Search algorithms

Algorithms are a constrained form of rewriting systems.

You may remember that, sometimes, several rewriting rules can be applied to the same term. This is a kind of **non-determinism**, i.e., the process requires an arbitrary choice. Like

$$\textsc{Not}(\textsc{And}(\underline{\textsc{Not}(\textsc{True})}, \textsc{Not}(\textsc{False}))) \to_1 \textsc{Not}(\textsc{And}(\underline{\textsc{False}}, \textsc{Not}(\textsc{False})))$$

$$\textsc{Not}(\textsc{And}(\textsc{False}, \underline{\textsc{Not}(\textsc{False})})) \to_2 \textsc{Not}(\textsc{And}(\textsc{False}, \underline{\textsc{True}}))$$

or

$$\textsc{Not}(\textsc{And}(\textsc{Not}(\textsc{True}), \underline{\textsc{Not}(\textsc{False})})) \to_2 \textsc{Not}(\textsc{And}(\textsc{Not}(\textsc{True}), \underline{\textsc{True}}))$$

$$\textsc{Not}(\textsc{And}(\underline{\textsc{Not}(\textsc{True})}, \textsc{True})) \to_1 \textsc{Not}(\textsc{And}(\underline{\textsc{False}}, \textsc{True}))$$

Search algorithms (cont)

It is possible to constrain the situation where the rules are applied. This is called a **strategy**.

For instance, one common strategy, called **call by value**, consists in rewriting the arguments of a function call into values before rewriting the function call itself.

Some further constraints can impose an order on the rewritings of the arguments, like rewriting them from left to right or from right to left. Algorithms rely on rewriting systems with strategies, but use a different language, easier to read and write. The important thing is that algorithms can always be expressed in terms of rewriting systems, if we want.

Search algorithms (cont)

The language we introduce now for expressing algorithms is different from a programming language, in the sense that it is less detailed.

Since you already have a working knowledge of programming, you will understand the language itself through examples.

If we start from a rewriting system, the idea consists to gather all the rules that define the computation of a given function and create its algorithmic definition.

Search/Booleans

Let us start with a very simple function of the $\textsc{Bool}$ specification:

$$\textsc{Not}(\textsc{True}) \to \textsc{False}$$
$$\textsc{Not}(\textsc{False}) \to \textsc{True}$$

Let us write the corresponding algorithm in the following way:

$\textsc{Not}(b)$
    if $b = \textsc{True}$
        then result $\leftarrow \textsc{False}$
        else result $\leftarrow \textsc{True}$

Writing $x \leftarrow A$ means that we **assign** the value of expression A to the variable x. Then the value of x is the value of A. Keyword **result** is a special variable whose value becomes the result of the function when it finishes.

The variable b is called a **parameter**.

Search/Booleans (cont)

You may ask: "Since we are defining the booleans, what is the meaning of a conditional **if** ... **then** ... **else** ...?"

We assume built-in booleans **true** and **false** in our algorithmic language. So, the expression $b = \textsc{True}$ may have value **true** or **false**.

The $\textsc{Bool}$ specification is *not* the built-in booleans.

Expression $b = \textsc{True}$ is not $b = \textbf{true}$ or even b.

Search/Booleans (cont)

Let us take the $\textsc{Bool.And}$ function:

$$\begin{aligned}\textsc{And}(\textsc{True},\textsc{True}) &\rightarrow \textsc{True}\\ \textsc{And}(x,\textsc{False}) &\rightarrow \textsc{False}\\ \textsc{And}(\textsc{False},x) &\rightarrow \textsc{False}\end{aligned}$$

$$\begin{array}{l}\textsc{And}(b_1, b_2)\\ \quad \textbf{if } b_1 = \textsc{False} \textbf{ or } b_2 = \textsc{False}\\ \qquad \textbf{then result} \leftarrow \textsc{False}\\ \qquad \textbf{else}\ \ \textbf{result} \leftarrow \textsc{True}\end{array}$$

Because there is an *order* on the operations, we have been able to gather the three rules into one conditional. Note that **or** is **sequential**: if the first argument evaluates to $\textsc{True}$ the second argument is not computed (this can save time and memory). Hence this test is better than **if** $(s_1, s_2) = (\textsc{True}, \textsc{True}) \ldots$

Search/Booleans (cont)

The $\textsc{Or}$ function, as we defined it is easy to write as an algorithm:

$$\textsc{Or}(b_1, b_2) \rightarrow \textsc{Not}(\textsc{And}(\textsc{Not}(b_1), \textsc{Not}(b_2)))$$

becomes simply

$\textsc{Or}(b_1, b_2)$
    result $\leftarrow \textsc{Not}(\textsc{And}(\textsc{Not}(b_1), \textsc{Not}(b_2)))$

Remember that $\leftarrow$ in an algorithm is not a rewriting step but an assignment. This function is defined in terms of other functions ($\textsc{Not}$ and $\textsc{Or}$) which are called using an underlying **call-by-value strategy**, i.e., the **arguments** are computed first, then passed associated to the parameters in order to compute the **body** of the (called) function.

