Breadth-First Extraction: Lessons from a Small Exercice in Algorithm Certification

Dominique Larchey-Wendling
Université de Lorraine, CNRS, LORIA
Vandœuvre-lès-Nancy, France
dominique.larchey-wendling@loria.fr

Ralph Matthes
Institut de Recherche en Informatique de Toulouse (IRIT)
CNRS and University of Toulouse, France
Ralph.Matthes@irit.fr

Abstract

By using pointers, breadth-first algorithms are very easy to implement efficiently in imperative languages. Implementing them with the same bounds on execution time in purely functional style can be challenging, as explained in Okasaki's paper at ICFP 2000 that even restricts the problem to binary trees but considers numbering instead of just traversal. The solution by Okasaki is modular and factors out the problem of implementing queues (FIFOs) with worst-case constant time operations. We certify those FIFO-based breadth-first algorithms on binary trees by extracting them from fully specified Coq terms, given an axiomatic description of FIFOs. In addition, we axiomatically characterize the strict and total order on branches that captures the nature of breadth-first traversal and propose alternative characterizations of breadth-first traversal of forests. We also propose efficient certified implementations of FIFOs by extraction, either with pairs of lists (with amortized constant time operations) or triples of lazy lists (with worst-case constant time operations), thus getting from extraction certified breadth-first algorithms with the optimal bounds on execution time.

Keywords Breadth-first algorithms, queues in functional programming, correctness by extraction, Coq

1 Introduction

Breadth-first algorithms form an important class of algorithms with many applications. The distinguishing feature is that the recursive process tries to be "equitable" in the sense that all elements of the structure with "distance" k from the starting point are treated before those at distance $k+1$. In particular, with infinite structures, this ensures "fairness" in that all possible branches are eventually pursued to arbitrary depth, in other words, the recursion does not get "trapped" in an infinite branch.[1] This phenomenon that breadth-first algorithms avoid is different from a computation that gets "stuck": even when "trapped", there may be still steady "progress" in the sense of producing more and more units of output in finite time. In this paper, we will not specify or certify algorithms to work on infinite structures although we expect the lazy reading of our extracted programs (e. g., if we chose to extract towards the Haskell language) to work properly for infinite input as well and thus be fair—however without any guarantees from the extraction process. Anyway, as mentioned above, breadth-first algorithms impose a stronger, quantitative notion of "equity" to address the order of traversal of the structure.

When looking out for problems to solve with breadth-first algorithms, plain traversal of a given structure is the easiest task; to make this traversal "observable", one simply prints the visited node labels or collects them in a list, in the functional programming paradigm, as we will do in this paper (leaving filtering of search "hits" aside). However, in the interest of efficiency, it is important to use a first-in, first-out queue (FIFO) to organize the waiting sub-problems (see, e. g., Paulson's book [9] concerning the language ML). We are here concerned with functional languages, and for them, constant-time (in the worst case) implementations of the FIFO operations were a scientific challenge, solved very elegantly by Okasaki [7].

Breadth-first traversal is commonly used to identify all nodes that are reachable in a graph from a given start node (see, e. g., [2]), and this allows to create a tree that captures the subgraph of reachable nodes, the "breadth-first tree" (for a given start node). In Okasaki's landmark paper at ICFP 2000 [8], the problem was that of *breadth-first numbering* (that had been known before): the traversal task is further simplified to start with a (binary) tree at its root, but the new challenge is to rebuild in a functional programming language the binary tree, where the labels of the nodes have been replaced by the value n if the node has been visited as the n-th one in the traversal. The progress achieved by Okasaki consists in separating the concerns of implementing breadth-first numbering from an efficient implementation of FIFOs: his algorithm works for any given FIFO and inherits optimal bounds on execution time from the given FIFO implementation. Thus, breadth-first numbering can be solved as efficiently with FIFOs as traversal,[2] and Okasaki reports in his paper that quite some of his colleagues did not come up with a FIFO-based solution when asked to find any solution.

[1] Meaning that recursion would be pursued solely in that branch.

[2] Jones and Gibbons [4] solved a variant of the problem with a "hard-wired" FIFO implementation—the one we review in Section 7.1—and thus do not profit from the theoretically most pleasing FIFO implementation of [7].

CPP 2019, January 14–15, 2019, Cascais/Lisbon, Portugal
2019.

In the present paper, using the Coq proof assistant,[3] we formalize and solve the breadth-first traversal problem for finite binary trees with a rich specification in the sense of certified programming, i. e. when the output of an algorithm is a dependent pair (v, C_v) composed of a value v together with a proof C_v certifying that v satisfies some (partial correctness) property.[4] The key ingredient for its functional correctness are two equations already studied by Okasaki [8], but there as definitional device. Using the very same equations, we formalize and solve the breadth-first numbering problem, out of which the Coq extraction mechanism [6][5] can extract the same algorithm as Okasaki's (however, we extract code in the OCaml[6] variant of the ML language), but here with all guarantees concerning the non-functional property of termination and the functional correctness, i. e., the algorithm indeed provides such a breadth-first numbering of the input tree (which is its *total correctness*).

As did Okasaki (and Gibbons and Jones for their solution [4]), we motivate the solution by first considering the natural extension to the problem of traversing or numbering a finite list of trees, henceforth called a *forest*. The forests are subsequently replaced by an abstract FIFO structure, which is finally instantiated for several implementations, including the worst-case constant-time implementation following [7] that is based on three lazy lists (to be extracted from coinductive lists in Coq for which we establish as invariant that they are always finite).

The breadth-first numbering problem can be slightly generalized to *breadth-first reconstruction*: the labels of the output tree are not necessarily natural numbers but come from a list of any type of length equal to the number of labels of the input tree, and the n-th list element replaces the n-th node in the traversal (which is a minor variant of "breadth-first labelling" considered by Jones and Gibbons [4]). A slight advantage of this problem formulation is that a FIFO-based solution is possible by structural recursion on that list parameter while the other algorithms were obtained by recursion over a measure (this is not too surprising since the length of that list coincides with the previously used measure).

In Section 2, we give background material, mostly on list notations and recursion/induction based on a decreasing measure, with a detailed toy example on the method we use throughout to obtain certified algorithms. Section 3 concentrates on background material concerning breadth-first traversal (specification, naive algorithm, distinction from depth-first traversal), including original material on a characterization of the breadth-first order, while Section 4 revolves around the mathematics of breadth-first traversal of forests that motivates the FIFO-based breadth-first algorithms. In Section 5, we use an abstractly specified datatype of FIFOs and deal with the different problems mentioned above: traversal, numbering, reconstruction. In Section 6, we elaborate on a level-based approach to numbering that thus is in the spirit of the naive traversal algorithm. Section 7 reports on the instantiation of the FIFO-based algorithms to two particular efficient FIFO implementations. We briefly present the Coq vernacular files in Section 8 and Section 9 concludes.

The full Coq development, that also formalizes the theoretical results (the characterizations) in addition to the certification of the extracted algorithms in the OCaml language, is available on GitHub:

https://github.com/DmxLarchey/BFE

2 Preliminaries

We introduce some short notations to represent the language of constructive type theory used in the proof assistant Coq. Then we introduce a method to justify termination of fixpoints (or inductive proofs) using measures over numbers.

2.1 Notations for constructive type theory

The type of propositions is denoted `Prop` while the type (family) of types is denoted `Type`. We use the inductive types of Booleans ($b : \mathbb{B} := 0 \mid 1$), of natural numbers ($n : \mathbb{N} := 0 \mid \mathrm{S}\, n$), of lists ($l : \mathbb{L}\, X := [] \mid x :: l$ with $x : X$) over a type parameter X. When $l = x_1 :: \cdots :: x_n :: []$, we define $|l| := n$ as the length of l and we may write $l = [x_1; \ldots; x_n]$ as well. We use $+\!\!+$ for the concatenation of two lists (called the "append" function). We write $x \in_{\mathrm{l}} l$ when x occurs in the list l. The function $\mathrm{rev} : \forall X,\ \mathbb{L}\, X \to \mathbb{L}\, X$ implements list reversal.

For a (heterogeneous) binary relation $R : X \to Y \to \mathtt{Prop}$, we define the lifting of R over lists $\forall^2 R : \mathbb{L}\, X \to \mathbb{L}\, Y \to \mathtt{Prop}$ by the following inductive rules, corresponding to the Coq standard library `Forall2` predicate:

$$\frac{}{\forall^2 R\ []\ []} \qquad \frac{R\ x\ y \qquad \forall^2 R\ l\ m}{\forall^2 R\ (x :: l)\ (y :: m)}$$

Intuitively, when $l = [x_1; \ldots; x_n]$ and $m = [y_1; \ldots; y_p]$, the predicate $\forall^2 R\ l\ m$ means $n = p \wedge R\, x_1\, y_1 \wedge \cdots \wedge R\, x_n\, y_n$.

2.2 Recursion on measures

We describe how to implement definitions of terms by induction on a measure of e. g. two arguments. Let us consider two types $X, Y : \mathtt{Type}$ and a measure $[\![\cdot,\cdot]\!] : X \to Y \to \mathbb{N}$. We explain how to build terms or proofs of type $t : P\, x\, y$ by induction on the measure $[\![x, y]\!]$. Hence, to build the term t, we are allowed to use instances of the induction hypothesis:

$$IH : \forall\, x'\, y',\ [\![x', y']\!] < [\![x, y]\!] \to P\, x'\, y'$$

i. e., the types $P\, x'\, y'$ with lower x'/y' parameters are (recursively) inhabited. The measures we use here are limited to $\mathbb{N}$

[3] https://coq.inria.fr, we have been using the current version 8.8.2, and the authoritative reference on Coq is https://coq.inria.fr/distrib/current/refman/

[4] for Coq, this method of rich specifications has been very much advocated in the Coq'Art, the first book on Coq [1].

[5] The authors of the current version are Filliâtre and Letouzey, see https://coq.inria.fr/refman/addendum/extraction.html.

