Martin Hofmann's case for non-strictly positive data types

Ulrich Berger
Dept. of Computer Science, Swansea University, United Kingdom
u.berger@swansea.ac.uk

Ralph Matthes
IRIT (CNRS and University of Toulouse)
Ralph.Matthes@irit.fr

Anton Setzer
Dept. of Computer Science, Swansea University, United Kingdom
a.g.setzer@swansea.ac.uk

Abstract

We describe the breadth-first traversal algorithm by Martin Hofmann that uses a non-strictly positive data type and carry out a simple verification in an extensional setting. Termination is shown by implementing the algorithm in the strongly normalising extension of system F by Mendler-style recursion. We then analyze the same algorithm by alternative verifications in an intensional setting, in a setting with non-strictly positive inductive definitions (not just non-strictly positive data types), and one by algebraic reduction. The verification approaches are compared in terms of notions of simulation and should elucidate the somewhat mysterious algorithm and thus make a case for other uses of non-strictly positive data types. Except for the termination proof, which cannot be formalised in Coq, all proofs were formalised in Coq and some of the algorithms were implemented in Agda and Haskell.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory; Software and its engineering → Abstract data types; Software and its engineering → Recursion; Software and its engineering → Software verification; Software and its engineering → Coroutines

Keywords and phrases non strictly-positive data types, breadth-first traversal, program verification, Mendler-style recursion, System F, theorem proving, Coq, Agda, Haskell

Digital Object Identifier 10.4230/LIPIcs.TYPES.2018.1

Supplement Material `https://github.com/rmatthes/breadthfirstalahofmann`

Funding The second and third authors got financial support by the COST action CA15123 EU-TYPES.
Ulrich Berger: Work supported by CORCON FP7 Marie Curie International Research Project, PIRSES-GA-2013-612638; COMPUTAL FP7 Marie Curie International Research Project, PIRSES-GA-2011-294962, CID FP7 Marie Curie International Research Project, H2020-MSCA-RISE-2016-731143
Anton Setzer: Work supported by EPSRC grant EP/G033374/1 *Theory and Applications of Induction Recursion*; CORCON FP7 Marie Curie International Research Project, PIRSES-GA-2013-612638; COMPUTAL FP7 Marie Curie International Research Project, PIRSES-GA-2011-294962, CID FP7 Marie Curie International Research Project, H2020-MSCA-RISE-2016-731143

Acknowledgements This paper is dedicated to the memory of the late Martin Hofmann. Martin was one of the leading researchers in the field of functional programming and type theory. This article is based on his notes [6, 7], which is only one example of the inspiration he has given to many researchers. His tragic unexpected death was a deep loss for the community.

1 Introduction

Given a finitely-branching tree t with labels at all nodes, there are different ways to traverse it, starting with its root. Depth-first traversal first goes along the entire left-most[1] branch until the leaf is reached and then backtracks and pursues with the next sibling. An efficient implementation of depth-first traversal is possible by using a stack of entry points into subtrees of t. In the beginning, t is pushed on the stack. While the stack is non-empty, a tree is popped from it, its root visited and its children pushed on the stack from right to left. If the tree is infinite, depth-first traversal does not visit all nodes in most cases. In particular, if the left-most branch is infinite, the algorithm will be confined to traverse this branch. (It visits all nodes if and only if all branches different from the right-most branch are finite.)

The described problem does not occur with breadth-first traversal. The latter means that it first visits the root, then the roots of all immediate subtrees from left to right[2], then in turn the roots of their immediate subtrees from left to right, etc. An efficient implementation is given by way of an efficiently implemented first-in, first-out queue (FIFO). The description of the algorithm is as before for depth-first traversal, but now with the FIFO operations. However, the immediate subtrees of the currently treated tree are put into the queue from left to right.

While these algorithms are easy to provide in imperative languages with worst-case linear execution time, functional programming languages only easily provide amortized linear execution time for the breadth-first traversal. (In functional programming, the "traversal" is replaced by the task to construct the list of all node labels in the order the imperative algorithm would traverse them.) Okasaki [12] presented for the first time an elegant and worst-case constant-time functional implementation of FIFO, thus yielding worst-case linear-time breadth-first traversal. However, there are also different functional implementations with worst-case linear time [8].

This paper is about breadth-first traversal in a functional programming language, but efficiency is not the concern here. Instead, we explore an algorithm for breadth-first traversal invented by Martin Hofmann, as presented in his posting [6] to the `TYPES forum` mailing list. In a draft [7], Martin Hofmann shows how he crafted the data type on which his proposal is based. There one also finds a sketch of a correctness proof by induction over binary trees.

We will first explain what is so special about Hofmann's algorithm. In dependent type theory one normally wants all programs to be terminating, i. e., the terms to be strongly normalizing. A well-established way of ensuring strong normalization is to restrict recursion to structural recursion on inductive structures obtained as least fixed points of monotone operators. Monotonicity is usually replaced by the stronger syntactic condition of positivity, which means that the expression that describes the operation must have its formal parameter at positive positions only. Positivity does not exclude going twice to the left of the arrow for the function type – only strict positivity would forbid that, but that latter is imposed in most implementations of type theory, including the Coq system and Agda. Non-strictly positive data types may not have a naive set-theoretic semantics [15], but they exist well in system F [3], i. e., polymorphic lambda-calculus [14], where they can be encoded as weakly initial algebras, in other words, as data types with constructors together with an iterator for programming structurally recursive functions. As evaluation in system F is strongly normalizing, all those structurally recursive programs are terminating.

[1] The choice of the left direction is only for definiteness of our description.
[2] This is again just for definiteness.

Hofmann's algorithm is based on the following non-strictly positive data type (our notation):

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{Inductive } \mathsf{Rou} :=} \\
\mid \ \mathsf{Over} & : & \mathsf{Rou} \\
\mid \ \mathsf{Next} & : & ((\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}) \to \mathsf{Rou}
\end{array}
$$

Rou stands for "routine", and there is the constructor Over for the routine that is not executing further, and the crucial non-strictly positive constructor Next that takes a functional of type $(\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}$ as argument to yield a composite routine.[3] Rou appears at a position twice to the left of $\to$ in the type of the argument, hence positively. As we mentioned above, such inductive definitions are ruled out in most proof assistants, notably in the Coq system and in Agda.

While there is a generic iterator for Rou in system F – as mentioned before – the recursive functions needed for the algorithm are not all instances of the iterator. Functions that would calculate the same values can be defined by iteration, but they would not reflect the algorithm properly. However, this shortcoming can be solved by using recursive functions in the style of Mendler [11] which can be provided by a (mild) extension of system F. A detailed account of these issues, which also settles the question of termination, is given in Sect. 5. Besides that, the paper concentrates on different correctness proofs, most of them based on simulations by related algorithms using different intermediate data types, with the aim to reveal and explain the internal structure of Hofmann's algorithm and to replace the impredicative type Rou by a predicative type while preserving the structural characteristics of the original algorithm.

Overview of the paper: After presenting an executable specification of breadth-first traversal as the concatenation of all levels (niveaux) of a tree (Sect. 2) we introduce the data type of routines and Hofmann's algorithm $\mathsf{breadthfirst}$ (Sect. 3) and prove its partial correctness (i. e., correctness assuming termination) following Hofmann's proof sketch (Sect. 4). Termination is proven in Sect. 5 by implementing the functions and data types in the strongly normalising extension of system F by Mendler-style recursion.

Having thus set the stage, we dive into the analysis of Hofmann's algorithm. We begin with a correctness proof (Sect. 6) based on a non-strictly positive inductive representation relation between routines and double lists (lists of lists) that does not require auxiliary functions. This proof does not require extensionality which is a natural prerequisite for Hofmann's correctness proof. Next we present a proof based on the natural extension of breadth-first traversal to forests (lists of trees) providing interesting insight into the internal structure of Hofmann's algorithm (Sect. 7). We give a meaning to the routine corresponding to a forest ts. It is the routine $(\mathsf{c}\,ts)$ computing the traversal of a forest ts while recursively calling $(\mathsf{c}\,(\mathsf{sub}\,ts))$ for the immediate subforest $(\mathsf{sub}\,ts)$ of ts. The function $\mathsf{extract}$ evaluates these recursive functions, and the function br in Hofmann's algorithm, that initially seems to be mysterious, is decoded as an operation which computes $(\mathsf{c}\,(t :: ts))$ from $(\mathsf{c}\,ts)$ and t.

Building on this insight we construct two predicative versions of this algorithm. The first one introduced in Sect. 8 is based on the observation that the routines occurring in the algorithm can be represented as lists of functions $\mathsf{List}\,\mathbb{N} \to \mathsf{List}\,\mathbb{N}$. Therefore we can replace the impredicative data type Rou by the predicative type $\mathsf{Rou}' := \mathsf{List}\,(\mathsf{List}\,\mathbb{N} \to \mathsf{List}\,\mathbb{N})$.

[3] $\mathsf{List}\,\mathbb{N}$ is the type of lists of natural numbers which are taken here for simplicity; any list type would be fine. The data type is tailor-made to our breadth-first traversal problem that requires to compute an element of $\mathsf{List}\,\mathbb{N}$.

Meaning is given to the routine corresponding to a forest ts as the routine $\mathsf{traverse}\,ts : \mathsf{Rou}'$ which is the list of functions appending the levels of the forest. As before, the function br' corresponding to br computes $(\mathsf{traverse}\,(t :: ts))$ from $(\mathsf{traverse}\,ts)$. The second predicative version (Sect. 9) observes that the functions in Rou' constructed in the algorithms are append functions, i. e., functions of the form $\lambda l\,.\,l' +\!\!+\, l$. They can be represented as lists of natural numbers, so we can replace Rou' by the simpler type $\mathsf{Rou}'' := \mathsf{List}^2\,\mathbb{N}$ of double lists. These double lists correspond to the list of levels in the specification of breadth-first traversal.

The findings are summarized in Sect. 10 where we show that the various algorithms and proofs all have the structure of a "simulation of systems". In addition we show that the two predicative algorithms provide a splitting of Hofmann's algorithm into two simpler phases. We round the paper off with a discussion of and pointers to the implementation and formalization of our work in the proof assistants Coq and Agda, highlighting the difficulties caused by non-strict positivity and how to overcome them (Sect. 11), and conclude with a reflection on what was achieved and an outlook to a possible extension of the domain of the algorithms to infinite trees.

2 Specification of breadth-first traversal

We fix the simplest setting to express the task of programming breadth-first traversal: our trees are not arbitrarily finitely branching but just binary, and they are even finite. As did Hofmann, we put labels on the inner nodes and the leaves. For simplicity, we restrict the type of labels to be the natural numbers but any other type could be used instead.

$$\begin{array}{lll} \text{Inductive } \mathsf{Tree} := & & \\ \mid\ \mathsf{Leaf} & : & \mathbb{N} \to \mathsf{Tree} \\ \mid\ \mathsf{Node} & : & \mathsf{Tree} \to \mathbb{N} \to \mathsf{Tree} \to \mathsf{Tree} \end{array}$$

We use the typing conventions

$$\begin{array}{lll} n & : & \mathbb{N} \\ l & : & \mathsf{List}\,\mathbb{N} \\ ls & : & \mathsf{List}^2\mathbb{N} \overset{\text{Def}}{=} \mathsf{List}\,(\mathsf{List}\,\mathbb{N}) \\ t, tl, tr & : & \mathsf{Tree}\quad (tl \text{ and } tr \text{ are typically used for the left and right subtree, respectively}) \end{array}$$

An extended use is made of the auxiliary function zip that "zips" the successive lists in both arguments using the append function for lists (denoted by $+\!\!+$). More precisely, our zip behaves like `zipWith (++)` (with `zipWith` in the Haskell basic library, and `(++)` the Haskell notation for append viewed as a function) for arguments of equal lengths but if lengths differ zip extends the shorter argument with empty lists wheras `zipWith (++)` truncates the longer argument.

$$\mathsf{zip} : \mathsf{List}^2\mathbb{N} \to \mathsf{List}^2\mathbb{N} \to \mathsf{List}^2\mathbb{N}$$
$$\mathsf{zip}\,[]\,ls = ls \qquad \mathsf{zip}\,(l :: ls)\,[] = l :: ls \qquad \mathsf{zip}\,(l :: ls)\,(l' :: ls') = (l +\!\!+ l') :: \mathsf{zip}\,ls\,ls'$$

▶ **Lemma 1** (basic properties of zip).

