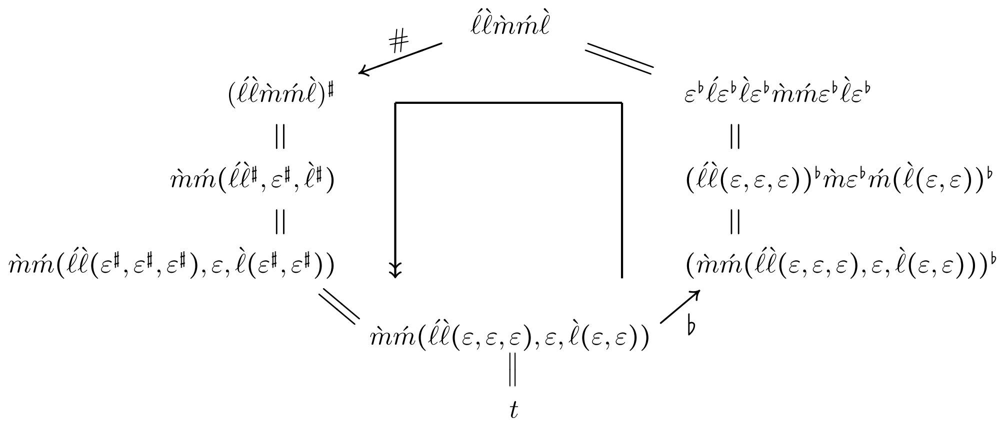


Figure 6: $\sharp \circ \flat$ is equivalent to the identity function

Prologue

So in preparation for the proof of `stratify_after_flatten_id`, let's consider `flatten_cert` and term induction.

flatten_cert

```
Fixpoint flatten_cert (ft : fterm) : fstring :=
  flatten (projT1 ft).
```

Excerpt #107

As we mention in Section 2.1.2 on French terms, `CoLoR`'s `term` data type does not inherently enforce the property of having arity (as in Definition 4, item 2). As is shown in Section 2.2.2, a term created by `stratify` is guaranteed to have arity. And so the proposition `forall t, stratify (flatten t) = t` *does not hold for any* `term` *in general*. More specifically, it doesn't hold for any `term` that doesn't have arity. We resolve this issue by only quantifying over `fterm` (see Excerpt #9), which is a `term` paired with a proof of its well-formedness. `flatten_cert` then simply performs `flatten` on the `term` part of this data type.

term induction

The default induction principle for terms automatically generated by Coq is `term_ind`:

```
Definition term_ind (P : term → Prop) (Q : terms → Prop) :=
  term_rect P Q.
```

Excerpt #108 CoLoR.Term.Varyadic.VTerm

This is generally not convenient however. We would like to be able to perform induction on the *whole* term, rather than have to perform a double induction on its head and arguments.

This principle is provided by `CoLoR` in the form of `term_ind_forall`, which uses the structure of terms. Given any head `f` and arguments `v`, if `P` holding for each individual argument in `v` (if `lforall P v`) means that `P` holds for the term in its entirety (that `P (Fun f v)` holds), then `P` holds for terms in general.

```
Lemma term_ind_forall :
  forall (P : term → Prop)
         (H1 : forall x, P (Var x))
         (H2 : forall f v, lforall P v → P (Fun f v)),
                 forall t, P t.
```

Excerpt #109 CoLoR.Term.Varyadic.VTerm

This is a one-step (single) induction principle, rather than the two-step (double) `term_ind`. Sometimes it can be useful to perform induction with seperate properties for full terms and for a list of terms,[2] but that is beyond the scope of this document.

Proof

We now proceed to prove `stratify_after_flatten_id`. For any `fterm F`, applying `stratify` to the result of `flatten_cert F` should again result in (the first projection of) `F`. That is,

```
3614  Lemma stratify_after_flatten_id :
3615    forall (F : fterm),
3616      stratify (flatten_cert F) = projT1 F.
```

Excerpt #110

As we will see in a moment, `stratify_after_flatten_id` is merely an interface for `stratify_after_flatten_id'`, packing together the `vterm` and its well-formedness into the `fterm` data type.

S111 Our proof of `stratify_after_flatten_id` begins by transformation of `flatten_cert` to `flatten`. This is done by dissecting the `fterm F`.

```
1 subgoals                                                    S111
______________________________________(1/1)
forall F : fterm, stratify (flatten_cert F) = projT1 F
```

`intro. destruct F. unfold flatten_cert.` (3620)

S112 We know that `projT1 (existT x v) = x`, so let's simplify our subgoal.

```
1 subgoals                                                    S112
x : vterm
v : vt_wellformed x
______________________________________(1/1)
stratify (flatten (projT1 (existT x v))) = projT1 (existT x v)
```

`simpl.` (3620)

S113 Now we arrive at the subgoal we were aiming at, with `x`'s well-formedness captured in the hypothesis `v`. We generalize `v` to make the subgoal match our auxiliary lemma, `stratify_after_flatten_id'`.

```
1 subgoals                                                    S113
x : vterm
v : vt_wellformed x
______________________________________(1/1)
stratify (flatten x) = x
```

`generalize v. clear v.` (3620)

S114 We apply `stratify_after_flatten_id'`. □

Let's prove this lemma next.

```
1 subgoals                                                    S114
x : vterm
______________________________________(1/1)
vt_wellformed x → stratify (flatten x) = x
```

`apply stratify_after_flatten_id'.` (3620)

[2] as done by Van Oostrom in `vt_transform_sub_inv`, `conttextterm2context` `vt_transform_var0_unique` (see `bitane_thesis_code.v`)

Our main lemma then is a slightly decomposed version of `stratify_after_flatten_id`.

```
Lemma stratify_after_flatten_id' :
  forall vt (p : vt_wellformed vt),
    stratify (flatten_cert (existT vt p)) = vt.
```

Excerpt #115

S116 We begin by doing *term induction* on `vt`.

```
1 subgoals                                                S116
______________________________________________(1/1)
forall (vt : vterm) (p : vt_wellformed vt),
stratify (flatten_cert (existT vt p)) = vt
```

`intro. apply term_ind_forall with (P := fun vt ⇒ forall p, stratify (flatten_cert (existT vt p)) = vt); intros.` (3597)

S117 After introducing the assumptions we begin our proof by induction.

We can dismiss the case in which `vt = Var x` on the base of `vt_wellformed (Var _)` having been defined as `False` (see Excerpt #8).

```
2 subgoal                                                 S117
vt : vterm
x : nat
p : vt_wellformed (Var x)
______________________________________________(1/2)
stratify (flatten_cert (existT (Var x) p)) = Var x
______________________________________________(2/2)
stratify (flatten_cert (existT (Fun f v) p)) = Fun f v
```

`inversion p.` (3598)

S118 As explained in the prologue (see page 32), the identity between `stratify (flatten vt)` and `vt` only holds if `vt` is well-formed. `flatten_cert` takes a sigma type, drops the second half, and performs `flatten`.

```
1 subgoals                                                S118
vt : vterm
f : fs_Sig
v : list (term fs_Sig)
H : lforall
      (fun vt : vterm ⇒
       forall p : vt_wellformed vt,
       stratify (flatten_cert (existT vt p)) = vt) v
p : vt_wellformed (Fun f v)
______________________________________________(1/1)
stratify (flatten_cert (existT (Fun f v) p)) = Fun f v
```

`simpl; simpl in H.` (3599)

S119 We want to distinguish cases on `p` without losing the original hypothesis, so we make a duplicate first using `pose`.

```
1 subgoals                                                S119
...
H : lforall (fun vt ⇒ vt_wellformed vt →
                      stratify (flatten vt) = vt) v
p : vt_wellformed (Fun f v)
______________________________________________(1/1)
stratify (interleave f (map flatten v)) = Fun f v
```

`pose proof p as p'. destruct p.` (3600)

Case `Fun f v` empty

S120 The first case of `p` (renamed `H0` by Coq) is `vterm_empty (Fun f v)`, in which case the current subgoal should be trivially true.

```
2 subgoal                                                 S120
...
H : lforall (fun vt ⇒ vt_wellformed vt →
                      stratify (flatten vt) = vt) v
H0 : vterm_empty (Fun f v)
p' : vt_wellformed (Fun f v)
______________________________________________(1/2)
stratify (interleave f (map flatten v)) = Fun f v
```

`simpl in H. destruct H0. subst.` (3601)

$\boxed{S_{121}}$ After substituting `nil` for `f` and `v`, we can simplify further. `map flatten nil` is `nil`, leading to `interleave nil nil`, which is also `nil`. The resulting subgoal is `stratify nil = Fun nil nil`, which is true by definition.

```
2 subgoal                                                    S121
vt : vterm
H : lforall (fun vt ⇒ vt_wellformed vt →
                         stratify (flatten vt) = vt) v
p' : vt_wellformed (Fun nil nil)
___________________________________________(1/2)
stratify (interleave nil (map flatten nil)) = Fun nil nil
```

`simpl. trivial.` (3601)

Case `Fun f v` not empty

$\boxed{S_{122}}$ The second case of `p` (renamed `H0`) is the case in which `Fun f v` is not the empty term. We first split `H0` into its constituent parts for better accessibility.

Because `Fun f v` is well-formed, we can peel off the first layer of `stratify`'s recursion. `f` will come out on top again, because `hoare_lt` holds between each node and its parent. This is done by `move_stratify_inward` (see page 37).

`map (fun x ⇒ stratify (flatten x))` will amount to the identity function for `v`, as follows from `H`. To save ourselves the trouble of proving this twice, we add this to our hypotheses, anticipating that it will be coming up in both subgoals generated by `move_stratify_inward` in one form or another.

```
1 subgoals                                                   S122
vt : vterm
f : fs_Sig
v : list (term fs_Sig)
H : lforall (fun vt ⇒ vt_wellformed vt →
                         stratify (flatten vt) = vt) v
H0 : vterm_not_empty (Fun f v) ∧
     ar f = length v ∧
     fs_incomparable f ∧
     Forall (fun x : vterm ⇒ hoare_lt (vt_head x) f) v ∧
     lforall vt_wellformed v
p' : vt_wellformed (Fun f v)
___________________________________________(1/1)
stratify (interleave f (map flatten v)) = Fun f v
```

`destruct H0; destruct H1; destruct H2; destruct H3.` (3602)
`cut (map (fun x ⇒ stratify (flatten x)) v = v);[intro|idtac].` (3603)
`rewrite move_stratify_inward.` (3604)

This then brings us to State S_{123}. Let's prove each of its three subgoals consecutively.