[6] http://ocaml.org

viewed as well-founded under the $<$ (strictly less) order. Any other type with a well-founded order would work as well.[7]

We prove the following theorem while carefully crafting the computational contents of the proof term, so that extraction yields an algorithm that is clean of spurious elements:

$$\texttt{Theorem measure\_double\_rect } (P : X \to Y \to \texttt{Type}) :\\ (\forall\, x\, y, (\forall\, x'\, y',\ [[x', y']] < [[x, y]] \to P\, x'\, y') \to P\, x\, y)\\ \to \forall\, x\, y,\ P\, x\, y.$$

It allows to build a term of dependent type $P\,x\,y$ by simultaneous induction on x and y using the decreasing measure $[[x, y]]$ to ensure termination. To ease the use of theorem measure_double_rect we define a *tactic notation* that is deployed this way:

$$\texttt{induction on } x\ y \texttt{ as } IH \texttt{ with measure } [[x, y]]$$

To be fully precise, we point out that the proof term of measure_double_rect is actually inlined in the induction-on tactic to ensure perfect extraction in the sense of the result being free of any artifacts.

2.3 Example: interleaving

As an example, we present the following implementations of the interleaving functions itl_1 and itl_2 of type $\mathbb{L}\,X \to \mathbb{L}\,X \to \mathbb{L}\,X$ where itl_1 is defined using the equations:

$$\text{itl}_1\ []\ m = m \qquad \text{itl}_1\ l\ [] = l$$
$$\text{itl}_1\ (x :: l)\ (y :: m) = x :: y :: \text{itl}_1\ l\ m$$

and itl_2 is defined by the equations

$$\text{itl}_2\ []\ m = m \qquad \text{itl}_2\ (x :: l)\ m = x :: \text{itl}_2\ m\ l$$

While itl_1 is trivial to define in Coq because its equations correspond to a structurally decreasing `Fixpoint`,[8] the term itl_2 is more complicated to define because, as l and m switch roles in its second equation, neither of these two arguments decreases structurally in the recursive call. Hence, it cannot be accepted *as is* as a `Fixpoint` in Coq.

To define and specify itl_2, we first implement a fully specified version of itl_2 as the functional equivalent of itl_1. We proceed with the following script, whose first line reads as: we want to define the list r together with a proof that r is equal to $\text{itl}_1\ l\ m$—a typical rich specification.

```
Definition itl2_full l m : {r : 𝕃 X | r = itl1 l m}.
Proof.
  induction on l m as loop with measure (|l| + |m|).
  revert loop; refine (match l with
    | nil  ↦ fun _    ↦ exist _ m 𝕆₁?
    | x :: l' ↦ fun loop ↦ let (r, Hr) := loop m l' 𝕆₂?
                          in exist _ (x :: r) 𝕆₃?
  end).
  ⋆ trivial.                                  (* proof of 𝕆₁? *)
  ⋆ simpl; omega.                             (* proof of 𝕆₂? *)
  ⋆ subst; rewrite itl1_eq; trivial. (* proof of 𝕆₃? *)
Defined.
```

It contains three proof obligations (PO)

$$\mathbb{O}_1^? \ //\ \ldots \vdash m = \text{itl}_1\ []\ m$$
$$\mathbb{O}_2^? \ //\ \ldots \vdash |m| + |l'| < |x :: l'| + |m|$$
$$\mathbb{O}_3^? \ //\ \ldots, H_r : r = \text{itl}_1\ m\ l' \vdash x :: r = \text{itl}_1\ (x :: l')\ m$$

proved with their respective short proofs scripts. The only difficulty is to first prove the following equation for itl_1:

```
Fact itl1_eq : ∀ x l m, itl1 (x :: l) m = x :: itl1 m l
```

and this is done by, e. g., nested structural induction on first l and then m. Notice that it could also be proved by measure induction using $|l| + |m|$. We remark that proof obligation $\mathbb{O}_2^?$ is of different nature than $\mathbb{O}_1^?$ and $\mathbb{O}_3^?$. Indeed, $\mathbb{O}_2^?$ is a *pre-condition*, in this case a *termination certificate* ensuring than the recursive call occurs on smaller inputs. On the other hand, $\mathbb{O}_1^?$ and $\mathbb{O}_3^?$ are *post-conditions* ensuring the functional correctness by type-checking (of the logical part of the rich specification). As a general rule in this paper, providing termination certificates will always be relatively easy because they reduce to proofs of strict inequations between arithmetic expressions, usually solved by the `omega` tactic.[9]

Considering POs $\mathbb{O}_2^?$ and $\mathbb{O}_3^?$, they contain the following hypothesis (hidden in the dots) which witnesses the induction on the measure $|l| + |m|$. It is called loop as indicated in the induction-on tactic used at the beginning of the script.

```
loop : ∀ l₀ m₀, |l₀|+|m₀| < |x :: l'|+|m| → {r | r = itl1 l₀ m₀}
```

It could also appear in $\mathbb{O}_1^?$ but since we do not need it to prove $\mathbb{O}_1^?$, we intentionally cleared it using `fun _ ↦ ...` Actually, loop is not used in the proofs of $\mathbb{O}_2^?$ or $\mathbb{O}_3^?$ either but it is necessary for the recursive call implemented in the `let ... := loop ... in ...` construct.

This peculiar way of writing terms as a combination of *programming style* and *combination of proof tactics* is possible

[7] If one wants to go beyond measures to implement a substantial fragment of general recursion, we invite the reader to consult [5].

[8] As a general rule in this paper, when equations can be straightforwardly implemented by structural induction on one of the parameters, we assume the corresponding Coq `Fixpoint` without further developments.

[9] Because termination certificates have a purely logical content, we do not care whether `omega` produces an "optimal" proof (b. t. w., it never does), but we appreciate its efficiency in solving those kinds of goals which we do not want to spend much time on, thus the gain is in time spent on developing the certified code. Efficiency of code execution is not touched since these proofs leave no traces in the extracted code.

in Coq by the use of the "swiss-army knife" tactic `refine`[10] that allows, via unification, to specify only parts of a term leaving holes to be filled later if unification fails to solve them. It is a major tool to finely control computational content while allowing great tactic flexibility on pure logical content.

We continue the definition of itl_2 as the first projection

```
Definition itl2 l m := proj1_sig (itl2_full l m).
```

and its specification using the second projection `proj2_sig` on the dependent pair `itl2_full` $l\ m$.

```
Fact itl2_spec l m : itl2 l m = itl1 l m.
```

Notice that by asking the extraction mechanism to inline the definition of `itl2_full`, we get the following extracted OCaml code for itl_2, the one that optimally reflects the equational specification:

```
let rec itl2 l m = match l with [] → m | x :: l → x :: itl2 m l
```

Of course, this outcome of the (automatic) code extraction had to be targeted when doing the proof of `itl2_full`: a trivial proof would have been to choose as r just $\mathsf{itl}_1\ l\ m$, and the extracted code would have been just that. Instead, we did pattern-matching on l and chose in the first case m and in the second case $x :: r$, with r obtained as first component of the recursive call loop $m\ l'$. The outer induction took care that loop stood for the function we were defining there.

Henceforth, we will take induction or recursion on measures for granted, assuming they correspond to the application of theorems like measure_double_rect via the use of the tactic `induction on ... as ... with measure ...`

There are other Coq tools that help at the interactive construction of recursive algorithms which are not structurally recursive but terminating according to some measure or well-founded order. The `Program Fixpoint` command (due to Sozeau [11] who further elaborated these ideas in the Equations package [12]) postpones proof obligations systematically: the user provides the computational content and the system generates POs. However, we feel that we lose some control over the generated POs. Though it is not the case with the above simple itl_2 example, in more complicated cases as we experience in this paper, POs might depend other POs introduced earlier in the proof script, possibly generating existential variables in later POs. In that case, as they postpone all the POs they cannot automatically solve, these packages might generate POs which are really hard to interpret for the user, as (s)he might not be able to identify where they come from and how they depend on each other.

With the `refine` tactic, we allow ourselves a much greater control and the possibility to interleave the proof of POs and computational content (CC). Using e. g. the `cycle` tactic, we can even reorder POs to favor the CC branch of the script.

[10]The `refine` tactic was originally implemented by J.-C. Filliâtre.

3 Traversal of binary trees

We present mostly standard material on traversal of binary trees that will lay the ground for the original contributions in the later sections of the paper. After defining binary trees and basic notions for them including their branches (Section 3.1), we insist on describing mathematically what constitutes breadth-first traversal by considering an order on the branches (Section 3.2) that we characterize axiomatically (our Theorem 3.2). We then contrast depth-first (Section 3.3) and breadth-first (Section 3.4) traversal in their most elementary forms, by paying attention that they indeed meet their specification in terms of the order of the visited branches.

3.1 Binary trees

We use the type of binary trees $(a, b : \mathbb{T}\ X := \langle x\rangle \mid \langle a, x, b\rangle)$ where x is of parameter type X.[11] We define the root $: \mathbb{T}\ X \to X$ of a tree by

$$\mathsf{root}\ \langle x\rangle := x \quad \text{and} \quad \mathsf{root}\ \langle \_, x, \_\rangle := x$$

and the subtrees $\mathsf{subt} : \mathbb{T}\ X \to \mathbb{L}\ (\mathbb{T}\ X)$ of a tree by

$$\mathsf{subt}\ \langle x\rangle := [] \quad \text{and} \quad \mathsf{subt}\ \langle a, \_, b\rangle := [a; b]$$

The size $\|\cdot\| : \mathbb{T}\ X \to \mathbb{N}$ is defined by

$$\|\langle x\rangle\| := 1 \quad \text{and} \quad \|\langle a, x, b\rangle\| := 1 + \|a\| + \|b\|$$

and we extend the measure to lists of trees by

$$\|[t_1; \ldots; t_n]\| := \|t_1\| + \cdots + \|t_n\|$$

hence $\|[]\| = 0$, $\|[t]\| = \|t\|$ and $\|l +\!\!+ m\| = \|l\| + \|m\|$. We will not need more complicated measures than $\|\cdot\|$ to justify the termination of the breadth-first algorithms to come.