(a) $\mathsf{zip}\,ls\,[] = ls$.

(b) $\mathsf{zip}\,ls_1\,(\mathsf{zip}\,ls_2\,ls_3) = \mathsf{zip}\,(\mathsf{zip}\,ls_1\,ls_2)\,ls_3$.

We create the list of labels for every horizontal section of the tree, starting with its root (niv refers to the French word "niveaux" for levels – the function collects the labels level-wise).

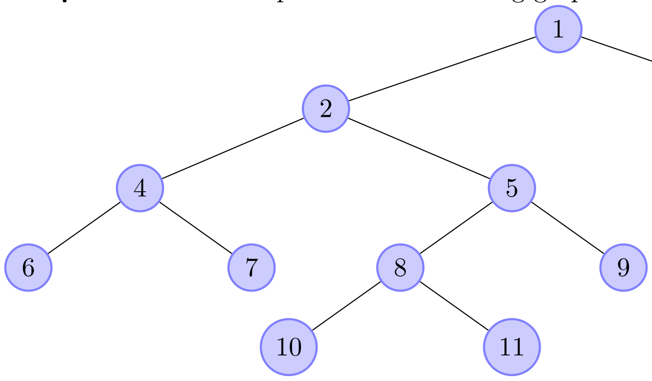
$$\mathsf{niv} : \mathsf{Tree} \to \mathsf{List}^2\mathbb{N}$$
$$\mathsf{niv}\,(\mathsf{Leaf}\,n) = [[n]] \qquad \mathsf{niv}\,(\mathsf{Node}\,t_1\,n\,t_2) = [n] :: \mathsf{zip}\,(\mathsf{niv}\,t_1)\,(\mathsf{niv}\,t_2)$$

From the definition, we see that niv is compositional, which the breadth-first traversal function is not (as also remarked in Hofmann's draft [7]). The latter is defined as follows:

$$\mathsf{breadthfirst}_{\mathsf{spec}} : \mathsf{Tree} \to \mathsf{List}\,\mathbb{N}$$
$$\mathsf{breadthfirst}_{\mathsf{spec}}\,t = \mathsf{flatten}\,(\mathsf{niv}\,t)$$

Here, $\mathsf{flatten} : \mathsf{List}^2\mathbb{N} \to \mathsf{List}\,\mathbb{N}$ denotes concatenation of all those lists (the monad multiplication of the list monad). We do not consider this description of $\mathsf{breadthfirst}_{\mathsf{spec}}$ as an implementation but as an executable specification.

▶ **Example 2.** Let t correspond to the following graphical representation:



Then $\mathsf{niv}\,t = [[1],[2,3],[4,5],[6,7,8,9],[10,11]]$ and $\mathsf{breadthfirst}\,t = [1,\ldots,11]$.

3 Definition of breadth-first traversal via routines

We again show the type Martin Hofmann came up with in his 1993 posting [6]:

Inductive Rou :=
| Over : Rou
| Next : $((\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}) \to \mathsf{Rou}$

The names of the constructors are not those chosen by Hofmann but were suggested to us by Olivier Danvy (since they are used for programming with coroutines). A routine of the form $(\mathsf{Next}\,f)$ comes with a functional f of type $(\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}$ whose argument can be seen as a "continuation", and $f\,k$, with k such a continuation, denotes a list that could be the result of our breadth-first traversal problem. In general, elements of Rou should be seen as encapsulations of routines for the computation of lists of natural numbers.

We use the typing conventions

c	:	Rou	(routines)
k	:	$\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}$	(continuations)
f	:	$(\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}$	

We define the following function (called apply by Hofmann) naively by pattern matching on its first argument and show that this is a legal definition of a terminating function below in Section 5:

$$\mathsf{unfold} : \mathsf{Rou} \to (\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}$$
$$\mathsf{unfold}\,\mathsf{Over} = \lambda k\,.\,k\,\mathsf{Over} \qquad \mathsf{unfold}\,(\mathsf{Next}\,f) = f$$

The name unfold seems justified (and more intuitive than Hofmann's choice of name) for the second case of the definition since it unfolds ($\mathsf{Next}\, f$) to its argument f. Unfolding Over is curious since it yields again an expression involving Over.

The traversal algorithm is expressed as a transformation on routines, instructed by the tree argument. It is by plain iteration on that tree argument ($\circ$ denotes composition of functions).

$$\begin{aligned}
&\mathsf{br} : \mathsf{Tree} \to \mathsf{Rou} \to \mathsf{Rou} \\
&\mathsf{br}\,(\mathsf{Leaf}\, n)\, c &&= \mathsf{Next}\,(\lambda k\,.\, n :: \mathsf{unfold}\, c\, k) \\
&\mathsf{br}\,(\mathsf{Node}\, tl\, n\, tr)\, c &&= \mathsf{Next}\,\big(\lambda k\,.\, n :: \mathsf{unfold}\, c\,(k \circ \mathsf{br}\, tl\, \circ\, \mathsf{br}\, tr)\big)
\end{aligned}$$

We define a function $\mathsf{extract}$ which computes a result from a given routine. Again, we naively define this function by pattern matching on the inductive type of routines, but we here allow ourselves a recursive call, as follows:

$$\begin{aligned}
&\mathsf{extract} : \mathsf{Rou} \to \mathsf{List}\,\mathbb{N} \\
&\mathsf{extract}\, \mathsf{Over} = [\,] \qquad \mathsf{extract}\,(\mathsf{Next}\, f) = f\, \mathsf{extract}
\end{aligned}$$

What is noteworthy here is that the recursive call is not to $\mathsf{extract}$ with some term smaller than ($\mathsf{Next}\, f$) in any sense. The term $\mathsf{extract}$ is even fed in as an argument to the term f, which is type-correct since $\mathsf{extract}$ is of the type of a continuation. In Section 5, we will show that this is a plain form of iteration, thus ensuring termination and well-definedness. As we are doing for unfold, we currently view the equations for $\mathsf{extract}$ as a specification, which allows us to carry out verification in the next section.

Hofmann's algorithm calculates the routine transformer br for the given tree, applies it to the trivial routine and then extracts the result from the output routine:

$$\begin{aligned}
&\mathsf{breadthfirst} : \mathsf{Tree} \to \mathsf{List}\,\mathbb{N} \\
&\mathsf{breadthfirst}\, t = \mathsf{extract}(\mathsf{br}\, t\, \mathsf{Over})
\end{aligned}$$

Of course, we have to make sure that $\mathsf{breadthfirst}$ is a total function and that its results agree with those of $\mathsf{breadthfirst}_{\mathsf{spec}}$.

4 Martin Hofmann's verification of partial correctness

Here, we follow the sketch in Hofmann's notes [6] and argue how functional correctness (i. e., the algorithm's result meets the specification) follows from the equational specification of unfold and $\mathsf{extract}$ and the definitions of the other functions (br and those used for the executable specification in Section 2).

We define a routine transformer that is instructed by a double list, by plain iteration on that list.

$$\begin{aligned}
&\gamma : \mathsf{List}^2\mathbb{N} \to \mathsf{Rou} \to \mathsf{Rou} \\
&\gamma\,[\,]\, c \;=\; c \qquad \gamma\,(l :: ls)\, c \;=\; \mathsf{Next}\Big(\lambda k\,.\, l +\!\!+ \big(\mathsf{unfold}\, c\,(k \circ \gamma\, ls)\big)\Big)
\end{aligned}$$

The following three lemmas (stated in Hofmann's notes [6] without their simple proofs shown below) on the function γ are all the preparations needed for the proof of functional correctness (cf. Theorem 6).

▶ **Lemma 3.** $\mathsf{extract}\,(\gamma\, ls\, \mathsf{Over}) = \mathsf{flatten}\, ls$.

Proof. Induction on ls.
$\mathsf{extract}\,(\gamma\,[]\,\mathsf{Over}) = \mathsf{extract}\,\mathsf{Over} = [] = \mathsf{flatten}\,[]\ .$

$\mathsf{extract}\,(\gamma\,(l :: ls)\,\mathsf{Over}) = \mathsf{extract}\,(\mathsf{Next}\,(\lambda k\,.\,l +\!\!+ (\mathsf{unfold}\,\mathsf{Over}\,(k \circ \gamma\,ls))))$
$= l +\!\!+ ((\mathsf{extract} \circ \gamma\,ls)\,\mathsf{Over}) \overset{\mathrm{IH}}{=} l +\!\!+ \mathsf{flatten}\,ls = \mathsf{flatten}(l :: ls)\ .$ ◀

By $\overset{ext}{=}$ we denote extensional, i. e., pointwise, equality of functions. The following lemma uses two instances of the principle of extensionality. The first states that functions $f : (\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}$ respect extensional equality, i. e., $k \overset{ext}{=} k'$ implies $f\,k = f\,k'$. The second states extensionality of the constructor $\mathsf{Next} : ((\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}) \to \mathsf{Rou}$ (w. r. t. extensional equality of its argument). The following two lemmas (4 and 5) and consequently Theorem 6 depend on extensionality for their proofs.

▶ **Lemma 4.** $\gamma\,ls \circ \gamma\,ls' \overset{ext}{=} \gamma\,(\mathsf{zip}\,ls\,ls')$.

Proof. Induction on ls and ls'.
$\gamma\,[] \circ \gamma\,ls' \overset{ext}{=} \gamma\,ls' = \gamma\,(\mathsf{zip}\,[]\,ls')\ .$

$\gamma\,ls \circ \gamma\,[] \overset{ext}{=} \gamma\,ls = \gamma\,(\mathsf{zip}\,ls\,[])\ .$

$$\begin{aligned}
&\gamma\,(l :: ls)\,(\gamma\,(l' :: ls')\,c)\\
&\quad= \gamma\,(l :: ls)\,(\mathsf{Next}\,(\lambda k'\,.\,l' +\!\!+ (\mathsf{unfold}\,c\,(k' \circ \gamma\,ls'))))\\
&\quad= \mathsf{Next}\,(\lambda k\,.\,l +\!\!+ (\mathsf{unfold}\,(\mathsf{Next}\,(\lambda k'\,.\,l' +\!\!+ (\mathsf{unfold}\,c\,(k' \circ \gamma\,ls')))))\,(k \circ \gamma\,ls))\\
&\quad= \mathsf{Next}\,(\lambda k\,.\,l +\!\!+ (l' +\!\!+ (\mathsf{unfold}\,c\,(k \circ \gamma\,ls \circ \gamma\,ls'))))\\
&\quad= \mathsf{Next}\,(\lambda k\,.\,l +\!\!+ (l' +\!\!+ (\mathsf{unfold}\,c\,(k \circ \gamma\,(\mathsf{zip}\,ls\,ls'))))) \quad \text{(by ind. hyp. and extensionality)}\\
&\quad= \gamma\,((l +\!\!+ l') :: \mathsf{zip}\,ls\,ls')\,c \quad \text{(by associativity of } +\!\!+\text{)}\\
&\quad= \gamma\,(\mathsf{zip}\,(l :: ls)\,(l' :: ls'))\,c\ .
\end{aligned}$$ ◀

▶ **Lemma 5.** $\mathsf{br}\,t \overset{ext}{=} \gamma\,(\mathsf{niv}\,t)$.

Proof. Induction on t.
$$\begin{aligned}
\mathsf{br}\,(\mathsf{Leaf}\,n)\,c &= \mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,c\,k) = \mathsf{Next}\,(\lambda k\,.\,[n] +\!\!+ (\mathsf{unfold}\,c\,k)) = \gamma\,[[n]]\,c\\
&= \gamma\,(\mathsf{niv}\,(\mathsf{Leaf}\,n))\,c\ .
\end{aligned}$$
$$\begin{aligned}
&\mathsf{br}\,(\mathsf{Node}\,t_1\,n\,t_2)\,c = \mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,c\,(k \circ \mathsf{br}\,t_1 \circ \mathsf{br}\ t_2))\\
&\overset{\text{IH, extensionality}}{=} \mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,c\,(k \circ \gamma\,(\mathsf{niv}\,t_1) \circ \gamma\,(\mathsf{niv}\,t_2)))\\
&\overset{\text{Lem. 4, extensionality}}{=} \mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,c\,(k \circ \gamma\,(\mathsf{zip}\,(\mathsf{niv}\,t_1)\,(\mathsf{niv}\,t_2))))\\
&= \gamma\,([n] :: \mathsf{zip}\,(\mathsf{niv}\,t_1)\,(\mathsf{niv}\,t_2))\,c = \gamma\,(\mathsf{niv}\,(\mathsf{Node}\,t_1\,n\,t_2))\,c\ .
\end{aligned}$$ ◀

From these lemmas, we now directly (without further inductive arguments) obtain the main result of this section.

