Stacks

Let us specify now a linear data structure, called **stack**.

A stack is similar to a pile of paper sheets on a table: we can only add a new sheet on its top (this is called **to push**) and remove one on its top (this is called **to pop**).

From this informal description, we understand that we shall need a constructor for the stack that takes an argument (like a sheet): it is a function. This is different from the boolean constructors which are constants ($\textsc{True}$ and $\textsc{False}$).

Stacks (cont)

How do we model the fact that the stack has changed after a pop or a push? The simplest is to imagine that we give the original stack as an argument and the function calls (pop/push) represent the modified stack.

Also, we do not want to specify actually the nature of the elements in the stack, in order to be general: we need a parameter type for the elements.

Stacks/Signature

Let us call $\textsc{Stack}$(item) the specification of a stack over the item type.

- **Parameter types**
  - The type item of the elements in the stack.
- **Defined types**
  - The type of the stacks is t.
- **Constructors**
  - $\textsc{Empty}$: t
    Expression $\textsc{Empty}$ represents the empty stack.
  - $\textsc{Push}$: item × t → t
    Expression $\textsc{Push}(e, s)$ denotes the stack s with element e pushed on top.

Stacks/Constructors

We need a **constant constructor** to stand for the empty stack, otherwise we would not know what stack remains after popping a stack containing only one element.

The type of $\textsc{Push}$ is item × t → t, which means it is a **non-constant constructor** (it is a special case of function, basically) which takes a pair made of an element and a stack and returns a new stack (with the element on top).

Here are some stacks:

- $\textsc{Empty}$
- $\textsc{Push}(e_1, \textsc{Push}(e_2, \textsc{Empty}))$

Stacks/Projections

We can complement this definition with other functions which allows us to extract information from a given stack. In particular, a function which gives us back the information which was given to some constructor is called a **projection**. *A projection is the inverse function of a constructor.*

The constant constructor $\textsc{Empty}$ has no inverse function, because it can be considered as equivalent to a function f defined as $\forall\, x\,.f(x) = \textsc{Empty}$, whose inverse f^{-1} is not a function because it maps $\textsc{Empty}$ to any x.

Thus we only care of **non-constant constructors**, i.e., the ones which take arguments.

Stacks/Projections and other functions

Since the specification $\textsc{Stack}$(item) has only one non-constant constructor, we have only one projection. We can also add a function $\textsc{Append}$.

Here is how the signature continues:

- **Projections**
  - $\textsc{Pop} : \mathsf{t} \rightarrow \mathsf{item} \times \mathsf{t}$
    This projection is the inverse of constructor $\textsc{Push}$.
- **Other functions**
  - $\textsc{Append} : \mathsf{t} \times \mathsf{t} \rightarrow \mathsf{t}$
    Expression $\textsc{Append}(s_1, s_2)$ represents a stack made of stack s_1 on top of stack s_2.

Stacks/Equations

Now the defining equations of the stack:

$$\begin{aligned}
\textsc{Pop} \circ \textsc{Push} &= \mathit{id} \\
\textsc{Append}(\textsc{Empty}, \textsc{Empty}) &= \textsc{Empty} \\
\textsc{Append}(\textsc{Empty}, \textsc{Push}(e, s)) &= \textsc{Push}(e, s) \\
\textsc{Append}(\textsc{Push}(e, s), \textsc{Empty}) &= \textsc{Push}(e, s) \\
\textsc{Append}(\textsc{Push}(e_1, s_1), \textsc{Push}(e_2, s_2)) &= \textsc{Push}(e_1, \textsc{Append}(s_1, \textsc{Push}(e_2, s_2)))
\end{aligned}$$

where $\overline{e} = (e_1, e_2)$, $\overline{s} = (s_1, s_2)$ and *id* is the identity function $\forall x . x \mapsto x$.

Stacks/Prefixing

If we refer to the type of stacks over elements of type item *outside its definition*, we have to write:

$$\textsc{Stack}(\text{item}).\text{t}$$

So the empty stack is noted $\textsc{Stack}(\text{item}).\textsc{Empty}$ outside the $\textsc{Stack}$ specification, in order to avoid confusion with $\textsc{Bin-tree}.(\text{node}).\textsc{Empty}$, for instance.

If the context is not ambiguous, e.g., we know that we are talking about stacks, we can omit the prefix "$\textsc{Stack}.$" and simply write $\textsc{Empty}$, for instance.