State S_{123} subgoal 1

$\boxed{S_{123}}$ We can see the heads are equal, so the real question is if `map stratify (map flatten v) = v`. This idea is captured by `f_equal`:

`forall f x y, x = y → f x = f y.`

```
                                                             S123
3 subgoal
vt : vterm
f : fs_Sig
v : list (term fs_Sig)
H : lforall (fun vt ⇒ vt_wellformed vt →
                         stratify (flatten vt) = vt) v
H0 : vterm_not_empty (Fun f v)
H1 : ar f = length v
H2 : fs_incomparable f
H3 : Forall (fun x : vterm ⇒ hoare_lt (vt_head x) f) v
H4 : lforall vt_wellformed v
p' : vt_wellformed (Fun f v)
H5 : map (fun x ⇒ stratify (flatten x)) v = v
___________________________________________(1/3)
Fun f (map stratify (map flatten v)) = Fun f v
___________________________________________(2/3)
vt_wellformed (Fun f (map stratify (map flatten v)))
___________________________________________(3/3)
map (fun x : vterm ⇒ stratify (flatten x)) v = v
```

`apply f_equal` (3605)

$\boxed{S_{124}}$ Merging the two uses of `map` results in the assumption we just made (`H5`).

```
3 subgoal                                                    S124
...
H5 : map (fun x ⇒ stratify (flatten x)) v = v
___________________________________________(1/3)
map stratify (map flatten v) = v
```

`rewrite map_map. assumption.` (3605)

State s_{123} subgoal 2

$\boxed{s_{125}}$ First we merge two uses of `map`. We then `rewrite H5` to get our subgoal to match the assumption `p'`.

```
2 subgoal                                              S125
...
p' : vt_wellformed (Fun f v)
H5 : map (fun x ⇒ stratify (flatten x)) v = v
________________________________________(1/2)
vt_wellformed (Fun f (map stratify (map flatten v)))
```

`rewrite map_map. rewrite H5.` (3606)

`assumption.` (3607)

State s_{123} subgoal 3

$\boxed{s_{126}}$ Lastly, we will prove our assumption that `map (fun x ⇒ stratify (flatten x))` does indeed amount to the identity function for `v`. We do this by induction on the length of `v`.

The induction base, the case where `v` equals `nil`, is dismissed, because by definition, for any `f`, `map f nil = nil`.

```
1 subgoals                                             S126
...
H : lforall (fun vt ⇒ vt_wellformed vt →
                        stratify (flatten vt) = vt) v
H4 : lforall vt_wellformed v
________________________________________(1/1)
map (fun x : vterm ⇒ stratify (flatten x)) v = v
```

`induction v. trivial.` (3608)

$\boxed{s_{127}}$ We then proceed with the induction step. By definition of `map`, for any `f`, `map f (x::xs)` is the same as `f x :: (map f xs)`. We simplify to uncover this fact.

```
2 subgoal                                              S127
vt : vterm
f : fs_Sig
a0 : term fs_Sig
v : list (term fs_Sig)
H : lforall (fun vt ⇒ vt_wellformed vt →
                        stratify (flatten vt) = vt) (a0::v)
H4 : lforall vt_wellformed (a0::v)
IHv : lforall (fun vt ⇒ vt_wellformed vt →
        stratify (flatten vt) = vt) v →
       lforall vt_wellformed v →
      map (fun x : vterm ⇒ stratify (flatten x)) v = v
________________________________________(1/2)
map (fun x : vterm ⇒ stratify (flatten x)) (a0::v) = a0::v
```

`simpl.` (3609)

$\boxed{s_{128}}$ An equation of the form `x::xs = y::ys` can be split by `f_equal` into `x = y` and `xs = ys`.

```
2 subgoal                                              S128
...
H : lforall (fun vt ⇒ vt_wellformed vt →
                        stratify (flatten vt) = vt) (a0::v)
H4 : lforall vt_wellformed (a0::v)
________________________________________(1/2)
stratify (flatten a0) :: map (fun x ⇒ stratify (flatten x)) v
= a0 :: v
```

`f_equal.` (3609)

$\boxed{s_{129}}$ The first part of the split is the subgoal `stratify (flatten a0) = a0`. If we fill in `a0` for `vt` in `H`, we get precisely this. `H`'s premise `vt_wellformed a0`, is fulfilled by `H4`.

```
3 subgoal                                              S129
...
H : lforall (fun vt ⇒ vt_wellformed vt →
                        stratify (flatten vt) = vt) (a0::v)
H4 : lforall vt_wellformed (a0::v)
________________________________________(1/3)
stratify (flatten a0) = a0
```

`apply H; apply H4.` (3610)

S130 The second part of the split is the subgoal `map (fun x ⇒ stratify (flatten x)) v = v`, which precisely matches the conclusion of the induction hypothesis `IHv`. The premises of `IHv` are fulfilled by `H` and `H4`, respectively. □

```
2 subgoal                                                    S130
...
H : lforall (fun vt ⇒ vt_wellformed vt →
                         stratify (flatten vt) = vt) (a0::v)
H4 : lforall vt_wellformed (a0::v)
IHv : lforall (fun vt ⇒ vt_wellformed vt →
          stratify (flatten vt) = vt) v →
        lforall vt_wellformed v →
       map (fun x : vterm ⇒ stratify (flatten x)) v = v
______________________________________________(1/2)
map (fun x : vterm ⇒ stratify (flatten x)) v = v
```

`apply IHv. apply H. apply H4.` (3611)

Epilogue

In this epilogue to Section 2.2.3, we address `move_stratify_inward`.

move_stratify_inward

Let `Fun f v` be a well-formed French term. Then `flatten (Fun f v)` equals `interleave f v` (by definition of `flatten`). If we want to apply `stratify` after `flatten`, that is, if we're working with `stratify (interleave f v)`, we can make use of the present seperation between maximal and non-maximal elements, so we don't have to recompute them. We thus take `f` to be the head and `v` to be the list of arguments over which `stratify` is mapped recursively.

```
Lemma move_stratify_inward :
  forall (f : fstring) (v : list fstring),
    vt_wellformed (@Fun fs_Sig f (map stratify v)) →
      stratify (interleave f v) = @Fun fs_Sig f (map stratify v).
```

Excerpt #131

S132 So what we want to show here essentially is that `stratify` and `interleave` cancel each other out. As we will see, this can be done without induction. Let's do `intros` and get started. We begin by unfolding `stratify`.

```
1 subgoals                                                   S132
______________________________________________(1/1)
forall (f : fstring) (v : list fstring),
vt_wellformed (Fun f (map stratify v)) →
stratify (interleave f v) = Fun f (map stratify v)
```

`intros. unfold stratify. rewrite unfold_1_stratify'.` (3576)

S133 We then employ `max_after_interleave_id` :

```
forall f v,
   vt_wellformed (Fun f (map stratify v))
     → fl_max (interleave f v) = f,
```

to substitute `f` for `fl_max (interleave f v)` on the left hand side of the equation. After having done this, we apply `f_equal` to drop `Fun f` from both sides of the equation.

```
1 subgoals                                                   S133
f : fstring
v : list fstring
H : vt_wellformed (Fun f (map stratify v))
______________________________________________(1/1)
Fun (fl_max (interleave f v))
    (map (fun e0 ⇒ stratify' (projT1 e0) (Acc_inv (
                 lengthOrder_wf (interleave f v)) (projT2 e0)))
         (split_fstring_sigT (interleave f v)))
= Fun f (map (fun fs ⇒ stratify' fs (lengthOrder_wf fs)) v)
```

`rewrite max_after_interleave_id; trivial. f_equal.` (3577)

S134 Next, we duplicate `H` as `H0` using `pose` and apply `split_fs_sigT_invertible` to it:

```
forall f v,
  vt_wellformed (Fun f (map stratify v)) →
    exists x,
      split_fstring_sigT (interleave f v) = x
    ∧ map (fun y ⇒ projT1 y) x = v.
```

```
1 subgoals                                                   S134
...
H : vt_wellformed (Fun f (map stratify v))
______________________________________________(1/1)
map (fun e0 ⇒ stratify' (projT1 e0) (Acc_inv (
            lengthOrder_wf (interleave f v)) (projT2 e0)))
    (split_fstring_sigT (interleave f v))
= map (fun fs ⇒ stratify' fs (lengthOrder_wf fs)) v
```

`pose proof H as H0. apply split_fs_sigT_invertible in H0.` (3578)
`do 2 destruct H5.` (3578)

$\boxed{S_{135}}$ Destructing `H0` of State S_{134} then has created witness `x`, described by (now) `H0` and `H1`. Let's replace `split_fstring_sigT (interleave f v)` by `x`.

```
1 subgoals                                                      S135
...
H : vt_wellformed (Fun f (map stratify v))
x : list {x' : fstring & lengthOrder x' (interleave f v)}
H0 : split_fstring_sigT (interleave f v) = x
H1 : map (fun y ⇒ projT1 y) x = v
______________________________________(1/1)
map (fun e0 ⇒ stratify' (projT1 e0) (Acc_inv (
            lengthOrder_wf (interleave f v)) (projT2 e0)))
    (split_fstring_sigT (interleave f v))
= map (fun fs ⇒ stratify' fs (lengthOrder_wf fs)) v
```

`rewrite H0` (3579)

$\boxed{S_{136}}$ Upon having a closer look, what can we tell about our witness `x`? It's a list of a sigma type. By `H1`, if we list the first projection of each element, we get our term's list of arguments, `v`. Likewise, by `H0`, if we interleave `f` with `v` and do `split_fstring_sigT`, we get `x` again.

We want to rewrite the right hand side using `H1`, but somehow Coq doesn't like that. Therefore we have done it via `transitivity` of equality.

```
1 subgoals                                                      S136
...
H : vt_wellformed (Fun f (map stratify v))
x : list {x' : fstring & lengthOrder x' (interleave f v)}
H0 : split_fstring_sigT (interleave f v) = x
H1 : map (fun y ⇒ projT1 y) x = v
______________________________________(1/1)
map (fun e0 ⇒ stratify' (projT1 e0) (Acc_inv (
            lengthOrder_wf (interleave f v)) (projT2 e0))) x
= map (fun fs ⇒ stratify' fs (lengthOrder_wf fs)) v
```

```
transitivity (map (fun f0 ⇒ stratify' f0 (lengthOrder_wf f0))
                   (map (fun y ⇒ projT1 y) x)). Focus 2.
```
(3581)

justification for `transitivity`

$\boxed{S_{137}}$ After substituting `map (fun y ⇒ projT1 y) x` for `v` indirectly, we strip away the context using `f_equal` and point to `H1` for justification of this substitution.

```
1 subgoals                                                      S137
...
H1 : map (fun y ⇒ projT1 y) x = v
______________________________________(1/1)
  map (fun fs ⇒ stratify' fs (lengthOrder_wf fs))
                                   (map (fun y ⇒ projT1 y) x)
= map (fun fs ⇒ stratify' fs (lengthOrder_wf fs)) v
```

`f_equal. assumption.` (3581)

main proof, continued

$\boxed{S_{138}}$ We now have `map` applied to `map`, which can be simplified using `map_map` (in general it holds that $f(g(x)) = f \circ g\ (x)$). Next, we transform the subgoal using `map_ext`: if these two functions mapped over `x` are the same in general, any maps over the same list should be the same as well.

```
1 subgoals                                                      S138
...
______________________________________(1/1)
map (fun e0 ⇒ stratify' (projT1 e0) (Acc_inv (
          lengthOrder_wf (interleave f v)) (projT2 e0))) x
= map (fun f0 ⇒ stratify' f0 (lengthOrder_wf f0))
      (map (fun x0 ⇒ projT1 x0) x)
```

`rewrite map_map. apply map_ext.` (3582)

$\boxed{S_{139}}$ This is true by virtue of `proof_irrelevance` (recall Section 2.2.1): the only difference is the second argument of `stratify'`. □

```
1 subgoals                                                      S139
...
______________________________________(1/1)
forall a0,
stratify'
     (projT1 a0)
     (Acc_inv (lengthOrder_wf (interleave f v)) (projT2 a0))
= stratify' (projT1 a0) (lengthOrder_wf (projT1 a0))
```

`intros. apply proof_irrelevance.` (3582)

This concludes the epilogue to Section 2.2.3.

