When Is a Container a Comonad?

Danel Ahman[1,⋆], James Chapman[2], and Tarmo Uustalu[2]

[1] Computer Laboratory, University of Cambridge,
15 J. J. Thomson Avenue, Cambridge CB3 0FD, United Kingdom
danel.ahman@cl.cam.ac.uk

[2] Institute of Cybernetics, Tallinn University of Technology,
Akadeemia tee 21, 12618 Tallinn, Estonia
{james,tarmo}@cs.ioc.ee

Abstract. Abbott, Altenkirch, Ghani and others have taught us that many parameterized datatypes (set functors) can be usefully analyzed via container representations in terms of a set of shapes and a set of positions in each shape. This paper builds on the observation that datatypes often carry additional structure that containers alone do not account for. We introduce directed containers to capture the common situation where every position in a datastructure determines another datastructure, informally, the sub-datastructure rooted by that position. Some natural examples are non-empty lists and node-labelled trees, and datastructures with a designated position (zippers). While containers denote set functors via a fully-faithful functor, directed containers interpret fully-faithfully into comonads. But more is true: every comonad whose underlying functor is a container is represented by a directed container. In fact, directed containers are the same as containers that are comonads. We also describe some constructions of directed containers. We have formalized our development in the dependently typed programming language Agda.

1 Introduction

Containers, as introduced by Abbott, Altenkirch and Ghani [1] are a neat representation for a wide class of parameterized datatypes (set functors) in terms of a set of shapes and a set of positions in each shape. They cover lists, colists, streams, various kinds of trees, etc. Containers can be used as a "syntax" for programming with these datatypes and reasoning about them, as can the strictly positive datatypes and polynomial functors of Dybjer [8], Moerdijk and Palmgren [16], Gambino and Hyland [9], and Kock [15]. The theory of this class of datatypes is elegant, as they are well-behaved in many respects.

This paper proceeds from the observation that datatypes often carry additional structure that containers alone do not account for. We introduce directed containers to capture the common situation in programming where every position in a datastructure determines another datastructure, informally, the

⋆ The first author was a summer intern at the Institute of Cybernetics, Tallinn University of Technology when the bulk of this work was carried out.

L. Birkedal (Ed.): FOSSACS 2012, LNCS 7213, pp. 74–88, 2012.
© Springer-Verlag Berlin Heidelberg 2012

sub-datastructure rooted by that position. Some natural examples of such datastructures are non-empty lists and node-labelled trees, and datastructures with a designated position or focus (zippers). In the former case, the sub-datastructure is a sublist or a subtree. In the latter case, it is the whole datastructure but with the focus moved to the given position.

We show that directed containers are no less neat than containers. While containers denote set functors via a fully-faithful functor, directed containers interpret fully-faithfully into comonads. They admit some of the constructions that containers do, but not others: for instance, two directed containers cannot be composed in general. Our main result is that every comonad whose underlying functor is the interpretation of a container is the interpretation of a directed container. So the answer to the question in the title of this paper is: a container is a comonad exactly when it is a directed container. In more precise terms, the category of directed containers is the pullback of the forgetful functor from the category of comonads to that of set functors along the interpretation functor of containers. This also means that a directed container is the same as a comonoid in the category of containers.

In Sect. 2, we review the basic theory of containers, showing also some examples. We introduce containers and their interpretation into set functors. We show some constructions of containers such as the coproduct of containers. In Sect. 3, we revisit our examples and introduce directed containers as a specialization of containers and describe their interpretation into comonads. We look at some constructions, in particular the focussed container (zipper) construction. Our main result, that a container is a comonad exactly when it is directed, is the subject of Sect. 4. In Sect. 5, we ask whether a similar characterization is possible for containers that are monads and hint that this is the case. We briefly summarize related work in Sect. 6 and conclude with outlining some directions for future work in Sect. 7

We spend a section on the background theory of containers as they are central for our paper but relatively little known, but assume that the reader knows about comonads, monoidal categories, monoidal functors and comonoids.

In our mathematics, we use syntax similar to the dependently typed functional programming language Agda [18]. If some function argument will be derivable in most contexts, we mark it as implicit by enclosing it/its type in braces in the function's type declaration and either give this argument in braces or omit it in the definition and applications of the function.

For lack of space, we have omitted all proofs from the paper. We have formalised our proofs in Agda; the development is available at `http://cs.ioc.ee/~danel/dcont.html`.

2 Containers

We begin with a recap of containers. We introduce the category of containers and the fully-faithful functor into the category of set functors defining the interpretation of containers and show that these are monoidal. We also recall some

basic constructions of containers. For proofs of the propositions in this section and further information, we refer the reader to Abbott et al. [1, 4].

2.1 Containers

Containers are a form of "syntax" for datatypes. A *container* $S \lhd P$ is given by a set $S : \mathsf{Set}$ of *shapes* and a shape-indexed family $P : S \to \mathsf{Set}$ of *positions*.

