Computerlinguistik

WS 2019/2020

Greg Kobele

October 17, 2019

1 Strings

A string is a formal object inspired by a written word. Informally, it is a sequence of objects. The precise nature of the objects is not really relevant; we can have strings of roman letters, of cyrillic letters, of numbers, of cars (waiting at a gas station), of people (waiting in line at the store), etc. This is important - we can't have strings without objects, but the nature of the objects is irrelevant for the properties of strings we are concerned with!

We will, building on the motivating inspiration of a written word, call the set of objects appearing in a string its *alphabet*, and will often use the Greek capital letters Σ (or Δ, or Γ) to name an alphabet. In principle, alphabets may be infinite, but we will be mostly interested in situations where they are finite. In contrast, we will only be interested in finite strings.

There are many ways of formalizing almost *everything*; strings are no different. One of the most fundamental things we would like to do with strings is to *concatenate* them. Intuitively, concatenating strings is like writing them down next to one another. For example, concatenating the string "obst" with the string "garten" gives the string "obstgarten". One important property of string concatenation is that it is insensitive to constituency; concatenating strings "abc" and "def" and "ghi" (in that order) gives rise to the string "abcdefghi", regardless of whether we first concatenate "abc" and "def" (to form "abcdef") and then concatenate that with "ghi", or whether we concatenate "abc" to the result of concatenating "def" and "ghi" (forming "defghi"). Indicating concatenation of strings via a dot ($\cdot$), we have that $"abc" \cdot ("def" \cdot "ghi") = ("abc" \cdot "def") \cdot "ghi"$. (Binary) operations which have this property are called *associative*. An adequate formalization of strings will allow us to define concatenation.

1.1 Strings as arrays

A first definition of strings views them as an ordered set of *positions*, which contain symbols. We begin by identifying positions with numbers, starting at 0 (we revisit this decision in section 1.1.1).[1] If a string has (say) five positions, these will be named 0,1,2,3,4. In other words, the set of positions in a string of length n will be $\{0, 1, \ldots, n-1\}$. It will be very convenient to have a simple notation for the set of all natural numbers less than another.[2] We define:

Definition 1. For any natural number $n \in \mathbb{N}$, let $[n] := \{k : k < n\}$.

A string of length n over an alphabet Σ is an assignment of symbols in Σ to each position in the string. What this means is that each position should be associated with exactly one symbol; this notion of an assigment is made precise as a *function*.

Definition 2. Given an alphabet Σ, and a natural number n, a string of length n over Σ is a function from $[n]$ to Σ.

Viewing a string as an association between positions and symbols (a function from positions to symbols) means that we can speak of the symbol at a particular position in terms of the result of applying the function to that position. We can visualize an example as in figure 1.

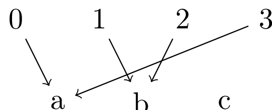

Figure 1: The string "abba" over the alphabet $\Sigma = \{a, b, c\}$

In the figure, the positions are written sequentially at the top, and the alphabet symbols are written sequentially at the bottom. Arrows link positions with symbols. The same alphabet symbol may be in many positions (in the figure, 'b' is in both positions 1 and 2), but each position must have exactly one symbol in it.

As a special case of our definition of a string, consider what happens when we have a string of length *0*. Then a string of length 0 is a function associating each position in [0] with a symbol. But, there *is* no position in [0];

[1] This has the unfortunate consequence of implying that the *first* position in a string is 0, the *second* is 1, etc.

[2] We write $\mathbb{N} = \{0, 1, 2, \ldots\}$ for the set of natural numbers.