▶ **Theorem 6.** $\mathsf{breadthfirst} \overset{ext}{=} \mathsf{breadthfirst}_{\mathsf{spec}}$, *i. e., for all trees t, we have* $\mathsf{breadthfirst}\,t = \mathsf{breadthfirst}_{\mathsf{spec}}\,t$.

Proof. $\mathsf{breadthfirst}\,t = \mathsf{extract}\,(\mathsf{br}\,t\,\mathsf{Over}) \overset{\text{Lem. 5}}{=} \mathsf{extract}\,(\gamma\,(\mathsf{niv}\,t)\,\mathsf{Over})$
$\overset{\text{Lem. 3}}{=} \mathsf{flatten}\,(\mathsf{niv}\,t) = \mathsf{breadthfirst}_{\mathsf{spec}}\,t\ .$ ◀

This completes the proof based on the sketch by Martin Hofmann.

5 Termination of Hofmann's algorithm

In his 1993 posting [6] Martin Hofmann argued about the existence of the functions unfold and $\mathsf{extract}$ through an impredicative encoding of data types in system F, equipped with parametric equality (equality that is defined as a logical relation by induction over the type of terms being equated, which is impredicative for the case of the universal quantifier). This is, in our opinion, not fully satisfactory, since a verification with parametric equality only shows the existence of a function that yields breadth-first traversal but does not verify the termination of the algorithm itself that is expressed by the defining equations.

Like Martin Hofmann, we are heading for a language-based termination guarantee: We implement the data types and functions of this algorithm in system F extended by Mendler-style recursion, which is known to be strongly normalising. In fact, all relevant data types (including Rou) and all functions defined by iteration can be defined in plain system F in the usual way [4]. Mendler's extension is only needed to properly model the algorithmic behaviour of the function unfold.

We begin with the system F encodings of the type Rou and the function $\mathsf{extract}$ as an example of a plain iteration, since in these cases the encoding is very similar to Mendler's encoding.

If we strip off the names of the constructors so as to fit into the scheme of categorical data types[4], we get Rou as least fixed point of the "functor" RouF, defined on types by

$$\mathsf{RouF}\,A := 1 + ((A \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}) \ \ ,$$

with the one-element type 1 (a. k. a. unit type with only inhabitant $*$) and the type constructor $+$ for disjoint sums (with injections inl and inr and case analysis operator $[s_0, s_1] : A_0 + A_1 \to C$ for $s_i : A_i \to C$, $i = 0, 1$). Clearly, the type A only occurs at a non-strictly positive position in the right-hand side. The usual impredicative encoding of least fixed points in system F (also called "Church encoding") yields as least fixed point of RouF

$$\mathsf{Rou}_{\mathsf{Imp}} := \forall A\,.\,(\mathsf{RouF}\,A \to A) \to A \ \ .$$

Iteration over Rou is then given by "catamorphisms" for RouF-algebras since Rou itself is the carrier of the initial RouF-algebra. Beware that initiality holds only with respect to a categorical semantics. Computationally, one only gets weak initiality, that is, the existence but not the uniqueness of the morphism (given by the iterator) in the standard commuting diagram for initial algebras. Moreover, the single[5] equation expressed by the commuting diagram is computationally directed: we will later use the symbol $\rhd^*$ for that relation, instead of the symmetric $=$ that appears in traditional categorical modeling.

This weak initiality principle already captures the behaviour of $\mathsf{extract}$ (but we will have to define $\mathsf{extract}$ differently later since also unfold needs to be taken care of). The details are as follows: We define the iterator

$$\mathsf{RouIt} : \forall A\,.\,(\mathsf{RouF}\,A \to A) \to \mathsf{Rou}_{\mathsf{Imp}} \to A \qquad \mathsf{RouIt}\,A\,s\,t = t\,A\,s$$

Due to positivity of RouF, there is a closed term $\mathsf{RouFmap}$, defined by case analysis on

[4] In the Haskell programming language, we would keep the constructors and define `data RouF a = Over | Next ((a -> List Nat) -> List Nat)`.

[5] before we make informal use of pattern matching that splits the rule into two rules

the sum as follows (slightly informally, for readability):

$$\begin{aligned}
&\mathsf{RouFmap} : \forall A, B\,.\,(A \to B) \to \mathsf{RouF}\,A \to \mathsf{RouF}\,B\\
&\mathsf{RouFmap}\,A\,B\,h^{A\to B}\,(\mathsf{inl}\,u^1) &&= \quad \mathsf{inl}\,u\\
&\mathsf{RouFmap}\,A\,B\,h^{A\to B}\,(\mathsf{inr}\,f^{(A\to\mathsf{List}\,\mathbb{N})\to\mathsf{List}\,\mathbb{N}}) &&= \quad \mathsf{inr}\,\big(\lambda k^{B\to\mathsf{List}\,\mathbb{N}}\,.\,f\,(k\circ h)\big)
\end{aligned}$$

This allows us to define the RouF-algebra $\mathsf{foldRou_{Imp}}$ with carrier $\mathsf{Rou_{Imp}}$:

$$\begin{aligned}
&\mathsf{foldRou_{Imp}} : \mathsf{RouF}\,\mathsf{Rou_{Imp}} \to \mathsf{Rou_{Imp}}\\
&\mathsf{foldRou_{Imp}}\,t\,A\,s = s\,\big(\mathsf{RouFmap}\,\mathsf{Rou_{Imp}}\,A\,(\mathsf{RouIt}\,A\,s)\,t\big)\quad.
\end{aligned}$$

The impredicative implementations of the constructors, $\mathsf{Over_{Imp}}$ and $\mathsf{Next_{Imp}}$, are now instances of $\mathsf{foldRou_{Imp}}$:

$$\begin{aligned}
&\mathsf{Over_{Imp}} &&:= &&\mathsf{foldRou_{Imp}}\,(\mathsf{inl}\,*) &&: &&\mathsf{Rou_{Imp}}\\
&\mathsf{Next_{Imp}} &&:= &&\mathsf{foldRou_{Imp}}\circ\mathsf{inr} &&: &&((\mathsf{Rou_{Imp}}\to\mathsf{List}\,\mathbb{N})\to\mathsf{List}\,\mathbb{N})\to\mathsf{Rou_{Imp}}
\end{aligned}$$

For convenience, we define ($\lambda\_$ is a void abstraction over unit type):

$$\begin{aligned}
&\mathsf{RouIt_{Imp}} : \forall A\,.\,A \to (((A\to\mathsf{List}\,\mathbb{N})\to\mathsf{List}\,\mathbb{N})\to A)\to\mathsf{Rou_{Imp}}\to A\\
&\mathsf{RouIt_{Imp}}\,A\,s_0\,s_1 = \mathsf{RouIt}\,A\,[\lambda\_\,.\,s_0\,,\,s_1]
\end{aligned}$$

We will write $\rhd$ for the one-step reduction relation of system F and $\rhd^*$ for its reflexive transitive closure. The characteristic reduction behaviour of $\mathsf{RouIt_{Imp}}$ is given by

$$\begin{aligned}
&\mathsf{RouIt_{Imp}}\,A\,s_0\,s_1\,\mathsf{Over_{Imp}} &&\rhd^* &&s_0\\
&\mathsf{RouIt_{Imp}}\,A\,s_0\,s_1\,(\mathsf{Next_{Imp}}\,f) &&\rhd^* &&s_1\Big(\lambda k^{A\to\mathsf{List}\,\mathbb{N}}.\,f\big(k\circ(\mathsf{RouIt_{Imp}}\,A\,s_0\,s_1)\big)\Big)
\end{aligned}$$

We can implement $\mathsf{extract}$, using the iterator with $A := \mathsf{List}\,\mathbb{N}$:

$$\begin{aligned}
&\mathsf{extract_{Imp}} : \mathsf{Rou_{Imp}} \to \mathsf{List}\,\mathbb{N}\\
&\mathsf{extract_{Imp}} = \mathsf{RouIt_{Imp}}\,(\mathsf{List}\,\mathbb{N})\,[]\,\big(\lambda g^{(\mathsf{List}\,\mathbb{N}\to\mathsf{List}\,\mathbb{N})\to\mathsf{List}\,\mathbb{N}}\,.\,g(\lambda l\,.\,l)\big)
\end{aligned}$$

and obtain proper recursive behaviour with three subsequent steps of β-reduction and one η-reduction step (that can be assumed in Church-style versions of system F):

$$\mathsf{extract_{Imp}}\,\mathsf{Over_{Imp}} \rhd^* [] \qquad \mathsf{extract_{Imp}}\,(\mathsf{Next_{Imp}}\,f) \rhd^* f\,\mathsf{extract_{Imp}}$$

The equational specification of unfold may seem innocuous, but Harper and Mitchell [5] have shown that even rewrite rules that just have the form of a projection may break termination when added to system F. Consider the type $S := \forall A, B\,.\,(A \to A) \to B \to B$, which is trivially inhabited by a term that maps constantly to the identity on B. A different inhabitant J' of S is *added* to system F, and the reduction relation of system F is extended by a specific rule for J': $J'\,A\,A\,f^{A\to A} \rhd f$ for any type A. It is easy to construct a term in this extension that rewrites in several steps to itself, hence creating an infinite loop.[6] However, unfold *is* terminating, albeit not for trivial reasons.

We use the extension of system F by Mendler-style recursion which is strongly normalizing [11]. Already Mendler's original work accommodates non-strictly positive inductive types, as our Rou, but it was later shown that even that restriction to positivity is not necessary for

[6] This is also presented in detail in a paper by the second author [10, p.122], together with a discussion of a variant of the scheme of inductive types with iteration for which termination fails.

strong normalization (see [9, Section 6.1.1] for a semantic and [1] for a syntactic proof). We describe only the instance of Mendler-style primitive recursion that governs the data type $\mathsf{Rou}_{\mathsf{Men}}$, which is the one obtained for RouF. Mendler's extension permits the construction of a RouF-algebra $\mathsf{foldRou}_{\mathsf{Men}}$ with carrier $\mathsf{Rou}_{\mathsf{Men}} \overset{\text{Def}}{=} \mu\,\mathsf{RouF}$ (with μ in the sense of Mendler), i. e., we have

$$\mathsf{foldRou}_{\mathsf{Men}} : \mathsf{RouF}\,\mathsf{Rou}_{\mathsf{Men}} \to \mathsf{Rou}_{\mathsf{Men}} \text{ with recursor } \mathsf{RouRec} : \forall A\,.\,\mathsf{Step}_{\mathsf{Men}}\,A \to \mathsf{Rou}_{\mathsf{Men}} \to A$$

where the type of step functions is

$$\mathsf{Step}_{\mathsf{Men}}A := \forall X\,.\,(X \to \mathsf{Rou}_{\mathsf{Men}}) \to (X \to A) \to \mathsf{RouF}\,X \to A \ \ .$$

A step function $s : \mathsf{Step}_{\mathsf{Men}}\,A$ transforms a function $X \to A$ into a function $\mathsf{RouF}\,X \to A$, possibly using a function $X \to \mathsf{Rou}_{\mathsf{Men}}$. RouRec takes a step function and then transforms elements of $\mathsf{Rou}_{\mathsf{Men}}$ into elements of A. We have the rewrite rule

$$\mathsf{RouRec}\,A\,s\,(\mathsf{foldRou}_{\mathsf{Men}}\,t) \rhd s\,\mathsf{Rou}_{\mathsf{Men}}\,(\lambda x^{\mathsf{Rou}_{\mathsf{Men}}}\,.\,x)\,(\mathsf{RouRec}\,A\,s)\,t \ \ .$$