A branch in a binary tree is described as a list of Booleans in $\mathbb{L}\,\mathbb{B}$ representing a list of left/right choices (0 for left and 1 for right.) The predicate $\mathsf{btb} : \mathbb{T}\ X \to \mathbb{L}\,\mathbb{B} \to \mathtt{Prop}$ is defined inductively by the rules below where $t /\!\!/ l\downarrow$ denotes $\mathsf{btb}\ t\ l$ and tells whether *l is a branch in t:*

$$\frac{}{t /\!\!/ []\downarrow} \qquad \frac{a /\!\!/ l\downarrow}{\langle a, x, b\rangle /\!\!/ 0 :: l\downarrow} \qquad \frac{b /\!\!/ l\downarrow}{\langle a, x, b\rangle /\!\!/ 1 :: l\downarrow}$$

We define the predicate $\mathsf{bpn} : \mathbb{T}\ X \to \mathbb{L}\,\mathbb{B} \to X \to \mathtt{Prop}$ inductively as well. The term $\mathsf{bpn}\ t\ l\ x$, also denoted $t /\!\!/ l \rightsquigarrow x$, tells whether *the node identified by l in t is decorated with x:*

$$\frac{}{t /\!\!/ [] \rightsquigarrow \mathsf{root}\ t} \qquad \frac{a /\!\!/ l \rightsquigarrow r}{\langle a, x, b\rangle /\!\!/ 0 :: l \rightsquigarrow r} \qquad \frac{b /\!\!/ l \rightsquigarrow r}{\langle a, x, b\rangle /\!\!/ 1 :: l \rightsquigarrow r}$$

We show the result that l is a branch of t if and only if it is decorated in t:

```
Fact btb_spec (t : T X) (l : LB) : t // l↓ ↔ ∃x, t // l ⇝ x.
```

[11]In [8], Okasaki considered unlabeled leaves. When we compare with his findings, we always tacitly adapt his definitions to cope with leaf labels, which adds only a small notational overhead.

We define $\sim_{\mathbb{T}} : \mathbb{T}\,X \to \mathbb{T}\,Y \to \mathtt{Prop}$, *structural equivalence of binary trees*, by two inductive rules

$$\frac{}{\langle x\rangle \sim_{\mathbb{T}} \langle y\rangle} \qquad \frac{a \sim_{\mathbb{T}} a' \qquad b \sim_{\mathbb{T}} b'}{\langle a, x, b\rangle \sim_{\mathbb{T}} \langle a', y, b'\rangle}$$

and structural equivalence of lists of binary trees as

$$l \sim_{\mathbb{LT}} m := \forall^2(\sim_{\mathbb{T}})\ l\ m$$

i. e., when $l = [a_1; \ldots; a_n]$ and $m = [b_1; \ldots; b_p]$, the equivalence $l \sim_{\mathbb{LT}} m$ means $n = p \wedge a_1 \sim_{\mathbb{T}} b_1 \wedge \cdots \wedge a_n \sim_{\mathbb{T}} b_n$.

Both $\sim_{\mathbb{T}}$ and $\sim_{\mathbb{LT}}$ are equivalence relations, and if $a \sim_{\mathbb{T}} b$ then $\|a\| = \|b\|$. This also holds for structurally equivalent lists of trees.

3.2 Ordering branches of trees

A *strict order* $<_R$ is an irreflexive and transitive (binary) relation. It is *total* if $\forall\, x\, y,\ x <_R y \vee y <_R x$. It is *decidable* if $\forall\, x\, y,\ \{x <_R y\} + \{\neg(x <_R y)\}$, where, as usual in type theory, a proof of the sum $A + B$ requires either a proof of A or a proof of B, and the information which one has been chosen.

The *dictionary order* of type $\prec_{\mathrm{dic}} : \mathbb{L}\,\mathbb{B} \to \mathbb{L}\,\mathbb{B} \to \mathtt{Prop}$ is the lexicographic product on lists of 0's or 1's defined by

$$\frac{}{[] \prec_{\mathrm{dic}} b :: l} \qquad \frac{}{0 :: l \prec_{\mathrm{dic}} 1 :: m} \qquad \frac{l \prec_{\mathrm{dic}} m}{b :: l \prec_{\mathrm{dic}} b :: m}$$

The *breadth-first order* on $\mathbb{L}\,\mathbb{B}$ of type $\prec_{\mathrm{bf}} : \mathbb{L}\,\mathbb{B} \to \mathbb{L}\,\mathbb{B} \to \mathtt{Prop}$ is defined by

$$\frac{|l| < |m|}{l \prec_{\mathrm{bf}} m} \qquad \frac{|l| = |m| \qquad l \prec_{\mathrm{dic}} m}{l \prec_{\mathrm{bf}} m}$$

i. e., the lexicographic product of *shorter* and *dictionary order* (in case of equal length).

Lemma 3.1. *$\prec_{\mathrm{dic}}$ and $\prec_{\mathrm{bf}}$ are decidable total strict orders.*

We characterize $\prec_{\mathrm{bf}}$ with the following four axioms:

Theorem 3.2. *Let $<_R : \mathbb{L}\,\mathbb{B} \to \mathbb{L}\,\mathbb{B} \to \mathtt{Prop}$ be a relation s. t.*

(A_1) *$<_R$ is a strict order (irreflexive and transitive)*
(A_2) $\forall\, x\, l\, m,\ l <_R m \leftrightarrow x :: l <_R x :: m$
(A_3) $\forall\, l\, m,\ |l| < |m| \to l <_R m$
(A_4) $\forall\, l\, m,\ |l| = |m| \to 0 :: l <_R 1 :: m$

Then $<_R$ is equivalent to $\prec_{\mathrm{bf}}$, i. e., $\forall\, l\, m,\ l <_R m \leftrightarrow l \prec_{\mathrm{bf}} m$. Moreover the relation $\prec_{\mathrm{bf}}$ satisfies (A_1)–(A_4).

3.3 Depth-first traversal

We easily define the very standard *depth-first traversal* algorithm $\mathrm{dft}_{\mathrm{std}} : \mathbb{T}\,X \to \mathbb{L}\,X$ by structural induction using

$$\begin{aligned}\mathrm{dft}_{\mathrm{std}}\,\langle x\rangle &:= x :: []\\ \mathrm{dft}_{\mathrm{std}}\,\langle a, x, b\rangle &:= x :: \mathrm{dft}_{\mathrm{std}}\,a +\!\!+ \mathrm{dft}_{\mathrm{std}}\,b\end{aligned}$$

We refine the algorithm so that it outputs branches instead of decorations as $\mathrm{dft}_{\mathrm{br}} : \mathbb{T}\,X \to \mathbb{L}\,(\mathbb{L}\,\mathbb{B})$ by

$$\begin{aligned}\mathrm{dft}_{\mathrm{br}}\,\langle x\rangle &:= [] :: []\\ \mathrm{dft}_{\mathrm{br}}\,\langle a, x, b\rangle &:= [] :: \mathsf{map}\,(0 :: \cdot)\,(\mathrm{dft}_{\mathrm{br}}\,a)\\ &\quad +\!\!+ \mathsf{map}\,(1 :: \cdot)\,(\mathrm{dft}_{\mathrm{br}}\,b)\end{aligned}$$

We recover the correspondence between branches and values in the following result:

$\mathtt{Thm}\ \mathsf{dft\_br\_std}\ (t : \mathbb{T}\,X) : \forall^2(\mathrm{bpn}\,t)\ (\mathrm{dft}_{\mathrm{br}}\,t)\ (\mathrm{dft}_{\mathrm{std}}\,t).$

i. e., in the tree $t : \mathbb{T}\,X$, the branches listed in $\mathrm{dft}_{\mathrm{br}}\,t$ are in one-to-one correspondence with the values listed in $\mathrm{dft}_{\mathrm{std}}\,t$ according to the relation $\mathrm{bpn}\,t : \mathbb{L}\,\mathbb{B} \to X \to \mathtt{Prop}$. That proves that $\mathrm{dft}_{\mathrm{std}}$ and $\mathrm{dft}_{\mathrm{br}}$ traverse the tree t in the same order, except that $\mathrm{dft}_{\mathrm{std}}$ outputs decorating values while $\mathrm{dft}_{\mathrm{br}}$ outputs branches.[12] As a consequence, $\mathrm{dft}_{\mathrm{br}}$ enumerates the branches of t, i. e., $l \in_{\mathbb{l}} \mathrm{dft}_{\mathrm{br}}\,t \leftrightarrow t \mathbin{/\!\!/} l \downarrow$. We can also show that the branches in $\mathrm{dft}_{\mathrm{br}}\,t$ are in strict ascending order according to the dictionary order $\prec_{\mathrm{dic}}$.

Theorem 3.3. *The list $\mathrm{dft}_{\mathrm{br}}\,t$ is strictly sorted w. r. t. $\prec_{\mathrm{dic}}$.*

3.4 Breadth-first traversal, a naive approach

The *zipping* function $\mathsf{zip} : \mathbb{L}\,(\mathbb{L}\,X) \to \mathbb{L}\,(\mathbb{L}\,X) \to \mathbb{L}\,(\mathbb{L}\,X)$ is defined by

$$\mathsf{zip}\ []\ m := m \qquad \mathsf{zip}\ l\ [] := l$$

$$\mathsf{zip}\ (x :: l)\ (y :: m) := (x +\!\!+ y) :: \mathsf{zip}\ l\ m$$

The level-wise function $\mathsf{niv} : \mathbb{T}\,X \to \mathbb{L}\,(\mathbb{L}\,X)$ — niv refers to the French word "niveaux" — is defined by[13]

$$\begin{aligned}\mathsf{niv}\,\langle x\rangle &:= [x] :: []\\ \mathsf{niv}\,\langle a, x, b\rangle &:= [x] :: \mathsf{zip}\,(\mathsf{niv}\,a)\,(\mathsf{niv}\,b)\end{aligned}$$

