Certification of Breadth-First Algorithms by Extraction

Dominique Larchey-Wendling, Ralph Matthes

Certification of Breadth-First Algorithms by Extraction

Dominique Larchey-Wendling[1(✉)] and Ralph Matthes[2]

[1] Université de Lorraine, CNRS, LORIA, Vandœuvre-lès-Nancy, France
dominique.larchey-wendling@loria.fr
[2] Institut de Recherche en Informatique de Toulouse (IRIT), CNRS and University of Toulouse, 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. Okasaki's solution 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 nodes in the graph with "distance" k from the starting point are treated before those at distance $k + 1$. In particular, with infinite (but finitely branching) structures, this ensures "fairness" in that all possible branches are eventually pursued to arbitrary depth, in other words, the

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

https://doi.org/10.1007/978-3-030-33636-3_3

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 choose 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" than just abstract fairness 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 (following the imperative programming style), or one 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 [15] on the ML language). Here, we are also 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 [13].

Breadth-first traversal is commonly [4] used to identify all nodes that are reachable in a graph from a given start node, and this allows creating 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 [14], the author proposes to revisit the problem of *breadth-first numbering*: the traversal task is further simplified to start at the root of a (binary) tree, but the new challenge is to rebuild the tree in a functional programming language, 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.

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

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

[2] Jones and Gibbons [7] solved a variant of the problem with a "hard-wired" FIFO implementation—the one we review in Sect. 7.1—and thus do not profit from the theoretically most pleasing FIFO implementation by Okasaki [13].

[3] https://coq.inria.fr, we have been using the current version 8.9.0, and the authoritative reference on Coq is https://coq.inria.fr/distrib/current/refman/.

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 [14], 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[5] mechanism [9] can extract the same algorithm as Okasaki's (however, we extract code in the OCaml[6] language), but here with all guarantees concerning the non-functional property of termination and the functional/partial correctness, both aspects together constituting its *total correctness*, i. e., the algorithm indeed provides such a breadth-first numbering of the input tree.

As did Okasaki (and Gibbons and Jones for their solution [7]), we motivate the solution by first considering the natural extension to the problem of traversing or numbering a *forest*, i. e., a finite list of trees. 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 Okasaki's paper [13] that is based on three lazy lists (to be extracted from coinductive lists in Coq for which we require as invariant that they are 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 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, i. e., a minor variant of "breadth-first labelling" considered by Jones and Gibbons [7]. A slight advantage of this problem formulation is that a FIFO-based solution is possible by structural recursion on that list argument 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 Sect. 2, we give background type theoretical material, mostly on list notation. Then we review existing tools for reasoning or defining terms by recursion/induction on a decreasing measure which combines more than one argument. We proceed to the short example of a simple interleaving algorithm that swaps its arguments when recurring on itself. In particular, we focus on how existing tools like `Program Fixpoint` or `Equations` behave in the context of extraction. Then we describe an alternative of our own—a tailored `induction-on` tactic—and we argue that it is more transparent to extraction than, e. g., `Equations`. Section 3 concentrates on background material concerning breadth-first traversal (specification, naive algorithm), including original material on a characterization of the breadth-first order, while Sect. 4 revolves around the mathematics of breadth-first traversal of forests that motivates the FIFO-based breadth-first algorithms. In Sect. 5, we use an abstractly specified datatype of FIFOs and deal with the different problems mentioned above: traversal,

[4] For Coq, this method of rich specifications has been very much advocated in the Coq'Art, the first book on Coq [3].

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

[6] http://ocaml.org.

numbering, reconstruction. In Sect. 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. Section 8 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

In Appendix A, we give a brief presentation of these Coq vernacular[7] files.

2 Preliminaries

We introduce some compact notations to represent the language of constructive type theory used in the proof assistant Coq. We describe the method called "certification by extraction" and illustrate how it challenges existing tools like `Program Fixpoint` or even `Equations` on the example of a simple interleaving algorithm. Then we introduce a tailored method to justify termination of fixpoints (or inductive proofs) using measures over, e. g., two arguments. This method is encapsulated into an `induction-on` tactic that is more transparent to extraction than the above-mentioned tools.

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 ($n : \mathbb{N} := 0 \mid \mathsf{S}\, n$), 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). The function $\mathsf{rev} : \forall X, \mathbb{L}\, X \to \mathbb{L}\, X$ implements list reversal and satisfies $\mathsf{rev}\ [] = []$ and $\mathsf{rev}(x :: l) = \mathsf{rev}\ l +\!\!+ x :: []$.

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:

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

Thus $\forall^2 R$ is the smallest predicate closed under the two above rules.[8] Intuitively, when $l = [x_1; \ldots; x_n]$ and $m = [y_1; \ldots; y_p]$, the predicate $\forall^2 R\ l\ m$ means $n = p \wedge R\ x_1\ y_1 \wedge \cdots \wedge R\ x_n\ y_n$.

[7] Vernacular is a Coq idiom for *syntactic sugar*, i. e., a human-friendly syntax for type theoretical notation—the "Gallina" language.

[8] This is implemented in Coq by two constructors for the two rules together with an induction (or elimination) principle that ensures its smallestness.

2.1 Certification by Extraction

Certification by extraction is a particular way of establishing the correctness of a *given implementation* of an algorithm. In this paper, algorithms are given as programs in the OCaml language.

Hence, let us consider an OCaml program t of type $X \to Y$. The way to certify such a program by extraction is to implement a Coq term

$$f_t : \forall x : X_t,\ \mathsf{pre}_t\ x \to \{y : Y_t \mid \mathsf{post}_t\ x\ y\}$$

where $\mathsf{pre}_t : X_t \to \texttt{Prop}$ is the *precondition* (i. e., the domain of use of the function) and $\mathsf{post}_t : X_t \to Y_t \to \texttt{Prop}$ is the (dependent) *postcondition* which (possibly) relates the input with the output. The precondition could be tighter than the actual domain of the program t and the postcondition may characterize some aspects of the functional behavior of t, up to its full correctness, i. e., when $\mathsf{pre}_t/\mathsf{post}_t$ satisfy the following: for any given x such that $\mathsf{pre}_t\ x$, the value $f_t\ x$ is the unique y such that $\mathsf{post}_t\ x\ y$ holds.

We consider that t is certified when the postcondition faithfully represents the intended behavior of t *and* when f_t extracts to t: the extracted term $\texttt{extract}(f_t)$ but also the extracted types $\texttt{extract}(X_t)$ and $\texttt{extract}(Y_t)$ have to match "exactly" their respectively given OCaml definitions, i. e., we want $t \equiv \texttt{extract}(f_t)$, $X \equiv \texttt{extract}(X_t)$ and $Y \equiv \texttt{extract}(Y_t)$. The above identity sign "$\equiv$" should be read as *syntactic identity*. This does not mean character by character equality between source codes but between the abstract syntax representations. Hence some slight differences are allowed, typically the name of bound variables which cannot always be controlled during extraction. Notice that $\mathsf{pre}_t\ x$ and $\mathsf{post}_t\ x\ y$ being of sort `Prop`, they carry only logical content and no computational content. Thus they are erased by extraction.

As a method towards certified development, extraction can also be used without having a particular output program in mind, in which case it becomes a tool for writing programs that are correct by construction. Of course, getting a clean output might also be a goal and thus, the ability to finely control the computational content is important. But when we proceed from an already written program t, this fine control becomes critical and this has a significant impact on the tools which we can use to implement f_t (see upcoming Sect. 2.3).

2.2 Verification, Certification and the Trusted Computing Base

"Certification" encompasses "verification" in the following way: it aims at producing a certificate that can be checked independently of the software which is used to do the certification—at least in theory. While verification implies trusting the verifying software, certification implies only trusting the software that is used to verify the certificate, hence in the case of Coq, possibly an alternative typechecker. Notice that one of the goals of the MetaCoq[9] project [1] is precisely to

[9] https://github.com/MetaCoq/metacoq.

produce a type-checker for (a significant fragment of) Coq, a type-checker which will itself be certified by Coq.