The individual constructors for $\mathsf{Rou}_{\mathsf{Men}}$ are obtained as in the impredicative encoding: $\mathsf{Over}_{\mathsf{Men}} := \mathsf{foldRou}_{\mathsf{Men}}\,(\mathsf{inl}\,*)$ and $\mathsf{Next}_{\mathsf{Men}}\,f := \mathsf{foldRou}_{\mathsf{Men}}\,(\mathsf{inr}\,f)$. Define the step terms for $\mathsf{extract}$ and unfold as follows (which could be mapped to terms of system F with unit and sum types):

$$\begin{array}{lcl}
\mathsf{s}_{\mathsf{extract}} : \mathsf{Step}_{\mathsf{Men}}\,(\mathsf{List}\,\mathbb{N}) & & \\
\mathsf{s}_{\mathsf{extract}}\,X\,i^{X\to\mathsf{Rou}_{\mathsf{Men}}}\,r^{X\to\mathsf{List}\,\mathbb{N}}\,(\mathsf{inl}\,u^1) & = & [] \\
\mathsf{s}_{\mathsf{extract}}\,X\,i^{X\to\mathsf{Rou}_{\mathsf{Men}}}\,r^{X\to\mathsf{List}\,\mathbb{N}}\,(\mathsf{inr}\,f^{(X\to\mathsf{List}\,\mathbb{N})\to\mathsf{List}\,\mathbb{N}}) & = & f\,r \\
\mathsf{s}_{\mathsf{unfold}} : \mathsf{Step}_{\mathsf{Men}}\,\mathsf{A}_{\mathsf{unfold}} \quad \text{where } \ \mathsf{A}_{\mathsf{unfold}} := (\mathsf{Rou}_{\mathsf{Men}} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N} & & \\
\mathsf{s}_{\mathsf{unfold}}\,X\,i^{X\to\mathsf{Rou}_{\mathsf{Men}}}\,r^{X\to A_{\mathsf{unfold}}}\,(\mathsf{inl}\,u^1) & = & \lambda k\,.\,k\,\mathsf{Over}_{\mathsf{Men}} \\
\mathsf{s}_{\mathsf{unfold}}\,X\,i^{X\to\mathsf{Rou}_{\mathsf{Men}}}\,r^{X\to A_{\mathsf{unfold}}}\,(\mathsf{inr}\,f^{(X\to\mathsf{List}\,\mathbb{N})\to\mathsf{List}\,\mathbb{N}}) & = & \lambda k\,.\,f\,(k\circ i)
\end{array}$$

Define the Mendler-style implementations:

$$\begin{array}{lclllcl}
\mathsf{extract}_{\mathsf{Men}} & : & \mathsf{Rou}_{\mathsf{Men}} \to \mathsf{List}\,\mathbb{N} & \qquad & \mathsf{extract}_{\mathsf{Men}} & = & \mathsf{RouRec}\,(\mathsf{List}\,\mathbb{N})\,\mathsf{s}_{\mathsf{extract}} \\
\mathsf{unfold}_{\mathsf{Men}} & : & \mathsf{Rou}_{\mathsf{Men}} \to \mathsf{A}_{\mathsf{unfold}} & & \mathsf{unfold}_{\mathsf{Men}} & = & \mathsf{RouRec}\,\mathsf{A}_{\mathsf{unfold}}\,\mathsf{s}_{\mathsf{unfold}}
\end{array}$$

Obviously, $\mathsf{extract}_{\mathsf{Men}}\,\mathsf{Over}_{\mathsf{Men}} \rhd^* []$, $\mathsf{extract}_{\mathsf{Men}}\,(\mathsf{Next}_{\mathsf{Men}}\,f) \rhd^* f\,\mathsf{extract}_{\mathsf{Men}}$ (as for the impredicative implementation) and $\mathsf{unfold}_{\mathsf{Men}}\,\mathsf{Over}_{\mathsf{Men}} \rhd^* \lambda k\,.\,k\,\mathsf{Over}_{\mathsf{Men}}$. Finally,

$$\mathsf{unfold}_{\mathsf{Men}}\,(\mathsf{Next}_{\mathsf{Men}}\,f) \rhd^* \lambda k\,.\,f\,(k\circ(\lambda x\,.\,x)) \rhd^* f \ \ ,$$

where the latter reduction has one β- and two η-reduction steps at the end. Thus, $\mathsf{extract}_{\mathsf{Men}}$ and $\mathsf{unfold}_{\mathsf{Men}}$ are implementations of Hofmann's functions, and the original defining equations become reductions in the sense of $\rhd^*$ of the Mendler-style extension of system F.

Of course, one can also encode any algebraic data types such as lists and trees and functions defined by iteration on elements of such types in Mendler's system. This can be done in a similar (but simpler) way as sketched above for Rou and $\mathsf{extract}$ in plain system F. Moreover, the interpretation is algorithmically faithful to the equational specification of these functions in the sense that the defining equations become one or more term rewriting steps in Mendler's terminating system. In summary we have the following

▶ **Theorem 7.** *The data types and functions involved in Hofmann's algorithm for breadth-first traversal can be algorithmically faithfully interpreted in the strongly normalising system of Mendler-style recursion. Therefore, Hofmann's algorithm is terminating.*

6 Verification by a non-strictly positive inductive relation

We now embark on giving alternative correctness proofs of Hofmann's algorithm. They explore different concepts and provide different intuitions for the correctness of this algorithm (see Section 10 for a mathematical assessment of their relations). The first and mathematically most challenging alternative proof given in this section uses a non-strictly positive inductive relation between routines $c : \mathsf{Rou}$ and double lists $ls : \mathsf{List}^2\mathbb{N}$ that, intuitively, states that c "represents" ls.

First, we define when a continuation k is an *extractor* for a binary relation $R \subseteq \mathsf{Rou} \times \mathsf{List}^2\mathbb{N}$ (seen as a candidate for a representation relation) and an "initial" double list ls'.

$$\mathsf{isextractor}(R, ls', k) \overset{\text{Def}}{\equiv} \forall c, ls'' \,.\, R(c, ls'') \to k\,c = \mathsf{flatten}\,(\mathsf{zip}\,ls'\,ls'') \enspace .$$

Note that R occurs negatively in the formula $\mathsf{isextractor}(R, ls', k)$. In intuitive words, this means that the weaker R is, the more constraints are imposed in order for k to be an extractor for R and ls'. The name "extractor" should convey the intuition that continuation k "extracts" the "right" result for ls'' out of all routines c representing ls'' in the sense of R with initialization ls'. Note that the formula for the prescribed result does not mention the niv operation of the original specification $\mathsf{breadthfirst}_{\mathsf{spec}}$. Lemma 8 below shows that $\mathsf{extract}$ is an extractor for a suitable representation relation R and initialization $ls' = []$.

With this auxiliary concept of extractor (which, after all, is only an abbreviation for a rather short formula of logic) we now define the representation relation $\mathsf{rep} \subseteq \mathsf{Rou} \times \mathsf{List}^2\mathbb{N}$ inductively by two rules. Not surprisingly, rep takes the role of relation R in the foregoing definition. The general relation argument R in the formulation of the notion of an extractor allows us to conveniently express the induction principle for rep (as can be seen in the proof of Lemma 8 below). The inductive definition of rep is as follows:

$$\frac{}{\mathsf{rep}(\mathsf{Over}, [])}\ (\text{over})$$

$$\frac{\forall k, ls' \,.\, \mathsf{isextractor}(\mathsf{rep}, ls', k) \to f\,k = l \mathbin{+\!\!+} \mathsf{flatten}\,(\mathsf{zip}\,ls'\,ls)}{\mathsf{rep}(\mathsf{Next}\,f, l :: ls)}\ (\text{next})$$

where in (next) the variables f, l, ls are implicitly universally quantified. The premise of the rule (next) contains the predicate rep positively (though not strictly positively) and therefore depends monotonically on it. By Tarski's fixed point theorem it follows that the smallest relation rep closed under the rules (over) and (next) exists.

Note that, since the premise of the rule (next) refers only to the result of applying f to k, the predicate rep respects extensional equality in the sense that if $f \overset{ext}{=} f'$, then $\mathsf{rep}(\mathsf{Next}\,f, l :: ls)$ iff $\mathsf{rep}(\mathsf{Next}\,f', l :: ls)$. Therefore, unlike the proofs in the previous section, the proofs of the following lemmas do not depend on extensionality principles.

The recursive function $\mathsf{extract}$, equationally specified in Section 3 as a continuation, is indeed an extractor for rep and the empty list:

▶ **Lemma 8.** $\mathsf{isextractor}(\mathsf{rep}, [], \mathsf{extract})$*, i. e.,* $\forall c, ls \,.\, \mathsf{rep}(c, ls) \to \mathsf{extract}\,c = \mathsf{flatten}\,ls$*.*

Proof. Setting $R_0(c, ls'') \overset{\text{Def}}{\equiv} \mathsf{extract}\,c = \mathsf{flatten}\,ls''$, $\mathsf{isextractor}(\mathsf{rep}, [], \mathsf{extract})$ is equivalent to $\mathsf{rep} \subseteq R_0$. We prove the latter by (non-strictly positive) induction, i. e., we show that the closure conditions (over) and (next) hold if rep is replaced by R_0.

(over): $R_0(\mathsf{Over}, [])$ means $\mathsf{extract}\,\mathsf{Over} = \mathsf{flatten}\,[]$, which holds since both sides equal $[]$.

(next): Assume $\forall k, ls' \,.\, \mathsf{isextractor}(R_0, ls', k) \to f\,k = l \mathbin{+\!\!+} \mathsf{flatten}\,(\mathsf{zip}\,ls'\,ls)$, which is our induction hypothesis. Since, trivially, $\mathsf{isextractor}(R_0, [], \mathsf{extract})$, the induction hypothesis yields $f\,\mathsf{extract} = l \mathbin{+\!\!+} \mathsf{flatten}\,ls$, which is equivalent to our goal, $R_0(\mathsf{Next}\,f, l :: ls)$. ◀

The following lemma shows that $\mathsf{br}\,t$, defined in Section 2 as a routine transformer, is well-behaved w. r. t. representation: if the argument routine c represents a (double) list ls, then the resulting routine represents $\mathsf{zip}\,(\mathsf{niv}\,t)\,ls$: [7]

▶ **Lemma 9.** $\mathsf{rep}(c, ls) \to \mathsf{rep}(\mathsf{br}\,t\,c, \mathsf{zip}\,(\mathsf{niv}\,t)\,ls)$.

Proof. Induction on $t : \mathsf{Tree}$.

Case $t = \mathsf{Leaf}\,n$**:** Assume $\mathsf{rep}(c, ls)$.
We have to show $\mathsf{rep}(\mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,c\,k), \mathsf{zip}\,[[n]]\,ls)$.

Subcase $ls = []$**:** Then $\mathsf{zip}\,[[n]]\,ls = [n] :: []$ and, since $\mathsf{rep}(c, [])$, $c = \mathsf{Over}$. Hence we have to show $\mathsf{rep}\big(\mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,\mathsf{Over}\,k), [n] :: []\big)$, i. e., for all k, ls', if $\mathsf{isextractor}(\mathsf{rep}, ls', k)$, then $n :: k\,\mathsf{Over} = [n] +\!\!+\, \mathsf{flatten}\,(\mathsf{zip}\,ls'\,[])$, i. e., $k\,\mathsf{Over} = \mathsf{flatten}\,ls'$. But the latter is obtained by instantiating the assumption $\mathsf{isextractor}(\mathsf{rep}, ls', k)$ with Over and $[]$.

Subcase $ls = l :: ls_0$**:** Then $\mathsf{zip}\,[[n]]\,ls = (n :: l) :: ls_0$ and, since $\mathsf{rep}(c, l :: ls)$, $c = \mathsf{Next}\,f$ with

$$(+) \qquad \forall k, ls'\,.\,\mathsf{isextractor}(\mathsf{rep}, ls', k) \to f\,k = l +\!\!+\, \mathsf{flatten}\,(\mathsf{zip}\,ls'\,ls_0) \enspace .$$

We have to show that $\mathsf{rep}\big(\mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,(\mathsf{Next}\,f)\,k), (n :: l) :: ls_0\big)$, i. e.,

$$\forall k, ls'\,.\,\mathsf{isextractor}(\mathsf{rep}, ls', k) \to n :: f\,k = (n :: l) +\!\!+\, \mathsf{flatten}\,(\mathsf{zip}\,ls'\,ls_0) \enspace .$$

But, cancelling n, this is exactly (+).

Case $t = \mathsf{Node}\,tl\,n\,tr$**:** By induction hypothesis, for all c, ls with $\mathsf{rep}(c, ls)$ and all $t' \in \{tl, tr\}$, $\mathsf{rep}(\mathsf{br}\,t'\,c, \mathsf{zip}\,(\mathsf{niv}\,t')\,ls)$.