$[0] = \emptyset$ is the empty set! This may seem puzzling at first: what should such a function do? It is easiest to see what to do in the context of a real-world example. Imagine that our job would be to give each subscriber the newspaper they ordered (i.e., we should implement a function from subscribers to newspapers). If we don't have any newspapers, then we are in trouble; we may be fired for not doing our job! But if we don't have any subscribers, then our job becomes trivial; we may as well relax on the beach, as we have finished before even starting! Returning to the function, we see that there is exactly one way of associating outputs with no inputs: do nothing. This is called the empty function, and in the world of strings, it is called the empty string, and is written with the Greek lower case ϵ.[3]

This discussion presupposes a notion of identity between strings, which we have not yet made explicit. Two strings are identical just in case the following holds:

1. they have the same set of positions

2. they have the same symbols in each of their positions

This can be written in a more mathematical idiom as follows.

Definition 3. Given two strings u, v of length n over alphabet Σ, we say that $u = v$ iff for every $i \in [n]$ it holds that $u(i) = v(i)$.

Consider how we might define string concatenation over these representations. As a concrete example, we may look at figure 2. Here we have representations of strings "ab" and "ba", which are concatenated to form the representation of the string "abba" which we saw above in figure 1.

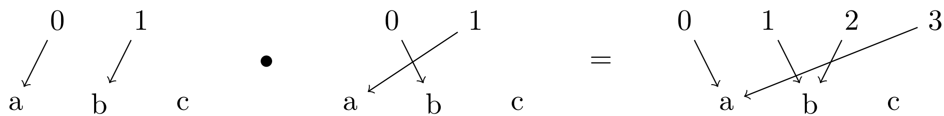

Figure 2: Concatenating "ab" with "ba" (over the alphabet $\Sigma = \{a, b, c\}$)

Studying the figure, we see that we must change the positions of the second string so that they begin, not at 0, but at the successor position to the end position of the first string (in the example, this is position 2). As the alphabets of the to-be-concatenated strings are the same, we can then simply unify the two copies into one, making sure that the arrows continue pointing

[3]In some places it is written with the Greek lower case λ.

to the same letters. Building on this idea, we can define a concatenation operation over these representations in the following way.

Definition 4. Let Σ be an alphabet, and let u and v be strings over Σ of lengths m and n respectively. Then $u \cdot v$ is the string over Σ of length $m+n$, such that

1. for each $i \in [m]$, $(u \cdot v)(i) = u(i)$
2. for each $i \in [m+n] - [m]$, $(u \cdot v)(i) = v(i - m)$

1.1.1 Generalizing

We began by identifying positions with initial sets of natural numbers. In this section, we relax this restriction, and allow positions to be anything. This will allow us to see what properties of natural numbers were actually needed (for example, every natural number can be uniquely decomposed as the product of primes - is **this** important to our notion of strings?!).

A string is determined by a set of positions, as before, and a function specifying what is 'at' each position. However, we also require an order on our positions, to specify which position immediately follows which other! Now we see that a string is determined by three sets:

1. the set of positions **Pos**
2. a functional relation **label** $\subseteq Pos \times Alph$
3. a relation $prec \subseteq Pos \times Pos$

We can now see that our use of natural numbers exploited the inherent ordering of natural numbers to encode the precedence relation.

With our generalization comes the need to revisit the question of string identity; now that positions are arbitrary, we can represent one and the same string in infinitely many ways.[4] We would like to define a notion of identity of string representations which ignores possible differences in the names of positions, and only considers the 'relevant' structural properties. One natural idea is to first try to rename the positions of the two objects we are comparing for equality, so that they have the same position names - then equality would really just be identity.

A renaming is then a bijective function from one position set to another. A function associates exactly one position in the target set with every position in the source set, and it being bijective requires that all positions in the

[4] This does not hold for the empty string, which has exactly one possible representation.

target set are used as names. Two objects are equal just in case the following are true:

1. we can rename the positions of the first as positions of the second, such that
2. the resulting structures are identical.

1.2 Strings as lists

The position-based formalization of strings given above is correct, and everything we are interested in doing can be carried out in these terms. However, just because something is *true* doesn't make it *useful*! As a concrete example, we have a conception of natural numbers, which is independent of how we represent them.[5] Using a decimal representation of numbers allows us to perform certain operations, like multiplication by 10, very easily: just add a 0 to the end of the representation. Other operations, like multiplication by 2, require actual work to compute. In contrast, a binary representation of the same makes multiplication by 2 a matter of adding a 0 to the end of the representation, but has the trade-off of making multiplication by 10 more difficult. Both representations of natural numbers (and infinitely many more!) are correct, but, depending on what we would like to do with them, one may be more useful than the other.