3 Decreasing Proof Order

In this chapter, the main theorem is developed, expanding upon the framework as described in Chapter 2.

3.1 Presenting the framework

In this section, the framework we set out in Chapter 2 is expanded upon to accomodate for the formalization of *Lemma 19*, the main theorem of this thesis.[3] Proving the core of Lemma 19 will be the topic of Section 3.2.

Below in Figure 7, an impression is given of the components involved. The main theorem revolves around an order on French strings called the *decreasing proof order*, denoted $\succ_{ilpo}$ (see §3.1.5). This order is an instance of the *lexicographic path order* (§3.1.4) induced by $\succ$ (§3.1.3). The order $\succ$, itself a *lexicographical order* (§3.1.1), compares two node labels, based firstly on their multiset , and secondly on their *area* (§3.1.2).

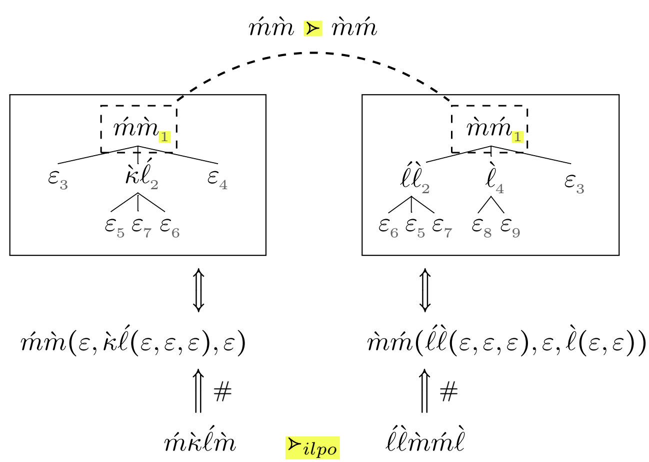


Figure 7: contents of Section 3.1

3.1.1 Lexicographic Order

The order on French strings featured in our main theorem compares labels in *two* dimensions: by their letters and by their area (see §3.1.2). To this end, the lexicographic order is used, which is defined formally as follows.

Definition 8. Let $>$ and $\sqsupset$ be orderings on sets A and B, respectively. For any two elements (a_1, b_1) and (a_2, b_2) of $A \times B$ then, the *lexicographic order* $(a_1, b_1) > \times_{lex} \sqsupset (a_2, b_2)$ holds, if either $a_1 > a_2$ or both $a_1 = a_2$ and $b_1 \sqsupset b_2$.

Example. Consider the set of paired numbers $\mathbb{N} \times \mathbb{N}$ and the greater-than relation $>$. Then $(9, 2) > \times_{lex} > (1, 4)$, as $9 > 1$, and $(4, 2) > \times_{lex} > (4, 1)$, as $4 = 4$ and $2 > 1$; but not $(4, 9) > \times_{lex} > (5, 0)$, as neither $4 > 5$ nor $4 = 5$.

Implementation

For our Coq implementation of this we have used the `CoLoR` libraries, which provide precisely this type of order.

```
Inductive lp_LexProd_Gt (A B : Type)
                        (eqL gtL : relation A)
                        (gtR     : relation B) : relation (A * B) :=
| GtL: forall a a' b b', gtL a a' →            (a, b) >lex (a', b')
| GtR: forall a a' b b', eqL a a' → gtR b b' → (a, b) >lex (a', b')
      where "a >lex b" := (lp_LexProd_Gt a b).
```

Excerpt #140 CoLoR.Util.Pair.LexOrder

Given two elements `a`, `a'` of type `A`, two elements `b`, `b'` of type `B`, two relations `eqL` and `gtL` on `A`, and one relation `gtR` on `B`, it holds that `lp_LexProd_Gt (a, b) (a', b')` if either `gtL a a'`, or both `eqL a a'` and `gtR b b'`.

[3] as with Lemma 12, the name was adopted directly from Van Oostrom[a] for sake of clarity

3.1.2 Area

Every French string has an *area* associated with it. For a formal description of the concept, the reader is referred to Van Oostrom[a], pp. 5–6. For our purposes here, such level of detail is not needed and would be cumbersome. In this section we give a short, intuitive description.

The accents on French letters indicate a measure of computational convergence. The larger the degree of convergence, the smaller the area. As such, a well-founded order can be established on this measure. In Figure 8 below, we see how the accents on a French string are first mapped to a sequence of diagonals and then to a triple. The middle value of this triple represents the area of that string. Its left (right) value is the number of gràve (acúte) accents.

Figure 8: Mapping French strings via strings of accents into triples[4]

Example. To see how this measure of convergence could help to establish an order on French strings, consider $\acute{m}\grave{m}$ and $\grave{m}\acute{m}$. They have the exact same multiset $\{m, m\}$, and so the order on letters does not help to establish an order between them. However, the former's area ($\wedge \mapsto 1$) is larger than the latter's ($\vee \mapsto 0$). (as the latter models a convergence of computation and the former a *divergence* of computation).

Implementation

An `fstring`'s area can be computed quite efficiently by associating with each `fletter` a `triple`, and then performing a left-to-right algorithm over the sequence of triples. Below, part of our implementation is shown.

```
Inductive triple : Type :=
  triple_cons : nat → nat → nat → triple.
...
Definition empty_triple := triple_cons 0 0 0.
Definition acute_triple := triple_cons 0 0 1.
Definition grave_triple := triple_cons 1 0 0.

Definition triple_prod (t1 t2 : triple) : triple :=
  match t1,t2 with
  | triple_cons n1 m1 k1,
  | triple_cons n2 m2 k2 ⇒ triple_cons (n1 + n2) (m1+k1 * n2+m2) (k1 + k2)
  end.
...
Fixpoint lab2trip (fs : fstring) : triple :=
  match fs with
  | nil     ⇒ empty_triple
  | x :: xs ⇒ triple_prod (fl2trip x) (lab2trip xs)
  end.
```
Excerpt #141

The conversion of an `fstring fs` to the area of `fs` is driven by `lab2trip`, which takes each `fletter x`, converts it to either an `acute_triple` or a `grave_triple`, depending on its accent, and uses `triple_prod` to compute from these the total area for `fs`. This will be the middle value of the final resulting `triple`.

[4]illustration adopted from Van Oostrom[a] p. 6

Example.
```
lab2trip (ḿ :: m̀ :: nil) = triple_prod acute_triple (triple_prod acute_triple empty_triple)
                          = triple_prod (triple_cons 0 0 1) (triple_prod (tc⁵ 1 0 0) (tc 0 0 0))
                          = triple_prod (triple_cons 0 0 1) (triple_cons (1+0)(0+0*0+0)(0+0))
                          = triple_cons (0+1)(0+1*1+0)(1+0)
                          = triple_cons 1 1 1
                                          ^
```

The middle value of the final resulting `triple` (1) is the area of `(ḿ :: m̀ :: nil)`.

3.1.3 Label less-than

A comparison of node labels between two French terms is made by combining the concepts explained in the previous two sections. The order $\succ$ is a lexicographical order on node labels, based on their multisets of letters firstly, and their areas secondly.

In Section 3.1.5 we describe the order on French strings around which our main theorem Lemma 19 revolves. This order is based on a comparison of node labels in their interpretation as French terms.

Definition 9. Let $\succ$ be a relation on the French term signature $L^{\sharp}_{\succ}$ (recall Definition 4) by interpreting each function symbol s in $L^{\sharp}_{\succ}$ as a tuple $\langle M, m\rangle$, where M is the multiset of letters in s, and m the area of s. These tuples are related by the combination of $\gg$ and $>$ (the multiset-extension of $\succ$, and greater-than, respectively). That is, they are related by $\gg \times_{lex} >$.

Example. $\acute{m}\grave{m} \succ \grave{m}\acute{m}$: again, their multisets are equal, but the former has greater area (1) than the latter (0).

Implementation

As we describe in Section 3.1.1, `CoLoR` provides an implementation of lexicographic orders, called `lp_LexProd_Gt`. This function is operated through the *module* `LexicographicOrder`. Modules are a convenient way to bundle the required proofs and properties for a set of functions, and have their accessibility coordinated automatically.[3] Part of this module's interior is shown below.

```
Module LexicographicOrder (A_ord B_ord : Ord).
  ...
  Notation L   := A_ord.S.A.
  Notation R   := B_ord.S.A.
  Notation eqL := A_ord.S.eqA.
  Notation eqR := B_ord.S.eqA.
  Notation gtL := A_ord.gtA.
  Notation gtR := B_ord.gtA.
  ...
  Definition LexProd_Gt (x y: L * R) := lp_LexProd_Gt eqL gtL gtR x y.
  ...
```
Excerpt #142 CoLoR.Util.Pair.LexOrder

To initialize an instance of `LexicographicOrder`, two parameters are required: `A_ord` and `B_ord`, which themselves are instances of another module, called `Ord` (see Excerpt #143 below).

Example. Let `(LexAmple NatOrd AlphOrd)` be an instance of `LexicographicOrder`, taking a module that models an order on natural numbers `nat`, and one that models an order on the English alphabet (say, `alpha`). Then `LexAmple.LexProd_Gt` is the lexicographic order on `(NatOrd.S.A * AlphOrd.S.A)`, ie. `(nat * alpha)`, using the domains and relations provided by these modules.

[5] we have used the shorthand `tc` here, for `triple_cons` to be able to fit the page

As convenient as the automatic coordination of proofs and properties might be, it can also lead to confusion as to where the actual definition of a function or value of a variable might be found. For this reason we (somewhat superfluously) explain the `Ord` module here as well.

```
Module Type Ord.

  Parameter A : Type.

  Declare Module Export S : Eqset with Definition A := A.

  Parameter gtA : relation A.
  Notation "X >A Y" := (gtA X Y) (at level 70).

  Parameter gtA_eqA_compat : forall x x' y y',
    x =A= x' → y =A= y' → x >A y → x' >A y'.

  Hint Resolve gtA_eqA_compat : sets.

End Ord.
```

Excerpt #143 CoLoR.Util.Relation.RelExtras

This module has three parameters: `A`, `gtA` and `gtA_eqA_compat`. The first (`A`) is its domain, and the second (`gtA`) the order on that domain. Note that this module again reaches to yet another module, of type `Eqset`, which pertains to decidable equality for `A`. The third parameter (`gtA_eqA_compat`) is a proof of compatibility between this equality relation and `gtA`.

Example. We initialize a module of type `Ord` to model the order we need on areas, by giving it the parameters `nat`, `gt` and a proof that `forall (n n' m m' : nat), n = n' → m = m' → gt n m → gt n' m'`.

```
Module LexMSTR :=
  LexOrder.LexicographicOrder MSOrd AROrd.
```

Excerpt #144

Back to `LexicographicOrder`. We initialize `LexMSTR` above to model $\gg \times_{lex} >$, our lexicographic order on $\widehat{L} \times \mathbb{N}$. The first parameter `MSOrd` models our order $\gg$ on the set of French strings $\widehat{L}$. This is done by referral to `MultisetListOrder`, a module also provided by `CoLoR`, which models the Dershowitz-Manna order on multisets,[6] implemented as instances of `list`. The second parameter `AROrd` is initialized by referral to the greater-than relation on natural numbers, as explained in the description of `Ord` above.

```
Definition lab_lt (f g : fstring) :=
  LexMSTR.LexProd_Lt (lab2pair f) (lab2pair g).
```

Excerpt #145

Finally, our relation $\succ$ (or rather $\prec$) on labels then is implemented as `lab_lt`, as shown in Excerpt #145 above. `LexProd_Lt` is simply a wrapper function for `lp_LexProd_Gt`, passing to it the values bundled when `LexMSTR` was initialized. Those implicit parameters marked gray in Excerpt #140 are actually fields in `LexicographicOrder`. `lab2pair fs` is defined as `(fs, lab2area fs)`, generating instances of type `(fstring * nat)`.[15]

This concludes our description of node label comparison by $\succ$ (implemented as `lab_lt`). In Section 3.1.4 we describe a lifting of orders from symbols to terms, applied in Section 3.1.5 to lift $\succ$ from node labels to conversions (that is, from node labels to French strings interpreted as French terms).