Assume $\mathsf{rep}(c, ls)$. We have to show $\mathsf{rep}(\mathsf{br}\,t\,c, \mathsf{zip}\,(\mathsf{niv}\,t)\,ls)$, i. e.,

$$\mathsf{rep}\big(\mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\;\;c\,(k \circ \mathsf{br}\,tl \circ \mathsf{br}\,tr)), \mathsf{zip}\,([n] :: \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr))\,ls\big) \enspace .$$

Subcase $ls = []$**:** Then $\mathsf{zip}\,([n] :: \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr))\,ls = [n] :: \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)$, and, since $\mathsf{rep}(c, [])$, $c = \mathsf{Over}$. Hence, we have to show that for all k, ls' such that $\mathsf{isextractor}(\mathsf{rep}, ls', k)$ we have $n :: (k \circ \mathsf{br}\,tl \circ \mathsf{br}\,tr)\,\mathsf{Over} = [n] +\!\!+\, \mathsf{flatten}\,(\mathsf{zip}\,ls'\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)))$, i. e.,

$$k\,(\mathsf{br}\,tl\,(\mathsf{br}\,tr\,\mathsf{Over})) = \mathsf{flatten}\,(\mathsf{zip}\,ls'\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr))) \enspace .$$

Using $\mathsf{isextractor}(\mathsf{rep}, ls', k)$, instantiated with
$c := \mathsf{br}\,tl\,(\mathsf{br}\,tr\,\mathsf{Over})$ and $ls'' := \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)$, our goal reduces to showing
$\mathsf{rep}\big(\mathsf{br}\,tl\,(\mathsf{br}\,tr\,\mathsf{Over}), \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)\big)$ which, by the first induction hypothesis, further reduces to $\mathsf{rep}(\mathsf{br}\,tr\,\mathsf{Over}, \mathsf{niv}\,tr)$. Finally, by the second induction hypothesis (with $ls := []$), the latter reduces to (over).

Subcase $ls = l :: ls_0$**:** Then
$\mathsf{zip}\,([n] :: \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr))\,ls = (n :: l) :: \mathsf{zip}\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr))\,ls_0$ and therefore, by the assumption $\mathsf{rep}(c, ls)$, we get $c = \mathsf{Next}\,f$ with

$$(++) \qquad \forall k, ls'\,.\,\mathsf{isextractor}(\mathsf{rep}, ls', k) \to f\,k = l +\!\!+\, \mathsf{flatten}\,(\mathsf{zip}\,ls'\,ls_0) \enspace .$$

We have to show
$\mathsf{rep}\big(\mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,c\,(k \circ \mathsf{br}\,tl \circ \mathsf{br}\,tr)), (n :: l) :: \mathsf{zip}\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr))\,ls_0\big)$,
i. e., for all k, ls' with $\mathsf{isextractor}(\mathsf{rep}, ls', k)$,

$$n :: f\,(k \circ \mathsf{br}\,tl \circ \mathsf{br}\,tr) = (n :: l) +\!\!+\, \mathsf{flatten}\big(\mathsf{zip}\,ls'\,(\mathsf{zip}\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr))\,ls_0)\big) \enspace .$$

[7] This descriptional pattern suggests to define representation of double list transformers by routine transformers in the usual style of logical relations. With that definition in place, the lemma could be stated as representation of $\mathsf{zip}\,(\mathsf{niv}\,t)$ by $\mathsf{br}\,t$.

Deleting n and using associativity for zip we end up with the goal $f\,(k \circ \mathsf{br}\,tl \circ \mathsf{br}\,tr) = l +\!\!+ \mathsf{flatten}\big(\mathsf{zip}\,(\mathsf{zip}\,ls'\,(\mathsf{zip}\,(\mathsf{niv} tl)\,(\mathsf{niv}\,tr)))\,ls_0\big)$. By $(++)$ it suffices to show

$$\mathsf{isextractor}\big(\mathsf{rep}, \mathsf{zip}\,ls'\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)), k \circ \mathsf{br}\,tl \circ \mathsf{br}\,tr\big).$$

Assume $\mathsf{rep}(c, ls'')$. We have to show

$$k\,(\mathsf{br}\,tl\,(\mathsf{br}\,tr\,c)) = \mathsf{flatten}\,\big(\mathsf{zip}\,(\mathsf{zip}\,ls'\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)))\,ls''\big)\ .$$

By the assumption $\mathsf{isextractor}(\mathsf{rep}, ls', k)$, and using associativity of zip, it suffices to show $\mathsf{rep}\big(\mathsf{br}\,tl\,(\mathsf{br}\,tr\,c), \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{zip}\,(\mathsf{niv}\,tr)\,ls'')\big)$. The first induction hypothesis reduces this to $\mathsf{rep}(\mathsf{br}\,tr\,c, \mathsf{zip}\,(\mathsf{niv}\,tr)\,ls'')$ and the second further to $\mathsf{rep}(c, ls'')$, which holds by assumption. ◀

Alternative proof of Theorem 6:
By the axiom (over), we have $\mathsf{rep}(\mathsf{Over}, [])$. Therefore, by Lemma 9, $\mathsf{rep}(\mathsf{br}\,t\,\mathsf{Over}, \mathsf{niv}\,t)$. Since, by Lemma 8, $\mathsf{isextractor}(\mathsf{rep}, [], \mathsf{extract})$, it follows $\mathsf{extract}\,(\mathsf{br}\,t\,\mathsf{Over}) = \mathsf{flatten}\,(\mathsf{niv}\,t)$, i. e., $\mathsf{breadthfirst}\,t = \mathsf{breadthfirst}_{\mathsf{spec}}\,t$.

7 Verification by interpreting routines as recursive programs

In this section we give a correctness proof, which is based on understanding the elements of Rou as recursive programs. We give a meaning to routines by defining what it means for a routine to compute the breadth-first traversal of a tree, and use this in order to state and prove in Lemma 12 the correctness condition fulfilled by the key operation br.

Following Okasaki [13], one can understand the breadth-first traversal of a tree by understanding the more general notion of the breadth-first traversal of elements of $\mathsf{Forest} := \mathsf{List}\,\mathsf{Tree}$. We use ts (for lists of trees) as variables for forests.

The obvious lifting of $\mathsf{breadthfirst}_{\mathsf{spec}}$ to forests is

$$\mathsf{breadthfirst}_{\mathsf{f,spec}} \overset{\mathrm{Def}}{=} \mathsf{flatten} \circ \mathsf{niv}_{\mathsf{f}} : \mathsf{Forest} \to \mathsf{List}\,\mathbb{N}\ ,$$

where $\mathsf{niv}_{\mathsf{f}}$ zips all $\mathsf{niv}\,t$ for t in ts, i. e.

$$\mathsf{niv}_{\mathsf{f}} : \mathsf{Forest} \to \mathsf{List}^2\,\mathbb{N}$$
$$\mathsf{niv}_{\mathsf{f}}\,[] = [] \qquad \mathsf{niv}_{\mathsf{f}}\,(t :: ts) = \mathsf{zip}\,(\mathsf{niv}\,t)\,(\mathsf{niv}_{\mathsf{f}}\,ts)$$

Clearly, $\mathsf{breadthfirst}_{\mathsf{spec}}\,t = \mathsf{breadthfirst}_{\mathsf{f,spec}}\,[t]$.

It is our goal to prove the correctness of Hofmann's algorithm via an embedding of forests into routines that is in a certain sense simpler than the embedding γ and explains the roles of the functions $\mathsf{br} : \mathsf{Tree} \to \mathsf{Rou} \to \mathsf{Rou}$ and $\mathsf{extract} : \mathsf{Rou} \to \mathsf{List}\,\mathbb{N}$.

Our programs will not recurse on the length of a forest but on its depth, and will access its roots and its immediate subforest:

- $\mathsf{depth} : \mathsf{Tree} \to \mathbb{N}$, $\mathsf{depth}(\mathsf{Leaf}\,n) = 1$, $\mathsf{depth}(\mathsf{Node}\,tl\,n\,tr) = \max\{\mathsf{depth}\,tl\,, \mathsf{depth}\,tr\} + 1$.
- $\mathsf{depth}_{\mathsf{f}} : \mathsf{Forest} \to \mathbb{N}$, $\mathsf{depth}_{\mathsf{f}}\,[t_1, \ldots, t_n] = \max\{0, \mathsf{depth}\,t_1, \ldots, \mathsf{depth}\,t_n\}$.
- $\mathsf{roots} : \mathsf{Forest} \to \mathsf{List}\,\mathbb{N}$
  $\mathsf{roots}\,[] = [] \quad \mathsf{roots}\,(\mathsf{Leaf}\,n :: ts) = \mathsf{roots}\,(\mathsf{Node}\,tl\,n\,tr :: ts) = n :: \mathsf{roots}\,ts$
- $\mathsf{sub} : \mathsf{Forest} \to \mathsf{Forest}$ calculates the immediate subforest:
  $\mathsf{sub}\,[] = [], \mathsf{sub}\,(\mathsf{Leaf}\,n :: ts) = \mathsf{sub}\,ts, \mathsf{sub}\,(\mathsf{Node}\,tl\,n\,tr :: ts) = tl :: tr :: \mathsf{sub}\,ts$.

▶ **Lemma 10.**
(a) $\mathsf{length}\,(\mathsf{niv}_{\mathsf{f}}\,ts) = \mathsf{depth}_{\mathsf{f}}\,ts$.
(b) For $ts \neq []$ we have $\mathsf{depth}_{\mathsf{f}}\,ts = \mathsf{depth}_{\mathsf{f}}\,(\mathsf{sub}\,ts) + 1$.

(c) If $ts \neq []$ then $\mathsf{niv_f}\ ts = \mathsf{roots}\ ts :: \mathsf{niv_f}\ (\mathsf{sub}\ ts)$.

Proof. Easy. ◀

We begin with the observation (which is made precise in Lemma 12 below) that the routines created in a run of the algorithm $\mathsf{breadthfirst}$ are either Over or of the form $(\mathsf{next}\,(\mathsf{addroots}\ ts)\ c)$ where

- $\mathsf{next} : (\mathsf{List}\ \mathbb{N} \to \mathsf{List}\ \mathbb{N}) \to \mathsf{Rou} \to \mathsf{Rou} \qquad \mathsf{next}\ g\ c = \mathsf{Next}\,(\lambda k\,.\,g\,(k\,c))$.
- $\mathsf{addroots} : \mathsf{Forest} \to \mathsf{List}\ \mathbb{N} \to \mathsf{List}\ \mathbb{N} \qquad \mathsf{addroots}\ ts = \mathsf{append}\,(\mathsf{roots}\ ts)$

We can regard these routines as recursive programs: Over is the routine which immediately terminates returning $[]$. The routine $(\mathsf{next}\ g\ c)$ makes a recursive call to c, and if the result returned there is l it returns $(g\,l)$. $\mathsf{extract}$ executes these recursive programs: We have $\mathsf{extract}\ \mathsf{Over} = []$ and $\mathsf{extract}\,(\mathsf{next}\ g\ c) = g\,(\mathsf{extract}\ c)$.

We now construct for $ts : \mathsf{Forest}$ the routine $(\mathsf{c}\ ts)$ which represents the computation of the breadth-first traversal of ts. If $ts = []$, then Over represents the traversal of ts which is $[]$. Otherwise, c represents the traversal of ts if it recursively calls a routine representing the traversal of $(\mathsf{sub}\ ts)$ and adds to the result $(\mathsf{roots}\ ts)$. More formally we define $\mathsf{c}\ ts : \mathsf{Rou}$ by recursion on the measure $\mathsf{depth_f}\ ts$:

$$\mathsf{c}\ ts = \begin{cases} \mathsf{Over} & \text{if } ts = [], \\ \mathsf{next}\,(\mathsf{addroots}\ ts)\,(\mathsf{c}\,(\mathsf{sub}\ ts)) & \text{otherwise.} \end{cases}$$

We show that $\mathsf{extract}$ evaluates the routines $\mathsf{c}\ ts$ to the breadth-first traversal of ts:

▶ **Lemma 11.** $\mathsf{extract} \circ \mathsf{c} \overset{ext}{=} \mathsf{breadthfirst_{f,spec}}$.

