Algebraic Specifications

Christian Rinderknecht

19 October 2008

Booleans/Signature

Let us start this section about search algorithms by presenting how to specify data structures without relying on a specific programming language. The method we introduce here is usually called **algebraic specification**.

Consider one of the simplest data structure you can imagine: the booleans. Let us call the following specification $\textsc{Bool}$. Here is its **signature**:

- **Defined types**
  - The type of the booleans is t.
- **Constructors**
  - $\textsc{True} : \mathsf{t}$
  - $\textsc{False} : \mathsf{t}$

Booleans/Signature (continued)

$\textsc{True}$ and $\textsc{False}$ are called constructors because they allow the construction of **values** of type t, i.e. booleans. This is why we write ": t" after these constructors.

For more excitement, let us add some usual **functions** whose arguments are booleans:

- $\textsc{Not} : \mathsf{t} \rightarrow \mathsf{t}$
  Expression $\textsc{Not}(b)$ is the negation of b;
- $\textsc{And} : \mathsf{t} \times \mathsf{t} \rightarrow \mathsf{t}$
  Expression $\textsc{And}(b_1, b_2)$ is the conjunction of b_1 and b_2;
- $\textsc{Or} : \mathsf{t} \times \mathsf{t} \rightarrow \mathsf{t}$
  Expression $\textsc{Or}(b_1, b_2)$ is the disjunction of b_1 and b_2.

Booleans/Equations

We can give mode information about the previous functions by means of **equations** (or **axioms**). A possible set of equations matching the signature is

$$\begin{aligned}
\textsc{Not}(\textsc{True}) &= \textsc{False}\\
\textsc{Not}(\textsc{False}) &= \textsc{True}\\
\textsc{And}(\textsc{True}, \textsc{True}) &= \textsc{True}\\
\textsc{And}(\textsc{True}, \textsc{False}) &= \textsc{False}\\
\textsc{And}(\textsc{False}, \textsc{True}) &= \textsc{False}\\
\textsc{And}(\textsc{False}, \textsc{False}) &= \textsc{False}\\
\forall\, b_1, b_2 \quad \textsc{Or}(b_1, b_2) &= \textsc{Not}(\textsc{And}(\textsc{Not}(b_1), \textsc{Not}(b_2)))
\end{aligned}$$

The signature and the equations make a **specification**.

Booleans/Equations (cont)

We note something interesting in the last equation: it contains **variables**, here b_1 and b_2. These variables must be of type t because they are arguments of function $\textsc{Or}$, whose type, as given by the signature, is $\mathsf{t} \times \mathsf{t} \rightarrow \mathsf{t}$.

It is very important to notice also that these variables are bound to a **universal quantifier**, $\forall$, at the beginning of the equation. This means, in particular, that we can rename these variables because they are just names which are local to the equation. For instance

$$\forall\, u, v \quad \textsc{Or}(u, v) = \textsc{Not}(\textsc{And}(\textsc{Not}(u), \textsc{Not}(v)))$$

would be equivalent.

Booleans/Simplifying the equations

We can simplify these equations before going on.

First we can omit the quantifiers (but remember they are implicitly present).

Second we remark that it suffices that one argument of $\textsc{And}$ is $\textsc{False}$ to make the call equal to $\textsc{False}$. In other words

$$\begin{aligned}
\textsc{Not}(\textsc{True}) &= \textsc{False}\\
\textsc{Not}(\textsc{False}) &= \textsc{True}\\
\textsc{And}(\textsc{True}, \textsc{True}) &= \textsc{True}\\
\textsc{And}(x, \textsc{False}) &= \textsc{False}\\
\textsc{And}(\textsc{False}, x) &= \textsc{False}\\
\textsc{Or}(b_1, b_2) &= \textsc{Not}(\textsc{And}(\textsc{Not}(b_1), \textsc{Not}(b_2)))
\end{aligned}$$

Booleans/Simplifying the equations (cont)

In order to be 100% confident, we must check whether the set of new equations is **equivalent** to the first set.

$$\begin{cases} \textsc{And}(\textsc{True}, \textsc{False}) = \textsc{False} \\ \textsc{And}(\textsc{False}, \textsc{True}) = \textsc{False} \\ \textsc{And}(\textsc{False}, \textsc{False}) = \textsc{False} \end{cases} \overset{?}{\Longleftrightarrow} \begin{cases} \forall x \quad \textsc{And}(x, \textsc{False}) = \textsc{False} \\ \forall x \quad \textsc{And}(\textsc{False}, x) = \textsc{False} \end{cases}$$

The new system is equivalent to $\begin{cases} \textsc{And}(\textsc{True}, \textsc{False}) = \textsc{False} \\ \textsc{And}(\textsc{False}, \textsc{False}) = \textsc{False} \\ \textsc{And}(\textsc{False}, \textsc{True}) = \textsc{False} \\ \textsc{And}(\textsc{False}, \textsc{False}) = \textsc{False} \end{cases}$

So the answer is yes.