[6] for more on multiset-comparison the reader is referred to the appendix (esp. Definition 19 and Definition 20)

3.1.4 Lexicographic Path Order

In Section 3.1.3 we described $\succ$, an order on node labels. In Section 3.1.5 we will see how this order is lifted from labels to terms over labels. But before exploring its application in our framework, let's have a look here at the general mechanics of this lifting.

Definition 10. Let $\succ$ be a strict order on a finite signature Σ. The *lexicographic path order* $\succ_{lpo}$ on terms over Σ induced by $\succ$ is (recursively) defined as follows. For any two such terms s and t, it holds that $s \succ_{lpo} t$, if:

(LPO1) $t \in \mathcal{V}ar(s)$[7] and $s \neq t$, or

(LPO2) $s = f(s_1, \dots, s_m)$, $t = g(t_1, \dots, t_n)$, and

(a) there exists i, $1 \le i \le m$: $s_i \succ_{lpo} t$ or $s_i = t$, or

(b) $f \succ g$ and for all j, $1 \le j \le n$: $s \succ_{lpo} t_j$, or

(c) $f = g$ and for all j, $1 \le j \le n$: $s \succ_{lpo} t_j$, and
there exists i, $1 \le i \le m$: $s_i \succ_{lpo} t_i$ and $s_1 = t_1$, $\dots$, $s_{i-1} = t_{i-1}$.

Example. Let $L^{\mathbb{N}}$ be the set of terms over $\mathbb{N}$, and $>$ the greater-than relation on $\mathbb{N}$.

Then $>_{lpo}$ is a *lifting* of $>$ from $\mathbb{N}$ to $L^{\mathbb{N}}$, and

$12(2,x,4)$	$>_{lpo}$	x	by clause LPO1	$\leftarrow x \in \mathcal{V}ar(12(2,x,4))$,
$1(1,2(9,3),3)$	$>_{lpo}$	$2(9,3)$	by clause LPO2a	$\leftarrow 2(9,3)$ is a direct subterm of $1(1,2(9,3),3)$,
$10(1,2,3)$	$>_{lpo}$	$9(4,5,6)$	by clause LPO2b	$\leftarrow 10 > 9$, $10 >_{lpo} 4$ by LPO2b, $10 >_{lpo} 5$ by LPO2b, etc.
$8(4,5,6,7)$	$>_{lpo}$	$8(4,5,0,7)$	by clause LPO2c	$\leftarrow$ the head symbols are equal, all direct subterms are equal except for the third, and $6 >_{lpo} 0$ by LPO2b.

Implementation

`CoLoR` provides the function `lt_lpo`, which is encapsulated in the module `LPO`, as shown below.

```
Module LPO (PT : VPrecedenceType).

  Module Export P := VPrecedence PT.
  Module Export S := Status PT.

  Inductive lt_lpo : relation term :=
    | lpo1 : forall f g ss ts,
        g <F f →
          (forall t, In t ts → lt_lpo t (Fun f ss)) →
            lt_lpo (Fun g ts) (Fun f ss)
    | lpo2 : forall f g,
        f =F= g →
          forall ss ts,
            (forall t, In t ts → lt_lpo t (Fun f ss)) →
              lex lt_lpo ts ss →
                lt_lpo (Fun g ts) (Fun f ss)
    | lpo3 : forall t f ss,
        ex (fun s ⇒ In s ss ∧ (s = t ∨ lt_lpo t s)) →
          lt_lpo t (Fun f ss).

  Definition mytau (f : Sig) (r : relation term) := lex r.

End LPO.
```

Excerpt #146 CoLoR.RPO.VLPO

`LPO` takes one parameter, an instance of the module `VPrecedenceType`, which contains precisely the required information to initialize a module of type `VPrecedence`. This module in turn models a strict order.

This concludes our description of lifting an order from symbols to terms over symbols. In the next section this is applied to establish a lifting of $\succ$ from node labels to French terms.

[7] for any given term x, $\mathcal{V}ar(x)$ denotes here the set of variables occurring in x

3.1.5 Decreasing Proof Order

This then brings us to the order featured in Lemma 19, as explained in Section 3.2.

Definition 11. The *decreasing proof order* on conversions (ie. French strings interpreted as French terms), $\succ_{ilpo}$, is the iterative lexicographic path order[8] induced by $\succ$. To accommodate the lexicographic aspect of this ordering, French term argument positions are given an arbitrary but fixed total order, based on accentuation. That order is specified as follows: if a node label's $i+1^{th}$ *letter* has a gràve (acúte) accent then the i^{th} *argument* of that node is evaluated before (after) the $i+1^{th}$ argument in a *leftmost* way of ordering.

Example. Consider àé$(\varepsilon_0, \varepsilon_1, \varepsilon_2)$. The $0+1^{st}$ label, à, has a grave accent, so the 0^{th} argument, ε_0, comes before the $0+1^{st}$ argument, ε_1. The $1+1^{nd}$ label, é, has an acute accent, so the 1^{st} argument, ε_1, comes after the $1+1^{nd}$ argument, ε_2. At this point all of the label's letters are considered, leaving the last argument's position unaltered. Two sequences are then compatible with this description, $\varepsilon_2, \varepsilon_0, \varepsilon_1$ and $\varepsilon_0, \varepsilon_2, \varepsilon_1$: in both cases, ε_0 comes before ε_1 and ε_1 comes after ε_2. This ambiguity does not agree with the requirement of "giving argument positions an arbitrary but *fixed* total order", so the *leftmostly* ordered of these is chosen, in our case $\varepsilon_0, \varepsilon_2, \varepsilon_1$.

Implementation

This order $\succ_{ilpo}$ is implemented by initializing `LPO` with `lab_lt`.

```
Module MyLPO := LPO fs_VPrecedence.
```

Excerpt #147

The command in Excerpt #147 has made `lt_lpo` (or, more precisely, `MyLPO.lt_lpo`) an operational relation on `vterm`: `lab_lt` is wrapped into the module `fs_VPrecedence`, the format required by `LPO`. The accentuation based argument position sequence is not yet in effect however. This sequence is brought about by `vt_transform`.

```
Fixpoint vt_transform (t : vterm) : vterm :=
  match t as t return vterm with
  | Var x   ⇒ Var x
  | Fun f v ⇒ Fun f (
                  projT1 (
                     args_rearrange' f ( (fix vts_transform (v : list vterm) : list vterm :=
                                           match v as v return (list vterm) with
                                           | nil       ⇒ nil
                                           | cons t' v' ⇒ cons (vt_transform t') (vts_transform v')
                                           end)
                                        v )
                        )
                     )
  end. (*V*)
```

Excerpt #148

The function `args_rearrange'` returns the arguments rearranged according to the order described in Definition 11, paired with proof that this is indeed a permutation of the original list. The recursive rearrangement of arguments is performed by the locally defined function `vts_transform`, as indicated by the dash-lined square.

```
Definition lt_lpo' (vt1 vt2 : vterm) :=
  lt_lpo (vt_transform vt1) (vt_transform vt2).
```

Excerpt #149

We thus define $\succ_{ilpo}$ (or rather, $\prec_{ilpo}$) as `lt_lpo'`. Note that, to modify the order of evaluation, we transform the terms and evaluate them using standard `lt_lpo`, leaving the actual (low-level) order of evaluation unaltered.

[8] the *iterative* lexicographic path order is an order equivalent to the lexicographic path order described in Section 3.1.4

Example. Consider Figure 9 below. This is taken from the graph displayed in Figure 1. Each of the circled numbers in Figure 1 represents a conversion step. This rewriting of conversions as modelled by French strings adheres to the decreasing proof order.

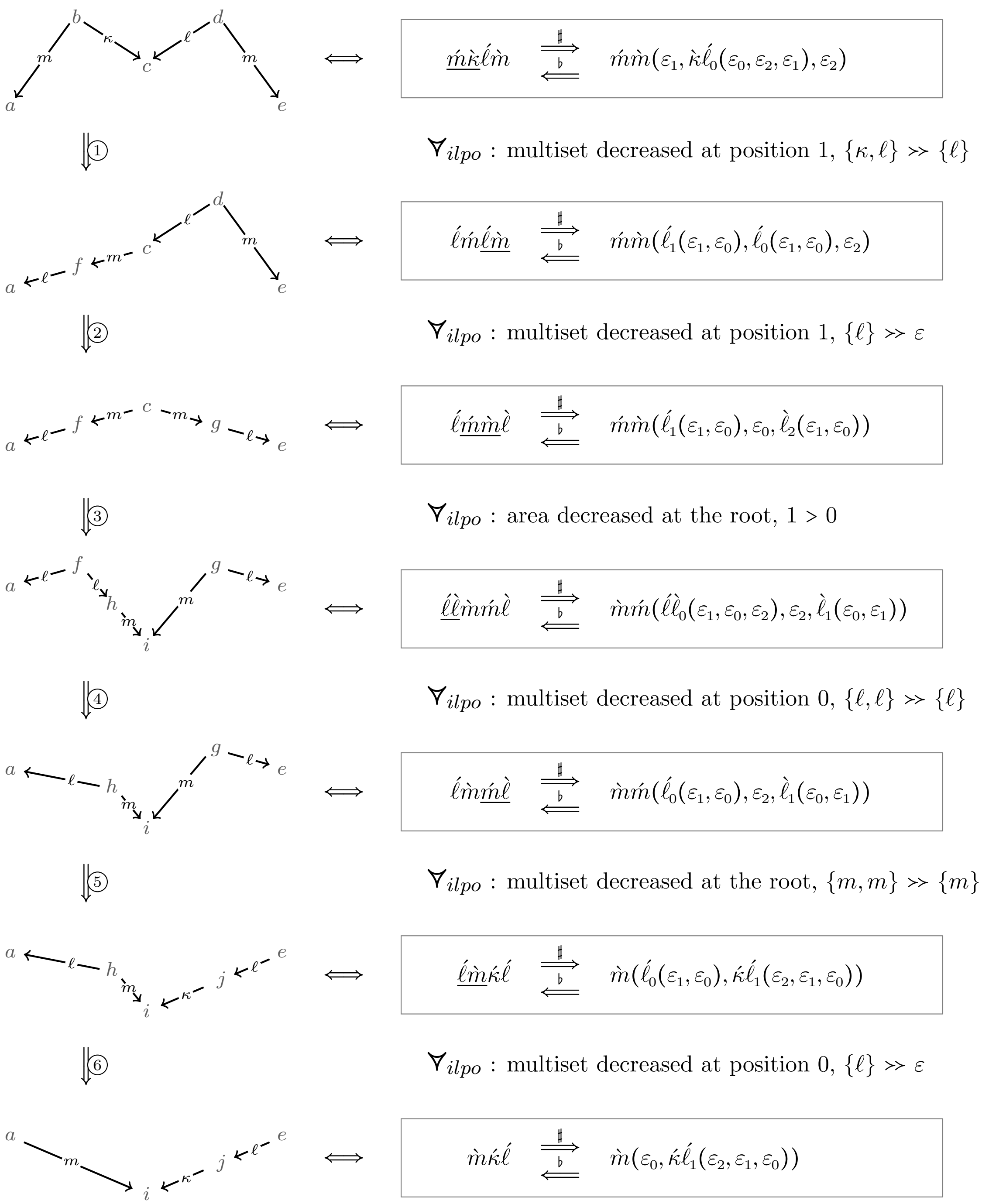


Figure 9: conversion example

3.2 Properties

This section covers the proof of Lemma 19.

