Substitution

Recap

Recall the definition of the set $Tree_{\Sigma}(\mathcal{X})$ where Σ is a signature and $\mathcal{X}$ is a denumerable set of *variables,* and that $\mathbf{Tree}_{\Sigma}(\mathcal{X})$ is a Σ-algebra with carrier set $Tree_{\Sigma}(\mathcal{X})$, with each symbol in signature Σ interpreted as a tree-forming operation over the set $Tree_{\Sigma}(\mathcal{X})$. Or more pertinently, $\mathbf{Tree}_{\Sigma}(\mathcal{X})$ represents a Σ-algebra where "syntax is the semantics".

Now let us consider any valuation $\sigma : \mathcal{X} \to Tree_{\Sigma}(\mathcal{X})$ where variables are mapped to syntactic elements from the carrier set of the tree algebra $Tree_{\Sigma}(\mathcal{X})$ rather than to semantic elements. Such a valuation is called a *substitution*. Now consider its Σ-homomorphic extension, namely $\hat{\sigma} : Tree_{\Sigma}(\mathcal{X}) \to Tree_{\Sigma}(\mathcal{X})$. This is a function that when applied to any tree $t \in Tree_{\Sigma}(\mathcal{X})$ will go through the tree from root to the leaves, and create a new tree $t' \in Tree_{\Sigma}(\mathcal{X})$ that maintains *every constant or function symbol* in t, but at the leaves of t which are variables $x \in \mathcal{X}$, the function $\sigma : \mathcal{X} \to Tree_{\Sigma}(\mathcal{X})$ is applied.

Since any tree can have only finitely many variables, the substitutions, by the Relevant Variables Lemma, need only specify the variables which are replaced by something different from those variables, i.e., *only those points where it is not identity.*

So there are many ways to define what a substitution is.

Version 1: A substitution is a total function from variables to trees that is identity "almost everywhere", i.e., at all except a finite number of points.
(What if it is identity everywhere?)

Version 2: A substitution is a finite domain partial function from variables to trees.
(What if the finite domain is empty?)

Version 3: A substitution in "defunctionalised form" is representable as a finite table with rows consisting of variables to be replaced and the corresponding trees with which to replace them. No variable appears in the first column of two different rows. (What if the table has no rows?)

A very common case is substitutions where only a single variable is replaced. This is written as $t[u/x]$ for trees $t, u \in Tree_{\Sigma}(\mathcal{X})$ and variable $x \in \mathcal{X}$. This is read as "substitute u for x in t". For this simpler case, we will state a fundamental result about substitutions: If we perform the substitution $[u/x]$ on t and then take the semantic *meaning* — with respect to Σ-algebra $\mathscr{A} = \langle A, \ldots \rangle$ — of the resulting tree as given by the (unique homomorphic extension of) any A-valuation $\rho_{\mathscr{A}} : \mathcal{X} \to A$, then we get the same result as finding the semantic value (wrt the same $\mathscr{A} = \langle A, \ldots \rangle$) of the tree t as given by the (unique homomorphic extension of) the A-valuation $\rho_{\mathscr{A}}[x \mapsto a] : \mathcal{X} \to A$, which is the same as valuation $\rho_{\mathscr{A}}$ but maps the variable to the value $a \in A$ that is the meaning wrt valuation $\rho_{\mathscr{A}}$ of the tree $u \in Tree_{\Sigma}(\mathcal{X})$.