From specifications to algorithms

Now, how do we go from a specification to an **algorithm**?

By algorithm, we mean an **operational description** of the specification, i.e., a series of explicit computations that respect the equations and yield the expected result.

A specification describes abstract properties (using notations like $\forall$, (x, y), $\varnothing$, $\sum$ etc. and leaving out explicit data structure definitions) and the algorithm is a calculus schema (this introduces an abstract computer model, including time).

From specifications to algorithms (cont)

One says that an algorithm is **correct with respect to its specification** when all the results of the algorithm satisfy the specification, i.e., there is no contradiction when the results are substituted into the equations of the specification.

Correctness is always a relative concept (a property of the algorithm in regard to its specification).

From specifications to algorithms (cont)

So, an algorithm is more detailed than a specification. But how much more?

Contrary to algorithms, **programs** depend on programming languages, as we mentioned before. Thus algorithms can be considered as a useful step from specification to programs, as a refinement step.

From specifications to algorithms (cont)

Algorithms can be implemented using different programming languages (featuring objects, or pointers, or exceptions etc.). The more we want to be free of using our favorite programming language, the more the language for expressing algorithms should be abstract.

However, this language may assume an abstract model of the computer which can be quite different from the one assumed by the chosen programming language. On the contrary, sometimes they are the same (this is the case of Prolog).

From specifications to algorithms (cont)

Coming back to our question: how do we go from an algebraic specification (which, by definition, describes formally a concept) to an algorithm (which describes formally a computation)?

One way is to rely on the two kinds of equations we have, recursive and non-recursive. In mathematics, the integer sequence we give page 28 can be transformed into a functional definition, i.e., U_n is expressed only in terms of n, a and b, by forming $U_{n+1} - U_n = b$ and summing both sides: $\sum_{n=0}^{p} (U_{n+1} - U_n) = \sum_{n=0}^{p} b \Leftrightarrow U_{p+1} - U_0 = (p+1)b$ $\Leftrightarrow U_{p+1} - a = (p+1)b \Leftrightarrow U_p = a + pb \Leftrightarrow U_n = a + nb$.

But this is an *ad hoc* technique (e.g., we rely on properties of the integer numbers, as addition), which cannot be applied to our algebraic specifications.

From specifications to algorithms (cont)

The general approach consists in finding a **computation scheme** instead of relying on a **reasoning**, as we did for the sequence.

This is achieved by looking at the equations in the specification and **orienting them**.

This means that an equation is then considered as a **rewriting step**, from the left side to the right side, or the reverse way.

From specifications to algorithms (cont)

Assume we have an equation $A = B$, where A and B are expressions.

The way to pass from this equality to a computation is to orient the sides as a rewriting step, i.e., wherever we find an occurrence of the left side, we replace it by the right side.

For instance, if we set $A \to B$, then it means that wherever we find a A, we can write instead a B.

But, in theory, $A = B$ is equivalent to $A \to B$ and $B \to A$. But if we keep both (symmetric) rewriting steps, we get a **non-terminating rewriting system**, as demonstrated by the following infinite chain: $A \to B \to A \to B \to \dots$

From specifications to algorithms (cont)

A rewriting system terminates if it stops on a value.

If we have a chain $X_1 \to X_2 \to \cdots \to X_n$ and there is no way to rewrite X_n, i.e., X_n does not appear in any part of a left-hand side, then if X_n is a value, the chain is terminating.

We want a rewriting system that terminates on a **value**.

A value is made of constructors only, whilst an **expression** is made of constructors and other functions. For example,

- $\textsc{True}$ is a value;
- $\textsc{Or}(\textsc{True}, \textsc{Not}(\textsc{False}))$ is an expression.

From specifications to algorithms (cont)

Now, how can we be sure we do not lose some property by just allowing one orientation for each equality, as just keeping $A \leftarrow B$ when $A = B$?

This problem is a **completeness problem** and is very difficult to tackle,[1] therefore we will not discuss it here.

So, how should we orient our equations?

[1] Refer to the Knuth-Bendix completion semi-algorithm.