Intuitively, shapes are "templates" for datastructures and positions identify "blanks" in these templates that can be filled with data. The datatype of lists is represented by $S \lhd P$ where the shapes $S = \mathsf{Nat}$ are the possible lengths of lists and the positions $P\,s = \mathsf{Fin}\,s = \{0, \ldots, s-1\}$ provide s places for data in lists of length s. Non-empty lists are obtained by letting $S = \mathsf{Nat}$ and $P\,s = \mathsf{Fin}\,(s+1)$ (so that shape s has $s+1$ rather than s positions). Streams are characterized by a single shape with natural number positions: $S = 1 = \{*\}$ and $P\,* = \mathsf{Nat}$. The singleton datatype has one shape and one position: $S = 1$, $P\,* = 1$.

A *morphism* between containers $S \lhd P$ and $S' \lhd P'$ is a pair $t \lhd q$ of maps $t : S \to S'$ and $q : \Pi\{s : S\}.\, P'\,(t\,s) \to P\,s$ (the shape map and position map). Note how the positions are mapped backwards. The intuition is that, if a function between two datatypes does not look at the data, then the shape of a datastructure given to it must determine the shape of the datastructure returned and the data in any position in the shape returned must come from a definite position in the given shape.

For example, the head function, sending a non-empty list to a single data item, is determined by the maps $t : \mathsf{Nat} \to 1$ and $q : \Pi\{s : \mathsf{Nat}\}.\, 1 \to \mathsf{Fin}\,(s+1)$ defined by $t\,_ = *$ and $q\,* = 0$. The tail function, sending a non-empty list to a list, is represented by $t : \mathsf{Nat} \to \mathsf{Nat}$ and $q : \Pi\{s : \mathsf{Nat}\}.\, \mathsf{Fin}\,s \to \mathsf{Fin}\,(s+1)$ defined by $t\,s = s$ and $q\,p = p + 1$. For the function dropping every second element of a non-empty list, the shape and position maps $t : \mathsf{Nat} \to \mathsf{Nat}$ and $q : \Pi\{s : \mathsf{Nat}\}.\, \mathsf{Fin}\,(s \div 2 + 1) \to \mathsf{Fin}\,(s+1)$ are $t\,s = s \div 2$ and $q\,\{s\}\,p = p * 2$. For reversal of non-empty lists, they are $t : \mathsf{Nat} \to \mathsf{Nat}$ and $q : \Pi\{s : \mathsf{Nat}\}.\, \mathsf{Fin}\,(s + 1) \to \mathsf{Fin}\,(s+1)$ defined by $t\,s = s$ and $q\,\{s\}\,p = s - p$. (See Prince et al. [19] for more similar examples.)

The *identity* morphism $\mathsf{id}^{\mathrm{c}}\{C\}$ on a container $C = S \lhd P$ is defined by $\mathsf{id}^{\mathrm{c}} = \mathsf{id}\,\{S\} \lhd \lambda\{s\}.\, \mathsf{id}\,\{P\,s\}$. The *composition* $h \circ^{\mathrm{c}} h'$ of container morphisms $h = t \lhd q$ and $h' = t' \lhd q'$ is defined by $h \circ^{\mathrm{c}} h' = t \circ t' \lhd \lambda\{s\}.\, q'\,\{s\} \circ q\,\{t'\,s\}$. Composition of container morphisms is associative, identity is the unit.

Proposition 1. *Containers form a category* **Cont**.

2.2 Interpretation of Containers

To map containers into datatypes made of datastructures that have the positions in some shape filled with data, we must equip containers with a "semantics".

For a container $C = S \lhd P$, we define its *interpretation* $[\![C]\!]^{\mathrm{c}} : \mathsf{Set} \to \mathsf{Set}$ on sets by $[\![C]\!]^{\mathrm{c}}\,X = \Sigma s : S.\, P\,s \to X$, so that $[\![C]\!]^{\mathrm{c}}\,X$ consists of pairs of a shape and an assignment of an element of X to each of the positions in this shape, reflecting

the "template-and-blanks" idea. The interpretation $[\![C]\!]^{\mathrm{c}} : \forall\{X\},\{Y\}.\,(X \to Y) \to (\Sigma s : S.\,P\,s \to X) \to \Sigma s : S.\,P\,s \to Y$ of C on functions is defined by $[\![C]\!]^{\mathrm{c}}\,f\,(s,v) = (s, f \circ v)$. It is straightforward that $[\![C]\!]^{\mathrm{c}}$ preserves identity and composition of functions, so it is a set functor (as any datatype should be).

Our example containers denote the datatypes intended. If we let C be the container of lists, we have $[\![C]\!]^{\mathrm{c}}\,X = \Sigma s : \mathsf{Nat}.\,\mathsf{Fin}\,s \to X \cong \mathsf{List}\,X$. The container of streams interprets into $\Sigma * : 1.\,\mathsf{Nat} \to X \cong \mathsf{Nat} \to X \cong \mathsf{Str}\,X$. Etc.