Search/Stacks

Let us consider again the stacks:
$\textsc{Pop}(\textsc{Push}(x)) \rightarrow x$ becomes

$\textsc{Pop}(s)$

if $s = \textsc{Empty}$
then error
else result $\leftarrow$ **???**

What is the problem here?

We want to define a projection (here $\textsc{Pop}$) without knowing the definition of the corresponding constructor ($\textsc{Push}$).

The reason why we do not define constructors with an algorithm is that we do not want to give too much details about the data structure, and so leave these details to the implementation (i.e., the program).

Because a projection is, by definition, the inverse of a constructor, we cannot define them explicitly with an algorithm.

Search/Stacks (cont)

With the example of this aborted algorithmic definition of projection POP, we realise that such definitions must be **complete**, i.e., they must handle all values satisfying the type of their arguments.

In the previous example, the type of the argument was STACK(node).t, so the case EMPTY had to be considered for parameter s.

Search/Stacks (cont)

In the rewriting rules, the erroneous cases are not represented because we don't want to give too much details at this stage. It is left to the algorithm to provide error detection and basic handling.

Note that in algorithms, we do not provide a sophisticated error handling: we just use a magic keyword **error**. This is because we leave for the program to detail what to do and maybe use some specific features of its language, like exceptions.

Search/Stacks (cont)

So let us consider the remaining function $\textsc{Append}$:

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

We gather all the rules into one function and choose a proper order:

$\textsc{Append}(s_3, s_2)$
if $s_3 = \textsc{Empty}$
 then result $\leftarrow s_2$ $\triangleright$ This is rule $\rightarrow_1$
 else $(e, s_1) \leftarrow \textsc{Pop}(s_3)$ $\triangleright$ This means $\textsc{Push}(e, s_1) = s_3$
  result $\leftarrow \textsc{Push}(e, \textsc{Append}(s_1, s_2))$ $\triangleright$ This is rule $\rightarrow_2$

Search/Queues

Let us come back to the $\textsc{Queue}$ specification:

$$\begin{aligned}\textsc{Dequeue}(\textsc{Enqueue}(e, \textsc{Empty})) &\rightarrow_1 (\textsc{Empty}, e)\\ \textsc{Dequeue}(\textsc{Enqueue}(e, q)) &\rightarrow_2 (\textsc{Enqueue}(e, q_1), e_1)\end{aligned}$$

where $q \neq \textsc{Empty}$ and where

$$\textsc{Dequeue}(q) \rightarrow_3 (q_1, e_1)$$

Search/Queues/Dequeuing

We can write the corresponding algorithmic function as

$\textsc{Dequeue}(q_2)$
if $q_2 = \textsc{Empty}$
then error
else $(e, q) \leftarrow \textsc{Enqueue}^{-1}(q_2)$
if $q = \textsc{Empty}$
then result $\leftarrow (q, e)$ $\qquad \triangleright$ Rule $\rightarrow_1$
else $(q_1, e_1) \leftarrow \textsc{Dequeue}(q)$ $\qquad \triangleright$ Rule $\rightarrow_3$
result $\leftarrow (\textsc{Enqueue}(e, q_1), e_1)$ $\triangleright$ Rrule $\rightarrow_2$

Termination is due to
$(e, q) = \textsc{Enqueue}^{-1}(q_2) \Rightarrow \mathcal{H}(q_2) = \mathcal{H}(q) + 1 > \mathcal{H}(q)$.

Search/Binary trees/Left prefix

Let us come back to the $\textsc{Bin-tree}$ specification and the left prefix traversal:

$$\textsc{Lpref}(\textsc{Empty}) \to \textsc{Stack}(\mathsf{node}).\textsc{Empty}$$
$$\textsc{Lpref}(\textsc{Make}(e, t_1, t_2)) \to \textsc{Push}(e, \textsc{Append}(\textsc{Lpref}(t_1), \textsc{Lpref}(t_2)))$$

We get the corresponding algorithmic function

$\textsc{Lpref}(t)$
if $t = \textsc{Empty}$
  then result $\leftarrow \textsc{Stack}(\mathsf{node}).\textsc{Empty}$
  else $(e, t_1, t_2) \leftarrow \textsc{Make}^{-1}(t)$
    result $\leftarrow \textsc{Push}(e, \textsc{Append}(\textsc{Lpref}(t_1), \textsc{Lpref}(t_2)))$