Booleans/Orienting the equations

Since we want to use an equation to compute the function it characterises, if one side of an equation contains an occurrence of the function call and the other not, we orient from the former side to the latter:

$$\begin{aligned}
\textsc{Not}(\textsc{True}) &\to_1 \textsc{False}\\
\textsc{Not}(\textsc{False}) &\to_2 \textsc{True}\\
\textsc{And}(\textsc{True},\textsc{True}) &\to_3 \textsc{True}\\
\textsc{And}(x,\textsc{False}) &\to_4 \textsc{False}\\
\textsc{And}(\textsc{False},x) &\to_5 \textsc{False}\\
\textsc{Or}(b_1,b_2) &\to_6 \textsc{Not}(\textsc{And}(\textsc{Not}(b_1),\textsc{Not}(b_2)))
\end{aligned}$$

Booleans/Orienting the equations (cont)

For example, here is a terminating chain for the expression $\textsc{Or}(\textsc{True}, \textsc{True})$:

$$
\begin{aligned}
\textsc{Or}(\textsc{True}, \underline{\textsc{And}(\textsc{True}, \textsc{False})}) &\to_4 \textsc{Or}(\textsc{True}, \underline{\textsc{False}}) \\
\underline{\textsc{Or}(\textsc{True}, \textsc{False})} &\to_6 \underline{\textsc{Not}(\textsc{And}(\textsc{Not}(\textsc{True}), \textsc{Not}(\textsc{False})))} \\
\textsc{Not}(\textsc{And}(\underline{\textsc{Not}(\textsc{True})}, \textsc{Not}(\textsc{False}))) &\to_1 \textsc{Not}(\textsc{And}(\underline{\textsc{False}}, \textsc{Not}(\textsc{False}))) \\
\textsc{Not}(\textsc{And}(\textsc{False}, \underline{\textsc{Not}(\textsc{False})})) &\to_2 \textsc{Not}(\textsc{And}(\textsc{False}, \underline{\textsc{True}})) \\
\textsc{Not}(\underline{\textsc{And}(\textsc{False}, \textsc{True})}) &\to_5 \textsc{Not}(\underline{\textsc{False}}) \\
\underline{\textsc{Not}(\textsc{False})} &\to_2 \underline{\textsc{True}}
\end{aligned}
$$

An important property of our system is that it allows different chains starting from the same expression but they all end on the same value, e.g., we could have applied $\to_2$ before $\to_1$.

Booleans/Orienting the equations (cont)

Here, we will always use a **strategy** which rewrites the arguments before the function calls.

For instance, given

$$\textsc{Or}(\textsc{And}(\textsc{True}, \textsc{True}), \textsc{False})$$

we will rewrite first

$$\textsc{And}(\textsc{True}, \textsc{True}) \rightarrow_3 \textsc{True}$$

and then

$$\textsc{Or}(\textsc{True}, \textsc{False}) \rightarrow_6 \textsc{True}$$

This strategy is named **call-by-value** because we rewrite the arguments into their values first before rewriting the function call.

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**
  - $\textsc{Empty} : \mathsf{t}$
    Expression $\textsc{Empty}$ represents the empty stack.
  - $\textsc{Push} : \mathsf{item} \times \mathsf{t} \to \mathsf{t}$
    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 $\times$ t $\rightarrow$ 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**
  - $\textsc{Pop} : \mathsf{t} \to \mathsf{item} \times \mathsf{t}$
    This projection is the inverse of constructor $\textsc{Push}$.
- **Other functions**
  - $\textsc{Append} : \mathsf{t} \times \mathsf{t} \to \mathsf{t}$
    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:

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

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:

$$\begin{aligned}
\textsc{Pop}(\textsc{Push}(x)) &\to x \\
\textsc{Append}(\textsc{Empty}, s) &\to_1 s \\
\textsc{Append}(s, \textsc{Empty}) &\to_2 s \\
\textsc{Append}(\textsc{Push}(e, s_1), s_2) &= \textsc{Push}(e, \textsc{Append}(s_1, s_2))
\end{aligned}$$

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*.

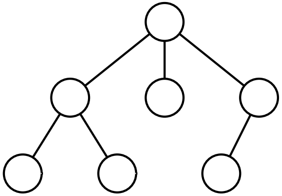
A node without non-empty subtrees is called a **leaf**.

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)

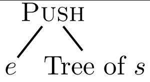
The **depth** of a node is the length of the path from the root to it (note this path is unique). Thus the depth of the root is 0 and the depth of the empty tree is undefined.