Proof. We show $\mathsf{extract}\,(\mathsf{c}\ ts) = \mathsf{breadthfirst_{f,spec}}\ ts$ by induction on $\mathsf{depth_f}\ ts$:
If $\mathsf{depth_f}\ ts = 0$ then $ts = []$, and $\mathsf{extract}\,(\mathsf{c}\ ts) = [] = \mathsf{flatten}\,(\mathsf{niv_f}\ ts) = \mathsf{breadthfirst_{f,spec}}\ ts$.
Otherwise by IH $\mathsf{extract}\,(\mathsf{c}\,(\mathsf{sub}\ ts)) = \mathsf{breadthfirst_{f,spec}}\,(\mathsf{sub}\ ts))$, and therefore, by Lemma 10

$$\begin{aligned} \mathsf{extract}\,(\mathsf{c}\ ts) &= \mathsf{extract}\,(\mathsf{next}\,(\mathsf{addroots}\ ts)\,(\mathsf{c}\,(\mathsf{sub}\ ts))) = \mathsf{addroots}\ ts\,(\mathsf{extract}\,(\mathsf{c}\,(\mathsf{sub}\ ts))) \\ &= \mathsf{roots}\ ts +\!\!+\ \mathsf{flatten}\,(\mathsf{niv_f}\,(\mathsf{sub}\ ts)) = \mathsf{flatten}\,(\mathsf{roots}\ ts :: \mathsf{niv_f}\,(\mathsf{sub}\ ts)) \\ &= \mathsf{flatten}\,(\mathsf{niv_f}\ ts) = \mathsf{breadthfirst_{f,spec}}\ ts\ . \end{aligned}$$ ◀

The next lemma is a key lemma for br. It shows that $(\mathsf{br}\ t\ c)$ translates a routine c computing the traversal of ts into a routine computing the traversal of $(t :: ts)$:

▶ **Lemma 12.** $\mathsf{br}\ t\ \circ \mathsf{c} \overset{ext}{=} \mathsf{c} \circ \mathsf{cons}\ t$.

Proof. We show $\mathsf{br}\ t\,(\mathsf{c}\ ts) = \mathsf{c}\,(t :: ts)$ by induction on $\mathsf{depth}\ t$:

Case 1 $ts = []$. Then $\mathsf{c}\ ts = \mathsf{Over}$.
Case 1.1 $t = \mathsf{Leaf}\ n$. We have

$$\begin{aligned} \mathsf{br}\ t\,(\mathsf{c}\ ts) &= \mathsf{next}\,(\mathsf{cons}\ n)\ \mathsf{Over} \qquad . \\ &= \mathsf{next}\,(\mathsf{addroots}\,(t :: ts))\,(\mathsf{c}\,(\mathsf{sub}\,(t :: ts))) = \mathsf{c}\,(t :: ts) \end{aligned}$$

Case 1.2 $t = \mathsf{Node}\ tl\ n\ tr$. Then by IH we get

$$\begin{aligned} \mathsf{br}\ t\,(\mathsf{c}\ ts) &= \mathsf{next}\,(\mathsf{cons}\ n)\,(\mathsf{br}\ tl\,(\mathsf{br}\ tl\,(\mathsf{c}\ ts))) \\ &= \mathsf{next}\,(\mathsf{cons}\ n)\,(\mathsf{c}\,(tl :: tr :: ts)) \\ &= \mathsf{next}\,(\mathsf{addroots}\,(t :: ts))\,(\mathsf{c}\,(\mathsf{sub}\,(t :: ts))) = \mathsf{c}\,(t :: ts) \end{aligned}$$

Case 2 Otherwise. Then $\mathsf{c}\ ts = \mathsf{next}\,(\mathsf{addroots}\ ts)\,(\mathsf{c}\,(\mathsf{sub}\ ts))$.
Case 2.1 $t = \mathsf{Leaf}\ n$.

$$\begin{aligned} \mathsf{br}\ t\,(\mathsf{c}\ ts) &= \mathsf{next}\,(\mathsf{cons}\ n \circ \mathsf{addroots}\ ts)\,(\mathsf{c}\,(\mathsf{sub}\ ts)) \\ &= \mathsf{next}\,(\mathsf{addroots}\,(t :: ts))\,(\mathsf{c}\,(\mathsf{sub}\,(t :: ts))) = \mathsf{c}\,(t :: ts) \end{aligned}$$

Case 2.2 $t = \mathsf{Node}\, tl\, n\, tl$. Then

$$\begin{aligned}
\mathsf{br}\, t\, (\mathsf{c}\, ts) &= \mathsf{next}\, (\mathsf{cons}\, n \circ \mathsf{addroots}\, ts)\, (\mathsf{br}\, tl\, (\mathsf{br}\, tl\, (\mathsf{c}\, (\mathsf{sub}\, ts)))) \\
&= \mathsf{next}\, (\mathsf{addroots}\, (t :: ts))\, (\mathsf{c}\, (tl :: tr :: (\mathsf{sub}\, ts))) \\
&= \mathsf{next}\, (\mathsf{addroots}\, (t :: ts))\, (\mathsf{c}\, (\mathsf{sub}\, (t :: ts))) = \mathsf{c}\, (t :: ts)
\end{aligned}$$

◀

Alternative proof of Theorem 6:
$\mathsf{breadthfirst}\, t = \mathsf{extract}\, (\mathsf{br}\, t\, \mathsf{Over}) = \mathsf{extract}\, (\mathsf{br}\, t\, (\mathsf{c}\, [])) = \mathsf{extract}\, (\mathsf{c}\, [t]) = \mathsf{breadthfirst}_{\mathsf{f,spec}}\, [t] = \mathsf{breadthfirst}_{\mathsf{spec}}\, t$. ◀

8 A predicative version of breadthfirst

In this section we present a variant of breadth-first traversal that, like Hofmann's algorithm, avoids the repeated use of list concatenation but is predicative since it doesn't use the data type of routines. Instead lists of functions are used as intermediate data type.

As observed in the previous section, the only elements of Rou created by the operations br and breadthfirst are Over and $\mathsf{next}\, g\, c$, where $g : \mathsf{List}\, \mathbb{N} \to \mathsf{List}\, \mathbb{N}$ and $c : \mathsf{Rou}$, and c is itself created by the algorithm. We can represent the elements of Rou that are defined inductively by these clauses as lists of functions $g : \mathsf{List}\, \mathbb{N} \to \mathsf{List}\, \mathbb{N}$, and therefore obtain them as those in the image of the function Φ defined as follows:

$$\mathsf{Rou}' = \mathsf{List}(\mathsf{List}\, \mathbb{N} \to \mathsf{List}\, \mathbb{N})$$
$$\Phi : \mathsf{Rou}' \to \mathsf{Rou} \qquad \Phi\, [] = \mathsf{Over} \qquad \Phi\, (g :: gs) = \mathsf{next}\, g\, (\Phi\, gs)$$

We denote elements of Rou' with the variable gs.

We translate br into a function br' referring to Rou' s. t. $\Phi \circ \mathsf{br}'\, t \overset{ext}{=} \mathsf{br}\, t \circ \Phi$:

$$\begin{aligned}
&\mathsf{br}' : \mathsf{Tree} \to \mathsf{Rou}' \to \mathsf{Rou}' \\
&\mathsf{br}'\, (\mathsf{Leaf}\, n)\, [] &&= \mathsf{cons}\, n :: [] \\
&\mathsf{br}'\, (\mathsf{Leaf}\, n)\, (g :: gs) &&= (\mathsf{cons}\, n \circ g) :: gs \\
&\mathsf{br}'\, (\mathsf{Node}\, tl\, n\, tr)\, [] &&= \mathsf{cons}\, n :: \mathsf{br}'\, tl\, (\mathsf{br}'\, tr\, []) \\
&\mathsf{br}'\, (\mathsf{Node}\, tl\, n\, tr)\, (g :: gs) &&= (\mathsf{cons}\, n \circ g) :: \mathsf{br}'\, tl\, (\mathsf{br}'\, tr\, gs)
\end{aligned}$$

The defining equations for br' are easily derived by transforming the right-hand side of the desired functional equation $\Phi\, (\mathsf{br}'\, t\, gs) = \mathsf{br}\, t\, (\Phi\, gs)$ into the form $\Phi\, gs'$ and then setting $\mathsf{br}'\, t\, gs = gs'$.

▶ **Lemma 13.** $\Phi \circ \mathsf{br}'\, t \overset{ext}{=} \mathsf{br}\, t \circ \Phi$.

Proof. One shows $\Phi\, (\mathsf{br}'\, t\, gs) = \mathsf{br}\, t\, (\Phi\, gs)$ by a straightforward induction on t and case analysis on gs (formalized in the Coq proof `br'_Lemma`, see Section 11). ◀

We can in the same way translate extract into a function $\mathsf{extract}'$ operating on Rou' s. t. $\mathsf{extract}' \overset{ext}{=} \mathsf{extract} \circ \Phi$: From this condition one can immediately derive its defining equations:

$$\mathsf{extract}' : \mathsf{Rou}' \to \mathsf{List}\, \mathbb{N} \qquad \mathsf{extract}'\, [] = [] \qquad \mathsf{extract}'\, (g :: gs) = g\, (\mathsf{extract}'\, gs)$$

▶ **Lemma 14.** $\mathsf{extract}' \overset{ext}{=} \mathsf{extract} \circ \Phi$.

Proof. We show $\mathsf{extract}'\, gs = \mathsf{extract}\, (\Phi\, gs)$ by induction on gs:

$$\begin{aligned}
\mathsf{extract}'\, [] &= [] = \mathsf{extract}\, (\Phi\, []) \\
\mathsf{extract}'\, (g :: gs) &= g\, (\mathsf{extract}'\, gs) = g\, (\mathsf{extract}\, (\Phi\, gs)) \\
&= \mathsf{extract}\, (\mathsf{next}\, g\, (\Phi\, gs)) = \mathsf{extract}\, (\Phi\, (g :: gs))
\end{aligned}$$

◀

Now we define $\mathsf{breadthfirst}' : \mathsf{Tree} \to \mathsf{List}\,\mathbb{N}$, $\mathsf{breadthfirst}'\,t = \mathsf{extract}'\,(\mathsf{br}'\,t\,[])$. It follows:

▶ **Lemma 15.** $\mathsf{breadthfirst}' \overset{ext}{=} \mathsf{breadthfirst}$.

Proof. $\mathsf{breadthfirst}'\,t = \mathsf{extract}'\,(\mathsf{br}'\,t\,[]) = \mathsf{extract}\,(\Phi\,(\mathsf{br}'\,t\,[])) = \mathsf{extract}\,(\mathsf{br}\,t\,(\Phi\,[]))$
$= \mathsf{extract}\,(\mathsf{br}\,t\,\mathsf{Over}) = \mathsf{breadthfirst}\,t$. ◀

In the next section 9 we will see how $\mathsf{breadthfirst}'$ can be reduced to $\mathsf{breadthfirst}''$ which is extensionally equal to $\mathsf{breadthfirst}_{\mathsf{spec}}$, giving an algebraic proof of the correctness of $\mathsf{breadthfirst}$. However, we can give as well a direct correctness proof of $\mathsf{breadthfirst}'$:
The routine computing the traversal of a $ts : \mathsf{Forest}$ having $\mathsf{niv}_{\mathsf{f}} = [l_1, \ldots, l_m]$ is given by $\mathsf{traverse}\,ts = [\mathsf{append}\,l_1, \ldots, \mathsf{append}\,l_n]$. A recursive definition (recursion on the measure $\mathsf{depth}\,ts$) of $\mathsf{traverse}\,ts : \mathsf{Rou}'$ is as follows:

$$\mathsf{traverse}\,ts = \begin{cases} [] & \text{if } ts = [], \\ \mathsf{addroots}\,ts :: \mathsf{traverse}\,(\mathsf{sub}\,ts) & \text{otherwise.} \end{cases}$$

▶ **Lemma 16.** $\mathsf{extract}' \circ \mathsf{traverse} \overset{ext}{=} \mathsf{breadthfirst}_{\mathsf{f,spec}}$.

▶ **Lemma 17.** $\mathsf{br}'\,t \circ \mathsf{traverse} \overset{ext}{=} \mathsf{traverse} \circ (\mathsf{cons}\,t)$.

Proof of Lemmas 16 and 17: One shows $\mathsf{extract}'\,(\mathsf{traverse}\,ts) = \mathsf{breadthfirst}_{\mathsf{f,spec}}\,ts$ by induction on $\mathsf{depth}\,ts$ and $\mathsf{br}'\,t\,(\mathsf{traverse}\,ts) = \mathsf{traverse}\,(t :: ts)$ by induction on t. ◀

We obtain an **alternative proof of Theorem 6** which contains as well the correctness of $\mathsf{breadthfirst}'$:

▶ **Theorem 18.** $\mathsf{breadthfirst} \overset{ext}{=} \mathsf{breadthfirst}' \overset{ext}{=} \mathsf{breadthfirst}_{\mathsf{spec}}$.