The $(n + 1)$-th element of $\mathsf{niv}\,t$ contains the labels of t in left-to-right order that have distance n to the root. We can then define $\mathrm{bft}_{\mathrm{std}}\,t := \mathsf{concat}(\mathsf{niv}\,t)$.[14] As with depth-first traversal, we lift breadth-first traversal to branches instead of decorations, defining $\mathrm{niv}_{\mathrm{br}} : \mathbb{T}\,X \to \mathbb{L}\,(\mathbb{L}\,(\mathbb{L}\,\mathbb{B}))$ by

$$\begin{aligned}\mathrm{niv}_{\mathrm{br}}\,\langle x\rangle &:= [[]] :: []\\ \mathrm{niv}_{\mathrm{br}}\,\langle a, x, b\rangle &:= [[]] :: \mathsf{zip}\,\Big(\mathsf{map}\,(\mathsf{map}\,(0 :: \cdot))\,(\mathrm{niv}_{\mathrm{br}}\,a)\Big)\\ &\qquad\qquad \Big(\mathsf{map}\,(\mathsf{map}\,(1 :: \cdot))\,(\mathrm{niv}_{\mathrm{br}}\,b)\Big)\end{aligned}$$

and then $\mathrm{bft}_{\mathrm{br}}\,t := \mathsf{concat}(\mathrm{niv}_{\mathrm{br}}\,t)$, and we show:

$\mathtt{Thm}\ \mathsf{niveaux\_br\_niveaux}\ t : \forall^2\ (\forall^2\ (\mathrm{bpn}\,t))\ (\mathrm{niv}_{\mathrm{br}}\,t)\ (\mathsf{niv}\,t).$

and then

$\mathtt{Theorem}\ \mathsf{bft\_br\_std}\ t : \forall^2(\mathrm{bpn}\,t)\ (\mathrm{bft}_{\mathrm{br}}\,t)\ (\mathrm{bft}_{\mathrm{std}}\,t).$

i. e., $\mathrm{bft}_{\mathrm{br}}\,t$ and $\mathrm{bft}_{\mathrm{std}}\,t$ traverse the tree t in the same order, except that $\mathrm{bft}_{\mathrm{std}}$ outputs decorating values and $\mathrm{bft}_{\mathrm{br}}$ outputs branches (just as for depth-first traversal). We also show that $\mathrm{bft}_{\mathrm{br}}\,t$ lists the branches of t in $\prec_{\mathrm{bf}}$-ascending order.

[12] Notice that, in general, one cannot recover the branch from the decoration, e. g., when $X = \mathtt{unit}$, all branches have the same decoration.
[13] The function niv is called levelorder traversal in [4].
[14] where concat := fold $(\cdot +\!\!+ \cdot)$ [], i. e., concat $[l_1; \ldots; l_n] = l_1 +\!\!+ \cdots +\!\!+ l_n$.

Theorem 3.4. *The list* $\mathrm{bft}_{\mathrm{br}}\ t$ *is strictly sorted w. r. t.* $\prec_{\mathrm{bf}}$.

4 Breadth-first traversal of a forest

We lift root and subt to lists of trees and define $\mathrm{roots} : \mathbb{L}(\mathbb{T}\,X) \to \mathbb{L}\,X$ and $\mathrm{subtrees} : \mathbb{L}(\mathbb{T}\,X) \to \mathbb{L}(\mathbb{T}\,X)$ by

$$\mathsf{roots} := \mathsf{map\ root} \qquad \mathsf{subtrees} := \mathsf{flat\_map\ subt}$$

where flat_map is the standard list operation given by

$$\mathsf{flat\_map}\ f\ [x_1; \ldots; x_n] = f\ x_1 +\!\!+ \cdots +\!\!+ f\ x_n$$

To justify the upcoming fixpoints/inductive proofs where recursive sub-calls occur on subtrees l, we show the following

$$\mathtt{Lemma}\ \mathsf{subtrees\_dec}\ l :\ l = [] \vee \|\mathsf{subtrees}\ l\| < \|l\|.$$

Hence we can justify termination of a recursive algorithm $f\ l$ with formal argument $l : \mathbb{L}(\mathbb{T}\,X)$ that is calling itself on $f(\mathrm{subtrees}\ l)$, as soon as the case $f\ []$ is computed without using recursive calls (to f). For this, we for instance use recursion of the measure $l \mapsto \|l\|$ but we may also use the binary measure $l, m \mapsto \|l +\!\!+ m\|$.

4.1 Equations for breadth-first traversal

We first characterize $\mathrm{bft}_{\mathrm{f}}$ with four equivalent equations.

Theorem 4.1 (Characterization of breadth-first traversal of forests – recursive part)**.** *Let* $\mathrm{bft}_{\mathrm{f}}$ *be any term of type* $\mathbb{L}(\mathbb{T}\,X) \to \mathbb{L}\,X$ *and consider the following equations:*

(1) $\forall\, l,\ \mathrm{bft}_{\mathrm{f}}\ l = \mathrm{roots}\ l +\!\!+ \mathrm{bft}_{\mathrm{f}}\ (\mathrm{subtrees}\ l)$*;*
(2) $\forall\, l\ m,\ \mathrm{bft}_{\mathrm{f}}\ (l +\!\!+ m) = \mathrm{roots}\ l +\!\!+ \mathrm{bft}_{\mathrm{f}}\ (m +\!\!+ \mathrm{subtrees}\ l)$*;*
(3) $\forall\, t\ l,\ \mathrm{bft}_{\mathrm{f}}\ (t :: l) = \mathrm{root}\ t :: \mathrm{bft}_{\mathrm{f}}\ (l +\!\!+ \mathrm{subt}\ t)$*;*

(Oka1) $\forall\, x\ l,\ \mathrm{bft}_{\mathrm{f}}\ (\langle x \rangle :: l) = x :: \mathrm{bft}_{\mathrm{f}}\ l$*;*
(Oka2) $\forall\, a\ b\ x\ l,\ \mathrm{bft}_{\mathrm{f}}\ (\langle a, x, b \rangle :: l) = x :: \mathrm{bft}_{\mathrm{f}}\ (l +\!\!+ [a; b])$.

We have the equivalence: (1) ↔ (2) ↔ (3) ↔ (Oka1 ∧ Oka2).

Proof. Equations (1) and (3) are clear instances of Equation (2). Then (Oka1 ∧ Oka2) is equivalent to (3) because they just represent a case analysis on t. So the only difficulty is to show (1) → (2) and (3) → (2). Both inductive proofs alternate the roles of l and m. So proving (2) from e. g. (1) by induction on either l or m is not possible. Following the example of the interleaving algorithm of Section 2.3, we proceed by induction on the measure $\|l +\!\!+ m\|$. □

The equations (Oka1) and (Oka2) are used in [8] as defining equations, while (3) is calculated from the specification in [4]. We single out Equation (2) above is a smooth gateway between subtrees-based breadth-first algorithms and FIFO-based breadth-first algorithms. Unlocking that "latch bolt" enabled us to show correctness properties of refined breadth-first algorithms.

Theorem 4.2 (Full characterization of breadth-first traversal of forests)**.** *Adding equation* $\mathrm{bft}_{\mathrm{f}}\ [] = []$ *to any one of the equations of Theorem 4.1 determines the function* $\mathrm{bft}_{\mathrm{f}}$ *uniquely.*

Proof. For any bft_1 and bft_2 satisfying both $\mathrm{bft}_1\ [] = []$, $\mathrm{bft}_2\ [] = []$ and e. g. $\mathrm{bft}_1\ l = \mathrm{roots}\ l +\!\!+ \mathrm{bft}_1\ (\mathrm{subtrees}\ l)$ and $\mathrm{bft}_2\ l = \mathrm{roots}\ l +\!\!+ \mathrm{bft}_2\ (\mathrm{subtrees}\ l)$, we show $\mathrm{bft}_1\ l = \mathrm{bft}_2\ l$ by induction on $\|l\|$. □

Notice that one should not confuse the unicity of the function—which is an extensional notion—with the unicity of the algorithms implementing such a function, because those are hopefully numerous.

4.2 Direct implementation of breadth-first traversal

We define $\mathrm{forest}_{\mathrm{dec}} : \mathbb{L}(\mathbb{T}\,X) \to \mathbb{L}\,X \times \mathbb{L}(\mathbb{T}\,X)$ such that $\mathrm{forest}_{\mathrm{dec}}\ l = (\mathrm{roots}\ l, \mathrm{subtrees}\ l)$ but using a more efficient simultaneous computation:

$$\begin{aligned} \mathrm{forest}_{\mathrm{dec}}\ [] &:= [] \\ \mathrm{forest}_{\mathrm{dec}}\ (\langle x \rangle :: l) &:= (x :: \alpha, \beta) \\ \mathrm{forest}_{\mathrm{dec}}\ (\langle a, x, b \rangle :: l) &:= (x :: \alpha, a :: b :: \beta) \\ &\text{where } (\alpha, \beta) := \mathrm{forest}_{\mathrm{dec}}\ l. \end{aligned}$$

Then we show one way to realize the equations of Theorem 4.1 into a Coq term.

Theorem 4.3 (Existence of breadth-first traversal of forests)**.** *One can define a Coq term* $\mathrm{bft}_{\mathrm{f}}$ *of type* $\mathbb{L}(\mathbb{T}\,X) \to \mathbb{L}\,X$ *s.t.*

1. $\mathrm{bft}_{\mathrm{f}}\ [] = []$
2. $\forall l,\ \mathrm{bft}_{\mathrm{f}}\ l = \mathrm{roots}\ l +\!\!+ \mathrm{bft}_{\mathrm{f}}\ (\mathrm{subtrees}\ l)$

and which extracts to the following OCaml code:

```
let rec bft_f l = match l with
    | [] → []
    | _ → let α, β = forest_dec l in α ++ bft_f β
```

Proof. We define the functional graph $\rightsquigarrow_{\mathrm{bft}}$ as binary relation of type $\mathbb{L}(\mathbb{T}\,X) \to \mathbb{L}\,X \to \mathtt{Prop}$ with the two following inductive rules:

$$\frac{}{[] \rightsquigarrow_{\mathrm{bft}} []} \qquad \frac{l \neq [] \qquad \mathrm{subtrees}\ l \rightsquigarrow_{\mathrm{bft}} r}{l \rightsquigarrow_{\mathrm{bft}} \mathrm{roots}\ l +\!\!+ r}$$

These rules follow the intended algorithm. We show that the graph $\rightsquigarrow_{\mathrm{bft}}$ is functional, i. e.

$$\mathsf{bft\_f\_fun} : \forall\, l\ r_1\ r_2,\ l \rightsquigarrow_{\mathrm{bft}} r_1 \to l \rightsquigarrow_{\mathrm{bft}} r_2 \to r_1 = r_2.$$

By induction on the measure $\|l\|$ we define a term

$$\mathsf{bft\_f\_full}\ l : \{r \mid l \rightsquigarrow_{\mathrm{bft}} r\}$$

where we proceed as explained in Section 2.3. We get $\mathrm{bft}_{\mathrm{f}}$ by the first projection $\mathrm{bft}_{\mathrm{f}}\ l := \mathtt{proj1\_sig}(\mathsf{bft\_f\_full}\ l)$ and derive the specification

$$\mathsf{bft\_f\_spec} : \forall l,\ l \rightsquigarrow_{\mathrm{bft}} \mathrm{bft}_{\mathrm{f}}\ l$$

with the second projection proj2_sig. Equations 1 and 2 follow straightforwardly from bft_f_fun and bft_f_spec. □

Hence we see that we can use the specifying Equations 1 and 2 of Theorem 4.3 to define the term $\mathrm{bft}_{\mathrm{f}}$. In the case of breadth-first algorithms, termination is not very complicated because one can use induction on a measure to ensure it. In

the proof, we just need to check that recursive calls occur on smaller arguments according to the given measure, and this follows from Lemma subtrees_dec.