Notation. Let [ ] and { } denote *optionality* and *arbitrary repetition*. Let $\vec{\ell}\succ$ ($\prec\vec{\ell}$) denote *any French letter to which at least one letter in the vector* $\vec{\ell}$ *is* $\succ$*-related* ($\prec$*-related*).

Example. $\{\ell\succ\}$ is an arbitrary string of letters to which ℓ is $\succ$-related, and $[\grave{m}]$ denotes either $\grave{m}$ or ε.

Lemma 19(a). *For all labels ℓ, m in L and all French strings s, r over L:* $s\hat{\ell}r \succ_{ilpo} s\{\ell\succ\}r$.

In Coq, this is formulated as follows.

```
Lemma Lemma19a :
  forall x s l r, x = (s ++ (l :: nil) ++ r) →
    forall L, hoare_lt L (l :: nil) →
      lt_lpo' (stratify (s ++ L         ++ r))
              (stratify (s ++ (l  :: nil) ++ r)).
```

Excerpt #150

3.2.1 Proof of properties: prologue

```
Definition tree_rel1 (phi : relation fstring) : relation vterm :=
  fun v1 v2 ⇒ exists (C : context fs_Sig) (v1' v2' : vterm),
    v1 = fill C v1' ∧ v2 = fill C v2' ∧ phi (vt_head v1') (vt_head v2'). (*V*)
```

Excerpt #151

`tree_rel1` states that two terms differ by exactly one subtree, and that the head of the one subtree is greater than the head of the other subtree.

3.2.2 Proof of properties

We now proceed to prove `Lemma19a` (as shown in Excerpt #150).

S152 We introduce our hypotheses. Rather than formulating an induction hypothesis, we simplify our subgoal first by application of `tree_rel1_lt_lab_lpo'`.

```
1 subgoals                                                        S152
______________________________________(1/1)
forall (x s : list fletter) (l : fletter) (r : list fletter),
x = s ++ (l :: nil) ++ r →
forall L : list fletter, hoare_lt L (l :: nil) →
lt_lpo' (stratify (s ++ L ++ r))
        (stratify (s ++ (l :: nil) ++ r))
```

`intros.` (10168)

S153 `tree_rel1_lt_lab_lpo'` states:

```
forall u t,
  vt_wellformed u → vt_wellformed t →
    tree_rel1 lab_lt u t → lt_lpo' u t.
```

Rather than proving `lt_lpo' u t` to follow from our assumptions, we can prove `tree_rel1 u t` to follow, which is a much narrower statement.

```
1 subgoals                                                        S153
x : list fletter
s : list fletter
l : fletter
r : list fletter
H : x = s ++ (l :: nil) ++ r
L : list fletter
H0 : hoare_lt L (l :: nil)
______________________________________(1/1)
lt_lpo' (stratify (s ++ L ++ r))
        (stratify (s ++ (l :: nil) ++ r))
```

`apply tree_rel1_lt_lab_lpo'; try apply vt_wellformed_stratify.` (10169)

S154 Our original lemma is thus divided in two steps: firstly from `tree_rel1 lab_lt` to `lt_lpo'`, and secondly from our original assumptions to `tree_rel1 lab_lt`. The former step is provided by Van Oostrom. Let's have a look then at the latter step.

```
1 subgoals                                                        S154
...
______________________________________(1/1)
tree_rel1 lab_lt (stratify (s ++ L ++ r))
                 (stratify (s ++ (l :: nil) ++ r))
```

`apply Lemma19a' with x; assumption.` (10170)

```
Lemma Lemma19a' :
  forall x s l r, x = (s ++ (l :: nil) ++ r) →
    forall L, hoare_lt L (l :: nil) →
      tree_rel1 lab_lt (stratify (s ++ L          ++ r))
                       (stratify (s ++ (l  :: nil) ++ r)).
```

Excerpt #155

S156 Having reduced the complexity of our conclusion, let's proceed to prove what remains. We do induction on the length of `s++(l::nil)++r` (here captured by `x`).

```
1 subgoals                                                        S156
______________________________________(1/1)
forall (x s : list fletter) (l : fletter) (r : list fletter),
x = s ++ (l :: nil) ++ r →
forall L : list fletter, hoare_lt L (l :: nil) →
tree_rel1 lab_lt (stratify (s ++ L ++ r))
                 (stratify (s ++ (l :: nil) ++ r))
```

`intro. apply well_founded_ind with (P := ...) (R := lengthOrder);`
`[apply lengthOrder_wf|idtac].` (10115)
`intro; intro IH; intros.` (10116)

$\boxed{S_{157}}$ This brings us to the following proof state.

We add to our assumption that

`forall e0 : fletter, In e0 L → fl_Lt e0 l`,

which follows from `H0` (see Excerpt #3).

```
1 subgoals                                                S157
x : list fletter
x0 : list fletter
IH : forall y, lengthOrder y x0 →
       forall s l r, y = s ++ (l :: nil) ++ r →
         forall L, hoare_lt L (l :: nil) →
           tree_rel1 lab_lt (stratify (s ++ L ++ r))
                            (stratify (s ++ (l :: nil) ++ r))
s : list fletter
l : fletter
r : list fletter
H : x0 = s ++ (l :: nil) ++ r
L : list fletter
H0 : hoare_lt L (l :: nil)
______________________________________(1/1)
tree_rel1 lab_lt (stratify (s ++ L ++ r))
                 (stratify (s ++ (l :: nil) ++ r))
```

```
cut (forall e0 : fletter, In e0 L → fl_Lt e0 l);[intro|idtac]. (10117)
  Focus 2. inversion H0. intros. apply H2 in H3. destruct H3.
        apply In_sgt in H3. rewrite ← H3. assumption. (10119)
```

$\boxed{S_{158}}$ Then we do case analysis on whether or not `l` is maximal in `s++(l::nil)++r`.

Note: for preservation of space, throughout this commentary we replace `s++(l::nil)++r` with its equivalent, `s++l::r`.

```
1 subgoals                                                S158
...
H0 : hoare_lt L (l :: nil)
H1 : forall e0 : fletter, In e0 L → fl_Lt e0 l
______________________________________(1/1)
tree_rel1 lab_lt (stratify (s ++ L ++ r))
                 (stratify (s ++ (l :: nil) ++ r))
```

```
cut ({not_below l (s ++ (l :: nil) ++ r)}
 + { ~ not_below l (s ++ (l :: nil) ++ r)});
        [intro|apply not_below_dec]. (10121)
                          destruct H2. (10121)
```

Case 1/2: `l` is maximal in `s++(l::nil)++r`

$\boxed{S_{159}}$ If `l` is maximal in `s++l::r`, the head of `stratify s++l::r` will differ from the head of `stratify s++L++r` precisely by `l`, since the letters maximal in `s` and `r` are in both heads, and from `H0` it follows that none of `L`'s letters are maximal.

We begin to show this by unfolding `tree_rel1`.

```
2 subgoal                                                 S159
...
H1 : forall e0 : fletter, In e0 L → fl_Lt e0 l
n : not_below l (s ++ (l :: nil) ++ r)
______________________________________(1/2)
tree_rel1 lab_lt (stratify (s ++ L ++ r))
                 (stratify (s ++ (l :: nil) ++ r))
```

```
unfold tree_rel1. (10123)
```

$\boxed{S_{160}}$ As the heads of our terms relate in `lab_lt`, we can simply compare the terms directly. To this end, we choose the empty context `Hole`, and fill in our terms. `fill Hole x` equals `x`, so this results in

```
stratify s++L++r = stratify s++L++r ∧
stratify s++l::r = stratify s++l::r ∧
lab_lt (vt_head (stratify s++l::r))
       (vt_head (stratify s++L++r)).
```

The first two conjuncts are trivial. Let's proceed by addressing the third conjunct.

```
2 subgoal                                                 S160
...
H1 : forall e0 : fletter, In e0 L → fl_Lt e0 l
n : not_below l (s ++ (l :: nil) ++ r)
______________________________________(1/2)
exists exists (C : context fs_Sig) (v1' v2' : vterm),
  stratify (s ++ L ++ r) = fill C v1' ∧
  stratify (s ++ (l :: nil) ++ r) = fill C v2' ∧
  lab_lt (vt_head v1') (vt_head v2')
```

```
                                   exists Hole. simpl. (10123)
exists (stratify (s ++ L ++ r)); exists (stratify (s ++ l :: r)). (10124)
                          split; trivial. split; trivial. (10125)
```

S161 The remaining subgoal is fully covered by `l_max_then_head_greater`, which states:

```
forall s r l L,
  (forall e, In e L → fl_Lt e l) →
  not_below l (s++l::r) →
    lab_lt (vt_head (stratify (s++L++r)))
           (vt_head (stratify (s++l::r))).
```

```
2 subgoal                                              S161
...
H1 : forall e0 : fletter, In e0 L → fl_Lt e0 l
n : not_below l (s ++ (l :: nil) ++ r)
______________________________________(1/2)
lab_lt (vt_head (stratify (s ++ L ++ r)))
       (vt_head (stratify (s ++ l :: r)))
```

`apply l_max_then_head_greater; assumption.` (10126)

Case 2/2: `l` is not maximal in `s++(l::nil)++r`

S162 If `l` is *not* maximal in `s++l::r`, then the heads of `stratify s++L++r` and `stratify s++l::r` are equal, since both `L` and `l` are not in them.