Proof. The first equation is Lemma 15. The 2nd equation follows as the alternative proof of Theorem 6 in Sect. 7 but using Lemmas 16 and 17 instead of Lemmas 11 and 12, respectively, and replacing Over by $[] : \mathsf{Rou}'$. ◀

9 A simplified predicative version of breadthfirst

The predicative algorithm for breadth-first traversal developed in the previous section can be simplified by observing that the type Rou' is only used with lists of functions that are formed from $(\mathsf{cons}\,n)$ by composition, i. e., functions of the form $\lambda l\,.\,l' +\!\!+\, l$ for some $l' : \mathsf{List}\,\mathbb{N}$. We can therefore denote them by elements of $\mathsf{List}\,\mathbb{N}$, and the elements of Rou' by elements of $\mathsf{List}^2\,\mathbb{N}$. Therefore, we define

$$\mathsf{Rou}'' := \mathsf{List}^2\,\mathbb{N}$$

$$\begin{array}{lll} \Psi : \mathsf{Rou}'' \to \mathsf{Rou}' & & \Psi\,ls = \mathsf{map}\,\mathsf{append}\,ls \\ \text{where} & \mathsf{map} : (A \to B) \to \mathsf{List}\,A \to \mathsf{List}\,B & \mathsf{map}\,f\,[l_1, \ldots, l_n] = [f\,l_1, \ldots, f\,l_n] \end{array}$$

We translate br' into a function br'' referring to Rou'':

$$\begin{array}{lcl} \mathsf{br}'' : \mathsf{Tree} \to \mathsf{Rou}'' \to \mathsf{Rou}'' & & \\ \mathsf{br}''\,(\mathsf{Leaf}\,n)\,[] & = & [[n]] \\ \mathsf{br}''\,(\mathsf{Leaf}\,n)\,(l :: ls) & = & \mathsf{cons}\,n\,l :: ls \\ \mathsf{br}''\,(\mathsf{Node}\,tl\,n\,tr)\,[] & = & [n] :: \mathsf{br}''\,tl\,(\mathsf{br}''\,tr\,[]) \\ \mathsf{br}''\,(\mathsf{Node}\,tl\,n\,tr)\,(l :: ls) & = & \mathsf{cons}\,n\,l :: \mathsf{br}''\,tl\,(\mathsf{br}''\,tr\,ls) \end{array}$$

▶ **Lemma 19.** $\Psi \circ \mathsf{br}''\,t \overset{ext}{=} \mathsf{br}'\,t \circ \Psi$.

Proof. We show $\Psi\,(\mathsf{br}''\,t\,ls) = \mathsf{br}'\,t\,(\Psi\,ls)$ by induction on t:

$$\begin{aligned}
\Psi\,(\mathsf{br}''\,(\mathsf{Leaf}\,n)\,[]) &= \Psi\,[[n]] = \mathsf{cons}\,n :: [] = \mathsf{br}'\,(\mathsf{Leaf}\,n)\,[] \\
\Psi\,(\mathsf{br}''\,(\mathsf{Leaf}\,n)\,(l :: ls)) &= \Psi\,(\mathsf{cons}\,n\,l :: ls) \\
&= (\mathsf{cons}\,n \circ \mathsf{append}\,l) :: \Psi\,ls \\
&= \mathsf{br}'\,(\mathsf{Leaf}\,n)\,(\mathsf{append}\,l :: \Psi\,ls) \\
\Psi\,(\mathsf{br}''\,(\mathsf{Node}\,tl\,n\,tr)\,[]) &= \Psi\,([n] :: \mathsf{br}''\,tl\,(\mathsf{br}''\,tr\,[])) \\
&= \mathsf{cons}\,n :: \mathsf{br}'\,tl\,(\mathsf{br}'\,tr\,[]) \\
&= \mathsf{br}'\,(\mathsf{Node}\,tl\,n\,tr)\,[] \\
\Psi\,(\mathsf{br}''\,(\mathsf{Node}\,tl\,n\,tr)\,(l :: ls)) &= \Psi(\mathsf{cons}\,n\,l :: (\mathsf{br}''\,tl\,(\mathsf{br}''\,tr\,ls))) \\
&= \mathsf{cons}\,n \circ \mathsf{append}\,l :: (\mathsf{br}'\,tl\,(\mathsf{br}'\,tr\,(\Psi\,ls))) \\
&= \mathsf{br}'\,(\mathsf{Node}\,tl\,n\,tr)\,(\mathsf{append}\,l :: \Psi\,ls)
\end{aligned}$$ ◀

▶ **Lemma 20.** $\mathsf{br}''\,t \overset{ext}{=} \mathsf{zip}\,(\mathsf{niv}\,t)$.

Proof. We show $\mathsf{br}''\,t\,ls = \mathsf{zip}\,(\mathsf{niv}\,t)\,ls$ by induction on t: For $t = \mathsf{Leaf}\,n$ this follows immediately by the definition of br''. In case of $t = \mathsf{Node}\,tl\,n\,tr$ and $ls = []$ we get using the IH $\mathsf{br}''\,t\,ls = [n] :: \mathsf{br}''\,tl\,(\mathsf{br}''\,tr\,[]) = [n] :: \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{zip}\,(\mathsf{niv}\,tr)\,[]) = [n] :: \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr) = \mathsf{niv}\,t = \mathsf{zip}\,(\mathsf{niv}\,t)\,[]$. In case of $t = \mathsf{Node}\,tl\,n\,tr$ and $ls = l' :: ls'$ we get using the IH $\mathsf{br}''\,t\,ls = \mathsf{cons}\,n\,l' :: \mathsf{br}''\,tl\,(\mathsf{br}''\,tr\,ls') = \mathsf{cons}\,n\,l' :: \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{zip}\,(\mathsf{niv}\,tr)\,ls') = \mathsf{cons}\,n\,l' :: \mathsf{zip}\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{zip}\,(\mathsf{niv}\,tr)))\,ls') = \mathsf{zip}\,([n] :: \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{zip}\,(\mathsf{niv}\,tr)))\,(l' :: ls') = \mathsf{zip}(\mathsf{niv}\,t)\,ls$. ◀

▶ **Lemma 21.** $\mathsf{flatten} \overset{ext}{=} \mathsf{extract}' \circ \Psi$.

Proof. By induction on the list argument:

$$\begin{aligned}
\mathsf{flatten}\,[] &= [] = \mathsf{extract}'\,[] \\
\mathsf{flatten}\,(l :: ls) &= l \mathbin{+\!\!+} \mathsf{flatten}\,ls = \mathsf{append}\,l\,(\mathsf{extract}'\,(\Psi\,ls)) = \mathsf{extract}'\,(\Psi\,(l :: ls))
\end{aligned}$$ ◀

Now we define $\mathsf{breadthfirst}'' : \mathsf{Tree} \to \mathsf{List}\,\mathbb{N}$ by $\mathsf{breadthfirst}''\,t = \mathsf{flatten}\,(\mathsf{br}''\,t\,[])$.

We obtain an **alternative proof of Theorem 6** which contains as well the correctness of $\mathsf{breadthfirst}'$ and $\mathsf{breadthfirst}''$:

▶ **Theorem 22.** $\mathsf{breadthfirst} \overset{ext}{=} \mathsf{breadthfirst}' \overset{ext}{=} \mathsf{breadthfirst}'' \overset{ext}{=} \mathsf{breadthfirst}_{\mathsf{spec}}$.

Proof. The first equation is Lemma 15. We prove the second equation:
$\mathsf{breadthfirst}''\,t = \mathsf{flatten}\,(\mathsf{br}''\,t\,[]) = \mathsf{extract}'\,(\Psi\,(\mathsf{br}''\,t\,[])) = \mathsf{extract}'\,(\mathsf{br}'\,t\,(\Psi\,[])) =$
$\mathsf{extract}'\,(\mathsf{br}'\,t\,[]) = \mathsf{breadthfirst}'\,t$.
Furthermore, by Lemma 20, we get
$\mathsf{breadthfirst}''\,t = \mathsf{flatten}\,(\mathsf{br}''\,t\,[]) = \mathsf{flatten}\,(\mathsf{zip}\,(\mathsf{niv}\,t)\,[]) = \mathsf{flatten}\,(\mathsf{niv}\,t) = \mathsf{breadthfirst}_{\mathsf{spec}}\,t$. ◀

10 Formal comparison of the obtained algorithms and proofs

In this section we isolate the common structure of the algorithms and proofs we have seen so far. Since, as remarked earlier, breadth-first traversal is not modular, all algorithms first compute some intermediate result (in a modular way) from which then the final result can be easily extracted. In fact, the program computing the intermediate result has an extra parameter which makes it possible to replace list concatenation (featuring in the specification) by function composition. We capture this common structure by the notion of a "system" and show that all proofs boil down to establishing a "simulation" relation between systems.

▶ **Definition 23.** *A* system *is a quadruple* $S = (A, \mathsf{Nil}, g, e)$ *where* $A : \mathsf{Set}$, $\mathsf{Nil} : A$, $g : \mathsf{Tree} \to A \to A$, *and* $e : A \to \mathsf{List}\,\mathbb{N}$.

- *S is* correct *(for breadth-first traversal) if* $e\,(g\,t\,\mathsf{Nil}) = \mathsf{breadthfirst_{spec}}\,t$ *for all trees* t.
- *Let* $S' = (A', \mathsf{Nil}', g', e')$ *be another system. A relation* R *on* $A \times A'$ *is a* simulation *between* S *and* S', $S \overset{R}{\sim} S'$, *if (1)* $R(\mathsf{Nil}, \mathsf{Nil}')$, *and, whenever* $R(a, a')$, *then (2)* $R(g\,t\,a, g'\,t\,a')$ *for all trees* t, *and (3)* $e\,a = e'\,a'$.
- *Let* S, S' *be systems.* S *and* S' *are* similar, $S \sim S'$, *if there exists a simulation between* S *and* S'.

▶ **Lemma 24.** *If* $S \sim S'$ *then* S *is correct if and only if* S' *is correct.*

Proof. If $S \overset{R}{\sim} S'$, then $R(g\,t\,\mathsf{Nil}, g'\,t\,\mathsf{Nil}')$, by (1) and (2), hence $e\,(g\,t\,\mathsf{Nil}) = e'\,(g'\,t\,\mathsf{Nil}')$, by (3). ◀

Note that if R is *functional*, i. e., defined as the graph of a function $\phi : A' \to A$, by setting $R(a, a')$ iff $a = \phi\,a'$, then the simulation conditions become (1) $\mathsf{Nil} = \phi\,\mathsf{Nil}'$, (2) $g\,t \circ \phi \overset{ext}{=} \phi \circ g'\,t$ for all trees t, and (3) $e \circ \phi \overset{ext}{=} e'$. In this situation we write $S \overset{\phi}{\leftarrow} S'$. All but one of the simulations described below are functional.

The specification of breadth-first traversal given in Section 2 corresponds to the system $S_{\mathsf{spec}} \overset{\mathrm{Def}}{\equiv} (\mathsf{List}^2\mathbb{N}, [], \mathsf{zip} \circ \mathsf{niv}, \mathsf{flatten})$. Correctness holds since $\mathsf{flatten}\,((\mathsf{zip} \circ \mathsf{niv})\,t\,[]) = \mathsf{flatten}\,(\mathsf{niv}\,t) = \mathsf{breadthfirst_{spec}}\,t$.

In the new view of systems, we may say that Hofmann defined his algorithm $\mathsf{breadthfirst}$ by the system $S_{\mathsf{MH}} \overset{\mathrm{Def}}{\equiv} (\mathsf{Rou}, \mathsf{Over}, \mathsf{br}, \mathsf{extract})$ (Sect. 3) and showed that $S_{\mathsf{MH}} \overset{\gamma_{\mathsf{Over}}}{\leftarrow} S_{\mathsf{spec}}$ where $\gamma_{\mathsf{Over}}\,ls \overset{\mathrm{Def}}{\equiv} \gamma\,ls\,\mathsf{Over}$ (Sect. 4). Condition (1) holds by the definition of γ, (2) holds by Lemmas 4 and 5, and (3) is Lemma 3.

The proofs given in Section 6 amount to showing $S_{\mathsf{MH}} \overset{\mathsf{rep}}{\sim} S_{\mathsf{spec}}$. (1) is the axiom (over), (2) is Lemma 9, and (3) is Lemma 8.