A morphism $h = t \lhd q$ between containers $C = S \lhd P$ and $C = S' \lhd P'$ is interpreted as a natural transformation between $[\![C]\!]^{\mathrm{c}}$ and $[\![C']\!]^{\mathrm{c}}$, i.e., as a polymorphic function $[\![h]\!]^{\mathrm{c}} : \forall\{X\}.\,(\Sigma s : S.\,P\,s \to X) \to \Sigma s' : S'.\,P'\,s' \to X$ that is natural. It is defined by $[\![h]\!]^{\mathrm{c}}\,(s,v) = (t\,s, v \circ q\,\{s\})$. $[\![-]\!]^{\mathrm{c}}$ preserves the identities and composition of container morphisms.

The interpretation of the container morphism h corresponding to the list head function $[\![h]\!]^{\mathrm{c}} : \forall\{X\}.\,(\Sigma s : \mathsf{Nat}.\,\mathsf{Fin}\,(s+1) \to X) \to \Sigma * : 1.\,1 \to X$ is defined by $[\![h]\!]^{\mathrm{c}}\,(s,v) = (*, \lambda *.\,v\,0)$.

Proposition 2. $[\![-]\!]^{\mathrm{c}}$ *is a functor from* **Cont** *to* $[\mathbf{Set},\mathbf{Set}]$.

Every natural transformation between container interpretations is the interpretation of some container morphism. For containers $C = S \lhd P$ and $C' = S' \lhd P'$, a natural transformation τ between $[\![C]\!]^{\mathrm{c}}$ and $[\![C']\!]^{\mathrm{c}}$, i.e., a polymorphic function $\tau : \forall\{X\}.\,(\Sigma s : S.\,P\,s \to X) \to \Sigma s' : S'.\,P'\,s' \to X$ that is natural, can be "quoted" to a container morphism $\ulcorner\tau\urcorner^{\mathrm{c}} = (t \lhd q)$ between C and C' where $t : S \to S'$ and $q : \Pi\{s : S\}.\,P'\,(t\,s) \to P\,s$ are defined by $\ulcorner\tau\urcorner^{\mathrm{c}} = (\lambda s.\,\mathsf{fst}\,(\tau\,\{P\,s\}\,(s,\mathsf{id}))) \lhd (\lambda\{s\}.\,\mathsf{snd}\,(\tau\,\{P\,s\}\,(s,\mathsf{id})))$.

For any container morphism h, $\ulcorner[\![h]\!]^{\mathrm{c}}\urcorner^{\mathrm{c}} = h$, and, for any natural transformation τ and τ' between container interpretations, $\ulcorner\tau\urcorner^{\mathrm{c}} = \ulcorner\tau'\urcorner^{\mathrm{c}}$ implies $\tau = \tau'$.

Proposition 3. $[\![-]\!]^{\mathrm{c}}$ *is fully faithful.*

2.3 Monoidal Structure

We have already seen the *identity* container $\mathsf{Id}^{\mathrm{c}} = 1 \lhd \lambda *.\,1$. The *composition* $C_0 \cdot^{\mathrm{c}} C_1$ of containers $C_0 = S_0 \lhd P_0$ and $C_1 = S_1 \lhd P_1$ is the container $S \lhd P$ defined by $S = \Sigma s : S_0.\,P_0\,s \to S_1$ and $P\,(s,v) = \Sigma p_0 : P_0\,s.\,P_1\,(v\,p_0)$. It has as shapes pairs of an outer shape s and an assignment of an inner shape to every position in s. The positions in the composite container are pairs of a position p in the outer shape and a position in the inner shape assigned to p. The (horizontal) composition $h_0 \cdot^{\mathrm{c}} h_1$ of container morphisms $h_0 = t_0 \lhd q_0$ and $h_1 = t_1 \lhd q_1$ is the container morphism $t \lhd q$ defined by $t\,(s,v) = (t_0\,s, t_1 \circ v \circ q_0\,\{s\})$ and $q\,\{s,v\}\,(p_0,p_1) = (q_0\,\{s\}\,p_0, q_1\,\{v\,(q_0\,\{s\}\,p_0)\}\,p_1)$. The horizontal composition preserves the identity container morphisms and the (vertical) composition of container morphisms, which means that $- \cdot^{\mathrm{c}} -$ is a bifunctor.

Cont has isomorphisms $\rho : \forall\{C\}.\,C \cdot^{\mathrm{c}} \mathsf{Id}^{\mathrm{c}} \to C$, $\lambda : \forall\{C\}.\,\mathsf{Id}^{\mathrm{c}} \cdot^{\mathrm{c}} C \to C$ and $\alpha : \forall\{C\}\,\{C'\},\{C''\}.\,(C \cdot^{\mathrm{c}} C') \cdot^{\mathrm{c}} C'' \to C \cdot^{\mathrm{c}} (C' \cdot^{\mathrm{c}} C'')$, defined by $\rho = \lambda(s,v).\,s \lhd \lambda\{s,v\}.\,\lambda p.\,(p,*)$, $\lambda = \lambda(*,v).\,v\,* \lhd \lambda\{*,v\}.\,\lambda p.\,(*,p)$, $\alpha = \lambda((s,v),v').\,(s, \lambda p.\,(v\,p, \lambda p'.\,v'\,(p,p'))) \lhd \lambda\{(s,v),v'\}.\,\lambda(p,(p',p'')).\,((p,p'),p'')$.