We begin our proof by duplicating our assumption `n`. We apply to this duplicate `l_not_max_then_heads_equal` to obtain that the heads are equal.

```
1 subgoals                                             S162
...
n : ~ not_below l (s ++ (l :: nil) ++ r)
______________________________________(1/1)
tree_rel1 lab_lt (stratify (s ++ L ++ r))
                 (stratify (s ++ (l :: nil) ++ r))
```

```
cut ( ~ not_below l (s ++ (l :: nil) ++ r)); trivial; intro Hx.
    apply l_not_max_then_heads_equal with (L := L) in Hx;
                                     [idtac|assumption].
```
(10129)

S163 We duplicate `Hx` so we can simplify it. Both formats will be needed later on.

```
1 subgoals                                             S163
...
n : ~ not_below l (s ++ (l :: nil) ++ r)
Hx : vt_head (stratify (s ++ L ++ r)) =
    vt_head (stratify (s ++ (l :: nil) ++ r))
______________________________________(1/1)
tree_rel1 lab_lt (stratify (s ++ L ++ r))
                 (stratify (s ++ (l :: nil) ++ r))
```

```
cut (vt_head (stratify (s ++ L ++ r)) =
  vt_head (stratify (s ++ (l :: nil) ++ r)));
                               trivial; intro Hx0.
```
(10131)

`rewrite unfold_1_stratify in Hx0, Hx0. simpl in Hx0.` (10132)

S164 Both heads being equal, we can be certain that `Hole` is not the context we're looking for. We thus gear the current subgoal towards an embedded context using `tree_rel1_heads_eq_then_sub`.

```
1 subgoals                                             S164
...
n : ~ not_below l (s ++ (l :: nil) ++ r)
Hx : vt_head (stratify (s ++ L ++ r)) =
    vt_head (stratify (s ++ (l :: nil) ++ r))
Hx0 : fl_max (s ++ L ++ r) = fl_max (s ++ l :: r)
______________________________________(1/1)
tree_rel1 lab_lt (stratify (s ++ L ++ r))
                 (stratify (s ++ (l :: nil) ++ r))
```

`apply tree_rel1_heads_eq_then_sub.` (10133)

S165 We're thus looking to pinpoint in our two terms `stratify s++L++r` and `stratify s++l::r` the single and only difference, and show that it bridges the relation `tree_rel1 lab_lt` between them. To this end, let's abstract from `stratify`, which currently obscures the underlying term structure.

We redefine our terms as `Fun f t1_args` and `Fun f t2_args`, respectively, and look for a single context `Cont f v1 C v2` and two terms `t1'` and `t2'`, such that filling `t1'` in `C` results in `Fun f t1_args`, and filling `t2'` in `C` results in `Fun f t2_args`.

Note that, both of our abstractions have `f` as the head label, since the heads are equal. Also note that in `Cont f v1 C v2`, all arguments `v1` ... `v2` are equal by necessity, except for `C`.

```
1 subgoals                                                S165
...
n : ~ not_below l (s ++ (l :: nil) ++ r)
Hx : vt_head (stratify (s ++ L ++ r)) =
     vt_head (stratify (s ++ (l :: nil) ++ r))
Hx0 : fl_max (s ++ L ++ r) = fl_max (s ++ l :: r)
______________________________________________(1/1)
exists (f : fs_Sig) (t1_args t2_args : list vterm),
  stratify (s ++ L ++ r) = Fun f t1_args ∧
  stratify (s ++ (l :: nil) ++ r) = Fun f t2_args ∧
  (exists (C : context fs_Sig)
          (v1 v2 : list vterm) (t1' t2' : vterm),
     Fun f t1_args = fill (Cont f v1 C v2) t1' ∧
     Fun f t2_args = fill (Cont f v1 C v2) t2' ∧
     lab_lt (vt_head t1') (vt_head t2'))
```

```
              exists (fl_max (s ++ L ++ r)).
      exists (vt_args (stratify (s ++ L ++ r))).
      exists (vt_args (stratify (s ++ l :: r))).   (10136)
             split. Focus 2. split. Unfocus.
```

S166 We have filled in the variables of our abstractions with fragments of our original terms. Then we split the complex subgoal into its three seperate components.

Firstly, the abstractions should actually match the original terms. As we assigned `fl_max s++L++r` to serve as the head of our abstraction, this matching trivially follows from `stratify`'s definition for `stratify s++L++r`.

```
3 subgoal                                                 S166
...
______________________________________________(1/3)
stratify (s ++ L ++ r) =
Fun (fl_max (s ++ L ++ r)) (vt_args (stratify (s ++ L ++ r)))
```

```
rewrite unfold_1_stratify. simpl. unfold fl_max. trivial.   (10137)
```

S167 The same holds for `stratify s++l::r`, except that we need to rewrite its head using `Hx0`.

```
2 subgoal                                                 S167
...
Hx0 : fl_max (s ++ L ++ r) = fl_max (s ++ l :: r)
______________________________________________(1/2)
stratify (s ++ (l :: nil) ++ r) =
Fun (fl_max (s ++ L ++ r)) (vt_args (stratify (s ++ l :: r)))
```

```
do 2 rewrite unfold_1_stratify. rewrite Hx0. simpl. trivial.   (10138)
```

S168 So let's construct from our assumptions the `context` we're looking for. The one and only difference between both terms is `L` and `l`. From `H1` we can infer that none of `L` will be in the node of `stratify s++L++r` corresponding `stratify s++l::r`'s node containing `l`, as all of `L` is smaller than `l` in `fl_Lt`. The node containing `l` will thus be greater than the corresponding node in `stratify s++L++r`, by one letter: `l`. This intuition is captured by the lemma `l_not_max_then_exists_not_eq`,[γ] as we'll see next.

```
1 subgoals                                                S168
...
H1 : forall e0 : fletter, In e0 L → fl_Lt e0 l
n : ~ not_below l (s ++ (l :: nil) ++ r)
______________________________________________(1/1)
exists
  (C : context fs_Sig) (v1 v2 : list vterm) (t1' t2' : vterm),
  Fun (fl_max (s ++ L ++ r)) (vt_args (stratify (s ++ L ++ r))) =
  fill (Cont (fl_max (s ++ L ++ r)) v1 C v2) t1' ∧
  Fun (fl_max (s ++ L ++ r)) (vt_args (stratify (s ++ l :: r))) =
  fill (Cont (fl_max (s ++ L ++ r)) v1 C v2) t2' ∧
  lab_lt (vt_head t1') (vt_head t2')
```

```
apply l_not_max_then_exists_not_eq with (L:=L) in n; trivial.   (10139)
```

S169 So from n and H1 in State S168, we have derived the current assumption n. Let's break down the complexity of this statement a bit by creating a witness for its existential component.

```
1 subgoals                                                S169
...
n : exists i,
      i < length (vt_args (stratify (s ++ (l :: nil) ++ r))) ∧
      (forall j,
         (j ≠ i → vt_arg j (stratify (s ++ L ++ r)) =
                   vt_arg j (stratify (s ++ (l :: nil) ++ r)))
       ∧ (j = i →
            exists s' r',
              length (s' ++ (l :: nil) ++ r') <
              length (s  ++ (l :: nil) ++ r)
            ∧ vt_arg j (stratify (s  ++ L ++ r)) =
                         stratify (s' ++ L ++ r')
            ∧ vt_arg j (stratify (s  ++ (l :: nil) ++ r)) =
                         stratify (s' ++ (l :: nil) ++ r')))
____________________________________________(1/1)
...
```

destruct n. destruct H2. (10140)

S170 Our witness is then x1. The x1$^{\text{th}}$ argument of stratify s++l::r is the subterm containing l. This is expressed in H3. All other arguments (any position j ≠ x1) are equal. The j$^{\text{th}}$ argument of both terms is stratify s'++L++r' for stratify s++L++r, and stratify s'++l::r' for stratify s++l::r.

Note that, again, both terms are equal except for L and l. Note also that these subterms are constructed such that their length is smaller than the term itself:

length 's++l::'r < length s++l::r.

From this we can establish a correspondence between each and every argument of both terms, but if we want to use this to formulate v1 and v2 for the present subgoal (see State S168), we should establish also that, when taken together, all the individual arguments add up to t1_args and t2_args, respectively.

```
1 subgoals                                                S170
...
x1 : nat
H2 : x1 < length (vt_args (stratify (s ++ (l :: nil) ++ r)))
H3 : forall j,
       (j ≠ x1 → vt_arg j (stratify (s ++ L ++ r)) =
                  vt_arg j (stratify (s ++ (l :: nil) ++ r)))
     ∧ (j = x1 →
          exists s' r',
            length (s' ++ (l :: nil) ++ r') <
            length (s  ++ (l :: nil) ++ r)
          ∧ vt_arg j (stratify (s  ++ L ++ r)) =
                       stratify (s' ++ L ++ r')
          ∧ vt_arg j (stratify (s  ++ (l :: nil) ++ r)) =
                       stratify (s' ++ (l :: nil) ++ r'))
____________________________________________(1/1)
...
```

```
cut (exists s1 r1,
    vt_args (stratify (s ++ (l :: nil) ++ r)) =
      s1 ++ (vt_arg x1 (stratify (s ++ (l :: nil) ++ r))) :: r1
 ∧ vt_args (stratify (s ++ L ++ r)) =
           s1 ++ (vt_arg x1 (stratify (s ++ L ++ r))) :: r1). (10145)
                                              intro. Focus 2.
```

S171 We infer this from vt_arg_construct (see epilogue, Excerpt #187), and add it to our list of hypotheses.

```
1 subgoals                                                S171
Hx : vt_head (stratify (s ++ L ++ r)) =
     vt_head (stratify (s ++ (l :: nil) ++ r))
____________________________________________(1/1)
exists s1 r1 : list vterm,
  vt_args (stratify (s ++ (l :: nil) ++ r)) =
  s1 ++ vt_arg x1 (stratify (s ++ (l :: nil) ++ r)) :: r1 ∧
  vt_args (stratify (s ++ L ++ r)) =
  s1 ++ vt_arg x1 (stratify (s ++ L ++ r)) :: r1
```

apply vt_arg_construct; trivial; try apply vt_wellformed_stratify. (10146) *vt_arg_construct requirements*

S172 A requirement of `vt_arg_construct` is that the smaller term (here `stratify s++L++r` be not empty. Since both terms are the result of `stratify`, from `vt_wellformed_stratify` it follows that both are well-formed. From `fl_max_neq_nil` it follows that the head of `stratify` (ie. `fl_max s++l::r`) is not empty. As the heads are equal, we can rewrite using `Hx`. We apply `head_not_empty`, which states that if the head of a well-formed term is not empty, neither is its list of arguments.

```
2 subgoal                                          S172
...
Hx : vt_head (stratify (s ++ L ++ r)) =
     vt_head (stratify (s ++ (l :: nil) ++ r))
________________________________________(1/2)
vterm_not_empty (stratify (s ++ L ++ r))
________________________________________(2/2)
forall j : nat, j ≠ x1 →
vt_arg j (stratify (s ++ L ++ r)) =
vt_arg j (stratify (s ++ (l :: nil) ++ r))
```

`apply head_not_empty; try apply vt_wellformed_stratify.` (10147)
`rewrite Hx. rewrite unfold_1_stratify. simpl.`
`apply fl_max_neq_nil. apply app_cons_not_nil'.` (10148)

S173 `vt_arg_construct` also requires that all arguments except for position `x1` are identical between both terms. This follows directly from `H3`.

```
1 subgoals                                         S173
...
H3 : forall j,
       (j ≠ x1 → vt_arg j (stratify (s ++ L ++ r)) =
                 vt_arg j (stratify (s ++ (l :: nil) ++ r)))
     ∧ (j = x1 →
           exists s' r',
             length (s' ++ (l :: nil) ++ r') <
             length (s  ++ (l :: nil) ++ r)
           ∧ vt_arg j (stratify (s  ++ L ++ r)) =
                      stratify (s' ++ L ++ r')
           ∧ vt_arg j (stratify (s  ++ (l :: nil) ++ r)) =
                      stratify (s' ++ (l :: nil) ++ r'))
________________________________________(1/1)
forall j : nat, j ≠ x1 →
vt_arg j (stratify (s ++ L ++ r)) =
vt_arg j (stratify (s ++ (l :: nil) ++ r))
```

`apply H3.` (10149)

main proof, continued

S174 Back to the main subgoal. From `H3` we can now construct each individual argument to our abstracted terms, and by `H4` we can bind these together to fit the mold of our abstract description, `Cont f v1 C v2`.