Substitution Lemma for ∑-algebras

Let $\sum$ be a signature, and let $\mathscr{A} = \langle A, \dots \rangle$ be a $\sum$-algebra. Suppose t, u are trees in $Tree_\Sigma(\mathscr{X})$, and $x \in \mathscr{X}$ is a variable, and let $t[u/x]$ denote the substitution of u for <u>all</u> occurrences of the variable x in t. (Every other variable is unchanged.)
Let $\rho \in [X \to A]$ be any A-assignment, and suppose $\widehat{\rho_{\mathscr{A}}}(u) = a$, where $\widehat{\rho_{\mathscr{A}}} \in [Tree\_\Sigma(X) \to A]$ denotes the unique $\sum$-homomorphic extension of ρ as per the interpretations of $\sum$'s symbols in the $\sum$-algebra $\mathscr{A} = \langle A, \dots \rangle$. Let $\rho[x \mapsto a] \in [X \to A]$ denote the A-valuation that is identical to ρ at all variables except at x, where it is given the value $a \in A$. Then

$$\widehat{\rho_{\mathscr{A}}}(t[u/x]) = \widehat{\rho[x \mapsto a]}_{\mathscr{A}}(t).$$

Proof is by induction on the *structure* (i.e., `ht`) of $t \in Tree_\Sigma(\mathscr{X})$ using the definitions of substitution and the unique homomorphic extensions $\widehat{\rho_{\mathscr{A}}}$ and $\widehat{\rho[x \mapsto a]}_{\mathscr{A}}$.
<u>Base cases</u>: (`ht` (t) = 0)

<u>Subcase</u> t is a leaf node $\bullet c$ labelled by 0-ary symbol c in $\sum$.
<u>Subcase</u> t is a leaf node $\bullet x$ labelled by variable x.
<u>Subcase</u> t is a leaf node $\bullet y$ labelled by variable $y \neq x$.

<u>Induction Hypothesis</u>: Suppose that for all $t' \in Tree_\Sigma(\mathscr{X})$ such that `ht` (t') ≤ n:
$\widehat{\rho_{\mathscr{A}}}(t'[u/x]) = \widehat{\rho[x \mapsto a]}_{\mathscr{A}}(t')$

<u>Induction step</u>: (`ht` (t) = n+1)

□

Exercise: Details are left as an exercise.

* **Exercise**: State (and prove) the Substitution Lemma for any general substitution.

Let us now implement in OCaml this notion of substitution to work with any abstract syntax, namely with **Tree**$_\Sigma(\mathscr{X})$ for any signature $\sum$ and a denumerable set of variables $\mathscr{X}$ (assuming $\sum$ and $\mathscr{X}$ are disjoint sets of symbols).

Representing arbitrary signatures in OCaml.

A naive idea is to represent a symbol in a signature as a pair consisting of
- a string representing the symbol,
- and an int representing the arity of the symbol

and then represent the signature as a *set* of such pairs; here we use lists as a representation for sets.

```
open List;;

type symbol = string * int;;

type signature = symbol list;;
```

An example signature is

```
let sig1 = [ ("0", 0); ("1", 0); ("+", 2); ("*", 2) ];;
```

Exercise: Write a program to check that a purported signature is indeed legitimate, i.e., that no symbol appears twice, and that the arities are all non-negative.
(Alternatively, you may allow treating the same string with different arities as different symbols.)

A tree for all seasons

For an arbitrary (but fixed) signature, we already have a sense of how to define trees as a data type: by using a constructor corresponding to each symbol. We now present a way to do so for any arbitrary signature represented as given above. There are more elegant ways of doing so in OCaml, e.g., using the module system and treating signatures as a parameter, but we will keep things simple for now.

A tree (over a given signature) is either
- a variable,
- or can be represented as an OCaml *record* that has a
  - root *node* which is a `symbol`, and
  - an ordered list of *children* which are its subtrees.

```
type tree = V of string
          | C of {node: symbol; children: tree list};;
```

Notice that this is a recursive definition. What are the base cases? How are leaves labelled by constants represented?

Of course for any purported tree, we should check if the symbols that appear in it are indeed symbols as defined in the signature, and whether the number of children agrees with the arity of the symbol at a node.