Extraction in Coq is very straightforward once the term has been fully specified and type-checked. Calling the command

```
Recursive Extraction some_coq_term
```

outputs the extracted OCaml program and this part is fully automatic, although it is very likely that the output is not of the intended shape on the first attempt. So there may be a back-and-forth process to fine-tune the computational content of Coq terms until their extraction is satisfactory. The method can scale to larger developments because of the principle of *compositionality*. Indeed, provided a development can be divided into manageable pieces, Coq contains the tools that help at composing small certified bricks into bigger ones. Verifying or certifying large monolithic projects is generally hard—whatever tools are involved—because guessing the proper invariants becomes humanly unfeasible.

Considering the Trusted Computing Base (TCB) of certified extraction in Coq, besides trusting a type-checker, it also requires trusting the extraction process. In his thesis [10], P. Letouzey gave a mathematical proof of correctness (w. r. t. syntactic and semantic desiderata) of the extraction principles that guide the currently implemented `Extraction` command. Arguably, there is a difference between principles and an actual implementation. The above-cited MetaCoq project also aims at producing a certified "extraction procedure to untyped lambda-calculus accomplishing the same as the Extraction plugin of Coq." Hence, for the moment, our work includes `Extraction` in its TCB but so do many other projects such as, e. g., the CompCert compiler.[10] Still concerning verifying Coq extraction in Coq, we also mention the Œuf[11] project [12], but there is work of similar nature also in the Isabelle community [6]. Notice that we expect no or little change in the resulting extracted OCaml programs (once MetaCoq or one of its competitors reaches its goal) since these must respect the computational content of Coq terms. As the principle of "certification by extraction" is to obtain programs which are correct by construction directly from Coq code, we consider certifying extraction itself to be largely orthogonal to the goal pursued here.

2.3 Extraction of Simple Interleaving with Existing Tools

This section explains some shortcomings of the standard tools that can be used to implement recursive schemes in Coq when the scheme is more complicated than just structural recursion. We specifically study the `Function`, `Program Fixpoint`, and `Equations` commands. After discussing their versatility, we focus on how they interact with the `Extraction` mechanism of Coq.

[10] http://compcert.inria.fr.
[11] http://oeuf.uwplse.org.

As a way to get a glimpse of the difficulty of the method, we begin with the case of the following simple interleaving function of type $\mathbb{L}\,X \to \mathbb{L}\,X \to \mathbb{L}\,X$ that merges two lists into one

$$[l_1; \ldots; l_n], [m_1; \ldots; m_n] \mapsto [l_1; m_1; l_2; m_2; \ldots; l_n; m_n]$$

by alternating the elements of both input lists. The algorithm we want to certify is the following one:

$$\texttt{let rec}\ \mathsf{itl}\ l\ m = \texttt{match}\ l\ \texttt{with}\ [] \to m \mid x :: l \to x :: \mathsf{itl}\ m\ l \tag{1}$$

where l and m switch roles in the second equation/match case for itl. Hence neither of these two arguments decreases structurally[12] in the recursive call and so, this definition cannot be used as such in a Coq `Fixpoint` definition. Notice that this algorithm has been identified as a challenge for program extraction in work by McCarthy *et al.* [11, p. 58], where they discuss an OCaml program of a function cinterleave that corresponds to our itl.

We insist on that specific algorithm—it is however trivial to define a function with the same functional behavior in Coq by the following equations:

$$\mathsf{itl}_{\mathrm{triv}}\ []\ m = m \qquad \mathsf{itl}_{\mathrm{triv}}\ l\ [] = l \qquad \mathsf{itl}_{\mathrm{triv}}\ (x :: l)\ (y :: m) = x :: y :: \mathsf{itl}_{\mathrm{triv}}\ l\ m$$

Indeed, these equations correspond to a *structurally decreasing* `Fixpoint` which is accepted nearly as is by Coq.[13] However, the algorithm we want to certify proceeds through the two following equations $\mathsf{itl}\ []\ m = m$ and $\mathsf{itl}\ (x :: l)\ m = x :: \mathsf{itl}\ m\ l$. While it is not difficult to show that $\mathsf{itl}_{\mathrm{triv}}$ satisfies this specification, in particular by showing

$$\texttt{Fact}\ \mathsf{itl\_triv\_fix\_1} : \forall\, x\, l\, m,\ \mathsf{itl}_{\mathrm{triv}}\,(x :: l)\, m = x :: \mathsf{itl}_{\mathrm{triv}}\, m\, l$$

by nested induction on l and then m, extraction of $\mathsf{itl}_{\mathrm{triv}}$, however, does not respect the expected code of itl, see Eq. (1).

While there is no structural decrease in Eq. (1), there is nevertheless an obvious decreasing measure in the recursive call of itl, i. e., $l, m \mapsto |l| + |m|$. We investigate several ways to proceed using that measure and discuss their respective advantages and drawbacks. The comments below are backed up by the file interleave.v corresponding to the following attempts. The use of Coq 8.9 is required for a recent version of the `Equations` package described below.

[12] Structural recursion is the built-in mechanism for recursion in Coq. It means that `Fixpoint`s are type-checked only when Coq can determine that at least one of the arguments of the defined fixpoint (e. g., the first, the second...) decreases structurally on each recursive call, i. e., it must be a strict sub-term in an inductive type; and this must be the same argument that decreases for each recursive sub-call. This is called the *guard condition* for recursion and it ensures termination. On the other hand, it is a very restrictive form a recursion and we study more powerful alternatives here.

[13] As a general rule in this paper, when equations can be straightforwardly implemented by structural induction on one of the arguments, we expect the reader to be able to implement the corresponding `Fixpoint` and so we do not further comment on this.

– Let us first consider the case of the Function command which extends the primitive `Fixpoint` command. It allows the definition of fixpoints on decreasing arguments that may not be structurally decreasing. However the `Function` method fails very quickly because it only allows for the decrease of *one of the arguments*, and with itl, only the decrease of a combination of both arguments can be used to show termination. We could of course pack the two arguments in a pair but then, this will modify the code of the given algorithm, a modification which will undoubtedly impact the extracted code;
– Let us now consider the case of the Program Fixpoint command [17]. We can define $\mathsf{itl}_{\mathrm{pfix}}$ via a fully specified term:

$$\texttt{Program Fixpoint}\ \ \mathsf{itl}^{\mathrm{full}}_{\mathrm{pfix}}\ l\ m\ \{\texttt{measure}\ (|l|+|m|)\} : \{r \mid r = \mathsf{itl}_{\mathrm{triv}}\ l\ m\} := \ldots$$

basically by giving the right-hand side of Eq. (1) in Coq syntax, and then apply first and second projections to get the result as $\mathsf{itl}_{\mathrm{pfix}}\ l\ m := \pi_1(\mathsf{itl}^{\mathrm{full}}_{\mathrm{pfix}}\ l\ m)$ and the proof that it meets its specification $\mathsf{itl}_{\mathrm{pfix}}\ l\ m = \mathsf{itl}_{\mathrm{triv}}\ l\ m$ as $\pi_2(\mathsf{itl}^{\mathrm{full}}_{\mathrm{pfix}}\ l\ m)$. The `Program Fixpoint` command generates proof obligations that are easy to solve in this case. Notice that defining $\mathsf{itl}_{\mathrm{pfix}}$ without going through the fully specified term is possible but then it is not possible to establish the postcondition $\mathsf{itl}_{\mathrm{pfix}}\ l\ m = \mathsf{itl}_{\mathrm{triv}}\ l\ m$;
– Alternatively, we can use the Equations package [18] which could be viewed as an extension/generalization of `Program Fixpoint`. In that case, we can proceed with a weakly specified term:

$$\texttt{Equations}\ \ \mathsf{itl}_{\mathrm{eqs}}\ l\ m : \mathbb{L}\ X\ \texttt{by wf}\ (|l| + |m|) < := \ldots$$

then derive the two equations that are used to define $\mathsf{itl}_{\mathrm{eqs}}$, i. e., $\mathsf{itl}_{\mathrm{eqs}}\ [\ ]\ m = m$ and $\mathsf{itl}_{\mathrm{eqs}}\ (x :: l)\ m := x :: \mathsf{itl}_{\mathrm{eqs}}\ m\ l$. These equations are directly obtained by making use of the handy `simp` tactic provided with `Equations`. With these two equations, it is then easy to show the identity $\mathsf{itl}_{\mathrm{eqs}}\ l\ m = \mathsf{itl}_{\mathrm{triv}}\ l\ m$.