Theorem 4.4 (Correctness of breadth-first traversal of forests)**.** $\forall t : \mathbb{T}\ X,\ \mathsf{bft}_{\mathrm{std}}\ t = \mathsf{bft}_{\mathrm{f}}\ [t]$.

Proof. We define $\rightsquigarrow_{\mathrm{niv}} : \mathbb{L}\,(\mathbb{T}\ X) \to \mathbb{L}\,(\mathbb{L}\,X) \to \mathsf{Prop}$ by

$$\frac{}{[] \rightsquigarrow_{\mathrm{niv}} []} \qquad \frac{l \neq [] \qquad \mathsf{subtrees}\ l \rightsquigarrow_{\mathrm{niv}} ll}{l \rightsquigarrow_{\mathrm{niv}} \mathsf{roots}\ l :: ll}$$

and we show that

$$l \rightsquigarrow_{\mathrm{niv}} ll \to m \rightsquigarrow_{\mathrm{niv}} mm \to l +\!\!+ m \rightsquigarrow_{\mathrm{niv}} \mathsf{zip}\ ll\ mm$$

holds, a property from which we deduce $[t] \rightsquigarrow_{\mathrm{niv}} \mathsf{niv}\ t$. We show $l \rightsquigarrow_{\mathrm{niv}} ll \to l \rightsquigarrow_{\mathrm{bft}} \mathsf{concat}\ ll$ and we deduce $[t] \rightsquigarrow_{\mathrm{bft}} \mathsf{concat}\ (\mathsf{niv}\ t)$ hence $[t] \rightsquigarrow_{\mathrm{bft}} \mathsf{bft}_{\mathrm{std}}\ t$. By bft_f_spec we have $[t] \rightsquigarrow_{\mathrm{bft}} \mathsf{bft}_{\mathrm{f}}\ [t]$ and we conclude with bft_f_fun. □

4.3 Properties of breadth-first traversal

We show that on a given forest shape, breadth-first traversal is an injective map;

$$\mathsf{Lemma\ bft\_f\_inj}\ l\ m : l \sim_{\mathbb{LT}} m \to \mathsf{bft}_{\mathrm{f}}\ l = \mathsf{bft}_{\mathrm{f}}\ m \to l = m.$$

Proof. By induction on the measure $l, m \mapsto \|l +\!\!+ m\|$ and then case analysis on l and m. We use (Oka1&Oka2) from Theorem 4.1 to rewrite $\mathsf{bft}_{\mathrm{f}}$ terms. The shape constraint $l \sim_{\mathbb{LT}} m$ ensures that the same equation is used for l and m. □

Hence, on a given tree shape, $\mathsf{bft}_{\mathrm{std}}$ is also injective:

Corollary 4.5. *If $t_1 \sim_{\mathbb{T}} t_2$ are two trees of type $\mathbb{T}\ X$ (of the same shape) and $\mathsf{bft}_{\mathrm{std}}\ t_1 = \mathsf{bft}_{\mathrm{std}}\ t_2$ then $t_1 = t_2$.*

Proof. From Lemma bft_f_inj and Theorem 4.4. □

4.4 Discussion

The algorithm described in Theorem 4.3 can be used to compute breadth-first traversal as a replacement of the naive $\mathsf{bft}_{\mathrm{std}}$ algorithm. We could also use other equations of Theorem 4.1, for instance using $\mathsf{bft}_{\mathrm{f}}\ [] = []$, (Oka1) and (Oka2) to define another algorithm. The problem with equation

$$\text{(Oka2)} \quad \mathsf{bft}_{\mathrm{f}}\ (\langle a, x, b\rangle :: l) = x :: \mathsf{bft}_{\mathrm{f}}\ (l +\!\!+ [a; b])$$

is that it implies the use of $+\!\!+$ to append two elements at the tail of l, which is a well-known culprit that transforms an otherwise linear-time algorithm into a quadratic one. But equation (Oka2) hints at replacing the list data-structure with a first-in, first-out queue (FIFO) for the argument of $\mathsf{bft}_{\mathrm{f}}$ which brings us to the central section of this paper.

5 FIFO-based breadth-first algorithms

Here come the certified algorithms in the spirit of Okasaki's paper [8]. They have the potential to be efficient, but this depends on the later implementation of the here considered axiomatic datatype of FIFOs (Section 5.1). We deal with traversal, numbering, reconstruction—in breadth-first order.

$$\begin{aligned}
\mathsf{fifo} &: \mathsf{Type} \to \mathsf{Type}\\
\mathsf{f2l} &: \forall X,\ \mathsf{fifo}\ X \to \mathbb{L}\,X\\
\mathsf{emp} &: \forall X,\ \{q \mid \mathsf{f2l}\ q = []\}\\
\mathsf{enq} &: \forall X\ q\ x,\ \{q' \mid \mathsf{f2l}\ q' = \mathsf{f2l}\ q +\!\!+ [x]\}\\
\mathsf{deq} &: \forall X\ q,\ \mathsf{f2l}\ q \neq [] \to \{p \mid \mathsf{f2l}\ q = \pi_1\ p :: \mathsf{f2l}(\pi_2\ p)\}\\
\mathsf{void} &: \forall X\ q,\ \{b : \mathbb{B} \mid b = 1 \leftrightarrow \mathsf{f2l}\ q = []\}
\end{aligned}$$

Figure 1. An axiomatization of first-in first-out queues.

5.1 Axiomatization of FIFOs

In Fig. 1, we give an axiomatic description of polymorphic first-in, first-out queues (a. k. a. FIFOs) by projecting them to lists with $\mathsf{f2l}\ \{X\} : \mathsf{fifo}\,X \to \mathbb{L}\,X$ where the notation $\{X\}$ marks X as an implicit parameter of f2l. Each axiom is fully specified using f2l: emp is the empty queue, enq the queuing function, deq the dequeuing function which assumes a non-empty queue as input, and void a Boolean test of emptiness. Notice that when q is non-empty, deq q returns a pair (x, q') where x is the dequeued value and q' the remaining queue.

A clean way of introducing such an abstraction in Coq that generates little overhead for program extraction towards OCaml is the use of a *module type* that collects the data of Fig. 1. Coq developments based on a hypothetical implementation of the module type are then organized as *functors* (i. e., modules depending on typed module parameters). Thus, all the Coq developments described in this section are such functors, and the extracted OCaml code is again a functor, now for the module system of OCaml, and with the module parameter that consists of the operations of Fig. 1 after stripping off the logical part, in other words, the parameter is nothing but a hypothetical implementation of a FIFO signature, viewed as a module type of OCaml.

Of course, the FIFO axioms have several realizations, including a trivial and inefficient one where f2l is the identity function (and the inefficieny comes from appending the new elements at the end of the list with enq). In Section 7, we realize these axioms with more efficient implementations following Okasaki's insights [7] that worst-case $O(1)$ FIFO operations are even possible in an elegant way with functional programming languages.

5.2 Breadth-first traversal

We use the equations which come from Theorem 4.3 in combination with Theorem 4.1.

$$\mathsf{bft}_{\mathrm{f}}\ [] = [] \qquad \mathsf{bft}_{\mathrm{f}}\ (\langle x\rangle :: l) = x :: \mathsf{bft}_{\mathrm{f}}\ l$$
$$\mathsf{bft}_{\mathrm{f}}\ (\langle a, x, b\rangle :: l) = x :: \mathsf{bft}_{\mathrm{f}}\ (l +\!\!+ [a; b])$$

They suggest the definition of an algorithm for breadth-first traversal where lists are replaced with queues (FIFOs) so that (linear-time) append at the end $(\ldots +\!\!+ [a; b])$ is turned into two primitive queue operations (enq (enq ... a) b). Hence, we implement FIFO-based breadth-first traversal.