Accordingly, we here present a different formalization of (and representation for) strings, which views them as being constructed from simpler pieces. We will say that a non-empty string contains a first symbol, and the rest of the string. In this way, larger strings can be built up out of smaller strings. Because we need to make reference to other strings when defining strings, we define at once the *set* of all strings over a certain alphabet.

Definition 5. Given an alphabet Σ, we define the strings over Σ as follows.

1. `[]` is a string over Σ
2. if w is a string over Σ, and $a \in \Sigma$, then $(a : w)$ is a string over Σ

implicit in the above definition is the further clause:

3. nothing else is a string over Σ

[5] This might be given by the Peano axioms, for example.

This definition tells us that a string (over Σ) can have exactly one of two general shapes; it can be empty, in which case it is `[]`, or non-empty, in which case it is of the form $a\!:\!w$ for some symbol 'a' and string w.[6] It gives us therefore not only a definition of strings, but also a procedure for determining whether something is a string over Σ.

Example 1. We attempt to verify that "abba" is indeed a string over $\{a, b, c\}$. Note that it is of the form $a\!:\!w$ (where $w =$ "bba"), and so is a string over $\{a, b, c\}$ iff $a \in \{a, b, c\}$ (which it is) and w is a string over $\{a, b, c\}$. So whether "abba" is a string depends on whether "bba" is a string. And "bba" (recall, it is of the form $b\!:\!w$, where $w =$ "ba") is a string iff $b \in \{a, b, c\}$ and "ba" is a string. Similarly, "ba" is a string iff "a" is, and "a" is a string iff `[]` is. But `[]` **is** a string (it says so in the definition). And so then "a" is, and then "ba" is, and then "bba" is, and then "abba" is, as we wanted to determine.

Thus, if we want to define a function over all strings, we need only to say what it does in these two circumstances. We call this a recursive definition. As an example, we define a function associating with each string its length, which should tell us how many symbols occur in it. Clearly, the length of the empty string, `[]`, is 0. An if we want to know the length of a string of the form $a\!:\!w$, it is one greater than the length of w.

Definition 6. The length of a string w, written $|w|$, is given inductively as follows.

- $|\texttt{[]}| = 0$
- $|a\!:\!w| = 1 + |w|$

What is unique about recursive definitions is that they define something in terms of itself.[7] That is, the definition appears *circular*. The naïve worry about circular definitions (such as of a word in a dictionary) is that you will be shunted from one place to another and back, and never make any progress. Not all definitions which refer to themselves are circular in this sense, however! One important aspect of the definition of lengh given above is that the size of the argument given to the function we are defining is steadlily decreasing (w is shorter than $a\!:\!w$); as all strings are finite, this means that our length function is guaranteed to eventually reach its base

[6] It is inconvenient to write $a\!:\!b\!:\!b\!:\!a\!:\!\texttt{[]}$ for the string "abba". We will simply write "abba" and convert in our minds between the simpler, orthographical writing convention and the official formal representation.

[7] We did this already in the definition of the set of strings over Σ.

case, where $w = \texttt{[]}$, and give us a concrete (i.e. non-self-referential) answer. We can illustrate the workings of this definition via an example.

Example 2. We trace through the action of the length function on the string *abba*.

$$\begin{aligned}
|a : b : b : a : \texttt{[]}\,| &= 1 + |b : b : a : \texttt{[]}\,| \\
&= 1 + 1 + |b : a : \texttt{[]}\,| \\
&= 1 + 1 + 1 + |a : \texttt{[]}\,| \\
&= 1 + 1 + 1 + 1 + |\,\texttt{[]}\,| \\
&= 1 + 1 + 1 + 1 + 0 \\
&= 4
\end{aligned}$$