Search/Binary trees/Left infix

Similarly, we can consider again the left infix traversal:

$$\textsc{Linf}(\textsc{Empty}) \rightarrow \textsc{Stack}(\mathsf{node}).\textsc{Empty}$$
$$\textsc{Linf}(\textsc{Make}(e, t_1, t_2)) \rightarrow \textsc{Append}(\textsc{Linf}(t_1), \textsc{Push}(e, \textsc{Linf}(t_2)))$$

Hence

$\textsc{Linf}(t)$
if $t = \textsc{Empty}$
  then result $\leftarrow \textsc{Stack}(\mathsf{node}).\textsc{Empty}$
  else $(e, t_1, t_2) \leftarrow \textsc{Make}^{-1}(t)$
    result $\leftarrow \textsc{Append}(\textsc{Linf}(t_1), \textsc{Push}(e, \textsc{Linf}(t_2)))$

Search/Binary trees/Left postfix

Similarly, we can consider again the left postfix traversal:

$$\textsc{Lpost}(\textsc{Empty}) \rightarrow \textsc{Stack}(\mathsf{node}).\textsc{Empty}$$

$$\textsc{Lpost}(\textsc{Make}(e, t_1, t_2)) \rightarrow$$

$$\textsc{Append}(\textsc{Lpost}(t_1), \textsc{Append}(\textsc{Lpost}(t_2), \textsc{Push}(e, \textsc{Empty})))$$

$\textsc{Lpost}(t)$

if $t = \textsc{Empty}$
  then **result** $\leftarrow \textsc{Stack}(\mathsf{node}).\textsc{Empty}$
  else $(e, t_1, t_2) \leftarrow \textsc{Make}^{-1}(t)$
    $n \leftarrow \textsc{Append}(\textsc{Lpost}(t_2), \textsc{Push}(e, \textsc{Empty}))$
    result $\leftarrow \textsc{Append}(\textsc{Lpost}(t_1), n)$

Search/Binary trees/Breadth-first

Let us recall the rewrite system of $\textsc{Roots}$ (see page 66):

$$
\begin{aligned}
\textsc{Roots}(\textsc{Forest}(\mathsf{node}).\textsc{Empty}) &\rightarrow_1 \textsc{Stack}(\mathsf{node}).\textsc{Empty}\\
\textsc{Roots}(\textsc{Push}(\textsc{Empty}, f)) &\rightarrow_2 \textsc{Roots}(f)\\
\textsc{Roots}(\textsc{Push}(\textsc{Make}(r, t_1, t_2), f)) &\rightarrow_3 \textsc{Push}(r, \textsc{Roots}(f))
\end{aligned}
$$

Rule $\rightarrow_2$ skips any empty tree in the forest.

Search/Binary trees/Breadth-first (cont)

The corresponding algorithmic definition is

$\textsc{Roots}(f_1)$

if $f_1 = \textsc{Forest}(\text{node}).\textsc{Empty}$
  then result $\leftarrow \textsc{Stack}(\text{node}).\textsc{Empty}$ $\quad\triangleright$ Rule $\rightarrow_1$
  else $(t, f) \leftarrow \textsc{Pop}(f_1)$ $\quad\triangleright$ $\textsc{Push}(t, f) = f_1$
    if $t = \textsc{Empty}$
      then result $\leftarrow \textsc{Roots}(f)$ $\quad\triangleright$ Rule $\rightarrow_2$
      else **result** $\leftarrow \textsc{Push}(\textsc{Root}(t), \textsc{Roots}(f))$ $\quad\triangleright$Rule $\rightarrow_3$

Search/Binary trees/Breadth-first (cont)

Let us recall the rewrite system of $\textsc{Next}$ (see page 67):

$$
\begin{aligned}
\textsc{Next}(\textsc{Forest}(\mathsf{node}).\textsc{Empty}) &\rightarrow_1 \textsc{Forest}(\mathsf{node}).\textsc{Empty}\\
\textsc{Next}(\textsc{Push}(\textsc{Empty},\, f)) &\rightarrow_2 \textsc{Next}(f)\\
\textsc{Next}(\textsc{Push}(\textsc{Make}(r, t_1, t_2),\, f)) &\rightarrow_3 \textsc{Push}(t_1, \textsc{Push}(t_2, \textsc{Next}(f)))
\end{aligned}
$$

Search/Binary trees/Breadth-first (cont)