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)		$1 +$ 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:

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

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

Queues/Signature

There is another common and useful linear data structure call **queue**.

As the stack, it is fairly intuitive, since we experience the concept when we are waiting at some place to get some goods or service.

Let us call QUEUE(item) the specification of a queue over elements of the item type.

- **Parameter types**
    - The type item of the elements in the queue.
- **Defined types**
    - The type of the queue is t.

Queues/Constructors and other functions

- **Constructors**
  - $\textsc{Empty}$: t
    Expression $\textsc{Empty}$ represents the empty queue.
  - $\textsc{Enqueue}$: item $\times$ t $\rightarrow$ t
    Expression $\textsc{Enqueue}(e, q)$ denotes the queue q with element e added at the end.
- **Other functions**
  - $\textsc{Dequeue}$: t $\rightarrow$ t $\times$ item
    Expression $\textsc{Dequeue}(q)$ denotes the pair made of the *first* element of q and the remaining queue. The queue q must not be empty.

Queues/Equations

$$\begin{aligned}
\textsc{Dequeue}(\textsc{Enqueue}(e, \textsc{Empty})) &= (\textsc{Empty}, e) \\
\textsc{Dequeue}(\textsc{Enqueue}\ \mathbf{(}e\mathbf{,}\ q\mathbf{)}) &= (\textsc{Enqueue}(e, q_1), e') \\
\text{where} \quad (q_1, e') &= \textsc{Dequeue}(q) \\
\text{and} \quad q &\neq \textsc{Empty}
\end{aligned}$$

They are easy to orient since q is a proper subterm of $\textsc{Enqueue}(e, q)$:

$$\begin{aligned}
\textsc{Dequeue}(\textsc{Enqueue}(e, \textsc{Empty})) &\to (\textsc{Empty}, e) \\
\textsc{Dequeue}(\textsc{Enqueue}(e, q)) &\to (\textsc{Enqueue}(e, q_1), e')
\end{aligned}$$

where $q \neq \textsc{Empty}$ and where $\textsc{Dequeue}(q) \to (q_1, e')$.
Note that we can remove the condition by replacing q by $\textsc{Enqueue}(\dots)$.

Trees

At this point it is important to understand the two usages of the word *tree*.

We introduced a map between terms and trees, because this new point of view gives some insights (e.g. the $\mathcal{H}$ function). In this context, a tree is another **representation** of the subject (terms) we are studying, it is a tool.

But this section now presents trees as a **data structure** on its own. In this context, a tree is the subject of our study, they are given. This is why, when studying the stacks (the subject), we displayed them as trees (an intuitive graphics representation).

Tree traversals

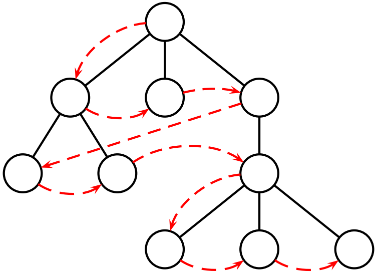
Given a tree, we can traverse it in many ways, but we must start from the root since we do not have any other node at this level. There are two main kind of traversals:

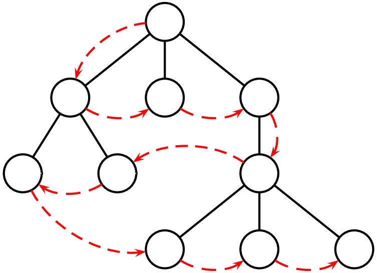
- **Breadth-first** traversal consists in walking the tree by increasing levels: first the root (level 0), the sons of the root (level 1), then the sons of the sons (level 2) etc.
- **Depth-first** traversal consists in walking the tree by reaching the leaves as soon as possible.

In both cases, we are finished when all the leaves have been encountered.

Tree traversals/Breadth-first

Let us consider two examples of breadth-first traversals.



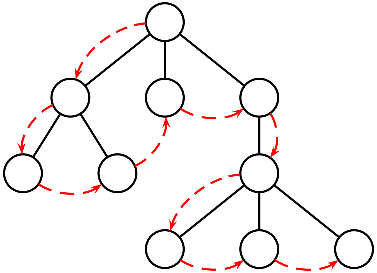


This is a **left to right** traversal.

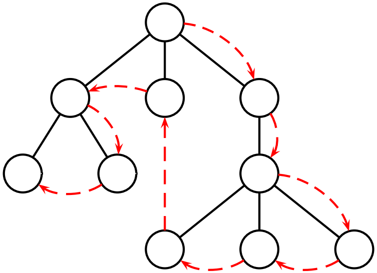
Many others are possible, like choosing randomly the next node of the following level.