Examples: We use the meta-language OCaml's definition facility make it easier for us to enter our generalised toy "object language" variables and trees.

```
let x = V "x";;
let y = V "y";;
let z = V "z";;
```

```
let zero = C {node = ("0", 0); children = []};;
let one = C {node = ("1", 0); children = []};;
let plus_zero_one = C {node = ("+", 2); children = [zero; one] };;
let times_one_x = C {node = ("*", 2); children = [one; x] };;
let plus_zero_y = C {node = ("+", 2); children = [zero; y] };;
let plus_timesonex_pluszeroy =
                 C {node = ("+", 2);
                    children = [times_one_x; plus_zero_y ] };;
let plus_timesonex_z =
                 C {node = ("+", 2);
                    children = [times_one_x; z ] };;
```

Exercise: Write a program to check that a purported tree over a given signature is indeed legitimate, i.e., that the node has exactly as many children as the arity of the symbol specifies.

Let us now define the usual function to find the height of a tree.

```
let rec ht t = match t with
    V _ -> 0
  | C r ->
      let (s,n) = r.node
      in (if n = 0
          then 0
          else 1+(fold_left max 0 (map ht r.children)
                 ))
;;
```

Examples

```
ht zero;;
ht x;;
ht plus_zero_one;;
ht plus_timesonex_pluszeroy;;
ht plus_timesonex_z ;;
```

The number of nodes in a tree can be computed by the following function `size`. Note the similarity (rather than the difference) to the function `ht`.

```
let rec size t = match t with
    V _ -> 1
  | C r -> 1+(fold_left (+) 0 (map size r.children) )
;;
```

Examples

```
size zero;;
size x;;
size plus_zero_one;;
size plus_timesonex_pluszeroy;;
size plus_timesonex_z ;;
```

Exercise: Write a function `vars` that given a tree, returns the set of variables that appear in it.

We are now ready to define an implementation of substitution of a collection of trees for a corresponding collection of variables in a given tree `t`. Again, note the similarity in the structure of the function `subst sigma` to the functions `ht`, `size` (and hopefully the function `vars` that you defined). Note that `subst` is a higher-order function that takes as its first argument the substitution `sigma` from the set of variables $\mathscr{X}$ to the set of trees $Tree_{\Sigma}(\mathscr{X})$.   In our OCaml encoding, `sigma` is of type `string -> tree`.

```
let rec subst sigma t = match t with
    V x -> sigma x
  | C r -> C { node = r.node;
                children = map (subst sigma) r.children }
;;
```

Notice that a substitution when applied to a tree only changes variables, which appear at some leaves. Otherwise applying a substitution involves going down tree recursively, preserving the structure of the input tree. All occurrences of a variable are changed uniformly, as specified by the substitution.

Here is an example substitution: the identity substitution.

```
let id_sub v = V v;;
```

Let us see how `subst` works when supplied the identity substitution `id_sub`.

```
subst id_sub zero ;;
subst id_sub one;;
subst id_sub x;;
subst id_sub y;;
subst id_sub z;;
subst id_sub plus_timesonex_pluszeroy ;;
subst id_sub plus_timesonex_z ;;
```

Exercise: Prove that forall $t \in Tree_{\Sigma}(\mathcal{X})$,

`subst id_sub` $t = t$

Here is another example of a substitution which replaces string “x” by a constant, and string “y” by a nontrivial tree, and leaves every other variable unchanged.

```
let sigma1 v = match v with
    "x" -> one
  | "y" -> plus_timesonex_pluszeroy
  | _ -> V v
;;
```

Let us see how `subst` works when supplied the identity substitution `sigma1`.

```
subst sigma1 zero ;;
subst sigma1 one;;
subst sigma1 x;;
subst sigma1 y;;
subst sigma1 z;;
subst sigma1 plus_timesonex_pluszeroy ;;
subst sigma1 plus_timesonex_z ;;
```

Continuing with our naïve representation of substitutions as functions of type `string -> tree`, let us define the composition of functions. This should be compared with the earlier definition of composition of functions `comp f g = g( f x).`
This form of composition is called "*Kleisli* composition".

```
let compose_subst s1 s2 t = subst s2 (subst s1 t);;
```

The substitution `id_sub` behaves as the left and right identity for the binary operation `compose_subst.`

Exercise: Prove that forall `s1: string -> tree,`
`compose_subst id_sub s1 = s1 = compose_subst s1 id_sub`

Exercise: State and prove that `compose_subst` is associative.