To sum up, considering the Coq implementation side only: `Function` fails because no single argument decreases; `Program Fixpoint` succeeds but via a fully specified term to get the postcondition; and `Equations` succeeds directly and generates the defining equations that are accessible through the `simp` tactic.

However, when we consider the extraction side of the problem, both `Program Fixpoint` and `Equations` do not give us the intended OCaml term of Eq. (1). On the left side of Fig. 1, we display the extracted code for $\mathsf{itl}_{\mathrm{pfix}}$ that was developed using `Program Fixpoint`, and on the right side the extracted code for $\mathsf{itl}_{\mathrm{eqs}}$ implemented using `Equations`.[14] Both present two drawbacks in our eyes. First, because "full" (resp. "eqs_0") are not made globally visible, it is impossible to inline their definitions with the `Extraction Inline` directive although this would be an obvious optimization. But then, also more problematic from an algorithmic point of view, there is the packing of the two arguments into a pair

[14] The actual extracted codes for both $\mathsf{itl}_{\mathrm{pfix}}$ and $\mathsf{itl}_{\mathrm{eqs}}$ are not that clean but we simplified them a bit to single out two specific problems.

```
let itl_pfix l m =
  let rec loop p =
    let l0   = fst p  in
    let m0   = snd p in
    let full = fun l2 m2
               → loop (l2, m2)
    in match l0 with
      | []      → m0
      | x :: l1 → x :: full m0 l1
  in loop (l, m)
```

```
let itl_eqs l m =
  let rec loop p =
    let m0   = snd p in
    let eqs0 = fun l2 m2 _
               → loop (l2, m2)
    in match fst p with
      | []      → m0
      | x :: l1 → x :: eqs0 m0 l1 _
  in loop (l, m)
```

Fig. 1. Extraction of $\mathsf{itl}_{\mathrm{pfix}}$ (`Program Fixpoint`) and $\mathsf{itl}_{\mathrm{eqs}}$ (`Equations`).

(l, m) prior to the call to the "loop" function that implements $\mathsf{itl}_{\mathrm{pfix}}$ (resp. $\mathsf{itl}_{\mathrm{eqs}}$). As a last remark, the two extracted codes look similar except that for $\mathsf{itl}_{\mathrm{eqs}}$, there is the slight complication of an extra *dummy* parameter _ added to the above definition of "eqs_0" that is then instantiated with a dummy argument _.

To sum up our above attempts, while the `Function` commands fails to handle itl because of the swap between arguments in Eq. (1), both `Program Fixpoint` and `Equations` succeed when considering the specification side, i. e., defining the function (which implicitly ensures termination) and proving its postcondition. However, extraction-wise, both generate artifacts, e. g., the pairing of arguments in this case, and which are difficult or impossible to control making the "certification by extraction" approach defined in Sect. 2.1 fail.

In the previously cited work by McCarthy *et al.* [11], the authors report that they successfully avoided "verification residues" as far as their generation of obligations for the certification of running time was concerned. We are heading for the absence of such residues from the extracted code. Let us hence now consider a last approach based on a finely tuned tactic which is both user-friendly and inlines inner fixpoints making it transparent to extraction. This is the approach we will be using for the breadth-first algorithms considered in here.

2.4 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 $X, Y : \mathtt{Type}$, a doubly indexed family $P : X \to Y \to \mathtt{Type}$ of types 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 smaller x'/y' arguments are (recursively) inhabited. The measures we use here are limited to $\mathbb{N}$ viewed as well-founded under the "strictly less" order. Any other type with a well-founded order would work as well.

To go beyond measures and implement a substantial fragment of general recursion, we invite the reader to consult work by the first author and Monin [8].

In the file wf_utils.v, we prove the following theorem while carefully crafting the computational content of the proof term, so that extraction yields an algorithm that is clean of spurious elements:

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

It allows building a term of 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 as follows:

$$\texttt{induction on}\ x\ y\ \texttt{as}\ \mathit{IH}\ \texttt{with measure}\ \|x, y\|$$

However, if the `induction-on` tactic was just about `apply`ing an instance of the term measure_double_rect, it would still leave artifacts in the extracted code much like the _ (resp. _) dummy parameter (resp. argument) in the $\mathsf{itl}_{\mathrm{eqs}}$ example of Fig. 1. So, to be precise in our description, the `induction-on` tactic actually builds the needed instance of the proof term measure_double_rect on a per-use basis, i. e., the proof term is inlined by the tactic itself. This ensures perfect extraction in the sense of the result being free of any artifacts. We refer to the file wf_utils.v for detailed explanations on how the inlining works.

From the point of view of specification, our `induction-on` tactic gives the same level of flexibility when compared to `Program Fixpoint` or `Equations`, at least when termination is grounded on a decreasing measure. However, when considering extraction, we think that it offers a finer control over the computational content of Coq terms, a feature which can be critical when doing certification by extraction. We now illustrate how to use the `induction-on` tactic from the user's point of view, on the example of the simple interleaving algorithm.

2.5 Back to Simple Interleaving

To define and specify $\mathsf{itl}_{\mathrm{on}}$ using the `induction-on` tactic, we first implement a fully specified version of $\mathsf{itl}_{\mathrm{on}}$ as the *functional* equivalent of $\mathsf{itl}_{\mathrm{triv}}$.[15] 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 $\mathsf{itl}_{\mathrm{triv}}\ l\ m$—a typical rich specification.

[15] Of course, it is not operationally equivalent.

```
Definition itl_on^full l m : {r : 𝕃 X | r = itl_triv 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, H_r) := loop m l' 𝕆₂?
                           in exist _ (x :: r) 𝕆₃? end).
  * trivial.                                   (* proof of 𝕆₁? *)
  * simpl; omega.                              (* proof of 𝕆₂? *)
  * subst; rewrite itl_triv_fix_1; trivial.    (* proof of 𝕆₃? *)
Defined.
```

The code inside the `refine`(...) tactic outputs terms like `exist _` $s\ \mathbb{O}_s^?$ where `_` is recovered by unification, and $\mathbb{O}_s^?$ is left open to be solved later by the user.[16] The constructor `exist` packs the pair $(s, \mathbb{O}_s^?)$ as a term of dependent type $\{r : \mathbb{L}\,X \mid r = \mathsf{itl}_{\mathrm{triv}}\ l\ m\}$ and thus $\mathbb{O}_s^?$ remains to be realized in the type $s = \mathsf{itl}_{\mathrm{triv}}\ l\ m$. This particular use of `refine` generates three *proof obligations* (also denoted PO)

$$\begin{array}{lll} \mathbb{O}_1^? & // & \ldots \vdash m = \mathsf{itl}_{\mathrm{triv}}\ []\ m \\ \mathbb{O}_2^? & // & \ldots \vdash |m| + |l'| < |x :: l'| + |m| \\ \mathbb{O}_3^? & // & \ldots, H_r : r = \mathsf{itl}_{\mathrm{triv}}\, m\, l' \vdash x :: r = \mathsf{itl}_{\mathrm{triv}}\ (x :: l')\ m \end{array}$$

later proved with their respective short proof scripts. Notice that our newly introduced `induction-on` tactic could alternatively be used to give another proof of itl_triv_fix_1 by measure induction on $|l| + |m|$. We remark that PO $\mathbb{O}_2^?$ is of different nature than $\mathbb{O}_1^?$ and $\mathbb{O}_3^?$. Indeed, $\mathbb{O}_2^?$ is a precondition, in this case a *termination certificate* ensuring that the recursive call occurs on smaller inputs. On the other hand, $\mathbb{O}_1^?$ and $\mathbb{O}_3^?$ are postconditions 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.[17]

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:

$$\mathsf{loop} : \forall\, l_0\, m_0,\ |l_0| + |m_0| < |x :: l'| + |m| \to \{r \mid r = \mathsf{itl}_{\mathrm{triv}}\, l_0\, m_0\}$$

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 _` $\mapsto \ldots$ Actually, loop is not used in the proofs

[16] In the Coq code, $\mathbb{O}_s^?$ is simply another `_` hole left for the `refine` tactic to either fill it by unification or postpone it. Because $\mathbb{O}_1^?$, $\mathbb{O}_2^?$ and $\mathbb{O}_3^?$ are postponed in this example, we give them names for better explanations.

