COMPSCI 250: Introduction to Computation

Lecture #20: Strings and String Operations
David Mix Barrington and Ghazaleh Parvini
20 October 2023

Strings and String Operations

- Review: Rules of Arithmetic for Naturals
- Peano Axioms for Strings
- Pseudo-Java for the string Class
- Defining the String Operations
- Proof By Induction For Strings
- Concatenating Strings Adds Lengths
- Concatenation is Associative
- Reversal of a Concatenation

Arithmetic for Naturals

- Last time we presented a series of proofs for rules of arithmetic on naturals:
- Addition is commutative [$x + y = y + x$]
- Addition is associative [$x+(y+z) = (x+y)+z$]
- Multiplication is commutative [$x * y = y * z$]
- We started to prove that multiplication is associative, but wound up needing to do the distributive law [$x*(y+z) = x * y + x * z$] first.

Arithmetic for Naturals

- The proofs used the definition of + in terms of successor, and that of * in terms of + and successor.
- Each proof used Generalization on all but one variable, letting them be arbitrary, and then ordinary induction on the rightmost variable.
- We had a base case for that variable being 0, which used the definitions for “+0” and “*0”.

Arithmetic on Naturals

- Then we had an inductive case, where we assumed that our statement was true for some value z and proved it also held for Sz.
- For example, with commutativity our IH was "x + y = y + x" and our inductive goal was then "x + Sy = Sy + x".
- We could turn "x + Sy" into "S(x + y)" and then into "S(y + x)", using the IH. But then the lemma that "S(y + x) = Sy + x" needed another proof, by induction on x.

Proof of the Lemma

- How do we prove $\forall x{:}P(x)$, where $P(x)$ is the statement "$S(y + x) = Sy + x$"?
- Our base case is "$S(y + 0) = Sy + 0$", which is true because $z + 0 = z$ for any z by definition.
- Our IH is then "$S(y + x) = Sy + x$" and our inductive goal "$S(y + Sx) = Sy + Sx$".
- $S(y + Sx) = S(S(y + x))$ (by definition of $+$) $= S(Sy + x)$ (by the IH) $= Sy + Sx$ (by definition of $+$ again, in the other direction)

The Rest of the Proofs

- You should have another look at the proofs of the three arithmetic rules in the previous lecture.
- None of them are difficult, in one sense, because there are rarely many choices about which rule to apply.
- The definition of addition lets us go back and forth between $z + Sx$ and $S(z + x)$.
- The definition of multiplication lets us go between $z \times Sx$ and $(z \times x) + z$.

Associativity and Distributivity

- As in the textbook, we'll start proving the associative law for multiplication, which is $\forall u: \forall v: \forall w: u \times (v \times w) = (u \times v) \times w$.
- We let u and v be arbitrary, and use induction on w with P(w): "$u \times (v \times w) = (u \times v) \times w$". The base case P(0) is "$u \times (v \times 0) = (u \times v) \times 0$", which reduces to "$0 = 0$" by known rules.
- We assume P(w) and try to prove P(Sw) which is "$u \times (v \times Sw) = (u \times v) \times Sw$".

Associativity and Distributivity

- The LHS reduces to $u \times ((v \times w) + v)$ by the definition, which is $(u \times (v \times w)) + (u \times v)$ by *distributivity*, which unfortunately we haven't proved yet.
- If we had done distributivity first, we could finish by using the IH to get $((u \times v) \times w) + (u \times v)$, and then the definition of multiplication to get $(u \times v) \times Sw$, the desired right-hand side.
- This makes proving the Distributive Law an important exercise — it is Problem 4.6.2.

Peano Axioms for Strings

- We define our string data type for any fixed alphabet Σ by induction, just as we defined the naturals.
- The basic way to make new strings from old is by appending a letter to a string.
- We can define five "Peano axioms" for strings, which are much like the Peano axioms for the naturals.

Peano Axioms for Strings

- 1. λ is a string.
- 2. If w is a string and a is a letter in Σ, then wa is a string.
- 3. If wa and vb are the same string, then $w = v$ and $a = b$ (i.e., no string is formed by appending in two different ways).
- 4. If and only if it is not λ, any string is equal to wa for some string w and letter a.
- 5. The only strings are those made from λ by the second axiom.