Proposition 4. *The category* **Cont** *is a monoidal category.*

There are also natural isomorphisms $\mathsf{e} : \mathsf{Id} \to [\![\mathsf{Id}^{\mathrm{c}}]\!]^{\mathrm{c}}$ and $\mathsf{m} : \forall\{C_0\},\{C_1\}.\ [\![C_0]\!]^{\mathrm{c}} \cdot [\![C_1]\!]^{\mathrm{c}} \to [\![C_0 \cdot^{\mathrm{c}} C_1]\!]^{\mathrm{c}}$ that are defined by $\mathsf{e}\,x = (*, \lambda *. x)$ and $\mathsf{m}\,(s, v) = ((s, \lambda p.\,\mathsf{fst}\,(v\,p)), \lambda\,(p, p').\,\mathsf{snd}\,(v\,p)\,p')$ and are coherent.

Proposition 5. *The functor* $[\![-]\!]^{\mathrm{c}}$ *is a monoidal functor.*

2.4 Constructions of Containers

Containers are closed under various constructions such as products, coproducts and constant exponentiation, preserved by interpretation.

- For two containers $C_0 = S_0 \lhd P_0$ and $C_1 = S_1 \lhd P_1$, their *product* $C_0 \times C_1$ is the container $S \lhd P$ defined by $S = S_0 \times S_1$ and $P\,(s_0, s_1) = P_0\,s_0 + P_1\,s_1$. It holds that $[\![C_0 \times C_1]\!]^{\mathrm{c}} \cong [\![C_0]\!]^{\mathrm{c}} \times [\![C_1]\!]^{\mathrm{c}}$.
- The *coproduct* $C_0 + C_1$ of containers $C_0 = S_0 \lhd P_0$ and $C_1 = S_1 \lhd P_1$ is the container $S \lhd P$ defined by $S = S_0 + S_1$, $P\,(\mathsf{inl}\,s) = P_0\,s$ and $P\,(\mathsf{inr}\,s) = P_1\,s$. It is the case that $[\![C_0 + C_1]\!]^{\mathrm{c}} \cong [\![C_0]\!]^{\mathrm{c}} + [\![C_1]\!]^{\mathrm{c}}$.
- For a set $K \in \mathsf{Set}$ and a container $C_0 = S_0 \lhd P_0$, the *exponential* $K \to C_0$ is the container $S \lhd P$ where $S = K \to S_0$ and $P\,f = \Sigma k : K.\,P\,(f\,k)$. We have that $[\![K \to C_0]\!]^{\mathrm{c}} \cong K \to [\![C_0]\!]^{\mathrm{c}}$.

3 Directed Containers

We now proceed to our contribution, directed containers. We define the category of directed containers and a fully-faithful functor interpreting directed containers as comonads, and discuss some examples and constructions.

3.1 Directed Containers

Datatypes often carry some additional structure that is worth making explicit. For example, each node in a list or non-empty list defines a sublist (a suffix). In container terms, this corresponds to every position in a shape determining another shape, the subshape corresponding to this position. The theory of containers alone does not account for such additional structure. Directed containers, studied in the rest of this paper, axiomatize subshapes and translation of positions in a subshape into the global shape.

A *directed container* is a container $S \lhd P$ together with three operations

- $\downarrow : \Pi s : S.\,P\,s \to S$ (the subshape for a position),
- $\mathsf{o} : \Pi\{s : S\}.\,P\,s$ (the root),
- $\oplus : \Pi\{s : S\}.\,\Pi p : P\,s.\,P\,(s \downarrow p) \to P\,s$ (translation of subshape positions into positions in the global shape).

satisfying the following two shape equations and three position equations:

1. $\forall\{s\}.\, s \downarrow \mathsf{o} = s$,
2. $\forall\{s, p, p'\}.\, s \downarrow (p \oplus p') = (s \downarrow p) \downarrow p'$,
3. $\forall\{s, p\}.\, p \oplus \{s\}\, \mathsf{o} = p$,
4. $\forall\{s, p\}.\, \mathsf{o}\{s\} \oplus p = p$,
5. $\forall\{s, p, p', p''\}.\, (p \oplus \{s\}\, p') \oplus p'' = p \oplus (p' \oplus p'')$.

(Using $\oplus$ as an infix operation, we write the first, implicit, argument next to the operation symbol when we want to give it explicitly.) Modulo the fact that the positions involved come from different sets, laws 3-5 are the laws of a monoid.