[17] 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.

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 in Coq by the use of the "swiss-army knife" tactic `refine`[18] that, via unification, allows the user 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 purely logical content.

We continue the definition of $\mathsf{itl}_{\mathrm{on}}$ as the first projection

$$\texttt{Definition}\ \mathsf{itl}_{\mathrm{on}}\ l\ m := \pi_1(\mathsf{itl}_{\mathrm{on}}^{\mathrm{full}}\ l\ m).$$

and its specification using the projection π_2 on the dependent pair $\mathsf{itl}_{\mathrm{on}}^{\mathrm{full}}\ l\ m$.

$$\texttt{Fact}\ \mathsf{itl\_on\_spec}\ l\ m : \mathsf{itl}_{\mathrm{on}}\ l\ m = \mathsf{itl}_{\mathrm{triv}}\ l\ m.$$

Notice that by asking the extraction mechanism to inline the definition of $\mathsf{itl}_{\mathrm{on}}^{\mathrm{full}}$, we get the following extracted OCaml code for $\mathsf{itl}_{\mathrm{on}}$, the one that optimally reflects the original specification of Eq. (1):

$$\texttt{let rec}\ \mathsf{itl}_{\mathrm{on}}\ l\ m = \texttt{match}\ l\ \texttt{with}\ [] \to m \mid x :: l \to x :: \mathsf{itl}_{\mathrm{on}}\ m\ l$$

Of course, this outcome of the (automatic) code extraction had to be targeted when doing the proof of $\mathsf{itl}_{\mathrm{on}}^{\mathrm{full}}$: a trivial proof would have been to choose as r just $\mathsf{itl}_{\mathrm{triv}}\ l\ m$, 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 $\mathsf{loop}\ m\ l'$. 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 use of the tactic `induction-on`.

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 (Sect. 3.1), we describe mathematically what constitutes breadth-first traversal by considering an order on the branches (Sect. 3.2) that we characterize axiomatically (our Theorem 1). We then look at breadth-first traversal in its most elementary form (Sect. 3.3), by paying attention that it indeed meets its specification in terms of the order of the visited branches.

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

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 the argument type X.[19] We define the $\mathsf{root} : \mathbb{T}\,X \to X$ of a tree, the subtrees $\mathsf{subt} : \mathbb{T}\,X \to \mathbb{L}\,(\mathbb{T}\,X)$ of a tree, and the size $\|\cdot\| : \mathbb{T}\,X \to \mathbb{N}$ of a tree by:

$$\begin{array}{rclcrclcrcl} \mathsf{root}\ \langle x\rangle & := & x & \quad & \mathsf{subt}\ \langle x\rangle & := & [] & \quad & \|\langle x\rangle\| & := & 1 \\ \mathsf{root}\ \langle \_, x, \_\rangle & := & x & & \mathsf{subt}\ \langle a, \_, b\rangle & := & [a; b] & & \|\langle a, x, b\rangle\| & := & 1 + \|a\| + \|b\| \end{array}$$

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).[20] We define the predicate $\mathsf{btb} : \mathbb{T}\,X \to \mathbb{L}\,\mathbb{B} \to \mathtt{Prop}$ 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:

$\mathtt{Fact}\ \mathsf{btb\_spec}\ (t : \mathbb{T}\,X)\ (l : \mathbb{L}\,\mathbb{B}) : t /\!\!/ l \downarrow \leftrightarrow \exists x,\ t /\!\!/ l \rightsquigarrow x.$

By two inductive rules, we define $\sim_{\mathbb{T}} : \mathbb{T}\,X \to \mathbb{T}\,Y \to \mathtt{Prop}$, *structural equivalence of binary trees*, and we lift it to structural equivalence of lists of binary trees $\sim_{\mathbb{L}\mathbb{T}} : \mathbb{L}\,(\mathbb{T}\,X) \to \mathbb{L}\,(\mathbb{T}\,Y) \to \mathtt{Prop}$

$$\frac{}{\langle x\rangle \sim_{\mathbb{T}} \langle y\rangle} \qquad \frac{a \sim_{\mathbb{T}} a' \qquad b \sim_{\mathbb{T}} b'}{\langle a, x, b\rangle \sim_{\mathbb{T}} \langle a', y, b'\rangle} \qquad l \sim_{\mathbb{L}\mathbb{T}} m := \forall^2(\sim_{\mathbb{T}})\ l\ m$$

i. e., when $l = [a_1; \ldots; a_n]$ and $m = [b_1; \ldots; b_p]$, the equivalence $l \sim_{\mathbb{L}\mathbb{T}} m$ means $n = p \wedge a_1 \sim_{\mathbb{T}} b_1 \wedge \cdots \wedge a_n \sim_{\mathbb{T}} b_n$ (see Sect. 2).

Both $\sim_{\mathbb{T}}$ and $\sim_{\mathbb{L}\mathbb{T}}$ are equivalence relations, and if $a \sim_{\mathbb{T}} b$ then $\|a\| = \|b\|$, i. e., structurally equivalent trees have the same size, and this holds for structurally equivalent lists of trees as well.

[19] In his paper [14], 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.

[20] We here intend to model branches from the root to nodes, rather than from nodes to leaves. It might have been better to call the concept paths instead of branches.

3.2 Ordering Branches of Trees

A *strict order* $<_R$ is an irreflexive and transitive (binary) relation. We say it is *total* if the associated partial order $<_R \cup =$ is total in the usual sense of being connex, or equivalently, if $\forall x\, y,\ \{x <_R y\} \vee \{x = y\} \vee \{y <_R x\}$. It is *decidable and total* if:

$$\forall\, x\, y,\ \{x <_R y\} + \{x = y\} + \{y <_R x\}$$

where, as usual in type theory, a proof of the sum $A + B + C$ requires either a proof of A, of B or of C together with the information whether the first, second or third summand has been proven.

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

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

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

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

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

Lemma 1. *$\prec_{\text{dic}}$ and $\prec_{\text{bf}}$ are decidable and total strict orders.*

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

Theorem 1. *Let $<_R : \mathbb{L}\,\mathbb{B} \to \mathbb{L}\,\mathbb{B} \to \texttt{Prop}$ 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 $\prec_{\text{bf}}$, i. e., $\forall\, l\, m,\ l <_R m \leftrightarrow l \prec_{\text{bf}} m$. Moreover the relation $\prec_{\text{bf}}$ satisfies (A_1)–(A_4).

3.3 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

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

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[21]

$$\mathsf{niv}\,\langle x\rangle := [x] :: [] \qquad \mathsf{niv}\,\langle a, x, b\rangle := [x] :: \mathsf{zip}\,(\mathsf{niv}\,a)\,(\mathsf{niv}\,b)$$

[21] The function niv is called "levelorder traversal" by Jones and Gibbons [7].