Tree traversals/Depth-first

Let us consider two examples of depth-first traversals.



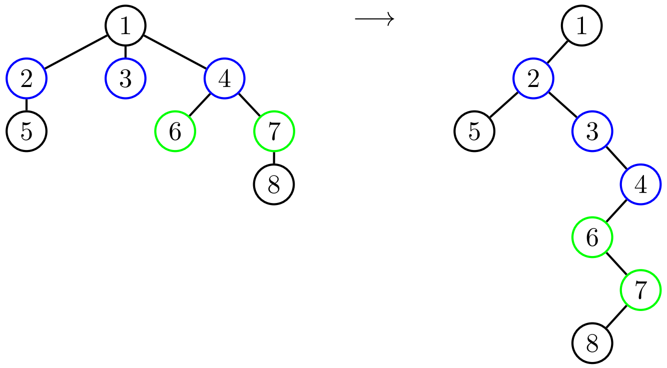
This is a **left to right** traversal.



This is a **right to left** traversal.

Binary trees

In order to simplify, let us consider here trees with two direct subtrees. They are called **binary trees**. We do not lose generality with this restriction: it is always possible to map any tree to a binary tree in a unique way: the **left son, right brother** technique:



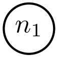
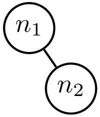
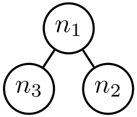
Binary trees/Signature

Let us formally define a a binary tree and call it $\textsc{Bin-tree}$(node), where node is the type of the nodes. Here is the **signature**:

- **Parameter types**
  - The type node of the nodes.
- **Defined types**
  - The type of the binary trees is t.
- **Constructors**
  - $\textsc{Empty} : \mathsf{t}$
    The constant $\textsc{Empty}$ is the empty tree (it is a constant function).
  - $\textsc{Make} : \mathsf{node} \times \mathsf{t} \times \mathsf{t} \rightarrow \mathsf{t}$
    The tree $\textsc{Make}(n, t_1, t_2)$ as root n and subtrees t_1 and t_2.

Binary trees/Signature (cont)

Let us show some examples of tree construction:

$\textsc{Empty}$	$\varnothing$
$\textsc{Make}(n_1, \textsc{Empty}, \textsc{Empty})$	
$\textsc{Make}(n_1, \textsc{Empty}, \textsc{Make}(n_2, \textsc{Empty}, \textsc{Empty}))$	
$\textsc{Make}(n_1, \textsc{Make}(n_3, \textsc{Empty}, \textsc{Empty}), \textsc{Make}(n_2, \textsc{Empty}, \textsc{Empty}))$	

Binary trees/Signature (cont)

The only projection for binary trees is the reverse function for $\textsc{Make}$:

$$\textsc{Make}^{-1} \circ \textsc{Make} = \mathit{id}$$

where *id* is the identity function. In other words

$$\forall\, n, t_1, t_2 \quad \textsc{Make}^{-1}(\textsc{Make}(n, t_1, t_2)) = (n, t_1, t_2)$$

We gave no name to the reverse of $\textsc{Make}$ because (n, t_1, t_2) does not correspond to a well-identified concept. Then let us choose more intuitive projections.

Binary trees/Signature (cont)

- **Projections**
  - $\textsc{Make}^{-1}$: $\mathsf{t} \to \mathsf{node} \times \mathsf{t} \times \mathsf{t}$
    This is the inverse function of constructor $\textsc{Make}$.
  - $\textsc{Root}$: $\mathsf{t} \to \mathsf{node}$
    Expression $\textsc{Root}(t)$ represents the root of tree t.
  - $\textsc{Left}$: $\mathsf{t} \to \mathsf{t}$
    Expression $\textsc{Left}(t)$ denotes the left subtree of tree t.
  - $\textsc{Right}$: $\mathsf{t} \to \mathsf{t}$
    Expression $\textsc{Right}(t)$ denotes the right subtree of tree t.

Binary trees/Equations

As we guessed with the case of $\textsc{Make}^{-1}$, we can complement our signature using **equations** (or **axioms**):

$$
\begin{aligned}
\forall\, n, t_1, t_2 &\quad \textsc{Root}(\textsc{Make}(n, t_1, t_2)) = n \\
\forall\, n, t_1, t_2 &\quad \textsc{Left}(\textsc{Make}(n, t_1, t_2)) = t_1 \\
\forall\, n, t_1, t_2 &\quad \textsc{Right}(\textsc{Make}(n, t_1, t_2)) = t_2
\end{aligned}
$$

The signature and the equations make a **specification**.