To help explain the operations and laws, we sketch in Fig. 1 a datastructure with nested sub-datastructures.

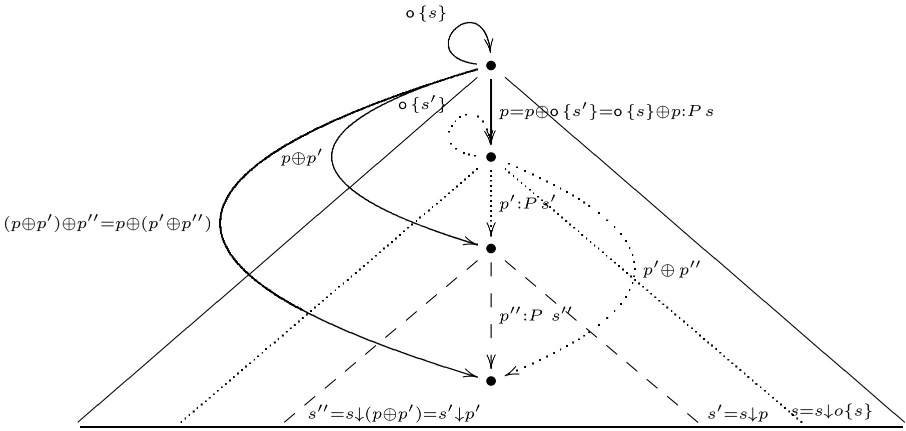

Fig. 1. A datastructure with two nested sub-datastructures

The global shape s is marked with a solid boundary and has a root position $\mathsf{o}\,\{s\}$. Then, any position p in s determines a shape $s' = s \downarrow p$, marked with a dotted boundary, to be thought of as the subshape of s given by this position. The root position in s' is $\mathsf{o}\,\{s'\}$. Law 3 says that its translation $p \oplus \mathsf{o}\,\{s'\}$ into a position in shape s is p, reflecting the idea that the subshape given by a position should have that position as the root.

By law 1, the subshape $s \downarrow \mathsf{o}\,\{s\}$ corresponding to the root position $\mathsf{o}\{s\}$ in the global shape s is s itself. Law 4, which is only well-typed thanks to law 1, stipulates that the translation of position p in $s \downarrow \mathsf{o}\,\{s\}$ into a position in s is just p (which is possible, as $P\,(s \downarrow \mathsf{o}\,\{s\}) = P\,s$).

A further position p' in s' determines a shape $s'' = s' \downarrow p'$. But p' also translates into a position $p \oplus p'$ in s and that determines a shape $s \downarrow (p \oplus p')$. Law 2 says that s'' and $s \downarrow (p \oplus p')$ are the same shape, which is marked by a dashed boundary in the figure. Finally, law 5 (well-typed only because of law 2) says that the two alternative ways to translate a position p'' in shape s'' into a position in shape s agree with each other.

Lists cannot form a directed container, as the shape 0 (for the empty list), having no positions, has no possible root position. But the container of *non-empty lists* (with $S = \mathsf{Nat}$ and $P\,s = \mathsf{Fin}\,(\mathsf{suc}\,s)$) is a directed container with respect to *suffixes* as (non-empty) sublists. The subshape given by a position p in a shape s (for lists of length $s+1$) is the shape of the corresponding suffix, given by $s \downarrow p = s - p$. The root $\mathsf{o}\{s\}$ is the position 0 of the head node. A position in the global shape is recovered from a position p' in the subshape of the position p by $p \oplus p' = p + p'$.

The "template" of non-empty lists of shape $s = 5$ (length 6) is given in Fig. 2. This figure also shows that the subshape determined by a position $p = 2$ in the global shape s is $s' = s \downarrow p = 5 - 2 = 3$ and a position $p' = 1$ in s' is rendered as the position $p \oplus p' = 2 + 1 = 3$ in the initial shape. Clearly one could

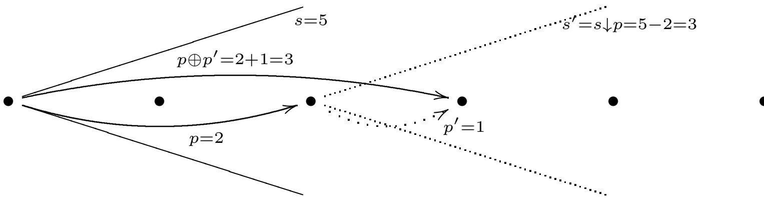

Fig. 2. The "template" of non-empty lists of shape 5 (length 6)

also choose prefixes as subshapes and the last node of a non-empty list as the root, but this gives an isomorphic directed container. Non-empty lists also give rise to an entirely different directed container structure that has *cyclic shifts* as "sublists" (this example was suggested to us by Jeremy Gibbons). The subshape at each position is the global shape ($s \downarrow p = s$). The root is still $\mathsf{o}\,\{s\} = 0$. The interesting part is that translation into the global shape of a subshape position is defined by $p \oplus \{s\}\, p' = (p + p') \mod s$, satisfying all the required laws.