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 $\mathsf{bft}_{\mathrm{std}}\, t := \mathsf{concat}(\mathsf{niv}\, t)$.[22] We lift breadth-first traversal to branches instead of decorations, defining $\mathsf{niv}_{\mathrm{br}} : \mathbb{T}\, X \to \mathbb{L}\,(\mathbb{L}\,(\mathbb{L}\,\mathbb{B}))$ by

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

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

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

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

Hence $\mathsf{bft}_{\mathrm{br}}\, t$ and $\mathsf{bft}_{\mathrm{std}}\, t$ traverse the tree t in the same order, except that $\mathsf{bft}_{\mathrm{std}}$ outputs decorating values and $\mathsf{bft}_{\mathrm{br}}$ outputs branches.[23] We moreover show that $\mathsf{bft}_{\mathrm{br}}\, t$ lists the branches of t in $\prec_{\mathrm{bf}}$-ascending order.

Theorem 2. *The list* $\mathsf{bft}_{\mathrm{br}}\ t$ *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 $\mathsf{roots} : \mathbb{L}\,(\mathbb{T}\, X) \to \mathbb{L}\, X$ and $\mathsf{subtrees} : \mathbb{L}\,(\mathbb{T}\, X) \to \mathbb{L}\,(\mathbb{T}\, X)$ by

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

where $\mathsf{flat\_map}$ 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 $\mathsf{subtrees}\ l$, 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 $f(\mathsf{subtrees}\ l)$, as soon as the case $f\,[]$ is computed without using recursive calls (to f). For this, we use, for instance, recursion on the measure $l \mapsto \|l\|$ but we may also use the binary measure $l, m \mapsto \|l +\!\!+ m\|$ when f has two arguments instead of just one.

4.1 Equational Characterization of Breadth-First Traversal

We first characterize $\mathsf{bft}_{\mathrm{f}}$—breadth-first traversal of a forest—with four equivalent equations:

[22] Where $\mathsf{concat} := \mathsf{fold}\ (\cdot +\!\!+ \cdot)\ []$, i. e., $\mathsf{concat}\ [l_1; \ldots; l_n] = l_1 +\!\!+ \cdots +\!\!+ l_n$.

[23] The $\mathsf{bpn}\ t$ relation, also denoted $l, x \mapsto t /\!\!/ l \rightsquigarrow x$, relates branches and decorations.

Theorem 3 (Characterization of breadth-first traversal of forests – recursive part). *Let* $\mathsf{bft_f}$ *be any term of type* $\mathbb{L}\,(\mathbb{T}\,X) \to \mathbb{L}\,X$ *and consider the following equations:*

(1) $\forall\, l,\ \mathsf{bft_f}\, l = \mathsf{roots}\, l +\!\!+ \mathsf{bft_f}\,(\mathsf{subtrees}\, l)$*;*
(2) $\forall\, l\, m,\ \mathsf{bft_f}\,(l +\!\!+ m) = \mathsf{roots}\, l +\!\!+ \mathsf{bft_f}\,(m +\!\!+ \mathsf{subtrees}\, l)$*;*
(3) $\forall\, t\, l,\ \mathsf{bft_f}\,(t :: l) = \mathsf{root}\, t :: \mathsf{bft_f}\,(l +\!\!+ \mathsf{subt}\, t)$*;*
(Oka1) $\forall\, x\, l,\ \mathsf{bft_f}\,(\langle x\rangle :: l) = x :: \mathsf{bft_f}\, l$*;*
(Oka2) $\forall\, a\, b\, x\, l,\ \mathsf{bft_f}\,(\langle a, x, b\rangle :: l) = x :: \mathsf{bft_f}\,(l +\!\!+ [a; b])$.

We have the equivalence: (1) $\leftrightarrow$ (2) $\leftrightarrow$ (3) $\leftrightarrow$ (Oka1 $\wedge$ Oka2).

Proof. Equations (1) and (3) are clear instances of Eq. (2). Then (Oka1 $\wedge$ Oka2) is equivalent to (3) because they just represent a case analysis on t. So the only difficulty is to show (1) $\to$ (2) and (3) $\to$ (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 simple interleaving algorithm of Sect. 2.5, we proceed by induction on the measure $\|l +\!\!+ m\|$. □

Equations (Oka1) and (Oka2) are used by Okasaki [14] as defining equations, while (3) is calculated from the specification by Jones and Gibbons [7]. We single out Eq. (2) above as 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 (Full characterization of breadth-first traversal of forests). *Adding equation* $\mathsf{bft_f}\,[] = []$ *to any one of the equations of Theorem 3 determines the function* $\mathsf{bft_f}$ *uniquely.*

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

Notice that one should not confuse the uniqueness of the function—which is an extensional notion—with the uniqueness of an algorithm implementing such a function, because there are hopefully numerous possibilities.[24]

4.2 Direct Implementation of Breadth-First Traversal

We give a definition of $\mathsf{forest_{dec}} : \mathbb{L}\,(\mathbb{T}\,X) \to \mathbb{L}\,X \times \mathbb{L}\,(\mathbb{T}\,X)$ such that, provably, $\mathsf{forest_{dec}}\ l = (\mathsf{roots}\ l, \mathsf{subtrees}\ l)$, but using a simultaneous computation:

$$\begin{array}{l} \mathsf{forest_{dec}}\ [] := ([], []) \quad \mathsf{forest_{dec}}\,(\langle x\rangle :: l) := (x :: \alpha, \beta) \\ \quad\quad \mathsf{forest_{dec}}\,(\langle a, x, b\rangle :: l) := (x :: \alpha, a :: b :: \beta) \end{array} \quad \text{where } (\alpha, \beta) := \mathsf{forest_{dec}}\ l.$$

[24] Let us stress that extensionality is not very meaningful for algorithms anyway.

Then we show one way to realize the equations of Theorem 3 into a Coq term:

Theorem 5 (Existence of breadth-first traversal of forests). *One can define a Coq term* $\mathsf{bft_f}$ *of type* $\mathbb{L}\,(\mathbb{T}\,X) \to \mathbb{L}\,X$ *s. t.*

1. $\mathsf{bft_f}\ [] = []$
2. $\forall l,\ \mathsf{bft_f}\ l = \mathsf{roots}\ l +\!\!+ \mathsf{bft_f}\,(\mathsf{subtrees}\ l)$

and s. t. $\mathsf{bft_f}$ *extracts to the following OCaml code:*[25]

```
let rec bft_f l = match l with [] -> [] | _ -> let α, β = forest_dec l in α @ bft_f β.
```

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

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

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

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

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 in Sect. 2.5. We get $\mathsf{bft_f}$ by the first projection $\mathsf{bft_f}\ l := \pi_1(\mathsf{bft\_f\_full}\ l)$ and derive the specification

$$\mathtt{Fact}\ \mathsf{bft\_f\_spec} : \forall l,\ l \rightsquigarrow_{\mathrm{bft}} \mathsf{bft_f}\ l.$$

with the second projection π_2. Equations 1 and 2 follow straightforwardly from $\mathsf{bft\_f\_fun}$ and $\mathsf{bft\_f\_spec}$. □

Hence we see that we can use the specifying Eqs. 1 and 2 of Theorem 5 to define the term $\mathsf{bft_f}$. 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 $\mathsf{subtrees\_dec}$.

Theorem 6 (Correctness of breadth-first traversal of forests). *For all* $t : \mathbb{T}\,X$*, we have* $\mathsf{bft_f}\,[t] = \mathsf{bft_{std}}\,t$.

Proof. We define $\rightsquigarrow_{\mathrm{niv}} : \mathbb{L}\,(\mathbb{T}\,X) \to \mathbb{L}\,(\mathbb{L}\,X) \to \mathtt{Prop}$ 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 $[t] \rightsquigarrow_{\mathrm{bft}} \mathsf{bft_{std}}\ t$. By $\mathsf{bft\_f\_spec}$ we have $[t] \rightsquigarrow_{\mathrm{bft}} \mathsf{bft_f}\ [t]$ and we conclude with $\mathsf{bft\_f\_fun}$. □

[25] Notice that list append is denoted `@` in OCaml and $+\!\!+$ in Coq.