Theorem 5.1. *There exists a fully specified Coq term*

$$\mathsf{bft\_fifo_f} : \forall q : \mathsf{fifo}\,(\mathbb{T}\,X), \{l : \mathbb{L}\,X \mid l = \mathsf{bft_f}\,(\mathsf{f2l}\,q)\}$$

which extracts to the following OCaml code:

```
let rec bft_fifo_f q =
  if void q then []
  else let t, q' = deq q
       in match t with
         | ⟨x⟩       → x :: bft_fifo_f q'
         | ⟨a, x, b⟩ → x :: bft_fifo_f (enq (enq q' a) b).
```

Proof. We proceed by induction on the measure $q \mapsto \|\mathsf{f2l}\ q\|$ following the method exposed in the toy interleave example of Section 2.3. The proof is structured around the computational content of the above OCaml code. Termination POs are easily solved by `omega`. Correctness post-conditions are proved using the above equations. □

Corollary 5.2. *There is a Coq term* $\mathsf{bft_{fifo}} : \mathbb{T}\,X \to \mathbb{L}\,X$ *such that* $\mathsf{bft_{fifo}}\ t = \mathsf{bft_{std}}\ t$ *holds for any* $t : \mathbb{T}\,X$. *Moreover,* $\mathsf{bft_{fifo}}$ *extracts to the following OCaml code:*

```
let bft_fifo t = bft_fifo_f (enq emp t).
```

Proof. Given a tree t, we instantiate $\mathsf{bft\_fifo_f}$ on the one element FIFO $(\mathsf{enq}\ \mathsf{emp}\ t)$ and thus derive the term

$$\mathsf{bft\_fifo\_full}\ (t : \mathbb{T}\,X) : \{l : \mathbb{L}\,X \mid l = \mathsf{bft_f}\ [t]\}.$$

The first projection $\mathsf{bft_{fifo}}\ t := \mathtt{proj1\_sig}\,(\mathsf{bft\_fifo\_full}\ t)$ gives us $\mathsf{bft_{fifo}}$ and we derive $\mathsf{bft_{fifo}}\ t = \mathsf{bft_{std}}\ t$ from the combination of the second projection $\mathsf{bft_{fifo}}\ t = \mathsf{bft_f}\ [t]$ with Theorem 4.4. □

5.3 Breadth-first numbering

Breadth-first numbering was the challenge proposed to the community by Okasaki and which led him to write [8]. It consists in redecorating a tree with numbers in breadth-first order. The difficulty was writing an efficient algorithm in purely functional style. We choose the easy way to specify the result of breadth-first numbering of a tree: the output of the algorithm should be a tree $t : \mathbb{T}\,\mathbb{N}$ preserving the input shape and of which the breadth-first traversal of $\mathsf{bft_{std}}\ t$ is of the form $[1; 2; 3 \ldots]$.

As usual with those breadth-first algorithms, we generalize the notions to lists of trees.

```
Definition is_bfn n l := ∃k, bft_f l = [n; ...; n + k[
```

Lemma 5.3. *Given a fixed shape, the breadth-first numbering of a forest is unique, i. e., for any* $n : \mathbb{N}$ *and any* $l, m : \mathbb{L}\,(\mathbb{T}\,\mathbb{N})$,

$$l \sim_{\mathbb{LT}} m \to \mathsf{is\_bfn}\ n\ l \to \mathsf{is\_bfn}\ n\ m \to l = m.$$

Proof. By Lemma bft_f_inj in Section 4.3. □

We give an equational characterization of breadth-first numbering of forests combined with list reversal.

Lemma 5.4. *Let* $\mathsf{bfn_f} : \mathbb{N} \to \mathbb{L}\,(\mathbb{T}\,X) \to \mathbb{L}\,(\mathbb{T}\,\mathbb{N})$ *be a term. Considering the following conditions:*

(E1) $\forall n,\ \mathsf{bfn_f}\ n\ [] = []$;
(E2) $\forall\, n\, x\, l,\ \mathsf{bfn_f}\ n\ (\langle x\rangle :: l) = \mathsf{bfn_f}\ (1+i)\ l +\!\!+ [\langle i\rangle]$
(E3) $\forall\, n\, a\, x\, b\, l,\ \exists\, a'\, b'\, l',$

$$\wedge \begin{cases} \mathsf{bfn_f}\ (1+n)\ (l +\!\!+ [a; b]) = b' :: a' :: l' \\ \mathsf{bfn_f}\ n\ (\langle a, x, b\rangle :: l) = l' +\!\!+ [\langle a', n, b'\rangle]; \end{cases}$$

(Bfn1) $l \sim_{\mathbb{LT}} \mathsf{rev}(\mathsf{bfn_f}\ l)$;
(Bfn2) $\mathsf{is\_bfn}\ n\ (\mathsf{rev}(\mathsf{bfn_f}\ l))$.

We have the equivalence: (E1 ∧ E2 ∧ E3) ↔ (Bfn1 ∧ Bnf2).

Proof. For the "only if" part, we essentially use Lemma 5.3. For the "if" part, we proceed by induction on the measure $i, l \mapsto \|l\|$ in combination with Theorem 4.3 and Theorem 4.1—equations (Oka1) and (Oka2). □

Although not explicitly written in Okasaki's paper [8], these equations hint at the use of FIFOs as a replacement for lists for both the input and output of $\mathsf{bfn_f}$. We define the specification bfn_fifo_f_spec corresponding to breadth-first numbering of FIFOs of trees

$$\mathtt{Def.}\ \mathsf{bfn\_fifo\_f\_spec}\ n\ q\ q' := \wedge \begin{cases} \mathsf{f2l}\ q \sim_{\mathbb{LT}} \mathsf{rev}\ (\mathsf{f2l}\ q') \\ \mathsf{is\_bfn}\ n\ (\mathsf{rev}\ (\mathsf{f2l}\ q')) \end{cases}$$

and we show the inhabitation of this specification.

Theorem 5.5. *There exists a fully specified Coq term*

$$\mathsf{bfn\_fifo_f} : \forall (n : \mathbb{N})\ (q : \mathsf{fifo}\ X),\ \{q' \mid \mathsf{bfn\_fifo\_f\_spec}\ n\ q\ q'\}$$

which extracts to the following OCaml code:[15]

```
let rec bfn_fifo_f n q =
  if void q then emp
  else let t, q0 = deq q in match t with
    | ⟨_⟩       → enq (bft_fifo_f (1 + n) q0) ⟨n⟩
    | ⟨a, _, b⟩ → let q1 = enq (enq q0 a) b in
                  let q2 = bfn_fifo_f (1 + n) q1 in
                  let a', q3 = deq q2 in
                  let b', q4 = deq q3
                  in enq q4 ⟨a', n, b'⟩.
```

Proof. To define $\mathsf{bfn\_fifo_f}$, we proceed by induction on the measure $n, q \mapsto \|\mathsf{f2l}\ q\|$ where the first argument does not participate in the measure. As in Section 2.3, we implement a proof script which mixes tactics and programming style using the `refine` tactic. We follow the computational content given by the above algorithm. As usual for these breadth-first algorithms, termination certificates are easy to solve thanks to the `omega` tactic. The only difficulties lie in the post-condition POs but these correspond to the (proofs of the) equations of Lemma 5.4. □

Corollary 5.6. *There is a Coq term* $\mathsf{bfn_{fifo}} : \mathbb{T}\,X \to \mathbb{T}\,\mathbb{N}$ *s. t. for any tree* $t : \mathbb{T}\,X$:

$$t \sim_{\mathbb{T}} \mathsf{bfn_{fifo}}\ t \quad \textit{and} \quad \mathsf{bft_{std}}\,(\mathsf{bfn_{fifo}}\ t) = [1; \ldots; \|t\|]$$

[15] The greyed `let` q_1 `= ... in` and `let` q_2 `= ... in` declarations are actually in-lined in the extracted code but we expand them here for readability.

and $\mathsf{bfn}_{\mathrm{fifo}}$ *extracts to the following OCaml code:*

```
let bfn_fifo t =
  let q1 = enq emp t in
  let q2 = bfn_fifo_f 1 q1 in
  let t', _ = deq q2 in t'.
```

Proof. Obvious consequence of Theorem 5.5 in conjunction with Theorem 4.4. □

5.4 Breadth-first reconstruction

Breadth-first reconstruction is a generalization of breadth-first numbering—see the introduction for its description. For simplicity (since all our structures are finite), we ask that the list of labels that serves as extra parameter has to be of the right length, i. e., has as many elements as there are labels in the input data-structure, while the "breadth-first labelling" function considered by Jones and Gibbons [4] just required it to be long enough so that the algorithm does not get stuck.

We define the specification of the breadth-first reconstruction of a FIFO q of trees in $\mathbb{T}\,X$ using a list $l : \mathbb{L}\,Y$ of labels

$$\mathtt{Def}\ \mathsf{bfr\_fifo\_f\_spec}\ q\ l\ q' := \wedge \begin{cases} \mathsf{f2l}\ q \sim_{\mathbb{LT}} \mathsf{rev}\ (\mathsf{f2l}\ q') \\ \mathsf{bft_f}\ (\mathsf{rev}\ (\mathsf{f2l}\ q')) = l \end{cases}$$

We can then define breadth-first reconstruction by structural induction on the list l of labels:

$$\mathtt{Fixpoint}\ \mathsf{bfr\_fifo_f}\ (q : \mathsf{fifo}\ (\mathbb{T}\,X))\ (l : \mathbb{L}\,Y)\ \{\mathtt{struct}\ l\} :$$
$$\|\mathsf{f2l}\ q\| = |l| \to \{q' \mid \mathsf{bfr\_fifo\_f\_spec}\ q\ l\ q'\}.$$

Notice the pre-condition $\|\mathsf{f2l}\ q\| = |l|$ stating that l contains as many labels as the total number of nodes in the FIFO q.[16] Since we use structural induction, there are no termination POs. There is however a pre-condition PO (easily proved) and post-condition POs are similar to those of the proof of Theorem 5.5. Extraction to OCaml outputs the following:

```
let rec bfr_fifo_f q = function
  | []    → emp
  | y :: l →
  let t, q0 = deq q in match t with
    | ⟨_⟩       → enq (bfr_fifo_f q0 l) ⟨y⟩
    | ⟨a, _, b⟩ → let q1 = enq (enq q0 a) b in
                  let q2 = bfr_fifo_f q1 l in
                  let a', q3 = deq q2 in
                  let b', q4 = deq q3
                  in enq q4 ⟨a', y, b'⟩.
```

Notice the similarity with the code of $\mathsf{bfn\_fifo_f}$ of Theorem 5.5.

Theorem 5.7. *There is a Coq term*

$$\mathsf{bfr}_{\mathrm{fifo}} : \forall\,(t : \mathbb{T}\,X)\ (l : \mathbb{L}\,Y),\ \|t\| = |l| \to \mathbb{T}\,Y$$

s. t. for any tree $t : \mathbb{T}\,X$, $l : \mathbb{L}\,Y$ *and* $H : \|t\| = |l|$ *we have:*

$$t \sim_{\mathbb{T}} \mathsf{bfr}_{\mathrm{fifo}}\ t\ l\ H \quad \textit{and} \quad \mathsf{bft}_{\mathrm{std}}\ (\mathsf{bfr}_{\mathrm{fifo}}\ t\ l\ H) = l.$$

[16] This condition could easily be weakened to $\|\mathsf{f2l}\ q\| \le |l|$ but in that case, the specification bfr_fifo_f_spec should be changed as well.