The corresponding algorithmic definition is

$\textsc{Next}(f_1)$

if $f_1 = \textsc{Forest}(\text{node}).\textsc{Empty}$
  then result $\leftarrow \textsc{Stack}(\text{node}).\textsc{Empty}$ ▷ This is rule $\rightarrow_1$
  else $(t, f) \leftarrow \textsc{Pop}(f_1)$ ▷ This means $\textsc{Push}(t, f) = f_1$
    if $t = \textsc{Empty}$
      then result $\leftarrow \textsc{Next}(f)$ ▷ This is rule $\rightarrow_2$
      else ▷ This is rule $\rightarrow_3$
        result $\leftarrow \textsc{Push}(\textsc{Left}(t), (\textsc{Push}(\textsc{Right}(t), \textsc{Next}(f))))$

Search/Binary trees/Breadth-first (cont)

Let us recall finally the rules for function $\mathcal{B}$:

$$\mathcal{B}(\textsc{Forest}(\mathsf{node}).\textsc{Empty}) \rightarrow \textsc{Stack}(\mathsf{node}).\textsc{Empty}$$
$$\mathcal{B}(f) \rightarrow \textsc{Append}(\textsc{Roots}(f), \mathcal{B}(\textsc{Next}(f)))$$

where $f \neq \textsc{Empty}$.

Search/Binary trees/Breadth-first (cont)

The corresponding algorithmic definition is

$\mathcal{B}(f)$

  if $f = \textsc{Empty}$
    then result $\leftarrow \textsc{Stack}(\text{node}).\textsc{Empty}$
    else result $\leftarrow \textsc{Append}(\textsc{Roots}(f), \mathcal{B}(\textsc{Next}(f)))$

And

$\text{BFS}(t)$

  result $\leftarrow \mathcal{B}(\textsc{Stack}(\text{node}).\textsc{Push}(t, \textsc{Empty}))$

Search/Binary trees/Breadth-first (cont)

Let us imagine we want to realise a breadth-first search on tree t *up to a given depth* d and return the encountered nodes. Let us reuse the name BFS to call such a function whose signature is then

$$\mathrm{BFS} : \textsc{Bin-Tree}(\mathsf{node}).\mathsf{t} \times \textbf{int} \rightarrow \textsc{Stack}(\mathsf{node}).\mathsf{t}$$

where int denotes the positive integers.

A possible defining equation can be:

$$\mathrm{BFS}_d(t) = \mathcal{B}_d(\textsc{Push}(t, \textsc{Empty})) \quad \text{with } d \geqslant 0$$

where $\mathcal{B}_d(f)$ is the stack of traversed nodes in the forest f up to depth $d \geqslant 0$ in a left-to-right breadth-first way.

Search/Binary trees/Breadth-first (cont)

Here are some possible equations defining $\mathcal{B}_d$:

$$
\begin{aligned}
\mathcal{B}_d(\textsc{Empty}) &= \textsc{Empty} \\
\mathcal{B}_0(f) &= \textsc{Roots}(f) \\
\mathcal{B}_d(f) &= \textsc{Append}(\textsc{Roots}(f), \mathcal{B}_{d-1}(\textsc{Next}(f))) \quad \text{if } d > 0
\end{aligned}
$$

The difference between $\mathcal{B}_d$ and $\mathcal{B}$ is the depth limit d.

Search/Binary trees/Breadth-first (cont)

In order to write the algorithm corresponding to $\mathcal{B}_d$, it is good practice not to use subscripts, like d, and use a regular parameter instead:

$\mathcal{B}(d, f)$

if $f = \text{Forest}(\text{node}).\text{Empty}$
  then result $\leftarrow \text{Stack}(\text{node}).\text{Empty}$
elseif $d = 0$
  then result $\leftarrow \text{Roots}(f)$
elseif $d > 0$
  then result $\leftarrow \text{Append}(\text{Roots}(f), \mathcal{B}(d-1, \text{Next}(f)))$
  else error

Search/Binary trees/Depth-first

We gave several algorithms for left-to-right depth-first traversals: prefix ($\textsc{Lpref}$), postfix ($\textsc{Lpost}$) and infix ($\textsc{Linf}$).

What if we want to limit the depth of such traversals, like $\textsc{Lpref}$?

$$\begin{aligned}
\textsc{Lpref}_{d}(\textsc{Empty}) &= \textsc{Empty}\\
\textsc{Lpref}_{0}(\textsc{Make}(e, t_1, t_2)) &= \textsc{Push}(e, \textsc{Empty})\\
\textsc{Lpref}_{d}(\textsc{Make}(e, t_1, t_2)) &= \textsc{Push}(e, \textsc{Append}(\textsc{Lpref}_{d-1}(t_1), \textsc{Lpref}_{d-1}(t_2)))
\end{aligned}$$

where $d > 0$.