4.3 Properties of Breadth-First Traversal

The *shape* of a tree (resp. forest) is the structure that remains when removing the values on the nodes and leaves, e. g., by mapping the base type X to the singleton type `unit`. Alternatively, one can use the $\sim_{\mathbb{T}}$ (resp. $\sim_{\mathbb{LT}}$) equivalence relation (introduced in Sect. 3.1) to characterize trees (resp. forests) which have identical shapes. We show that on a given forest shape, breadth-first traversal is an injective map:

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

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 3 to rewrite $\mathsf{bft_f}$ 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, $\mathsf{bft_{std}}$ is also injective:

Corollary 1. *If* $t_1 \sim_{\mathbb{T}} t_2$ *are two trees of type* $\mathbb{T}\,X$ *(of the same shape) and* $\mathsf{bft_{std}}\ t_1 = \mathsf{bft_{std}}\ t_2$ *then* $t_1 = t_2$.

Proof. From Lemma bft_f_inj and Theorem 6. □

4.4 Discussion

The algorithm described in Theorem 5 can be used to compute breadth-first traversal as a replacement for the naive $\mathsf{bft_{std}}$ algorithm. We could also use other equations of Theorem 3, for instance using $\mathsf{bft_f}\ [] = []$, (Oka1) and (Oka2) to define another algorithm. The problem with equation

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

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 $\mathsf{bft_f}$ 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 [14]. They have the potential to be efficient, but this depends on the later implementation of the axiomatic datatype of FIFOs considered here (Sect. 5.1). We deal with traversal, numbering, reconstruction—in breadth-first order.

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

Fig. 2. An axiomatization of first-in first-out queues.

5.1 Axiomatization of FIFOs

In Fig. 2, 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 argument*[26] 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. 2. 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. 2 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 (or refinements), including a trivial and inefficient one where f2l is the identity function (and the inefficiency comes from appending the new elements at the end of the list with enq). In Sect. 7, we refine these axioms with more efficient implementations following Okasaki's insights [13] that worst-case $\mathcal{O}(1)$ 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 3 and Theorem 5:

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

[26] When a parameter X is marked implicit using the $\{X\}$ notation, it usually means that Coq is going to be able to infer the value of the argument by unification from the constraints in the context. In the case of $\mathsf{f2l}\ l$, it means X will be deduced from the type of l which should unify with $\mathsf{fifo}\ X$. While not strictly necessary, the mechanism of implicit arguments greatly simplifies the readability of Coq terms. In particular, it avoids an excessive use of dummy arguments _.

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 ($\mathsf{enq}\ (\mathsf{enq}\ \ldots\ a)\ b$). Hence, we implement FIFO-based breadth-first traversal.

Theorem 7. *There exists a fully specified Coq term*

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

s. t. $\mathsf{bft\_fifo_f}$ *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 interleave example of Sect. 2.5. The proof is structured around the computational content of the above OCaml code. Termination POs are easily solved by `omega`. Postconditions for correctness are proved using the above equations. □

Corollary 2. *There is a Coq term* $\mathsf{bft_{fifo}} : \mathbb{T}\,X \to \mathbb{L}\,X$ *s.t.* $\mathsf{bft_{fifo}}\ t = \mathsf{bft_{std}}\ t$ *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. From 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 $\mathsf{bft_{fifo}}\ t := \pi_1(\mathsf{bft\_fifo\_full}\ t)$ 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 6. □

5.3 Breadth-First Numbering

Breadth-first numbering was the challenge proposed by Okasaki to the community and which led him to write his paper [14]. 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 2. *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 Sect. 4.3. □

We give an equational characterization of breadth-first numbering of forests combined with list reversal. In the equations below, we intentionally consider a $\mathsf{bfn_f}$ function that outputs the *reverse* of the numbering of the input forest, so that, when viewed as FIFOs of trees (instead of lists of trees), both the input FIFO over $\mathbb{T}\,X$ and the output FIFO over type $\mathbb{T}\,\mathbb{N}$ correspond to left dequeuing and right enqueuing.[27] That said, Eqs. (E2) and (E3) correspond to (Oka1) and (Oka2) respectively, augmented with an extra argument used for keeping track of the numbering.

Lemma 3. *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) $\forall\, n\, x\, l,\ \mathsf{bfn_f}\ n\ (\langle x\rangle :: l) = \mathsf{bfn_f}\ (1+n)\ l +\!\!+ [\langle n\rangle]$;
(E3) $\forall\, n\, a\, x\, b\, l,\ \exists\, a'\, b'\, l',$
$\mathsf{bfn_f}\ (1+n)\ (l +\!\!+ [a; b]) = b' :: a' :: l' \wedge \mathsf{bfn_f}\ n\ (\langle a, x, b\rangle :: l) = l' +\!\!+ [\langle a', n, b'\rangle]$;
(Bfn1) $\forall\, n\, l,\ l \sim_{\mathbb{LT}} \mathsf{rev}(\mathsf{bfn_f}\ n\ l)$;
(Bfn2) $\forall\, n\, l,\ \mathsf{is\_bfn}\ n\ (\mathsf{rev}(\mathsf{bfn_f}\ n\ l))$.

We have the equivalence: $(\mathrm{E1} \wedge \mathrm{E2} \wedge \mathrm{E3}) \leftrightarrow (\mathrm{Bfn1} \wedge \mathrm{Bfn2})$.

Proof. From right to left, we essentially use Lemma 2. For the reverse direction, we proceed by induction on the measure $i, l \mapsto \|l\|$ in combination with Theorem 5 and Theorem 3—Equations (Oka1) and (Oka2). □

Although not explicitly written in Okasaki's paper [14], these equations hint at the use of FIFOs as a replacement for lists for both the input and output of $\mathsf{bfn_f}$. Let's see this informally for (E3): $\mathsf{bfn_f}\ n$ is to be computed on a non-empty list viewed as a FIFO, and left dequeuing gives a composite tree $\langle a, x, b\rangle$ and the remaining list/FIFO l. The subtrees a and b are enqueued to the right of l and $\mathsf{bfn_f}\ (1+n)$ called on the resulting list/FIFO. (E3) guarantees that a' and b' can be dequeued to the left from the output list/FIFO. Finally, $\langle a', n, b'\rangle$ is enqueued to the right to give the correct result, thanks to (E3). This construction will be formalized in Theorem 8.

We define the specification bfn_fifo_f_spec corresponding to breadth-first numbering of FIFOs of trees

$$\texttt{Definition}\ \mathsf{bfn\_fifo\_f\_spec}\ n\ q\ q' := \mathsf{f2l}\ q \sim_{\mathbb{LT}} \mathsf{rev}\,(\mathsf{f2l}\ q') \wedge \mathsf{is\_bfn}\ n\ (\mathsf{rev}(\mathsf{f2l}\ q'))$$

and we show the inhabitation of this specification.

[27] The fact that input and output FIFOs operate in mirror to each other was already pointed out by Okasaki in [14]. Using reversal avoids defining two types of FIFOs or bi-directional FIFOs to solve the issue.