The (spec.-like) algorithm $\lambda t\,.\,\mathsf{breadthfirst_{f,spec}}\,[t]$ of Section 7 works with forests as the intermediate data type. The underlying system is $S_{\mathsf{forest}} \overset{\mathrm{Def}}{\equiv} (\mathsf{Forest}, [], \mathsf{cons}, \mathsf{breadthfirst_{f,spec}})$. Correctness of this system is easily established via the functional simulation $S_{\mathsf{spec}} \overset{\mathsf{niv_f}}{\leftarrow} S_{\mathsf{forest}}$ ((2) holds by definition of $\mathsf{niv_f}$, (3) is trivial). However, the point of S_{forest} is to provide a new correctness proof for S_{MH}. This is achieved by showing $S_{\mathsf{MH}} \overset{\mathsf{c}}{\leftarrow} S_{\mathsf{forest}}$. (1) holds by definition of c, (2) is Lemma 12, and (3) is Lemma 11.

The first predicative version of breadth-first traversal introduced in Section 8 defines the system $S_{\mathsf{pred1}} \overset{\mathrm{Def}}{\equiv} (\mathsf{Rou}', [], \mathsf{br}', \mathsf{extract}')$ and proves the simulation $S_{\mathsf{MH}} \overset{\Phi}{\leftarrow} S_{\mathsf{pred1}}$. The simulation conditions (2),(3) are shown in Lemmas 13 and 14, while (1) holds by definition of Φ. The correctness of S_{pred1} is shown via the simulation $S_{\mathsf{pred1}} \overset{\mathsf{traverse}}{\leftarrow} S_{\mathsf{forest}}$.

The simplified predicative algorithm in Section 9 is defined by the system $S_{\mathsf{pred2}} \overset{\mathrm{Def}}{\equiv} (\mathsf{List}^2\mathbb{N}, [], \mathsf{br}'', \mathsf{flatten})$. S_{pred2} is in fact (extensionally) equal to S_{spec} since $\mathsf{br}'' \overset{ext}{=} \mathsf{zip} \circ \mathsf{niv}$, by Lemma 20. We show $S_{\mathsf{pred1}} \overset{\Psi}{\leftarrow} S_{\mathsf{pred2}}$: the simulation conditions (2),(3) are given by the Lemmas 19 and 21, while (1) holds by definition of Ψ.

The following diagram gives an overview of the simulations:

$$\begin{tikzcd}
 & & S_{\mathsf{MH}} & & \\
S_{\mathsf{forest}} \arrow[urr, "\mathsf{c}"] \arrow[rrrr, "\mathsf{traverse}"] \arrow[dr, "\mathsf{niv_f}"'] & & & & S_{\mathsf{pred1}} \arrow[ull, "\Phi"'] \\
 & S_{\mathsf{spec}} \arrow[uur, "\mathsf{rep}"] & \overset{ext}{=} & S_{\mathsf{pred2}} \arrow[uul, "\gamma_{\mathsf{Over}}"'] \arrow[ur, "\Psi"'] &
\end{tikzcd}$$

In fact, the functions in the diagram are fully commutative assuming extensionality (regarding rep all we know at this stage is that it is a simulation, but we don't know its relationship to the simulation defined by γ_{Over}):

▶ **Lemma 25.**
(a) $\gamma_{\mathsf{Over}} \overset{ext}{=} \Phi \circ \Psi$.
(b) $\mathsf{traverse} \overset{ext}{=} \Psi \circ \mathsf{niv_f}$.
(c) $\mathsf{c} \overset{ext}{=} \Phi \circ \mathsf{traverse} \overset{ext}{=} \gamma_{\mathsf{Over}} \circ \mathsf{niv_f}$.

Proof. $\Phi\,(\Psi\,ls) = \gamma_{\mathsf{Over}}\,ls$ can be easily shown by induction on ls. However, the proof uses the extensionality principle (cf. Section 4). The equation $\mathsf{traverse}\,ts = \Psi(\mathsf{niv_f}\,ts)$ is obvious from the definition of $\mathsf{traverse}$. $\mathsf{c}\,ts = \Phi\,(\mathsf{traverse}\,ts)$ follows by induction on $\mathsf{depth}\,ts$. $\mathsf{c} \overset{ext}{=} \gamma_{\mathsf{Over}} \circ \mathsf{niv_f}$ follows from the previous equations. ◀

In particular, the simulations $S_{\mathsf{MH}} \overset{\Phi}{\leftarrow} S_{\mathsf{pred1}} \overset{\Psi}{\leftarrow} S_{\mathsf{pred2}}$ provide a splitting of Hofmann's simulation $S_{\mathsf{MH}} \overset{\gamma_{\mathsf{Over}}}{\leftarrow} S_{\mathsf{spec}}$ into simpler components.

11 Implementation and formalization in proof assistants

Here, we comment on our (partial) implementation of the presented ideas in Coq and Agda, that is publicly available in a Git repository [2]. The *Coq system* does not allow any inductive data type beyond strictly positive ones.[8] We overcome this by working with a version of Coq augmented by the plugin `TypingFlags` provided by Simon Boulier.[9] The effect of this plugin is to disable the checks for strict positivity, guardedness and termination. If, in such a development, one has established Lemma `lem` (for example), then `Print Assumptions lem` reveals for which constructions the plugin has forced Coq to accept them. For the formalization of Theorem 6, the forced acceptance only concerns the inductive data type Rou and the recursive function extract (and we also referred to `Logic.FunctionalExtensionality.functional_extensionality`, which is nothing but assuming equality of pointwise equal functions). The formalization and its verification present no difficulties at all, given the detailed proofs we provide in the paper. Thus, all of the elaborated mathematical developments in the Sections 2 to 10, with the notable exception of Section 5 (that is situated outside of Coq since it reflects on the term evaluation mechanism)

[8] See the Coq reference manual, in particular `https://coq.inria.fr/distrib/current/refman/language/cic.html#positivity-condition`.
[9] Plugin available at `https://github.com/SimonBoulier/TypingFlags/`.

are fully formalized in Coq, under the above provisos, i. e., with forced acceptance by Coq of the type Rou, the function extract, the relation rep and its induction principle `rep_ind` that is "manually" defined and not generated by the system, and by sometimes employing extensionality. For the recapitulations in form of the four formalized correctness proofs of S_{MH} – through Hofmann's function γ, through the relation rep, through forests and through the two predicative systems, lines of the form `Print Assumptions S_MH_correct*` reveal what is assumed beyond the core of Coq: Rou and extract in all cases since the algorithm is expressed in terms of them, rep and its induction principle only for the second proof, and extensionality only for the first and fourth proof.

Agda has the feature that using pragmas one can switch off strict positivity checks locally for data types and termination checks locally for functions. This allowed us to implement the functions used in the paper. Using quantification of set levels we were able to write down a substantial part of the operations defined in System F in Sect. 5, and after using postulates and the REWRITE pragma as well the extension by Mendler recursion. This allowed us to check that the reductions hold (at least that the left-hand and right-hand side of a reduction have the same normal form). Carrying out the proofs not requiring extensionality is still work in progress.

12 Conclusion and further work

In this paper we studied an intriguing algorithm by Martin Hofmann for the breadth-first traversal of finite binary trees which uses a non-strictly positive data type Rou of routines. We completed Hofmann's proof sketch of correctness (Sect. 4) and provided a justification for the termination of the algorithm by reduction to Mendler-style recursion in system F (Sect. 5). Furthermore we presented various alternative breadth-first traversal algorithms and correctness proofs with the aim to provide an explanation of Hofmann's somewhat mysterious construction. In Sect. 6 we transformed the data type Rou into a non-strictly positive inductive relation rep between routines and double lists and proved directly that the algorithm maps a tree to a routine that represents its levels from which correctness follows immediately. While the proof in Sect. 6 exploits non-strict positive induction as a proof principle, the other proofs only use structural induction (on lists or trees) but instead introduce new constructions that explain the roles of the components of Hofmann's algorithm and break it (the algorithm) into smaller, simpler, parts. The proof in Sect. 7 proves the correctness of Hofmann's algorithm breadthfirst via a simulation by a straightforward extension of breadth-first traversal to forests (which is closely related to the common approach to breadth-first traversal [13]). This reveals that the crucial component, br, of breadthfirst performs – via this simulation – nothing but the cons-operation on lists of trees. Through an analysis of the behaviour of breadthfirst we showed in Section 8 how to replace the impredicative type Rou of routines by the type Rou′ of lists of list functions and provided a predicative version, breadthfirst′, of breadthfirst. In Section 9, this predicative algorithm is further simplified by observing that only functions of the form $\lambda l \,.\, l' +\!\!+\, l$ are needed which can be represented by the list l'. Section 10 isolates the common structure of the algorithms by the notion of a *system* and the common structure of the correctness proofs by the notion of a *simulation*. In addition it shows that the simulation $S_{\mathsf{MH}} \overset{\gamma_{\mathsf{Over}}}{\leftarrow} S_{\mathsf{spec}}$, which corresponds to Hofmann's original proof, is split into the two, simpler and predicative, simulations $S_{\mathsf{MH}} \overset{\Phi}{\leftarrow} S_{\mathsf{pred1}} \overset{\Psi}{\leftarrow} S_{\mathsf{pred2}}$.

All algorithms were implemented and verified in the proof assistant Coq using various tweaks and extensions to accommodate non-strict positivity and some algorithms were implemented in Agda and Haskell [2].

Is the mystery of non-strictly positive breadth-first traversal now completely solved? Far from it. Looking at the algorithms it is quite clear that they should work for infinite (and hence non-well-founded) binary trees as well. This is confirmed by experiments with implementations in Haskell [2]. In order to formally prove this, coinductive data types and proof principles will be required which rely on the productivity of algorithms instead of the well-foundedness of their inputs. Carrying this out in current proof systems (whose capabilities of dealing with coinduction are still in their infancy) will be an exciting challenge.

Another mysterious algorithm that can be formulated with a non-strictly positive inductive type similar to the type of routines is a solution to the "same-fringe problem" that was suggested to us by Olivier Danvy. The problem is well-known: testing whether two finite trees have the same fringe, i. e., the same left-to-right listing of labels at their leaves. This problem is essentially different from breadth-first traversal since it relies on trees being finite. Its analysis is left to further work.

References

1 Andreas Abel and Ralph Matthes. Fixed points of type constructors and primitive recursion. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, Karpacz, Poland, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 2004. URL: `https://doi.org/10.1007/978-3-540-30124-0_17`.

2 Ulrich Berger, Ralph Matthes, and Anton Setzer. Git repository of code supplementing the present paper. `https://github.com/rmatthes/breadthfirstalahofmann`, November 2018 – August 2019.

3 Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse de Doctorat d'État, Université de Paris VII, 1972.

4 Jean-Yves Girard. *Proofs and types.* Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, Cambridge, 1989.

5 Robert Harper and John C. Mitchell. Parametricity and variants of Girard's J operator. *Information Processing Letters*, 70:1–5, 1999.

6 Martin Hofmann. Non strictly positive datatypes in system F, 15 Feb, 14:40:03 GMT 1993. Email on types mailing list, `http://www.seas.upenn.edu/~sweirich/types/archive/1993/msg00027.html`.

7 Martin Hofmann. Approaches to recursive datatypes – a case study, April 1995. LaTeX draft, 5 pages. Circulated by email.

8 Geraint Jones and Jeremy Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Technical Report No. 71, Dept of Computer Science, University of Auckland, 1993. IFIP Working Group 2.1 working paper 705 WIN-2. URL: `http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/linear.ps.gz`.

9 Ralph Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types.* Doktorarbeit (PhD thesis), LMU München, 1998. Available via the homepage `http://www.irit.fr/~Ralph.Matthes/works.html`.

10 Ralph Matthes. Tarski's fixed-point theorem and lambda calculi with monotone inductive types. *Synthese*, 133(1):107–129, 2002.

11 Paul F. Mendler. Inductive definition in type theory. Technical Report 87-870, Cornell University, Ithaca, N.Y., September 1987. PhD. Thesis (Paul F. Mendler = Nax P. Mendler).

12 Chris Okasaki. Simple and efficient purely functional queues and deques. *J. Funct. Program.*, 5(4):583–592, 1995.

13 Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN*

International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000., ICFP '00, pages 131–136, New York, NY, USA, 2000. ACM. URL: `http://doi.acm.org/10.1145/351240.351253`.

14 John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

15 John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 1984. URL: `https://doi.org/10.1007/3-540-13346-1_7`.