The container of *streams* ($S = 1$, $P\,* = \mathsf{Nat}$) carries a very trivial directed container structure given by $* \downarrow p = *$, $\mathsf{o} = 0$ and $p \oplus p' = p + p'$. Fig. 3 shows how a position $p = 2$ in the only possible global shape $s = *$ and a position $p' = 2$ in the equal subshape $s' = s \downarrow p = *$ give back a position $p + p = 4$ in the global shape.

Similarly to the theory of containers, one can also define morphisms between directed containers. A *morphism* between directed containers $(S \lhd P, \downarrow, \mathsf{o}, \oplus)$ and $(S' \lhd P', \downarrow', \mathsf{o}', \oplus')$ is a morphism $t \lhd q$ between the containers $S \lhd P$ and $S' \lhd P'$ that satisfies three laws:

- $\forall \{s, p\}.\, t\,(s \downarrow q\,p) = t\,s \downarrow' p$,
- $\forall \{s\}.\, \mathsf{o}\,\{s\} = q\,(\mathsf{o}'\,\{t\,s\})$,
- $\forall \{s, p, p'\}.\, q\,p \oplus \{s\}\, q\,p' = q\,(p \oplus' \{t\,s\}\, p')$.

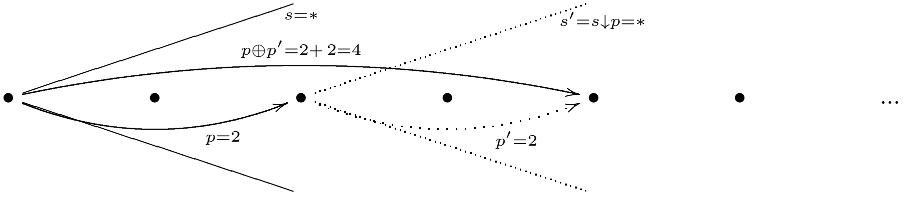

Fig. 3. The template of streams

Recall the intuition that t determines the shape of the datastructure that some given datastructure is sent to and q identifies for every position in the datastructure returned a position in the given datastructure. These laws say that the positions in the sub-datastructure for any position in the resulting datastructure must map back to positions in the corresponding sub-datastructure of the given datastructure. This means that they can receive data only from those positions, other flows are forbidden.

The container representations of the head and drop-even functions for non-empty lists are directed container morphisms. But that of reversal is not.

The identities and composition of **Cont** can give the identities and composition for directed containers, since for every directed container $E = (C, \downarrow, \mathsf{o}, \oplus)$, the identity container morphism $\mathsf{id}^{\mathrm{c}}\,\{C\}$ is a directed container morphism and the composition $h \circ^{\mathrm{c}} h'$ of two directed container morphisms is also a directed container morphism.

Proposition 6. *Directed containers form a category* **DCont**.

3.2 Interpretation of Directed Containers

As directed containers are containers with some operations obeying some laws, a directed container should denote not just a set functor, but a set functor with operations obeying some laws. The correct domain of denotation for directed containers is provided by comonads on sets.

Given a directed container $E = (S \lhd P, \downarrow, \mathsf{o}, \oplus)$, we define its *interpretation* $[\![E]\!]^{\mathrm{dc}}$ to be the set functor $D = [\![S \lhd P]\!]^{\mathrm{c}}$ (i.e., the interpretation of the underlying container) together with two natural transformations

$$
\begin{array}{l}
\varepsilon : \forall\{X\}.(\Sigma s : S.\, P\, s \to X) \to X \\
\varepsilon\,(s, v) = v\,(\mathsf{o}\,\{s\}) \\
\delta : \forall\{X\}.\,(\Sigma s : S.\, P\, s \to X) \to \Sigma s : S.\, P\, s \to \Sigma s' : S.\, P\, s' \to X \\
\delta\,(s, v) = (s, \lambda p.\, (s \downarrow p, \lambda p'.\, v\,(p \oplus \{s\}\, p')))
\end{array}
$$

The directed container laws ensure that the natural transformations ε, δ make the counit and comultiplication of a comonad structure on D.

Intuitively, the counit extracts the data at the root position of a datastructure (e.g., the head of a non-empty list), the comultiplication, which produces a datastructure of datastructures, replaces the data at every position with the sub-datastructure corresponding to this position (e.g., the corresponding suffix or cyclic shift).

The interpretation $[\![h]\!]^{\mathrm{dc}}$ of a morphism h between directed containers $E = (C, \downarrow, \mathsf{o}, \oplus)$, $E' = (C', \downarrow', \mathsf{o}', \oplus')$ is defined by $[\![h]\!]^{\mathrm{dc}} = [\![h]\!]^{\mathrm{c}}$ (using that h is a container morphism between C and C'). The directed container morphism laws ensure that this natural transformation between $[\![C]\!]^{\mathrm{c}}$ and $[\![C']\!]^{\mathrm{c}}$ is also a comonad morphism between $[\![E]\!]^{\mathrm{dc}}$ and $[\![E']\!]^{\mathrm{dc}}$.