This abstract definition allows you to use the **programming language** you prefer to implement the specification $\textsc{Bin-tree}$(node), where the type node is a **parameter**

Binary trees/Equations (cont)

As a remark, let us show how $\textsc{Root}$, $\textsc{Left}$ and $\textsc{Right}$ can actually be defined by composing projections — hence they are projections themselves.

The only thing we need is the projections p_1, p_2 and p_3 on 3-tuples:

$$
\begin{aligned}
p_1(a, b, c) &= a\\
p_2(a, b, c) &= b\\
p_3(a, b, c) &= c
\end{aligned}
$$

Then it is obvious that we could have defined $\textsc{Root}$, $\textsc{Left}$ and $\textsc{Right}$ only with basic projections:

$$
\begin{aligned}
\textsc{Root} &= p_1 \circ \textsc{Make}^{-1}\\
\textsc{Left} &= p_2 \circ \textsc{Make}^{-1}\\
\textsc{Right} &= p_3 \circ \textsc{Make}^{-1}
\end{aligned}
$$

Let us define $\textsc{Fst}(x, y) = x$ and $\textsc{Snd}(x, y) = y$.

Binary trees/Equations (cont)

Note that our definition of BIN-TREE is incomplete on purpose: taking the root of an empty tree is undefined (i.e. there is no equation about this).

The reason is that we want the implementation, i.e. the program, to refine and handle such kind of situations.

For instance, if your programming language features **exceptions**, you may use them and raise an exception when taking the root of an empty tree. So, it is up to the programmer to make the required tests prior to call a partially defined function.

Binary trees/Equations (cont)

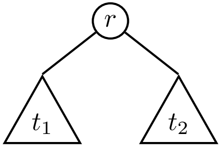
The careful reader may have noticed that some fundamental and necessary equations were missing:

- $\forall\, n, t_1, t_2 \quad \textsc{Make}(n, t_1, t_2) \neq \textsc{Empty}$
  This equation states that the **constructors** of trees (i.e. $\textsc{Empty}$ and $\textsc{Make}$) are unique.
- $\forall\, n, t_1, t_2, n', t'_1, t'_2 \quad \textsc{Make}(n, t_1, t_2) = \textsc{Make}(n', t'_1, t'_2) \Longrightarrow (n, t_1, t_2) = (n', t'_1, t'_2)$
  This equation states that the constructors with parameters (here $\textsc{Make}$) are **injective functions**.

These kind of equations (i.e. uniqueness and injection of constructors) are in fact always desirable, that is why they are usually assumed without explicit statement.

Binary trees/Left to right traversals

Consider a non-empty binary tree:



A depth-first traversal from left to right visits first node r, then the left subtree t_1 and finally the right subtree t_2. But if we want to keep track of the visited nodes, we have several ways.

- We can record r, then nodes of t_1 and finally nodes of t_2: this is **left prefix traversal**;
- we can record nodes of t_1, then r and nodes of t_2: this is a **left infix traversal**;
- we can record nodes of t_1, then nodes of t_2 and finally r: this is a **left postfix traversal**.

Binary trees/Left prefix traversal

Let us augment the specification $\textsc{Bin-tree}$(node) with a new function realising a **left prefix traversal**. In order to record the traversed nodes, we need an additional structure. Let us take a stack and call our traversal $\textsc{Lpref}$.

The additional signature is straightforward:

$$\textsc{Lpref} : \textsc{Bin-Tree}(\textsf{node}).\textsf{t} \rightarrow \textsc{Stack}(\textsf{node}).\textsf{t}$$

The corresponding equations are

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

where we omitted the prefixes "$\textsc{Bin-Tree}$(node)" and "$\textsc{Stack}$(node)".

Binary trees/Left prefix traversal (cont)

These equations must obviously be oriented from left to right:

$$\begin{aligned}\textsc{Lpref}(\textsc{Empty}) &\rightarrow \textsc{Empty}\\ \textsc{Lpref}(\textsc{Make}(e, t_1, t_2)) &\rightarrow \textsc{Push}(e, \textsc{Append}(\textsc{Lpref}(t_1), \textsc{Lpref}(t_2)))\end{aligned}$$

where we omitted the specification prefixes.

This is a left to right traversal if the evaluation strategy computes the value of arguments *from left to right*.