The Pseudo-Java `string` class

- We can think of our string operations as being built up from basic `string` methods in our pseudo-Java programming language.
- Remember that unlike real Java `String` objects, pseudo-Java string values are primitives.
- We have a method to test whether a `string` is empty, a method to append a letter, and two "inverses" for the append operation.

The Pseudo-Java `string` class

- The inverse methods throw an exception if called on an empty string. If called on a string `wa`, `last` returns a, the last letter, and `allButLast` returns the string `w`.

```
public static boolean isEmpty(string w){...}

public static string append (string w, char a)
{…}

public static char last (string w) {...}

public static string allButLast (string w) {...}
```

Clicker Question #1

```
public static int foo (string w) {
    while (!isEmpty(w) && last(w) == ‘a’)
        w = allButLast(w);
    if (isEmpty(w)) return 1;
    while (!isEmpty(w) && last(w) == ‘b’)
        w = allButLast(w);
    if (isEmpty(w)) return 2;
    else return 3;  }
```

- Which statement is true of this method?
- (a) it may throw an exception
- (b) it cannot return 1
- (c) it cannot return 2
- (d) it will return a value and could be 1, 2, or 3

Not the Answer

Clicker Answer #1

```
public static int foo (string w) {
   while (!isEmpty(w) && last(w) == ‘a’)
      w = allButLast(w);
   if (isEmpty(w)) return 1;
   while (!isEmpty(w) && last(b) == ‘b’)
      w = allButLast(w);
   if (isEmpty(w)) return 2;
   else return 3;  }
```

- Which statement is true of this method?
- (a) it may throw an exception guarded by &&
- (b) it cannot return 1 returns 1 on “aa”
- (c) it cannot return 2 returns 2 on “bb”

  returns 3 on “aba”
- (d) it will return a value and could be 1, 2, or 3

Defining String Operations

- We defined operations on naturals recursively, first saying what the operation does with argument 0 and then defining what argument n+1 does based on what argument n does.
- Here we can do much the same thing for strings.
- Each operation comes from a simple recursive definition.

Length and Concatenation

- The code for these methods follows fairly directly from the inductive definitions.

```
public static natural length (string w) {
    if (isEmpty(w)) return 0;
    return
       successor(length(allButLast(w)));
}
public static string cat(string w, string x) {
    if (isEmpty(x)) return w;
    return append
       (cat(w, allButLast(x)), last(x));
}
```

The Code for Reversal

- There's two interesting wrinkles in this code for the reversal operation.
- We need the `cat` operation to be defined.
- Since `cat` takes two `string` arguments, we have an implicit type cast from the character `last(w)` to a string.

```
public static string rev (string w) {
    if (isEmpty(w)) return w;
    return cat(last(w),
               rev(allButLast(w)));}
```

Clicker Question #2

```
public static string mixup (string w) {
  return rev(cat(rev(allButLast(w)),
                 allButLast(rev(w))));
}
```

- What does this method output on input “xyz”?
- (a) “yxxz”
- (b) “yxzy”
- (c) “yzxy”
- (d) “zxyz”

Not the Answer

Clicker Answer #2

```
public static string mixup (string w) {
  return rev(cat(rev(allButLast(w)),
                 allButLast(rev(w))));
}
```

- What does this method output on input “xyz”?
- (a) “yxxz”
- (b) “yxzy”
- (c) “yzxy”
- (d) “zxyz”

aBL(w) == “zy”
rev(aBL(w) == “yx”
rev(w) == “zyx”
aBL(rev(w)) = “zy”
cat == “yxzy”
rev(cat) = “yzxy”

Proof by Induction for Strings

- As we noted above, an alternate version of the fifth Peano Axiom for strings allows us to prove statements of the form ∀x: P(x), where x is of type string, by induction on all strings.
- We need a base case of P(λ), and then an inductive case for each letter a in Σ, of the form ∀w: P(w) → P(wa).
- With binary strings we must prove P(w) → P(w0) and P(w) → P(w1) for arbitrary w (or just prove P(w) → (P(w0) ∧ P(w1)).

Proof By Induction for Strings

- Each of our recursive definitions defines f(wa), for example, in terms of f(w).
- So if we can phrase our statement P(w) so that it talks about f(w), then information about f(w) should be useful in talking about f(wa) when we prove P(wa).
- We'll finish the lecture by doing three such inductive proofs.