Theorem 8. *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:

```
let rec bfn_fifo_f n q =
  if void q then emp
  else let t, q0 = deq q in match t with
    | ⟨_⟩       → enq (bfn_fifo_f (1 + n) q0) ⟨n⟩
    | ⟨a, _, b⟩ → let b', q1 = deq (bfn_fifo_f (1 + n) (enq (enq q0 a) b)) in
                  let a', q2 = deq q1
                  in enq q2 ⟨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 parameter does not participate in the measure. As in Sect. 2.5, we implement a proof script which mixes tactics and programming style using the `refine` tactic. We strictly follow the above algorithm to design $\mathsf{bfn\_fifo_f}$. Of course, proof obligations like termination certificates or postconditions are generated by Coq and need to be addressed. As usual for these breadth-first algorithms, termination certificates are easy to solve thanks to the `omega` tactic. The only difficulties lie in the postcondition POs but these correspond to the (proofs of the) equations of Lemma 3. □

Corollary 3. *There is a Coq term* $\mathsf{bfn_{fifo}} : \mathbb{T}\,X \to \mathbb{T}\,\mathbb{N}$ *s. t. for any tree* $t : \mathbb{T}\,X$*:*

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

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

$$\mathtt{let}\ \mathsf{bfn_{fifo}}\ t = \mathtt{let}\ t', \_ = \mathsf{deq}\ \big(\mathsf{bfn\_fifo_f}\ 1\ (\mathsf{enq}\ \mathsf{emp}\ t)\big)\ \mathtt{in}\ t'.$$

Proof. Obvious consequence of Theorem 8 in conjunction with Theorem 6. □

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 argument 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 [7] 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

$$\mathtt{Definition}\ \mathsf{bfr\_fifo\_f\_spec}\ q\ l\ q' := \mathsf{f2l}\ q \sim_{\mathbb{L}\mathbb{T}} \mathsf{rev}\,(\mathsf{f2l}\ q') \wedge \mathsf{bft_f}\,(\mathsf{rev}\,(\mathsf{f2l}\ q')) = l.$$

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

$$\texttt{Fixpoint}\ \mathsf{bfr\_fifo_f}\ q\ l\ \{\texttt{struct}\ l\} : \|\mathsf{f2l}\ q\| = |l| \to \{q' \mid \mathsf{bfr\_fifo\_f\_spec}\ q\ l\ q'\}.$$

Notice the precondition $\|\mathsf{f2l}\ q\| = |l|$ stating that l contains as many labels as the total number of nodes in the FIFO q.[28] Since we use structural induction, there are no termination POs. There is however a precondition PO (easily proved) and postcondition POs are similar to those of the proof of Theorem 8. 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 b', q1 = deq (bfr_fifo_f (enq (enq q0 a) b) l) in
                  let a', q2 = deq q1
                  in enq q2 ⟨a', y, b'⟩.
```

Notice the similarity with the code of $\mathsf{bfn\_fifo_f}$ of Theorem 8.

Theorem 9. *There is a Coq term* $\mathsf{bfr_{fifo}} : \forall\,(t : \mathbb{T}\,X)\,(l : \mathbb{L}\,Y),\ \|t\| = |l| \to \mathbb{T}\,Y$ *such that for any tree* $t : \mathbb{T}\,X$, $l : \mathbb{L}\,Y$ *and* $H : \|t\| = |l|$ *we have:*

$$t \sim_{\mathbb{T}} \mathsf{bfr_{fifo}}\ t\ l\ H \qquad \textit{and} \qquad \mathsf{bft_{std}}\,(\mathsf{bfr_{fifo}}\ t\ l\ H) = l.$$

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

$$\texttt{let}\ \mathsf{bfr_{fifo}}\ t\ l = \texttt{let}\ t', \_ = \mathsf{deq}\ \left(\mathsf{bfr\_fifo_f}\ (\mathsf{enq}\ \mathsf{emp}\ t)\ l\right)\ \texttt{in}\ t'.$$

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

6 Numbering by Levels

Okasaki reports in his paper [14] 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 $\mathsf{bft_{std}}$. In his Sect. 4, he describes the "cleanest" of all those solutions, and this small section is devoted to get it by extraction (and thus with certification through the method followed in this paper).

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

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

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

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

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

Since we will need to use both $\mathsf{children_f}$ and $\mathsf{rebuild_f}$ for varying values of the type parameter K, we define them as fully polymorphic here.[29] We then fix $X : \mathsf{Type}$ for the remainder of this section.

The algorithms $\mathsf{children_f}$ and $\mathsf{rebuild_f}$ are (nearly) those defined in [14, Figure 5] but that paper does not provide a specification for $\mathsf{rebuild_f}$ and thus cannot show the correctness result which follows. Instead of a specification, Okasaki offers an intuitive explanation of $\mathsf{rebuild_f}$ [14, p. 134]. Here, we will first rephrase the following lemma in natural language: the task is to obtain the breadth-first numbering of list t_s, starting with index n. We consider the list $\mathsf{subtrees}\ t_s$ of all immediate subtrees, hence of all that is "at the next level" (speaking in Okasaki's terms), and assume that c_s is the result of breadth-first numbering of those, but starting with index $|t_s| + n$, so as to skip all the roots in t_s whose number is $|t_s|$. Then, $\mathsf{rebuild_f}\ n\ t_s\ c_s$ is the breadth-first numbering of t_s. In view of this description, the first three definition clauses of $\mathsf{rebuild_f}$ are unavoidable, and the last one gives a dummy result for a case that never occurs when running algorithm $\mathsf{bfn\_level_f}$ in Theorem 10 below.

Lemma 4. *The function* $\mathsf{rebuild_f}$ *satisfies the following specification: for any* n, $t_s : \mathbb{L}\,(\mathbb{T}\,X)$ *and* $c_s : \mathbb{L}\,(\mathbb{T}\,\mathbb{N})$, *if both* $\mathsf{subtrees}\ t_s \sim_{\mathbb{LT}} c_s$ *and* $\mathsf{is\_bfn}\ (|t_s| + n)\ c_s$ *hold then* $t_s \sim_{\mathbb{LT}} \mathsf{rebuild_f}\ n\ t_s\ c_s$ *and* $\mathsf{is\_bfn}\ n\ (\mathsf{rebuild_f}\ n\ t_s\ c_s)$.

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 $Y, Z : \mathsf{Type}$, $n : \mathbb{N}$, $t_s : \mathbb{L}\,(\mathbb{T}\,Y)$, $t'_s : \mathbb{L}\,(\mathbb{T}\,Z)$ and any $c_s : \mathbb{L}\,(\mathbb{T}\,\mathbb{N})$, if $t_s \sim_{\mathbb{LT}} t'_s$ 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 2. □

The lemma suggests the recursive algorithm contained in the following theorem: in place of c_s as argument to $\mathsf{rebuild_f}$, it uses the result of the recursive call on $\mathsf{bfn\_level_f}$ with the index shifted by the number of terms in t_s (hence the number of roots in t_s—which is different from Okasaki's setting with labels only at inner nodes) and the second component of $\mathsf{children_f}\ t_s$.

Theorem 10. *There is a fully specified Coq term*

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

[29] Hence the $\forall\{K\}$ where K is declared as implicit.

which extracts to the following OCaml code:

$$\texttt{let rec}\ \mathsf{bfn\_level_f}\ i\ t_s = \texttt{match}\ t_s\ \texttt{with}$$
$$\mid [] \to []$$
$$\mid \_ \to \texttt{let}\ n, s_s = \mathsf{children_f}\ t_s\ \texttt{in}\ \mathsf{rebuild_f}\ i\ t_s\ (\mathsf{bfn\_level_f}\ (n+i)\ s_s).$$

Proof. By induction on the measure $i, l \mapsto \|l\|$. The non-trivial correctness PO is a consequence of Lemma 4. □

Corollary 4. *There is a Coq term* $\mathsf{bfn_{level}} : \mathbb{T}\,X \to \mathbb{T}\,\mathbb{N}$ *s. t. for any tree* $t : \mathbb{T}\,X$*:*

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

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

$$\texttt{let}\ \mathsf{bfn_{level}}\ t = \texttt{match}\ \mathsf{bfn\_level_f}\ 1\ [t]\ \texttt{with}\ t' :: \_ \to t'.$$

Proof. Direct application of Theorems 6 and 10. □

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 Sect. 5.1, the Coq developments of Sects. 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. 2 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. 2.

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 [7] 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 by Okasaki [13], 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 [13].