Stacks/Recursive equations

An interesting point in the previous equations is that the function APPEND is defined on terms of itself. This kind of equation is called **recursive**.

This is not new for you. In high school you became familiar with **integer sequences** defined by equations like

$$U_{n+1} = b + U_n$$
$$U_0 = a$$

This is exactly equivalent to

$$U(n+1) = b + U(n)$$
$$U(0) = a$$

Only the notation differs. The meaning is the same.

Stacks/Simplifying the equations

We can ease the notation by omitting the quantifiers $\forall$ in equations. Also, we can simplify a little the equations for $\textsc{Append}$ by noting that if one of the stack is empty, then the result is always the other stack:

$$\begin{cases} & \textsc{Append}(\textsc{Empty}, \textsc{Empty}) = \textsc{Empty} \\ \forall e, s & \textsc{Append}(\textsc{Empty}, \textsc{Push}(e, s)) = \textsc{Push}(e, s) \\ \forall e, s & \textsc{Append}(\textsc{Push}(e, s), \textsc{Empty}) = \textsc{Push}(e, s) \end{cases}$$

$$\overset{?}{\iff} \begin{cases} \textsc{Append}(\textsc{Empty}, s) = s \\ \textsc{Append}(s, \textsc{Empty}) = s \end{cases}$$

Stacks/Simplifying the equations (cont)

The way to check this is to note that there are only two kinds of stacks, empty and no-empty, so we can replace s respectively by $\textsc{Empty}$ and $\textsc{Push}(e, s)$ in the new system:

$$\begin{cases} \textsc{Append}(\textsc{Empty}, s) = s \\ \textsc{Append}(s, \textsc{Empty}) = s \end{cases} \Leftrightarrow \begin{cases} \textsc{Append}(\textsc{Empty}, \textsc{Empty}) = \textsc{Empty} \\ \textsc{Append}(\textsc{Empty}, \textsc{Push}(e, s)) = \textsc{Push}(e, s) \\ \textsc{Append}(\textsc{Empty}, \textsc{Empty}) = \textsc{Empty} \\ \textsc{Append}(\textsc{Push}(e, s), \textsc{Empty}) = \textsc{Push}(e, s) \end{cases}$$

The first and third equations are the same. The system is the same as the original one.

Stacks/Orienting the equations

Let us call **term** the objects constructed using the functions of the specification, e.g., $\textsc{Empty}$ and $\textsc{Push}(e, \textsc{Empty})$ are terms. The e in the latter term is called a **variable** (and is a special case of term).

We call **subterm** a term embedded in a term. For instance

- $\textsc{Empty}$ is a subterm of $\textsc{Push}(e, \textsc{Empty})$;
- $\textsc{Push}(e_1, \textsc{Empty})$ is a subterm of $\textsc{Push}(e_2, \textsc{Push}(e_1, \textsc{Empty}))$;
- e is a subterm of $\textsc{Push}(e, \textsc{Empty})$;
- e is a subterm of e (it is not a *proper* subterm, though).

Stacks/Orienting the equations (cont)

How do we orient

$$\begin{aligned}
\textsc{Pop}(\textsc{Push}(x)) &= x \\
\textsc{Append}(\textsc{Empty}, s) &= s \\
\textsc{Append}(s, \textsc{Empty}) &= s \\
\textsc{Append}(\textsc{Push}(e, s_1), s_2) &= \textsc{Push}(e, \textsc{Append}(s_1, s_2))
\end{aligned}$$

Stacks/Orienting the equations (cont)

The first three ones are easy to orient since no function call of the defined function appear on both sides:

$$\begin{aligned} \textsc{Pop}(\textsc{Push}(x)) &\rightarrow x \\ \textsc{Append}(\textsc{Empty}, s) &\rightarrow_1 s \\ \textsc{Append}(s, \textsc{Empty}) &\rightarrow_2 s \\ \textsc{Append}(\textsc{Push}(e, s_1), s_2) &= \textsc{Push}(e, \textsc{Append}(s_1, s_2)) \end{aligned}$$

The last equation is a recursive equation, i.e., there is a function call of the defined function on both side of the equality. How should we orient it?