Concatenation Adds Lengths

- Our first proof relates a string operation to an operation on naturals.
- When we concatenate two strings, we add their lengths. Let's prove the statement ∀u: ∀v: |uv| = |u| + |v|, where we use "|u|" to mean the length of u.
- We let u be an arbitrary string and use string induction on v.
- The statement P(v) is "|uv| = |u| + |v|", or "length(cat(u, v)) == plus(length(u), length(v))".

Concatenation Adds Lengths

- The base case P(λ) says that $|u\lambda| = |u| + |\lambda|$, which is true because the definitions tell us that $u\lambda = u$, $|\lambda| = 0$, and $|u| = |u| + 0$.
- We assume P(v) and look at P(va), which says $|u(va)| = |u| + |va|$.
- To prove this we will need to use the inductive clauses of two recursive definitions, that of concatenation and that of length.

Concatenation Adds Lengths

- The definition of concatenation says that u(va) = (uv)a, and the definition of length then says that |u(va)| = |(uv)a| = successor(|uv|).
- The definition of length says that |va| = successor(|v|), and the definition of addition says that |u| + successor(|v|) = successor(|u| + |v|).
- We finish by using the IH to replace |uv| by |u| + |v|. This completes the inductive step for arbitrary v and a.

Clicker Question #3

- Let $\Sigma = \{a, b, \ldots, z\}$. Define a function blaze(w) from Σ^* to Σ^* by the rules blaze(λ) = λ, and for any string w and any letter a, blaze(wa) = rev(cat(last(rev(aw)), rev(blaze(w))))).

  What does blaze(w) return for any string w?

- (a) w itself
- (b) the reversal of w
- (c) a, followed by the reversal of w
- (d) no output -- exception is thrown

Not the Answer

Clicker Question #3

- Let $\Sigma = \{a, b, \ldots, z\}$. Define a function blaze(w) from Σ^* to Σ^* by the rules blaze(λ) = λ, and for any string w and any letter a, blaze(wa) = rev(cat(last(rev(aw)), rev(blaze(w))))

  What does blaze(w) return for any string w?

- (a) w itself
- (b) the reversal of w
- (c) a, followed by the reversal of w
- (d) no output -- exception is thrown

rev(aw)==wa
last(rev(aw))==a
Let z = blaze(w)
rev(cat(a, rev(z)))==za
blaze(wa)==blaze(w)a
blaze is the identity function

Concatenation is Associative

- Now we prove ∀u:∀v:∀w: (uv)w = u(vw), where we use parentheses to indicate the order of operations. We let u and v be arbitrary, and use string induction on w with P(w) as "(uv)w = u(vw)" or "cat(cat(u, v), w) == cat(u, cat(v, w))".
- The base case P(λ) is "(uv)λ = u(vλ)", which reduces to uv = uv by the definition of concatenating with λ.

Concatenation is Associative

- We assume P(w) and try to prove P(wa), which says "(uv)(wa) = u(v(wa))". (Again we must be careful of notation, as we are using the same notation for appending and concatenation.)
- The LHS is ((uv)w)a, and the RHS is u((vw)a) which we can convert to (u(vw))a, each time using the definition of concatenation.
- The IH of "(uv)w = u(vw)" now lets us prove that the LHS equals the RHS, by appending an a to each side of this equation.

Reversal of a Concatenation

- Finally we prove the rule relating reversal and concatenation, the statement $\forall u{:}\forall v{:}$ $(uv)^R = (v^R)(u^R)$. (For example, $(\text{“bulldog”})^R = (\text{“dog”})^R(\text{“bull”})^R =$ “godllub”.) We'll let u be arbitrary and use string induction on v.

- The base case $P(\lambda)$ is “$(u\lambda)^R = \lambda^R u^R$”. We can prove this with the rules $u\lambda = u$ and $\lambda^R = \lambda$, and the theorem $\lambda u = u$, which is easy to prove by induction on u.

Reversal of a Concatenation

- So we assume P(v), "$(uv)^R = v^R u^R$", and try to prove P(va), "$(u(va))^R = (va)^R u^R$".
- The LHS is $((uv)a)^R$ by the definition of concatenation, and $a(uv)^R$ by the definition of reversal. (Note that this last is the concatenation of the two strings a and $(uv)^R$.)
- The RHS is $(av^R)u^R$ by the definition of reversal, and then $a(v^R u^R)$ by associativity of concatenation from the previous slide. We can now equate these forms of the LHS and the RHS by using the IH once. This completes the inductive step and thus also the proof.