Moreover, $\mathsf{bfr}_{\mathrm{fifo}}$ *extracts to the following OCaml code:*

```
let bfr_fifo t l =
  let q1 = enq emp t in
  let q2 = bfr_fifo_f q1 l in
  let t', _ = deq q2 in t'.
```

Proof. Direct application of $\mathsf{bfr\_fifo_f}\ (\mathsf{enq}\ \mathsf{emp}\ t)\ l$. □

6 Numbering by levels

Okasaki reports in [8] on his colleagues' attempts to solve the breadth-first numbering problem and mentions that most of them were level-oriented, as is the original traversal function $\mathsf{bft}_{\mathrm{std}}$. In his Section 4, he describes the "cleanest" of all those solutions, and this small section is devoted to obtaining it by extraction (and thus with certification through the method followed in this paper).

We define $\mathsf{children_f} : \mathbb{L}\,(\mathbb{T}\,X) \to \mathbb{N} \times \mathbb{L}\,(\mathbb{T}\,X)$ such that $\mathsf{children_f}\ l = (|l|, \mathsf{subtrees}\ l)$ but using a more efficient simultaneous computation:

$$\begin{aligned} \mathsf{children_f}\ [] &:= (0, []) \\ \mathsf{children_f}\ (\langle\_\rangle :: l) &:= (1 + n, m) \\ \mathsf{children_f}\ (\langle a, \_, b\rangle :: l) &:= (1 + n, a :: b :: m) \\ &\quad \text{where } (n, m) := \mathsf{children_f}\ l. \end{aligned}$$

and $\mathsf{rebuild_f} : \mathbb{N} \to \mathbb{L}\,(\mathbb{T}\,X) \to \mathbb{L}\,(\mathbb{T}\,\mathbb{N}) \to \mathbb{L}\,(\mathbb{T}\,\mathbb{N})$

$$\begin{aligned} &\mathsf{rebuild_f}\ n\ []\ \_ := [] \\ &\mathsf{rebuild_f}\ n\ (\langle\_\rangle :: t_s)\ c_s := \langle n\rangle :: \mathsf{rebuild_f}\ (1+n)\ t_s\ c_s \\ &\mathsf{rebuild_f}\ n\ (\langle\_, \_, \_\rangle :: t_s)\ (a :: b :: c_s) := \\ &\qquad \langle a, n, b\rangle :: \mathsf{rebuild_f}\ (1+n)\ t_s\ c_s \\ &\text{and otherwise } \mathsf{rebuild_f}\ \_\ \_\ \_ := []. \end{aligned}$$

The algorithms $\mathsf{children_f}$ and $\mathsf{rebuild_f}$ are (nearly) those defined in [8, Figure 5] but that paper does not provide a specification for $\mathsf{rebuild_f}$ and thus cannot show the correctness result which follows.

Lemma 6.1. *The function* $\mathsf{rebuild_f}$ *satisfies the following specification: for any n,* t_s *and* c_s*, if both*

$$\mathsf{subtrees}\ t_s \sim_{\mathbb{LT}} c_s \quad \textit{and} \quad \mathsf{is\_bfn}\ (|t_s| + i)\ c_s$$

hold then

$$t_s \sim_{\mathbb{LT}} \mathsf{rebuild_f}\ n\ t_s\ c_s \quad \textit{and} \quad \mathsf{is\_bfn}\ i\ (\mathsf{rebuild_f}\ n\ t_s\ c_s).$$

Proof. First we show by structural induction on t_s that for any $n : \mathbb{N}$ and $t_s : \mathbb{L}\,(\mathbb{T}\,\mathbb{N})$,

if $\mathsf{roots}\ t_s = [n; n{+}1; \ldots]$ then $\mathsf{rebuild_f}\ n\ t_s\ (\mathsf{subtrees}\ t_s) = t_s$

Then we show that for any $n : \mathbb{N}$, $t_s : \mathbb{L}\,(\mathbb{T}\,X)$, $t'_s : \mathbb{L}\,(\mathbb{T}\,Y)$ and any $c_s : \mathbb{L}\,(\mathbb{T}\,\mathbb{N})$,

$$\text{if } t_s \sim_{\mathbb{LT}} t'_s \text{ then } \mathsf{rebuild_f}\ n\ t_s\ c_s = \mathsf{rebuild_f}\ n\ t'_s\ c_s$$

This second proof is by structural induction on proofs of the $t_s \sim_{\mathbb{LT}} t'_s$ predicate. The result follows using Lemma 5.3. □

Theorem 6.2. *There is a fully specified Coq term*

$\mathsf{bfn\_level_f} : \forall\,(n : \mathbb{N})\;(l : \mathbb{L}\,(\mathbb{T}\,x)), \{m \mid l \sim_{\mathbb{LT}} m \wedge \mathsf{is\_bfn}\,n\,m\}$

which extracts to the following OCaml code:

```
let rec bfn_level_f i t_s = match t_s with
  | [] → []
  | _ → let n, s_s = children_f t_s in
        let cs = bfn_level_f (n + i) s_s
        in rebuild_f i t_s c_s
```

Proof. By induction on the measure $i, t_s \mapsto \|t_s\|$. The non-trivial correctness PO is a consequence of Lemma 6.1. □

Corollary 6.3. *There is a Coq term* $\mathsf{bfn_{level}} : \mathbb{T}\,X \to \mathbb{T}\,\mathbb{N}$ *such that for any tree* $t : \mathbb{T}\,X$*:*

1. $t \sim_{\mathbb{T}} \mathsf{bfn_{level}}\;t$*;*
2. $\mathsf{bft_{std}}\;(\mathsf{bfn_{level}}\;t) = [1; \ldots; \|t\|]$

and $\mathsf{bfn_{level}}$ *extracts to the following OCaml code:*

```
let bfn_level t = match bfn_level_f 1 [t] with t' :: _ → t'.
```

Proof. Direct application of Theorems 4.4 and 6.2. □

7 Efficient functional FIFOs

We discuss the use of our breadth-first algorithms that are parameterized over the abstract FIFO datatype with the implementations of efficient and purely functional FIFOs. As described in Section 5.1, the Coq developments of Sections 5.2, 5.3 and 5.4 take the form of (Coq module) functors and their extracted code is structured as (OCaml module) functors. Using these functors means instantiating them with an implementation of the parameter, i. e., a module of the given module type. Formally, this is just application of the functor to the argument module. In our case, we implement Coq modules of the module type corresponding to Fig. 1 and then apply our (Coq module) functors to those modules. The extraction process yields the application of the extracted (OCaml module) functors to the extracted FIFO implementations. This implies a certification that the application is justified logically, i. e., that the FIFO implementation indeed satisfies the axioms of Fig. 1.

7.1 FIFOs based on two lists

It is an easy exercise to implement our abstract interface for FIFOs based on pairs (l, r) of lists, with list representation $\mathsf{f2l}\;(l, r) := l +\!\!+ \mathsf{rev}\;r$. The enq operation adds the new element to the *front* of r (seen as the tail of the second part), while the deq operation "prefers" to take the elements from the *front* of l, but if l is empty, then r has to be carried over to l, which requires reversal. It is well-known that this implementation still guarantees amortized constant-time operations if list reversal is done efficiently (in linear time). As before, we obtain the implementation by automatic code extraction from constructions with the rich specifications that use measure induction for deq.

We can then instantiate our algorithms to this specific implementation, while Jones and Gibbons [4] calculated a dedicated breadth-first traversal algorithm for this implementation from the specification.

We have the advantage of a more modular approach and tool support for the code generation (once the mathematical argument in form of the rich specification is formalized in Coq). Moreover, we can benefit from a theoretically yet more efficient and still elegant implementation of FIFOs, the one devised in [7], to be discussed next.

7.2 FIFOs based on three lazy lists

While amortized constant-time operations for FIFOs seem acceptable — although imperative programming languages can do better — Okasaki showed that also functional programming languages allow an elegant implementation of worst-case constant time FIFO operations [7].

The technique he describes relies on lazy evaluation. To access those data structures in terms extracted from Coq code, we use coinductive types, in particular finite or infinite streams (also called "colists"):

```
CoInductive 𝕊 X := ⟨⟩ : 𝕊 X | _ # _ : X → 𝕊 X → 𝕊 X.
```

However, this type is problematic just because of the infinite streams it contains: since our inductive arguments are based on measures, we cannot afford having such infinite streams occur in the course of execution of our algorithms. Hence we need to guard our lazy lists with a non-informative *finiteness predicate* which is erased by the extraction process.

```
Inductive lfin : 𝕊 X → Prop :=
  | lfin_nil  : lfin ⟨⟩
  | lfin_cons : ∀ x s, lfin s → lfin (x # s).
```

We can then define the type of (finite) lazy lists as:

```
Definition 𝕃_l X := {s : 𝕊 X | lfin s}.
```

Compared to regular lists $\mathbb{L}\,X$, for a lazy list $(s, Hs) : \mathbb{L}_l\,X$, on the one hand, we can also do pattern-matching on s but on the other hand, we cannot define `Fixpoints` by structural induction on s. We replace it with structural induction on the proof of the predicate $H_s : \mathsf{lfin}\;s$. Although a bit cumbersome, this allows to work with such lazy lists as if they were regular lists, and this practice is fully compatible with extraction because the guards of type $\mathsf{lfin}\;s : \mathtt{Prop}$, being purely logical, are erased at extraction time. Here is the extraction of the type $\mathbb{L}_l$ in OCaml:

```
type α llist   = α __llist Lazy.t
and  α __llist = Lnil | Lcons of α * α llist
```

Here, we see the outcome of the effort: the (automatic) extraction process instructs OCaml to use lazy lists instead of standard lists (a distinction that does not even exist in the lazy language Haskell).