Unifiers

Definition: A unifier of two trees t_1 and t_2 *(if it exists)* is *any* substitution *sigma* such that `subst` *sigma* t_1 = `subst` *sigma* t_2, i.e., it returns identical trees.

Definition: Let $Unif(t_1, t_2)$ define the set of *all* possible substitutions that unify trees t_1 **and** t_2. Note that t_1 and t_2 are not always unifiable.

Questions:
What if there are no such unifying substitutions?
Can there be more than one unifier of t_1 and t_2?
Exercise:
Give examples where t_1 and t_2 have no unifiers.
Give examples where t_1 and t_2 have multiple unifiers.

Exercise: Prove that for all $t_1, t_2 \in Tree_\Sigma(\mathcal{X})$: $Unif(t_1, t_2) = Unif(t_2, t_1)$.

Definition: We define an ordering on substitutions *s1* and *s2*, written *s1* ≤ *s2*, if there exists any substitution *s'* such that `compose_subst` *s1 s'* = *s2*. We say that *s1* is more general than *s2*.

Exercise: Prove that the ordering ≤ is a pre-order, i.e., it is
- reflexive: for all substitutions *s*, *s* ≤ *s*, and
- transitive: for all substitutions *s1, s2, s3*, if *s1* ≤ *s2* and *s2* ≤ *s3*, then *s1* ≤ *s3*.

Since unifiers of two trees t_1 and t_2 are substitutions, the pre-ordering notion of "more general than" also applies to unifiers: If *s1* and *s2* are both in $Unif(t_1, t_2)$, then *s1* is a more general unifier of t_1 and t_2 than *s2* if *s1* ≤ *s2*.
Note here that the s' used to show *s1* ≤ *s2* need not be in $Unif(t_1, t_2)$.

Definition: *s* in *Unif(t1, t2)* -- if it exists -- is *a most general unifier* of t_1 and t_2, written $mgu(t_1, t_2)$, if for every other unifying substitution *s'* in $Unif(t_1, t_2)$, we have *s* ≤ *s*'.

Note that we say "a most general unifier" -- most general unifiers of two tree not be unique. That is, both *s1* and *s2* can be *mgu*s in $Unif(t_1, t_2)$, with both *s1* ≤ *s2* and *s2* ≤ *s1*, but *s1 =/= s2*.
However, two different *mgu*s do not differ greatly: they only do so to the extent of variable renamings. For example: both substitutions *{*`"x" ↦ V "y"`*}* and *{* `"y" ↦ V "x"`*}* are *mgu*s in *Unif(*`V "x"`, `V "y"`*)*

Exercise: Verify that
{`"x" ↦ V "y"`*}* ≤ *{* `"y" ↦ V "x"`*}*, and
{ `"y" ↦ V "x"`*}* ≤ *{*`"x" ↦ V "y"`*}*.

An algorithm to compute Most General Unifiers
... if they exist.

Given two trees *t* and *u*, there exists an algorithm to compute their *mgu* in *Unif(t,u)*, if it exists, and which fails otherwise.

Let us write a specification of this algorithm, which we call *mgu (t, u),* in a functional style. It is defined expectedly by case analysis.

Case analysis on *t* and *u*.

If both *t* and *u* are variables:
- if they are the same variable, i,e, for some string *x,*
  t is `V` *x* and *u* is `V` *x*, then return the identity substitution `id_sub` as the *mgu*.

- if they are different variables, i,e, for some string *x*, *y* which are different,
  t is `V` *x* and *u* is `V` *x*, then without loss of generality, return the substitution *{ x* ↦ `V` *y}* as the *mgu*.

