Stacks

Until now, we studied basic arithmetic and boolean expressions by means of rewrite systems, which provided the formal definition of their evaluation.

In order to check the expressive power of rewrite systems, let us show how we can use them to define precisely a **data structure** which is very useful in programming: the **stack**.

This approach allows us to define and understand the concept of stack without referring to features like **pointers**, which depend on specific **computer architectures** or **operating system libraries** etc.

For instance, in the case of C, it is not possible to study stacks without the notions of **memory**, **array** or **memory allocation and deallocation** (`malloc` and `free`), integer or **pointer arithmetic** etc.

Stacks/From the analogy to the formal notation

A stack is similar to a box which is open on the top and that contains a pile of paper sheets: 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**).

How do we model the fact that the stack has changed after a pop or a push? The simplest is to imagine that we rewrite the operation on a given stack into a (modified) stack.

Also, we do not want to specify actually the nature of the elements in the stack, in order to be general.

Items in stacks can be of any kind, so we do not define them: *items are independent from the concept of stack.*

Stacks/Examples

Let us define the “stack values”:

- let us note $\textsc{Empty}$ the empty stack;
- let us note $\textsc{Push}(i, s)$ the stack resulting of pushing item i onto the stack (value) s.

For example:

- $\textsc{Empty}$ is the empty stack;
- $\textsc{Push}(i, \textsc{Empty})$ is the stack containing only item i;
- $\textsc{Push}(i_1, \textsc{Push}(i_2, \textsc{Empty}))$ is the stack which only contains item i_1 on the top and item i_2 below (the bottom).

Stacks/Comparison with C

In C, we need the standard library. Then we need to define the stack data structure using pointers. Since it is recursive, we need a trick to declare it:

```
#include<stdlib.h>
typedef struct stack_ stack;
struct stack_ { int item; stack* next; };
```

Then, we need to take care of deallocating a stack:

```
void free_stack (stack* s) {
  if (s != NULL) { stack* sub_stack = (*s).next;
                   free(s);
                   free_stack(sub_stack);
                 }
}
```

Stacks/Comparison with C (cont)

The definition of Push in C must take care of allocating memory and checking whether this implies a memory overflow. If so, we must exit but not forget to deallocate the previous stack, if not Empty. Note the type of `push`: parameter `s` must be a pointer to a pointer because traditional C does not include reference passing. Also, the type of the items must be given (`int`).

```
void push (stack** s, int item) {
  stack* new_stack = malloc(sizeof(stack));
  if (new_stack == NULL) { if (s != NULL) free_stack(*s);
                           exit (2); }
  (*new_stack).item = item;
  (*new_stack).next = *s;
  *s = new_stack;
}
```

Stacks/Comparison with C (cont)

On this simple example, it is easy to see how the concept of stack is buried in many important details in C: memory management, address of pointers as arguments, tricky recursive type definition, fixed type for the items in the stack.

The approach with rewrite systems makes clear what the concept of stack is and what it is not necessary. In particular, our formalism does not make room for the concept of in-place modification: there is not even he concept of memory. Contrast this to the C implementation of `push`, which modifies the *given* stack in memory.

Stacks/Comparison with C (cont)

Also, in C we must give the type of the elements in the stack, even if the function does not use them: this is because the pointer arithmetic needs to know the size of each cell.

Then, once we understand what it is, we can more safely translate our understanding into programs because we already have in mind what we want to implement, and we can focus on the specific features of the language.

Stacks/Discussion

Until now, there are only stacks (or “stack values”) but no *computable* operation on them.

Operation $\textsc{Push}$ is not computable, i.e., there is no rewrite rule for it: $\textsc{Push}$ is only used to define what a new stack is, given another and an item to push on top of it.

This is different from the operator $\times$ on integers, because there were rewrite rules like $2 \times 5 \rightarrow 10$. There are no such rules for $\textsc{Push}$.

Stacks/Discussion (cont)

In other words, EMPTY and PUSH define the concept of stack, they are *values*.

But how about popping an item from the stack? This is should be part of the concept too, because an item can only be popped from the top of a stack. Should not this be part of the definition of the concept of stack?

Stacks/Popping

It is true that the way items are taken from a stack is part of the definition of the concept of stack, but in an operational way, not ontological.

In other words, popping, while participating to the definition of what a stack is, actually de-structures a stack. Therefore, popping can be defined in terms of a sub-stack, i.e., it can be defined as a computable operation.