Okasaki [7] found a simple way to implement simple FIFOs[17] efficiently using triples of lazy lists. By efficiently, he means where enqueue and dequeue operations take *constant time* (in the worst case). We follow the proposed implementation using our own lazy lists $\mathbb{L}_l\, X$. Of course, his proposed code, though of beautiful simplicity, does not provide the full specifications for correctness proof. Some important invariants are present though. The main difficulty we faced was giving a correct specification for his intermediate rotate and make functions.

We do not enter more into the details of this implementation which is completely orthogonal to breadth-first algorithms. We invite the reader to check that we do get the exact intended extraction of FIFOs as triples of lazy lists.

7.3 Some remarks about practical complexity

However, our experience with the extracted algorithms for breadth-first numbering in OCaml indicate for smaller (with size < 2k) randomly generated input trees that the FIFOs based on three lazy lists are responsible for a factor of approximately 5 in execution time in comparison with the 2-list-based implementation. For large trees (with size > 64k), garbage collection severely hampers the theoretic linear-time behaviour. A precise analysis is out of scope for this paper. Moreover, we are not aware of work on certified run-time behaviour for code that is obtained through the extraction mechanism of Coq.

8 Code correspondence

In this small section, we briefly describe the Coq vernacular files behind our paper that is hosted at https://github.com/DmxLarchey/BFE. Besides giving formal evidence for the more theoretical characterizations, it directly allows program extraction, see the README section on the given web page.

There are 26 Coq vernacular files, here presented in useful order:

- `list_utils.v`: One of the biggest files, all concerning list operations, list permutations, the lifting of relations to lists (Section 2.1) and segments of the natural numbers – auxiliary material with use at many places.
- `wf_utils.v`: The subtle tactics for measure recursion in one or two arguments with a $\mathbb{N}$-valued measure function (Section 2.2) – this is crucial for smooth extraction throughout the paper.
- `llist.v`: Some general material on coinductive lists, in particular proven finite ones (including append for those), but also the rotate operation of Okasaki [7], relevant in Section 7.2.
- `wf_example.v`: the example of itl_2 from which the desired code is extracted
- `zip.v`: zipping with a rich specification and relations with concatenation – just auxiliary material
- `sorted.v`: consequences of a list being sorted, in particular absence of duplicates in case of strict orders – auxiliary material for Section 3.2.
- `increase.v`: small auxiliary file for full specification of breadth-first traversal (Section 3.4)
- `bt.v`: The largest file in this library, describing binary trees (Section 3.1), their branches and orders on those (Section 3.2) in relation with depth-first and breadth-first traversal and structural relations on trees and forests (again Section 3.1).
- `fifo.v`: the module type for abstract FIFOs (Section 5.1).
- `fifo_triv.v`: The trivial implementation of FIFOs through lists, mentioned in Section 5.1.
- `fifo_2lists.v`: An efficient implementation that has amortized $O(1)$ operations, see [7], see Section 7.1.
- `fifo_3llists.v`: The much more complicated FIFO implementation that is slower but has worst-case $O(1)$ operations, invented by Okasaki [7]; see Section 7.2.
- `dft_std.v`: Depth-first traversal, see Section 3.3.
- `bft_std.v`: Breadth-first traversal naively with levels (specified with the traversal of branches in suitable order), presented in Section 3.4.
- `dft_bft_compare.v`: Only the result that the respective versions that compute branches compute the same branches, and their lists form a permutation.
- `bft_forest.v`: Breadth-first traversal for forests of trees, paying much attention to the recursive equations that can guide the definition and/or verification (Section 4.1).
- `bft_inj.v`: Structurally equal forests with the same outcome of breadth-first traversal are equal, shown in Section 4.3.
- `bft_fifo.v`: Breadth-first traversal given an abstract FIFO, described in Section 5.2.
- `bfn_spec_rev.v`: A characterization of breadth-first numbering, see Lemma 5.4.
- `bfn_fifo.v`: The certified analogue of Okasaki's algorithm for breadth-first numbering [8], in Section 5.3.
- `bfn_trivial.v`: Just the instance of the previous with the trivial implementation of FIFOs.
- `bfn_level.v`: A certified reconstruction of `bfnum` on page 134 (section 4 and Figure 5) of Okasaki's article [8]. For its full specification, we allow ourselves to use breadth-first numbering obtained in `bfn_trivial.v`.
- `bfr_fifo.v`: Breadth-first reconstruction, a slightly more general task (see next file) than breadth-first numbering, presented in Section 5.4.
- `bfr_bfn_fifo.v`: Shows the claim that breadth-first numbering is an instance of breadth-first reconstruction (although they have been obtained with different induction principles).
- `extraction.v`: This operates extraction on-the-fly.
- `benchmarks.v`: Extraction towards `.ml` files.

[17] He also found a simple solution for double-ended FIFOs.

9 Final remarks

This paper shows that, despite their simplicity, breadth-first algorithms for finite binary trees present an interesting case for algorithm certification, in particular when it comes to obtain certified versions of efficient implementations in functional programming languages, as those considered by Okasaki [8].

Contributions. For this, we used the automatic extraction mechanism of the Coq system, whence we call this "breadth-first extraction." Fine-tuning the proof constructions so that the extraction process could generate the desired code—without any need for subsequent code polishing—was an engineering challenge, and the format of our proof scripts based on a hand-crafted measure induction tactic (expressed in the Ltac language for Coq [3] that is fully usable as user of Coq—as opposed to Coq developers only) should be reusable for a wide range of algorithmic problems and thus allow their solution with formal certification by program extraction.

We also considered variations on the algorithms that are not optimized for efficiency but illustrate the design space and also motivate the FIFO-based solutions. And we used the Coq system as theorem prover to formally verify some more abstract properties in relation with our breadth-first algorithms. This comprises as original contributions an axiomatic characterization of relations on paths in binary trees to be the breadth-first order (Theorem 3.2) in which the paths are visited by the breadth-first traversal algorithm. (Section 3.4). This also includes the identification of four different but logically equivalent ways to express the recursive behaviour of breadth-first traversal on forests (Theorem 4.1) and an equational characterization of breadth-first numbering of forests (Lemma 5.4). Among the variations, we mention that breadth-first reconstruction (Section 5.4) is amenable to a proof by structural recursion on the list of labels that is used for the relabeling while all the other proofs needed induction w. r. t. measures.

Perspectives. As mentioned in the introduction, code extraction of our constructions towards lazy languages such as Haskell would yield algorithms that we expect to work properly on infinite trees (the forests and FIFOs would still contain only finitely many elements, but those could then be infinite). The breadth-first nature of the algorithms would ensure fairness (hinted at also in the introduction). However, our present method does not certify in any way that use outside the specified domain of application (in particular, the non-functional correctness criterion of productivity is not guaranteed). We would have to give coinductive specifications and corecursively create their proofs, which would be a major challenge in Coq (cf. the experience of the second author with coinductive rose trees in Coq [10] where the restrictive guardedness criterion of Coq had to be circumvented in particular for corecursive constructions).

As further related work, we mention a yet different linear-time breadth-first traversal algorithm by Jones and Gibbons [4] that, as the other algorithms of their paper, is calculated from the specification, hence falls under the "algebra of programming" paradigm. Our methods should apply for that algorithm, too. And there is also their breadth-first reconstruction algorithm that relies on lazy evaluation of streams—a version of it which is reduced to breadth-first numbering has been discussed by Okasaki [8] to relate his work to theirs. To obtain such kind of algorithms would be a major challenge for the research in certified program extraction.

Acknowledgments

The authors got financial support by the COST action CA15123 EUTYPES. The first author got financial support by the TICAMORE joint ANR-FWF project.

References

[1] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer Verlag.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. 1989. *Introduction to Algorithms.* The MIT Press and McGraw-Hill Book Company.

[3] David Delahaye. 2002. A Proof Dedicated Meta-Language. *Electr. Notes Theor. Comput. Sci.* 70, 2 (2002), 96–109. https://doi.org/10.1016/S1571-0661(04)80508-5

[4] Geraint Jones and Jeremy Gibbons. 1993. *Linear-time Breadth-first Tree Algorithms: An Exercise in the Arithmetic of Folds and Zips.* Technical Report No. 71. Dept of Computer Science, University of Auckland.

[5] Dominique Larchey-Wendling and Jean-François Monin. 2018. Simulating Induction-Recursion for Partial Algorithms. In *24th International Conference on Types for Proofs and Programs, TYPES 2018, Abstracts*, José Espírito Santo and Luís Pinto (Eds.). University of Minho, Braga, Portugal. http://www.loria.fr/~larchey/papers/TYPES_2018_paper_19.pdf

[6] Pierre Letouzey. 2002. A New Extraction for Coq. In *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Selected Papers (Lecture Notes in Computer Science)*, Herman Geuvers and Freek Wiedijk (Eds.), Vol. 2646. Springer, 200–219.

[7] Chris Okasaki. 1995. Simple and Efficient Purely Functional Queues and Deques. *J. Funct. Program.* 5, 4 (1995), 583–592.

[8] Chris Okasaki. 2000. Breadth-first numbering: lessons from a small exercise in algorithm design. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*., Martin Odersky and Philip Wadler (Eds.). ACM, 131–136.

[9] Lawrence C. Paulson. 1991. *ML for the working programmer.* Cambridge University Press.

[10] Celia Picard and Ralph Matthes. 2012. Permutations in Coinductive Graph Representation. In *Coalgebraic Methods in Computer Science - 11th International Workshop, CMCS 2012, Revised Selected Papers (Lecture Notes in Computer Science)*, Dirk Pattinson and Lutz Schröder (Eds.), Vol. 7399. Springer, 218–237.

[11] Matthieu Sozeau. 2006. Subset Coercions in Coq. In *Types for Proofs and Programs, International Workshop, TYPES 2006, Revised Selected Papers (Lecture Notes in Computer Science)*, Thorsten Altenkirch and Conor McBride (Eds.), Vol. 4502. Springer, 237–252.

[12] Matthieu Sozeau. 2010. Equations: A Dependent Pattern-Matching Compiler. In *Interactive Theorem Proving, First International Conference, ITP 2010. Proceedings (Lecture Notes in Computer Science)*, Matt Kaufmann and Lawrence C. Paulson (Eds.), Vol. 6172. Springer, 419–434.