If one of *t* and *u* is a variable and the other not a variable:
- without loss of generality (why can we assume this)
  assume that *t* is `V` *x*, and *u* is `C` *r*, for some *r*.
  Then we return the substitution *{ x ↦ u}* as the *mgu*.

Now this is actually wrong in general. We will see why this is wrong later. However, it is "almost correct". We just need to clarify when it is correct, and when it is not correct, and what to do in that case.

If both *t* and *u* are <u>not</u> variables:.
So *u* is `C` *r* and *u* is `C` *r'*, for some *r* and *r'*.

- if *r*.`node` =/= *r'*.`node` (i.e., the roots of the trees have different symbols),
  then the algorithm FAILs (in an implementation, we may raise an exception). There is no *mgu*.

- if *r*.`node` = *r'*.`node` (i.e., the roots of the trees have the same symbol),
  then we recur on each pair of corresponding children of *t* and *u,* trying to unify each corresponding pair of children....
  .... but we need to do so in *some serial order*, not in parallel!!

```
suppose r.children = [t1;  ...  ;  tk]
 and r'.children = [u1;  ...  ;  uk]

let s0 = id_sub
 let s1 = compose_subst s0 (mgu (subst s0 t1, subst s0 u1))
   let s2 = compose_subst s1 (mgu (subst s1 t2, subst s1 u2))
     ......
      let sk = compose_subst $s_{k-1}$ (mgu (subst $s_{k-1}$ tk, subst $s_{k-1}$ uk))

return $s_k$ as the mgu.

If a FAIL occurs at any stage, the whole process FAILs, and there is no unifier.
```

We now identify and correct the error in the case when *t* is `V x`, and *u* is `C r`, for some *r* — where we wanted to return the substitution *{ x ↦ u}* as the *mgu*. Why is it wrong, in general, to claim that *{ x ↦ u}* is the *mgu?*

Imagine trying to unify `V "x"` and
`C { node = ("f", 1); children = [ V "x" ] }`.

If we take *{ x ↦* `C { node = ("f", 1); children = [ V "x" ] }` *}* as their unifier, and apply this substitution on both the trees, `V "x"` and
`C { node = ("f", 1); children = [ V "x" ] },` we find that they do not yield the same result. Why aren't they the same? Because the variable `"x"` appears in both trees, in one at the root position, and in the other at a non-root position. So when replaced, we will get two different trees. In fact no matter how many times we apply this substitution, the two trees will always be different, the second one always of greater height. So this cannot be the unifier we seek.

However, if this variable `V` *x* does not appear in *u,* then indeed this substitution is the *mgu.* So we have a simple fix (This is called an "Occurs-check"): check if the variable occurs in *u*. If it does, then the algorithm FAILs, and there is no *mgu.* Otherwise, return the substitution *{ x ↦ u}* as the *mgu.*

Exercise: Write a program `mgu` *(t, u)* that given two trees *t* and *u* encoded as above, returns their *mgu* if it exists and raises and exception `Fail` otherwise.

Note that it may be useful to write the code in a manner where there is a "running" substitution as the partial *mgu* computed so far; this is given an initial default value. What should this be? Note that in the algorithm, we take a pair of equal-length lists of trees, and process them as list of pairs of trees t_i and u_i. Note also that we apply the running substitution to the both t_i and u_i. And that the computed *mgu* s_i is Kleisli-composed with the previous "running" substitution s_{i-1}. All of this can be easily written as an iteration, namely using functions such as `zip`, `map` and `fold_left,` or some variation on those programs.

Exercise: Present an informal argument that your program `mgu` *(t, u)* that given two trees *t* and *u* encoded as above, returns their *mgu* if it exists and raises and exception `Fail` otherwise. Is `mgu` *(t, u)* `=` `mgu` *(u, t)* ? If not, how does it differ? Why is this acceptable?