It is important to distinguish between the moment when a node is encountered and when it is added in the resulting stack. Therefore, if the arguments of LPREF are computed from right to left, the traversal is from right to left, but the nodes in the final stack will be ordered from left to right (as specified).

Binary trees/Left postfix traversal

The additional signature for left postfix traversal is:

$$\textsc{Lpost} : \textsc{Bin-Tree}(\mathsf{node}).\mathsf{t} \to \textsc{Stack}(\mathsf{node}).\mathsf{t}$$

The corresponding equations are

$$\begin{aligned}\textsc{Lpost}(\textsc{Empty}) &= \textsc{Empty}\\ \textsc{Lpost}(\textsc{Make}\ (e,\ t_1,\ t_2)) &= \\ \textsc{Append}(\textsc{Lpost}(t_1), \textsc{Append}(\textsc{Lpost}(t_2), \textsc{Push}(e, \textsc{Empty})))\end{aligned}$$

where we omitted the specification prefixes.

Binary trees/Left postfix traversal (cont)

We orient these equations from left to right:

$$\begin{aligned}\textsc{Lpost}(\textsc{Empty}) &\rightarrow \textsc{Empty}\\ \textsc{Lpost}(\textsc{Make}(e, t_1, t_2)) &\rightarrow\\ \textsc{Append}(\textsc{Lpost}(t_1), \textsc{Append}(\textsc{Lpost}(t_2), \textsc{Push}(e, \textsc{Empty})))\end{aligned}$$

where we omitted the specification prefixes.

Binary trees/Left infix traversal

The additional signature for left infix traversal is simply:

$$\textsc{Linf} : \textsc{Bin-Tree}(\mathsf{node}).\mathsf{t} \to \textsc{Stack}(\mathsf{node}).\mathsf{t}$$

The corresponding equations are

$$\begin{aligned}\textsc{Linf}(\textsc{Empty}) &= \textsc{Empty}\\ \textsc{Linf}(\textsc{Make}\ (e,\ t_1,\ t_2)) &= \textsc{Append}(\textsc{Linf}(t_1), \textsc{Push}(e, \textsc{Linf}(t_2)))\end{aligned}$$

where we omitted the specification prefixes.

Binary trees/Left infix traversal (cont)

We must orient these equations from left to right:

$$\begin{aligned}\textsc{Linf}(\textsc{Empty}) &\rightarrow \textsc{Empty}\\ \textsc{Linf}(\textsc{Make}(e, t_1, t_2)) &\rightarrow \textsc{Append}(\textsc{Linf}(t_1), \textsc{Push}(e, \textsc{Linf}(t_2)))\end{aligned}$$

where we omitted the specification prefixes.

Binary trees/Breadth-first traversal

Let us consider the pictures of page 46.

The idea it that we need to find at each level the nodes belonging to the next level, adding them to the previous nodes and repeat the search. So we need to handle at each level a **forest**, i.e. a set (or stack) of trees, not just nodes because nodes do not contain the subtrees (thus the next level nodes).

Therefore let us add first a function to the signature of $\textsc{Bin-tree}$(node):

$$\mathcal{B} : \textsc{Stack}(\textsc{Bin-tree}(\text{node}).\text{t}).\text{t} \rightarrow \textsc{Stack}(\text{node}).\text{t}$$

such that expression $\mathcal{B}(f)$ is a stack of the nodes of forest f traversed in a breadth-first way from left to right.

Let specify $\textsc{Forest}(\text{node}) = \textsc{Stack}(\textsc{Bin-tree}(\text{node}).\text{t})$

Binary trees/Breadth-first traversal (cont)

Then we define a function

$$\mathrm{BFS} : \textsc{Bin-tree}(\mathsf{node}).\mathsf{t} \rightarrow \textsc{Stack}(\mathsf{node}).\mathsf{t}$$

such that expression $\mathrm{BFS}(t)$ is a stack of the nodes of the tree t traversed in a breadth-first way from left to right. The corresponding equation is simply

$$\mathrm{BFS}(t) = \mathcal{B}(\textsc{Stack}(\mathsf{node}).\textsc{Push}(t, \textsc{Empty}))$$

Now, in order to define $\mathcal{B}(f)$ we need to get

- the roots of the forest f,
- the forest rooted at level 1 of the forest f.

Binary trees/Breadth-first traversal (cont)

Let $\textsc{Roots}$ have the signature

$$\textsc{Roots} : \textsc{Forest}(\mathsf{node}).\mathsf{t} \to \textsc{Stack}(\mathsf{node}).\mathsf{t}$$

The corresponding equations are not difficult to guess:

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

These equations must be oriented from left to right (do you see why?).