Stacks/Popping (cont)

Actually, this is very easy: let us take $\textsc{Pop}$ as the exact inverse function of $\textsc{Push}$:

$$\textsc{Pop}(\textsc{Push}(i, s)) \to s \quad \langle\textsc{Pop}\rangle$$

Note that there is no rewrite rule for expression $\textsc{Pop}(\textsc{Empty})$ because there is no way to pop an item from an empty stack.

Stacks/Popping in C

In contrast, in C, the Pop function would we implemented as follows. Again, the programmer must take care of passing the address of the pointer to the top of the stack, s. In case of error, we return an error code, so the popped element, if any, must be an argument whose address is passed (top). Some errors are improbable, like s being NULL, but mandatory in general.

```
int pop (stack** s, int* top) {
  if (s != NULL)
    if (*s != NULL) { *top = (**s).item;
                      *s = (**s).next;
                      return 0; }
    return 2;
  return 1;
}
```

Stacks/Popping in C (cont)

Again, the concept of popping is buried in too many low-level details, like handling three error cases, even if improbable. It is not obvious at all, just by looking at the code, that POP is actually exactly the inverse function of PUSH.

Note also that the C implementation pop modifies the given stack, because it would be expensive to make a copy of it. On the other hand, modifying an argument, called **side-effect**, is always risky because the caller must be aware of it.

There is no such notion of side-effect in our rewrite systems: it is not necessary in order to understand what a stack actually is.

Stacks/Top

Now let us consider a operator which is similar to popping. The function $\textsc{Pop}$ returns the stack that remains when we ignore the top element. The function $\textsc{Top}$ instead returns this top element and ignores the remaining stack. This is fairly simple to define:

$$\textsc{Top}(\textsc{Push}(x, s)) \rightarrow x \quad \langle\textsc{Top}\rangle$$

Just like $\textsc{Pop}$, it does not apply to empty stacks. In C, we do not need a side-effect on the stack (only on the item):

```
int top (stack* s, int* item) {
  if (s == NULL) return 1;
  *item=(*s).item;
  return 0;}
```

Stacks/Appending

Let expression $\textsc{Append}(s_1, s_2)$ represent the stack made of stack s_1 on top of stack s_2.

Let us express exactly what we mean using a rewrite system defined by inference rules.

$$\begin{aligned}
\textsc{Append}(\textsc{Empty}, \textsc{Empty}) &\rightarrow \textsc{Empty}\\
\textsc{Append}(\textsc{Empty}, \textsc{Push}(e, s)) &\rightarrow \textsc{Push}(e, s)\\
\textsc{Append}(\textsc{Push}(e, s), \textsc{Empty}) &\rightarrow \textsc{Push}(e, s)\\
\textsc{Append}(\textsc{Push}(e_1, s_1), \textsc{Push}(e_2, s_2)) &\rightarrow
\end{aligned}$$

$\textsc{Push}(e_1, \textsc{Append}(s_1, \textsc{Push}(e_2, s_2)))$
Let us explain how we find theses rewrite rules.

Stacks/Appending (cont)

The first step consists in understanding that we must find rules of the shape

$$\textsc{Append}(s_1, s_2) \rightarrow \cdots$$

Second, we understand that s_1 and s_2 are stacks. We already know that stacks are built by $\textsc{Empty}$ or $\textsc{Push}$. Thus, we replace each stack by all of its possible basic values: $\textsc{Empty}$ and $\textsc{Push}(\ldots, \ldots)$. This leads to four cases:

$$\begin{aligned} \textsc{Append}(\textsc{Empty}, \textsc{Empty}) &\rightarrow \cdots \\ \textsc{Append}(\textsc{Empty}, \textsc{Push}(e, s)) &\rightarrow \cdots \\ \textsc{Append}(\textsc{Push}(e, s), \textsc{Empty}) &\rightarrow \cdots \\ \textsc{Append}(\textsc{Push}(e_1, s_1), \textsc{Push}(e_2, s_2)) &\rightarrow \cdots \end{aligned}$$

Stacks/Appending (cont)

Now we need to think about the interpretation of each case. An analogy and a picture is helpful. In the first case, we want append two empty stacks. It is obvious that the resulting stack will be empty too:

$$\textsc{Append}(\textsc{Empty}, \textsc{Empty}) \rightarrow \textsc{Empty}$$

In the second and third case, we append an empty stack and a non-empty one. The order in which we append them is irrelevant and the result is always the non-empty one:

$$\textsc{Append}(\textsc{Empty}, \textsc{Push}(e, s)) \rightarrow \textsc{Push}(e, s)$$
$$\textsc{Append}(\textsc{Push}(e, s), \textsc{Empty}) \rightarrow \textsc{Push}(e, s)$$