Since **Comonads**(**Set**) inherits its identities and composition from [**Set**, **Set**], $[\![-]\!]^{\mathrm{dc}}$ also preserves the identities and composition.

Proposition 7. $[\![-]\!]^{\mathrm{dc}}$ *is a functor from* **DCont** *to* **Comonads**(**Set**).

Similarly to the case of natural transformations between container interpretations, one can also "quote" comonad morphisms between directed container interpretations into directed container morphisms. For any directed containers $E = (C, \downarrow, \mathsf{o}, \oplus)$, $E' = (C', \downarrow', \mathsf{o}', \oplus')$ and any morphism τ between the comonads $[\![E]\!]^{\mathrm{dc}}$ and $[\![E']\!]^{\mathrm{dc}}$ (which is a natural transformation between $[\![C]\!]^{\mathrm{c}}$ and $[\![C']\!]^{\mathrm{c}}$), the container morphism $\ulcorner \tau \urcorner^{\mathrm{dc}} = \ulcorner \tau \urcorner^{\mathrm{c}}$ between the underlying containers C and C' is also a directed container morphism between E and E'. The directed container morphism laws follow from the comonad morphism laws.

From what we already know about interpretation and quoting of container morphisms, it is immediate that $\ulcorner [\![h]\!]^{\mathrm{dc}} \urcorner^{\mathrm{dc}} = h$ for any directed container morphism h and that $\ulcorner \tau \urcorner^{\mathrm{dc}} = \ulcorner \tau' \urcorner^{\mathrm{dc}}$ implies $\tau = \tau'$ for any comonad morphisms τ and τ' between directed container interpretations.

Proposition 8. $[\![-]\!]^{\mathrm{dc}}$ *is fully faithful.*

The *identity container* $\mathsf{Id}^{\mathrm{c}} = 1 \lhd \lambda *. 1$ extends trivially to an identity directed container whose denotation is isomorphic to the identity comonad. But, similarly to the situation with functors and comonads, composition of containers fails to yield a composition monoidal structure on **DCont**.

3.3 Constructions of Directed Containers

We now show some constructions of directed containers. While some standard constructions of containers extend to directed containers, others do not.

Coproducts. Given two directed containers $E_0 = (S_0 \lhd P_0, \downarrow_0, \mathsf{o}_0, \oplus_0)$, $E_1 = (S_1 \lhd P_1, \downarrow_1, \mathsf{o}_1, \oplus_1)$, their coproduct is $(S \lhd P, \downarrow, \mathsf{o}, \oplus)$ whose underlying container $S \lhd P$ is the coproduct of containers $S_0 \lhd P_0$ and $S_1 \lhd P_1$. All of the directed container operations are defined either using $\downarrow_0, \mathsf{o}_0, \oplus_0$ or $\downarrow_1, \mathsf{o}_1, \oplus_1$ depending on the given shape. This means that the subshape is given by $\mathsf{inl}\, s \downarrow p = \mathsf{inl}\, (s \downarrow_0 p)$ and $\mathsf{inr}\, s \downarrow p = \mathsf{inr}\, (s \downarrow_1 p)$, the root position is given by $\mathsf{o}\, \{\mathsf{inl}\, s\} = \mathsf{o}_0\, \{s\}$ or $\mathsf{o}\, \{\mathsf{inr}\, s\} = \mathsf{o}_1\, \{s\}$ and the position in the initial shape is given by $p \oplus \{\mathsf{inl}\, s\}\, p' = p \oplus_0 \{s\}\, p'$ and $p \oplus \{\mathsf{inr}\, s\}\, p' = p \oplus_1 \{s\}\, p'$. Its interpretation is isomorphic to the coproduct of comonads $[\![E_0]\!]^{\mathrm{dc}}$ and $[\![E_1]\!]^{\mathrm{dc}}$.

Directed containers from monoids. Any monoid $(M, e, \bullet)$ gives rise to a directed container $E = (S \lhd P, \downarrow, \mathsf{o}, \oplus)$ where there is only one shape $*$ (with $S = 1$) whose positions $P * = M$ are the elements in the carrier set. The subshape operation $* \downarrow p = *$ thus becomes trivial as there is only one shape to return. Furthermore, the root position $\mathsf{o}\,\{*\} = e$ in the shape $*$ is the unit of the monoid and the position in the initial shape is given by using the monoid operation $p \oplus \{*\}\, p' = p \bullet p'$. The interpretation of this directed container is the comonad (D, ε, δ) where $D\,X = M \to X$, $\varepsilon = \lambda f.\, f\, e$, $\delta = \lambda f.\, \lambda p, p'.\, f\,(p \bullet p')$.