Binary trees/Breadth-first traversal (cont)

We have to define now a function which, given a forest, returns the forest at level 1, i.e. all the subtrees rooted at level 1 for each tree in the initial forest. Let us call it

$$\textsc{Next} : \textsc{Forest}(\mathsf{node}).\mathsf{t} \rightarrow \textsc{Forest}(\mathsf{node}).\mathsf{t}$$

The equations are

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

Note the similarities between $\textsc{Next}$ and $\textsc{Roots}$. Note also that t_1 and t_2 may be $\textsc{Empty}$, for the sake of simplicity.

Binary trees/Breadth-first traversal (cont)

Now we can write the equations of $\mathcal{B}$, using $\textsc{Roots}$ and $\textsc{Next}$:

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

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

Match the traversal as defined formally here against a figure page 46.

Let us orient these equations from left to right.

In order to show that the resulting rewriting system terminates, we have to compare the height of the *values* of f and $\textsc{Next}(f)$.

Binary trees/Breadth-first traversal (cont)

The intuition is that expression $\textsc{Next}(f)$ denotes a *larger* forest than f, because it is made of the direct subtrees of f (if they are all non-empty, then we get twice as more trees than in f), but the trees are strictly *smaller*.

Since we traverse the levels in increasing order, theses trees will finally be empty. But the second equation discards empty trees, so in the end, we really get an empty forest.

With this reasoning we show that the terminating orientation is

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

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

$$\textsc{Not}(\textsc{And}(\underline{\textsc{Not}(\textsc{True})}, \textsc{Not}(\textsc{False}))) \rightarrow_1 \textsc{Not}(\textsc{And}(\underline{\textsc{False}}, \textsc{Not}(\textsc{False})))$$
$$\textsc{Not}(\textsc{And}(\textsc{False}, \underline{\textsc{Not}(\textsc{False})})) \rightarrow_2 \textsc{Not}(\textsc{And}(\textsc{False}, \underline{\textsc{True}}))$$

or

$$\textsc{Not}(\textsc{And}(\textsc{Not}(\textsc{True}), \underline{\textsc{Not}(\textsc{False})})) \rightarrow_2 \textsc{Not}(\textsc{And}(\textsc{Not}(\textsc{True}), \underline{\textsc{True}}))$$
$$\textsc{Not}(\textsc{And}(\underline{\textsc{Not}(\textsc{True})}, \textsc{True})) \rightarrow_1 \textsc{Not}(\textsc{And}(\underline{\textsc{False}}, \textsc{True}))$$

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 $b =$ **true** or even b.

Search/Booleans (cont)

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

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

```
$\textsc{And}(b_1, b_2)$
    if $b_1 = \textsc{False}$ or $b_2 = \textsc{False}$
        then result $\leftarrow \textsc{False}$
        else  result $\leftarrow \textsc{True}$
```

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:
$\textsc{Pop}(\textsc{Push}(x)) \to x$ 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 $\leftarrow (q, e)$                              $\triangleright$ Rule $\rightarrow_1$
            else $(q_1, e_1) \leftarrow \textsc{Dequeue}(q)$             $\triangleright$ 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:

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

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

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

```
$\textsc{Lpost}(t)$
    **if** $t = \textsc{Empty}$
        **then result** $\leftarrow \textsc{Stack}(\text{node}).\textsc{Empty}$
        **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

```
ROOTS($f_1$)
    if $f_1$ = FOREST(node).EMPTY
        then result ← STACK(node).EMPTY                          ▷ Rule $\rightarrow_1$
        else ($t$, $f$) ← POP($f_1$)                      ▷ PUSH($t$, $f$) = $f_1$
             if $t$ = EMPTY
                then result ← ROOTS($f$)                        ▷ Rule $\rightarrow_2$
                else result ← PUSH(ROOT($t$), ROOTS($f$))       ▷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 $\leftarrow \textsc{Stack}(\text{node}).\textsc{Empty}$        $\triangleright$ This is rule $\rightarrow_1$
        else $(t, f) \leftarrow \textsc{Pop}(f_1)$        $\triangleright$ This means $\textsc{Push}(t, f) = f_1$
            if $t = \textsc{Empty}$
                then result $\leftarrow \textsc{Next}(f)$        $\triangleright$ This is rule $\rightarrow_2$
                else        $\triangleright$ 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 $\mathcal{B}$:

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

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

```
$\mathrm{BFS}(t)$
    **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

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

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