Let's begin by constructing the subterms in position `x1`, containing `L` and `l` respectively, by cutting[4] the trivially true assumption `x1 = x1` and then applying `H3` to it. We destruct the result to obtain our witnesses.

```
1 subgoals                                         S174
...
H3 : forall j,
       (j ≠ x1 → vt_arg j (stratify (s ++ L ++ r)) =
                 vt_arg j (stratify (s ++ (l :: nil) ++ r)))
     ∧ (j = x1 →
           exists s' r',
             length (s' ++ (l :: nil) ++ r') <
             length (s  ++ (l :: nil) ++ r)
           ∧ vt_arg j (stratify (s  ++ L ++ r)) =
                      stratify (s' ++ L ++ r')
           ∧ vt_arg j (stratify (s  ++ (l :: nil) ++ r)) =
                      stratify (s' ++ (l :: nil) ++ r'))
H4 : exists s1 r1 : list vterm,
             vt_args (stratify (s ++ (l :: nil) ++ r)) =
       s1 ++ vt_arg x1 (stratify (s ++ (l :: nil) ++ r)) :: r1 ∧
             vt_args (stratify (s ++ L ++ r)) =
       s1 ++ vt_arg x1 (stratify (s ++ L ++ r)) :: r1
________________________________________(1/1)
exists
  (C : context fs_Sig) (v1 v2 : list vterm) (t1' t2' : vterm),
  Fun (fl_max (s ++ L ++ r)) (vt_args (stratify (s ++ L ++ r))) =
  fill (Cont (fl_max (s ++ L ++ r)) v1 C v2) t1' ∧
  Fun (fl_max (s ++ L ++ r)) (vt_args (stratify (s ++ l :: r))) =
  fill (Cont (fl_max (s ++ L ++ r)) v1 C v2) t2' ∧
  lab_lt (vt_head t1') (vt_head t2')
```

`cut (x1 = x1); trivial; intro.`
`apply H3 in H5. do 3 destruct H5. destruct H6.` (10150)

assembling the context

S175 Although it may be the case that `l` is in the head of subterm in position `x1` of `stratify s++l::r`'s arguments, it might not be maximal in that subterm either. For the latter case we mobilize our induction hypothesis. We assert that `tree_rel1 lab_lt` holds between them.

```
1 subgoals                                                    S175
...
x2 : list fletter
x3 : list fletter
H5 : length (x2 ++ (l :: nil) ++ x3) <
     length (s  ++ (l :: nil) ++ r)
H6 : vt_arg x1 (stratify (s ++ L ++ r)) = stratify (x2 ++ L ++ x3)
H7 : vt_arg x1 (stratify (s ++ (l :: nil) ++ r)) =
     stratify (x2 ++ (l :: nil) ++ x3)
______________________________________________(1/1)
...
```

```
cut (tree_rel1 lab_lt (stratify (x2 ++ L ++ x3))
                        (stratify (x2 ++ (l :: nil) ++ x3))). (10152)
                                                intro. Focus 2.
```

S176 To prove this assertion, we apply `IH`.

```
1 subgoals                                                    S176
...
IH : forall y, lengthOrder y x0 →
       forall s l r, y = s ++ (l :: nil) ++ r →
         forall L, hoare_lt L (l :: nil) →
           tree_rel1 lab_lt (stratify (s ++ L ++ r))
                              (stratify (s ++ (l :: nil) ++ r))
H : x0 = s ++ (l :: nil) ++ r
H0 : hoare_lt L (l :: nil)
H5 : length (x2 ++ (l :: nil) ++ x3) <
     length (s  ++ (l :: nil) ++ r)
______________________________________________(1/1)
tree_rel1 lab_lt (stratify (x2 ++ L ++ x3))
  (stratify (x2 ++ (l :: nil) ++ x3))
```

```
apply IH with (x2 ++ (l :: nil) ++ x3). (10153)
```

S177 The witnesses we created earlier conform precisely to the requirements of `IH`.

```
3 subgoal                                                     S177
...
H : x0 = s ++ (l :: nil) ++ r
H0 : hoare_lt L (l :: nil)
H5 : length (x2 ++ (l :: nil) ++ x3) <
     length (s  ++ (l :: nil) ++ r)
______________________________________________(1/3)
lengthOrder (x2 ++ (l :: nil) ++ x3) x0
______________________________________________(2/3)
x2 ++ (l :: nil) ++ x3 = x2 ++ (l :: nil) ++ x3
______________________________________________(3/3)
hoare_lt L (l :: nil)
```

```
rewrite H. assumption.
             trivial.
         assumption. (10154)
```

S178 Now that we have proven our assertion, let's `unfold tree_rel1` to uncover a witness for the context we're assembling.

```
1 subgoals                                                    S178
...
H8 : tree_rel1 lab_lt (stratify (x2 ++ L ++ x3))
                        (stratify (x2 ++ (l :: nil) ++ x3))
______________________________________________(1/1)
...
```

```
unfold tree_rel1 in H8. (10155)
```

main proof, continued

S179 After unfolding `tree_rel1` in our assertion `H8`, we `destruct` it to introduce the witnesses we set out to create.

```
1 subgoals                                                          S179
...
H4 : exists s1 r1 : list vterm,
            vt_args (stratify (s ++ (l :: nil) ++ r)) =
      s1 ++ vt_arg x1 (stratify (s ++ (l :: nil) ++ r)) :: r1 ∧
            vt_args (stratify (s ++ L ++ r)) =
      s1 ++ vt_arg x1 (stratify (s ++ L ++ r)) :: r1
H8 : exists (C : context fs_Sig) (v1' v2' : vterm),
      stratify (x2 ++ L ++ x3) = fill C v1' ∧
      stratify (x2 ++ (l :: nil) ++ x3) = fill C v2' ∧
      lab_lt (vt_head v1') (vt_head v2')
______________________________________________(1/1)
exists
  (C : context fs_Sig) (v1 v2 : list vterm) (t1' t2' : vterm),
  Fun (fl_max (s ++ L ++ r)) (vt_args (stratify (s ++ L ++ r))) =
  fill (Cont (fl_max (s ++ L ++ r)) v1 C v2) t1' ∧
  Fun (fl_max (s ++ L ++ r)) (vt_args (stratify (s ++ l :: r))) =
  fill (Cont (fl_max (s ++ L ++ r)) v1 C v2) t2' ∧
  lab_lt (vt_head t1') (vt_head t2')
```

```
do 4 destruct H8; destruct H9. do 3 destruct H4.
```
(10155)

S180 We assign the witnesses to their proper position in our subgoal. The context `C` we were looking for is `x4`, as demonstrated by `H8` and `H9`. The arguments in positions other than `x1` are represented by `x7` and `x8`, as shown by `H4` and `H11`. The subterm to fill `C` in `stratify s++L++r` is `x5`, and the subterm for `stratify s++l::r x6`, as shown by `H8` and `H9`.

```
1 subgoals                                                          S180
...
x7 : list vterm
x8 : list vterm
H4 :          vt_args (stratify (s ++ (l :: nil) ++ r)) =
     x7 ++ vt_arg x1 (stratify (s ++ (l :: nil) ++ r)) :: x8
H11 :          vt_args (stratify (s ++ L ++ r)) =
      x7 ++ vt_arg x1 (stratify (s ++ L ++ r)) :: x8
H6 : vt_arg x1 (stratify (s ++ L ++ r)) =
              stratify (x2 ++ L ++ x3)
H7 : vt_arg x1 (stratify (s ++ (l :: nil) ++ r)) =
              stratify (x2 ++ (l :: nil) ++ x3)
x4 : context fs_Sig
x5 : vterm
x6 : vterm
H8 : stratify (x2 ++ L ++ x3) = fill x4 x5
H9 : stratify (x2 ++ (l :: nil) ++ x3) = fill x4 x6
H10 : lab_lt (vt_head x5) (vt_head x6)
______________________________________________(1/1)
exists
  (C : context fs_Sig) (v1 v2 : list vterm) (t1' t2' : vterm),
  Fun (fl_max (s ++ L ++ r)) (vt_args (stratify (s ++ L ++ r))) =
  fill (Cont (fl_max (s ++ L ++ r)) v1 C v2) t1' ∧
  Fun (fl_max (s ++ L ++ r)) (vt_args (stratify (s ++ l :: r))) =
  fill (Cont (fl_max (s ++ L ++ r)) v1 C v2) t2' ∧
  lab_lt (vt_head t1') (vt_head t2')
```

```
exists x4. exists x7. exists x8. exists x5. exists x6. simpl.
split.
```
(10156)

S181 After assigning the witnesses, three statements remain to be proven. The first of these is that the abstract term for `stratify s++L++r` we just constructed actually matches with `stratify s++L++r`. This follows from `H11`, `H8` and `H6`.

```
2 subgoal                                                           S181
...
H11 :          vt_args (stratify (s ++ L ++ r)) =
      x7 ++ vt_arg x1 (stratify (s ++ L ++ r)) :: x8
H6 : vt_arg x1 (stratify (s ++ L ++ r)) =
              stratify (x2 ++ L ++ x3)
H8 : stratify (x2 ++ L ++ x3) = fill x4 x5
______________________________________________(1/2)
Fun (fl_max (s ++ L ++ r)) (vt_args (stratify (s ++ L ++ r))) =
Fun (fl_max (s ++ L ++ r)) (x7 ++ fill x4 x5 :: x8)
```

```
rewrite H11. rewrite ← H8. rewrite ← H6. trivial.
split.
```
(10157)

$\boxed{S_{182}}$ For the `stratify s++l::r`, this follows from `H4`, `H9` and `H7`.

```
2 subgoal                                                    S182
...
H4 :        vt_args (stratify (s ++ (l :: nil) ++ r)) =
    x7 ++ vt_arg x1 (stratify (s ++ (l :: nil) ++ r)) :: x8
H7 : vt_arg x1 (stratify (s ++ (l :: nil) ++ r)) =
             stratify (x2 ++ (l :: nil) ++ x3)
H9 : stratify (x2 ++ (l :: nil) ++ x3) = fill x4 x6
____________________________________(1/2)
Fun (fl_max (s ++ L ++ r)) (vt_args (stratify (s ++ l :: r))) =
Fun (fl_max (s ++ L ++ r)) (x7 ++ fill x4 x6 :: x8)
```

`simpl in H4. rewrite H4. rewrite ← H9. rewrite ← H7. trivial.` (10158)

$\boxed{S_{183}}$ The less-than relation for labels holds between the head of `x5` and `x6`, as stated by `H10`.