Stacks/Appending (cont)

The last case is the more difficult one: we append two non-empty stacks. The top item of the stack to be appended on top of the other is e_1. This item will also be on the top of the resulting stack. That is why we expect something like

$$\textsc{Append}(\textsc{Push}(e_1, s_1), \textsc{Push}(e_2, s_2)) \to \textsc{Push}(e_1, \dots)$$

Now, the stack below e_1 in the result can be computed by appending s_1 to the second stack, $\textsc{Push}(e_2, s_2)$:

$$\begin{aligned}&\textsc{Append}(\textsc{Push}(e_1, s_1), \textsc{Push}(e_2, s_2)) \to \\ &\qquad\qquad \textsc{Push}(e_1, \overline{\textsc{Append}(s_1, \textsc{Push}(e_2, s_2))})\end{aligned}$$

Stacks/Appending (cont)

By looking carefully to the rewrite rules, we can devise a simpler one.

Indeed, in the case of one of the stacks is empty, the result will always be the other stack. Thus, we only need to check if the first or the second stack is empty:

$$\textsc{Append}(\textsc{Empty}, s) \to s$$
$$\textsc{Append}(s, \textsc{Empty}) \to s$$

How can we check that this is correct?

Stacks/Appending (cont)

Remember that s can be replaced by any stack, so let us try with the two basic stacks:

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

Stacks/Appending (cont)

The third rule is redundant. If we remove it, we find the original system. Thus both systems are equivalent.

There is a different, though, but not with respect of the rewrites themselves (both systems are sound) but about the *strategy*: the original system was deterministic while the second is non-deterministic: in the case when the two stacks are empty, two rules are possible.

Stacks/Appending (cont)

We can simplify further. The last rule shows that during the rewrite, the sub-expression $\textsc{Push}(e_2, s_2)$ is invariant, i.e., that it is simply copied, without none of its sub-expressions (e_2 and s_2) being used some place else in the result.
This shows that the last rule does not care whether the second stack, i.e., the one that will be below the first one, is empty or not: it is just copied. Therefore we can simply write

$$\textsc{Append}(\textsc{Push}(e_1, s_1), s) \rightarrow \textsc{Push}(e_1, \textsc{Append}(s_1, s))$$

We can rename s into s_2, to make it more regular:

$$\textsc{Append}(\textsc{Push}(e_1, s_1), s_2) \rightarrow \textsc{Push}(e_1, \textsc{Append}(s_1, s_2))$$

Stacks/Appending (cont)

Finally:

$$\begin{aligned}
\textsc{Append}(\textsc{Empty}, s) &\rightarrow s & \langle\textsc{Append}_1\rangle\\
\textsc{Append}(s, \textsc{Empty}) &\rightarrow s & \langle\textsc{Append}_2\rangle\\
\textsc{Append}(\textsc{Push}(e_1, s_1), s_2) &\rightarrow \textsc{Push}(e_1, \textsc{Append}(s_1, s_2)) & \langle\textsc{Append}_3\rangle
\end{aligned}$$

The new last rule adds to the non-determinism. Let $\sigma_1(s) = \textsc{Push}(e_1, s_1)$ and $\sigma_2(s_2) = \textsc{Empty}$:

$$\textsc{Append}(\underline{\textsc{Push}(e_1, s_1)}, \textsc{Empty}) \rightarrow \overline{\textsc{Push}(e_1, s_1)} \quad \sigma_1\langle\textsc{Append}_2\rangle$$

$$\textsc{Append}(\textsc{Push}(e_1, s_1), \underline{\textsc{Empty}}) \rightarrow$$
$$\textsc{Push}(e_1, \underline{\textsc{Append}(s_1, \overline{\textsc{Empty}})}) \sigma_2\langle\textsc{Append}_3\rangle$$

Stacks/Appending (cont)

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 $\textsc{Append}_2$, if the second stack is empty, we can conclude in one step.

If rule $\textsc{Append}_3$ is successively preferred, we have to traverse all the elements of the first stack before terminating.

Exercise. Implement in C the $\textsc{Append}$ function

1. using loops;
2. without loops.