Stacks/Orienting the equations (cont)

Let us colour both calls to $\textsc{Append}$:

$$\textsc{Append}\ \mathbf{(}\textsc{Push}\ \mathbf{(}e\mathbf{,}\ s_1\mathbf{)}\mathbf{,}\ s_2\mathbf{)} = \textsc{Push}(e, \textsc{Append}\ \mathbf{(}s_1\mathbf{,}\ s_2\mathbf{)})$$

Let us colour only the differences between the two:

$$\textsc{Append}(\textsc{Push}\ \mathbf{(}e\mathbf{,}\ s_1\mathbf{)}, s_2) = \textsc{Push}(e, \textsc{Append}(s_1, s_2))$$

Obviously, s_1 is a *proper* subterm of $\textsc{Push}(e, s_1)$, so the value of s_1 is included in the value of $\textsc{Push}(e, s_1)$. The call-by-value strategy implies that the value of $\textsc{Append}(s_1, s_2)$ is included in the value of $\textsc{Append}(\textsc{Push}(e, s_1), s_2)$. Therefore we must orient the equation from left to right.

Stacks/Orienting the equations (cont)

What is the use of $\textsc{Append}(s, \textsc{Empty}) \rightarrow_2 s$?

It is actually useless because the first argument of $\textsc{Append}$ will always become $\textsc{Empty}$, since we replace it by a proper subterm at each rewriting, the second rewriting rule $\textsc{Append}(\textsc{Empty}, s) \rightarrow_1 s$ always applies at the end. So we only need:

$$\begin{aligned} \textsc{Append}(\textsc{Empty}, s) &\rightarrow s \\ \textsc{Append}(\textsc{Push}(e, s_1), s_2) &\rightarrow \textsc{Push}(e, \textsc{Append}(s_1, s_2)) \end{aligned}$$

The only difference is the **complexity**, that is, in this framework of rewriting systems, the number of steps needed to reach the result. With rule $\rightarrow_2$, if the second stack is empty, we can conclude in one step. Without it, we have to traverse all the elements of the first stack before terminating.

Stacks/Terms as trees

Let us find a model which clarifies these ideas: the concept of **tree**.

A tree is either

- the empty set
- or a tuple made of a **root** and other trees, called **subtrees**.

This is a **recursive definition** because the object (here, the tree) is defined by case and by grouping objects of the same kind (here, the subtrees).

A root could be further refined as containing some specific information.

It is usual to call **nodes** the root of a given tree and the roots of all its subtrees, *transitively*.

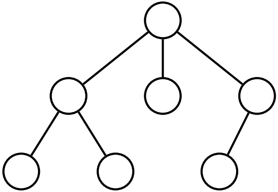
A node without non-empty subtrees is called a **leaf**.

Stacks/Terms as trees (cont)

If we consider trees as relationships between nodes, it is usual to call a root the **parent** of the roots of its direct subtrees (i.e., the ones immediately in the tuple). Conversely, these roots are **sons** of their parent (they are ordered).

It is also common to call subtree any tree included in it according to the subset relationship (otherwise we speak of *direct* subtrees).

Trees are often represented in a top-down way, the root being at the top of the page, nodes as circles and the relationship between nodes as **edges**. For instance:



Stacks/Terms as trees (cont)

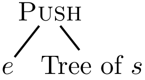
The **depth** of a node is the length of the path from the root to it (note this path is unique). Thus the depth of the root is 0 and the depth of the empty tree is undefined.

The **height** of a tree is the maximal depth of its nodes. For example, the height of the tree in the previous page is 2.

A **level** in a tree is the set of all nodes with a given depth. Hence it is possible to define level 0, level 1 etc. (may be empty).

Stacks/Height of terms

This leads us to consider *values* as *trees* themselves. Each constructor corresponds to a node, and each argument corresponds to a subtree. By definition:

Term	Tree	Height
Empty	Empty	0
Push(e, s)	 Push e Tree of s	$1 +$ height of tree of s

Now we can think the "size" of a value as the **height of the corresponding tree**.

Stacks/Height of terms (cont)

Let us define a function, called **height** and written $\mathcal{H}$, for each term denoting a stack in the following way:

$$\mathcal{H}(\textsc{Empty}) = 0$$
$$\forall\, e, s \quad \mathcal{H}(\textsc{Push}(e, s)) = \mathcal{H}(s) + 1$$

where x is a variable denoting an element and s a variable denoting a stack.