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"):

$$\texttt{CoInductive}\ \mathbb{S}\,X := \langle\rangle : \mathbb{S}\,X \mid \_ \# \_ : \ X \to \mathbb{S}\,X \to \mathbb{S}\,X.$$

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

$$\begin{aligned}&\texttt{Inductive}\ \mathsf{lfin} : \mathbb{S}\,X \to \texttt{Prop} :=\\ &\quad \mid \mathsf{lfin\_nil} \quad : \mathsf{lfin}\ \langle\rangle\\ &\quad \mid \mathsf{lfin\_cons} : \forall\, x\ s,\ \mathsf{lfin}\ s \to \mathsf{lfin}\,(x \# s).\end{aligned}$$

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

$$\texttt{Definition}\ \mathbb{L}_l\,X := \{s : \mathbb{S}\,X \mid \mathsf{lfin}\ s\}.$$

Compared to regular lists $\mathbb{L}\,X$, for a lazy list $(s, H\,s) : \mathbb{L}_l\,X$, 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 working with such lazy lists as if they were regular lists, and this practice is fully compatible with extraction because the guards of type $\mathsf{lfin}\,s : \texttt{Prop}$, 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 [13] found a simple way to implement simple FIFOs[30] 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, being of beautiful simplicity, does not provide the full specifications for a correctness proof. Some important invariants are present, though. The main difficulty we faced was to give 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; see the files `llists.v`, `fifo_3llists.v` and `extraction.v` described in Appendix A.

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 below 2k nodes) 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 over 64k nodes), garbage collection severely hampers the theoretic linear-time behaviour. A precise analysis is out of scope for this paper.

8 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 [14].

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 [5] 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.

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

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 1) in which the paths are visited by the breadth-first traversal algorithm. (Sect. 3.3). This also includes the identification of four different but logically equivalent ways to express the recursive behaviour of breadth-first traversal on forests (Theorem 3) and an equational characterization of breadth-first numbering of forests (Lemma 3). Among the variations, we mention that breadth-first reconstruction (Sect. 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 binary 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 [16] 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 [7] 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 [14] to relate his work to theirs. To obtain such kind of algorithms would be a major challenge for the research in certified program extraction.

Another Related Research Direction. We mentioned directly related work throughout the paper, and we discussed certification of the program extraction procedure in Sect. 2.2. Let's briefly indicate a complementary approach. We state in our Theorems 5, 7, 8, 9 and 10 the OCaml code we wanted to obtain by extraction (and that we then got), but there is no tool support to start with that code and to work towards the (fully specified) Coq terms. The `hs-to-coq`[31] tool [19] transforms Haskell code (in place of OCaml that we chose to use) into Coq code and provides means to subsequent verification provided the Haskell code does not exploit non-termination.

[31] https://github.com/antalsz/hs-to-coq.

Acknowledgments. We are most grateful to the anonymous reviewers for their thoughtful feedback that included numerous detailed suggestions for improvement of the presentation.

A Code Correspondence

Here, 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 doing program extraction, see the README section on the given web page.
We are here presenting 24 Coq vernacular files in useful order:

- `list_utils.v`: One of the biggest files, all concerning list operations, list permutations, the lifting of relations to lists (Sect. 2) 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 (Sect. 2.4) – 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 [13], relevant in Sect. 7.2.
- `interleave.v`: The example of interleaving with three different methods in Sects. 2.3 (with existing tools—needs Coq v8.9 with package `Equations`) and Sect. 2.5 (with our method).
- `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 Sect. 3.2.
- `increase.v`: Small auxiliary file for full specification of breadth-first traversal (Sect. 3.3).
- `bt.v`: The largest file in this library, describing binary trees (Sect. 3.1), their branches and orders on those (Sect. 3.2) in relation with breadth-first traversal and structural relations on trees and forests (again Sect. 3.1).
- `fifo.v`: the module type for abstract FIFOs (Sect. 5.1).
- `fifo_triv.v`: The trivial implementation of FIFOs through lists, mentioned in Sect. 5.1.
- `fifo_2lists.v`: An efficient implementation that has amortized $\mathcal{O}(1)$ operations (see, e. g., the paper by Okasaki [13]), described in Sect. 7.1.
- `fifo_3llists.v`: The much more complicated FIFO implementation that is slower but has worst-case $\mathcal{O}(1)$ operations, invented by Okasaki [13]; see Sect. 7.2.
- `bft_std.v`: Breadth-first traversal naively with levels (specified with the traversal of branches in suitable order), presented in Sect. 3.3.
- `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 (Sect. 4.1).

– `bft_inj.v`: Structurally equal forests with the same outcome of breadth-first traversal are equal, shown in Sect. 4.3.
– `bft_fifo.v`: Breadth-first traversal given an abstract FIFO, described in Sect. 5.2.
– `bfn_spec_rev.v`: Characterization of breadth-first numbering, see Lemma 3.
– `bfn_fifo.v`: The certified analogue of Okasaki's algorithm for breadth-first numbering [14], in Sect. 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 (Sect. 4 and Fig. 5) of Okasaki's article [14]. 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 Sect. 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.

References

1. Anand, A., Boulier, S., Cohen, C., Sozeau, M., Tabareau, N.: Towards certified meta-programming with typed Template-Coq. In: Avigad, J., Mahboubi, A. (eds.) ITP 2018. LNCS, vol. 10895, pp. 20–39. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94821-8_2
2. Andronick, J., Felty, A.P. (eds.): Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, 8–9 January 2018. ACM (2018). http://dl.acm.org/citation.cfm?id=3176245
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07964-5
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. The MIT Press and McGraw-Hill Book Company (1989)
5. Delahaye, D.: A proof dedicated meta-language. Electr. Notes Theor. Comput. Sci. **70**(2), 96–109 (2002). https://doi.org/10.1016/S1571-0661(04)80508-5
6. Hupel, L., Nipkow, T.: A verified compiler from Isabelle/HOL to CakeML. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 999–1026. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_35
7. Jones, G., Gibbons, J.: Linear-time breadth-first tree algorithms: an exercise in the arithmetic of folds and zips. Technical report, No. 71, Department of Computer Science, University of Auckland, May 1993
8. Larchey-Wendling, D., Monin, J.F.: Simulating induction-recursion for partial algorithms. In: Espírito Santo, J., Pinto, L. (eds.) 24th International Conference on Types for Proofs and Programs, TYPES 2018, Abstracts. University of Minho, Braga (2018). http://www.loria.fr/~larchey/papers/TYPES_2018_paper_19.pdf

9. Letouzey, P.: A new extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39185-1_12
10. Letouzey, P.: Programmation fonctionnelle certifiée - L'extraction de programmes dans l'assistant Coq. Ph.D. thesis, Université Paris-Sud, July 2004. https://www.irif.fr/~letouzey/download/these_letouzey_English.pdf
11. McCarthy, J.A., Fetscher, B., New, M.S., Feltey, D., Findler, R.B.: A Coq library for internal verification of running-times. Sci. Comput. Program. **164**, 49–65 (2018). https://doi.org/10.1016/j.scico.2017.05.001
12. Mullen, E., Pernsteiner, S., Wilcox, J.R., Tatlock, Z., Grossman, D.: Œuf: minimizing the Coq extraction TCB. In: Andronick and Felty [2], pp. 172–185. https://doi.org/10.1145/3167089
13. Okasaki, C.: Simple and efficient purely functional queues and deques. J. Funct. Program. **5**(4), 583–592 (1995)
14. Okasaki, C.: Breadth-first numbering: lessons from a small exercise in algorithm design. In: Odersky, M., Wadler, P. (eds.) Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), pp. 131–136. ACM (2000)
15. Paulson, L.C.: ML for the Working Programmer. Cambridge University Press, Cambridge (1991)
16. Picard, C., Matthes, R.: Permutations in coinductive graph representation. In: Pattinson, D., Schröder, L. (eds.) CMCS 2012. LNCS, vol. 7399, pp. 218–237. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32784-1_12
17. Sozeau, M.: Subset coercions in Coq. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 237–252. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74464-1_16
18. Sozeau, M.: Equations: a dependent pattern-matching compiler. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 419–434. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_29
19. Spector-Zabusky, A., Breitner, J., Rizkallah, C., Weirich, S.: Total Haskell is reasonable Coq. In: Andronick and Felty [2], pp. 14–27. https://doi.org/10.1145/3167092