□

```
1 subgoals                                                   S183
...
H10 : lab_lt (vt_head x5) (vt_head x6)
____________________________________(1/1)
lab_lt (vt_head x5) (vt_head x6)
```

`assumption.` (10159)

3.2.3 Proof of properties: epilogue

In this section some lemmas are addressed that were initially skipped over.

```
Lemma l_max_then_head_greater :
  forall s r l L,
    (forall e, In e L → fl_Lt e l) →
    not_below l (s ++ (l :: nil) ++ r) →
    lab_lt (vt_head (stratify (s ++  L        ++ r)))
           (vt_head (stratify (s ++ (l  :: nil) ++ r))).
```

Excerpt #184

```
Lemma l_not_max_then_heads_equal :
  forall s r l L,
    (forall e, In e L → fl_Lt e l) →
    ~ not_below l (s ++ (l :: nil) ++ r) →
        (vt_head (stratify (s ++  L        ++ r))) =
        (vt_head (stratify (s ++ (l  :: nil) ++ r))).
```

Excerpt #185

```
Lemma tree_rel1_heads_eq_then_sub :
  forall t1 t2,
      (exists f t1_args t2_args,
        t1 = Fun f t1_args ∧ t2 = Fun f t2_args ∧
          exists (C : context fs_Sig) v1 v2 t1' t2',
            Fun f t1_args = fill (Cont f v1 C v2) t1' ∧
            Fun f t2_args = fill (Cont f v1 C v2) t2' ∧
            lab_lt (vt_head t1') (vt_head t2')
      ) → tree_rel1 lab_lt t1 t2.
```

Excerpt #186

```
Lemma vt_arg_construct :
  forall x1 L l,
    vt_wellformed L → vt_wellformed l →
      vt_head L = vt_head l → vterm_not_empty L → x1 < length (vt_args l) →
        (forall j, j ≠ x1 → vt_arg j L = vt_arg j l)
          → exists s1 r1, vt_args l = s1 ++ vt_arg x1 l :: r1 ∧
                          vt_args L = s1 ++ vt_arg x1 L :: r1.
```

Excerpt #187

4 Conclusion

Confluence is an important property of term rewriting systems. Since any algorithm can be modelled as such, the decreasing diagrams technique is very useful for establishment of this property. Van Oostrom has further refined the technique by means of the decreasing proof order in his 2012 article.[a] We have formalized the main two lemmas of this method, thereby verifying its correctness. Firstly, the framework we have written in support of our formalization is correct, as described in Chapter 2. Secondly, several essential properties of the decreasing proof order have been established, as described in Chapter 3.

5 Appendix

Some basic notions formally defined for ease of reference.

Definition 12. Given a set S, an *order* on S is a (binary) relation R on elements of S. We call R a *total order* on S if, for any two elements x and y from S, it holds that Rxy or Ryx. If there are two elements x and y from S such that neither Rxy nor Ryx, we call (S, R) a *partial order*.

Example. Consider the set of natural numbers $\mathbb{N}$, ie. $\{0, 1, 2, \dots\}$. The relation $\geq$ is a total order on $\mathbb{N}$, as for any two numbers x and y it either holds that $x \geq y$ or $y \geq x$. The relation $=$ is a partial order on $\mathbb{N}$, since not all numbers are equal.

Definition 13. A relation R on S is called *reflexive* if $\forall s \in S : Rss$. It's called *irreflexive* if $\forall s \in S : \neg Rss$.

Example. The equality relation $=$ on natural numbers is reflexive, as any number is equal to itself. Likewise, the greater-than relation $>$ is irreflexive, as no number is greater than itself.

Definition 14. A relation R on S is called *symmetrical* if $\forall x, y \in S : Rxy \rightarrow Ryx$. It's called *asymmetrical* if $\forall x, y \in S : Rxy \rightarrow \neg Ryx$.

Example. The equality relation is symmetrical, should it be the case that $x = y$ then $y = x$ must also be the case. The greater-than relation is asymmetrical: if $x > y$, then $y \ngtr x$.

Definition 15. A relation R on S is called *transitive* if $\forall x, y, z \in S : Rxy \rightarrow Ryz \rightarrow Rxz$.

Example. The greater-than relation is transitive. An example of a non-transitive relation would be inequality. If $x \neq y$ and $y \neq z$, it doesn't follow that $x \neq z$.

Definition 16. A relation R on S is called a *strict order* on S if R is both transitive and irreflexive on S. It's called a *preorder* on S if R is both transitive and reflexive on S.

Example. The greater-than relation is transitive. An example of a non-transitive relation would be inequality. If $x \neq y$ and $y \neq z$, it doesn't follow that $x \neq z$.

Definition 17. A relation $<$ on S is called *well-founded* if every non-empty subset S_i of S has a *minimal element*. That is, in each such subset S_i there is an element m such that $\forall s \in S_i : s \nless m$.

Example. The less-than relation $<$ is well-founded on the set of natural numbers $\mathbb{N}$. Let S_n be a non-empty subset of $\mathbb{N}$. It must be the case that S_n have a smallest element. $<$ is not well-founded on the set of integers $\mathbb{Z}$, ie. $\{\dots, -2, -1, 0, 1, 2, \dots\}$. Consider its subset $\{\dots, -3, -2, -1\}$, the set of negative numbers. No matter the element, there is always a smaller element in that set.

Definition 18. Given two sets X and Y, let $<$ be a relation between elements from X and Y. We say that X *dominates* Y in $<$ if, $\forall y_i \in Y : \exists x_i \in X : y_i < x_i$.

Example. Consider a set S: $\{a, b, c, d\}$, and a partial order $\prec$ on S: $\{(a, b), (b, c), (b, d), (a, d)\}$ We then say that $\{d\}$, $\{c, d\}$, and $\{a, b, c, d\}$ all dominate $\{a, b\}$. Likewise, $\{d\}$ and $\{c, d\}$ are dominated by no sets.

Definition 19. A *multiset* is a set S paired with a map $\mu : S \rightarrow \mathbb{N} \setminus \{0\}$, where for any $e \in S, \mu e$ denotes the *multiplicity* (ie. the number of occurrences) of e in S. We use only finite multisets. Two multisets $\langle M, \mu_M \rangle$ and $\langle N, \mu_N \rangle$ are considered equal if $\forall x \in M \cup N : \mu_M(x) = \mu_N(x)$.

Example. A multiset can be considered a set allowing multiple occurences per element. Let $M = \{a, b, c, d\}$ and $\mu_M = \{(a, 3), (b, 1), (c, 4), (d, 2)\}$. Then $\langle M, \mu_M \rangle$ equals $\{a, a, a, b, c, c, c, c, d, d\}$.

Definition 20. Given a set S, let $\mathcal{M}(S)$ denote the set of all finite multisets on S. Then, given a relation $<$ on S, the *Dershowitz-Manna* ordering $<_{DM}$ for all $M, N \in \mathcal{M}(S)$ is defined as: $M <_{DM} N$ if and only if $\exists X \exists Y \in \mathcal{M}(S)$ such that: $X \neq \varnothing, \quad X \subseteq N, \quad M = (N - X) + Y,$ and X dominates Y in $<$.

Example. A more instructive way to think about this order is to suppose that M is constructed from N by first removing the elements in X and then adding the elements in Y. Because of the requirement that X dominates Y in $<$, each element that is added is smaller than some removed element, rendering a smaller multiset.

References

[a] Van Oostrom, V. *A proof order for decreasing diagrams - Interpreting conversions in involutive monoids* (2012). Unpublished.

[b] Terese. *Term Rewriting Systems* (2003). Cambridge University Press.

[c] Bertot, Y., Castéran, P.: *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions* (2004). Springer.

[d] Baader, F., Nipkow, T.: *Term rewriting and all that* (1998). Cambridge University Press.

[e] Zankl, H. *Confluence by Decreasing Diagrams – Formalized* (2013). 24th International Conference on Rewriting Techniques and Applications, pp. 352–367.

Index

Index (Coq definitions)

Index (Coq theorems)

Bookmarks

[1] Reference Manual. *Definition of functions by recursion over inductive objects*
`https://coq.inria.fr/distrib/8.4pl3/refman/Reference-Manual003.html#sec54`

[2] Reference Manual. *Extraction of programs in Objective Caml and Haskell*
`https://coq.inria.fr/distrib/8.4pl3/refman/Reference-Manual025.html`

[3] Reference Manual. *The Module System*
`https://coq.inria.fr/distrib/8.4pl3/refman/Reference-Manual004.html#sec80`

[4] Reference Manual. *Controlling the proof flow:* `cut`
`https://coq.inria.fr/distrib/8.4pl3/refman/Reference-Manual010.html#hevea_default556`

[5] Reference Manual. *Controlling the proof flow:* `pose`
`https://coq.inria.fr/distrib/8.4pl3/refman/Reference-Manual010.html#hevea_tactic53`

[6] Reference Manual. *Sorts*
`https://coq.inria.fr/distrib/8.4pl3/refman/Reference-Manual003.html#hevea_default15`

[7] Reference Manual. *Abstractions:* `fun`
`https://coq.inria.fr/distrib/8.4pl3/refman/Reference-Manual003.html#sec32`

[8] Reference Manual. *Explicit applications:* `@`
`https://coq.inria.fr/distrib/8.4pl3/refman/Reference-Manual004.html#hevea_default136`

[9] Reference Manual. *Performing computations:* `unfold`
`https://coq.inria.fr/distrib/8.4pl3/refman/Reference-Manual010.html#hevea_tactic137`

[10] Reference Manual. *Recursive functions:* `fix`
`https://coq.inria.fr/distrib/8.4pl3/refman/Reference-Manual003.html#sec39`

[11] Reference Manual. *Inferable subterms:* `_`
`https://coq.inria.fr/distrib/8.4pl3/refman/Reference-Manual003.html#hevea_default25`

[12] Reference Manual. *Equality:* `f_equal`
`https://coq.inria.fr/distrib/8.4pl3/refman/Reference-Manual010.html#sec423`

[13] Standard Library. *Peano natural numbers*
`https://coq.inria.fr/distrib/8.4pl3/stdlib/Coq.Init.Datatypes.html#lab3`

[14] Standard Library. *Container datatypes:* `list`
`https://coq.inria.fr/distrib/8.4pl3/stdlib/Coq.Init.Datatypes.html#list`

[15] Standard Library. *Container datatypes:* `prod`
https://coq.inria.fr/distrib/8.4pl3/stdlib/Coq.Init.Datatypes.html#prod

[16] Standard Library. *Case analysis and induction:* `case_eq`
https://coq.inria.fr/distrib/8.4pl3/refman/Reference-Manual010.html#hevea_tactic70

[17] Standard Library. *Existential and universal predicates over lists*
https://coq.inria.fr/distrib/8.4pl3/stdlib/Coq.Lists.List.html#lab373

[18] Standard Library. *Subsets and Sigma-types*
https://coq.inria.fr/distrib/8.4pl3/stdlib/Coq.Init.Specif.html

[19] Standard Library. *Applying functions to the elements of a list:* `map`
https://coq.inria.fr/distrib/8.4pl3/stdlib/Coq.Lists.List.html#lab363

[20] CoLoR Library. *Contexts and replacement of the hole*
http://color.inria.fr/doc/CoLoR.Term.Varyadic.VContext.html#term

[21] CoLoR Library. *strict order*
http://color.inria.fr/doc/CoLoR.Util.Relation.RelExtras.html#StrictOrder

[22] CoLoR Library. `lforall`
http://color.inria.fr/doc/CoLoR.Util.List.ListForall.html

[23] CoLoR Library. *Module types for setoids with decidable equality:* `Eqset`
http://color.inria.fr/doc/CoLoR.Util.Relation.RelExtras.html#Eqset

[24] CoLoR Library. *An order on lists derived from the order on multisets:* `MultisetListOrder`
http://color.inria.fr/doc/CoLoR.Util.Multiset.MultisetListOrder.html#MultisetListOrder

[25] CoLoR Library. *Recursive path orderings are monotonic well-founded strict orders:* `VPrecedence`
http://color.inria.fr/doc/CoLoR.RPO.VPrecedence.html