Cofree directed containers. The cofree directed container on a container $C = S_0 \lhd P_0$ is $E = (S \lhd P, \downarrow, \mathsf{o}, \oplus)$ where the underlying container is defined as $S = \nu Z.\, \Sigma s : S_0 . P_0\, s \to Z$ and $P = \mu Z.\, \lambda(s, v).\, 1 + \Sigma p : P_0\, s . Z\,(v\, p)$. The subshapes are defined by $(s, v) \downarrow \mathsf{inl}\, * = (s, v)$ and $(s, v) \downarrow \mathsf{inr}\,(p, p') = v\, p \downarrow p'$. The root position is defined by $\mathsf{o}\,\{s, v\} = \mathsf{inl}\, *$ and subshape positions by $\mathsf{inl}\, * \oplus \{s, v\}\, p'' = p''$ and $\mathsf{inr}\,(p, p') \oplus \{s, v\}\, p'' = \mathsf{inr}\,(p, p' \oplus \{v\, p\}\, p'')$. The interpretation $[\![E]\!]^{\mathrm{dc}} = (D, \varepsilon, \delta)$ of this directed container has its underlying functor given by $D\,X = \nu Z.\, X \times [\![C]\!]^{\mathrm{c}}\, Z$ and is the cofree comonad on the functor $[\![C]\!]^{\mathrm{c}}$.

A different directed container, the cofree recursive directed container on C is obtained by replacing the ν in the definition of S with μ. The interpretation has its underlying functor given by $D\,X = \mu Z.\, X \times [\![C]\!]^{\mathrm{c}}\, Z$ and is the cofree recursive comonad on $[\![C]\!]^{\mathrm{c}}$.

There is no general way to endow the product of the underlying containers of two directed containers $E_0 = (S_0 \lhd P_0, \downarrow_0, \mathsf{o}_0, \oplus_0)$ and $E_1 = (S_1 \lhd P_1, \downarrow_1, \mathsf{o}_1, \oplus_1)$ with the structure of a directed container. One can define $S = S_0 \times S_1$ and $P\,(s_0, s_1) = P_0\, s_0 + P_1\, s_1$, but there are two choices o_0 and o_1 for o. Moreover, there is no general way to define $p \oplus p'$. But this should not be surprising, as the product of the underlying functors of two comonads is not generally a comonad. Also, the product of two comonads would not be a comonad structure on the product of the underlying functors.

3.4 Focussing

Another interesting construction turning any container into a directed container is "focussing".

Datastructures with a focus. Any container $C = S_0 \lhd P_0$ defines a directed container $(S \lhd P, \downarrow, \mathsf{o}, \oplus)$ as follows. We take $S = \Sigma s : S_0 .\, P_0\, s$, so that a shape is a pair of a shape s, the "shape proper", and an arbitrary position p in that shape, the "focus". We take $P\,(s, p) = P_0\, s$, so that a position in the shape (s, p) is a position in the shape proper s, irrespective of the focus. The subshape determined by position p' in shape (s, p) is given by keeping the shape proper but changing the focus: $(s, p) \downarrow p' = (s, p')$. The root in the shape (s, p) is the focus p such that $\mathsf{o}\,\{s, p\} = p$. Finally, we take the translation of positions from the subshape (s, p') given by position p' to shape (s, p) to be the identity, by defining

$p' \oplus \{s, p\}\, p'' = p''$. All directed container laws are satisfied. This directed container interprets into the canonical comonad structure on the functor $\partial[\![C]\!]^{\mathrm{c}} \times \mathsf{Id}$ where ∂F denotes the derivative of the functor F.

Zippers. Inductive (tree-like) datatypes with a designated focus position are isomorphic to the zipper types of Huet [13]. A zipper datastructure encodes a tree with a focus as a pair of a context and a tree. The tree is the subtree of the global tree rooted by the focus and the context encodes the rest of the global tree. On zippers, changing the focus is supported via local navigation operations for moving one step down into the tree or up or aside into the context.

Zipper datatypes are directly representable as directed containers. We illustrate this on the example of zippers for non-empty lists. Such a zipper is a pair of a list (the context) and a non-empty list (the suffix determined by the focus position). Accordingly, by defining $S = \mathsf{Nat} \times \mathsf{Nat}$, the shape of a zipper is a pair (s_0, s_1) where s_0 is the shape of the context and s_1 is the shape of the suffix. For positions, it is convenient to choose $P\,(s_0, s_1) = \{-s_0, \ldots, s_1\}$ by allocating the negative numbers in the interval for positions in the context and non-negative numbers for positions in the suffix. The root position is $\mathsf{o}\,\{s_0, s_1\} = 0$, i.e., the focus. The subshape for each position is given by $(s_0, s_1) \downarrow p = (s_0 + p, s_1 - p)$ and translation of subshape positions by $p \oplus \{s_0, s_1\}\, p' = p + p'$.

Fig. 4 gives an example of a non-empty list with focus with its shape fixed to $s = (5, 6)$. It should be clear from the figure how the $\oplus$ operation works on positions $p = 4$ and $p' = -7$ to get back the position $p \oplus p' = -3$ in the initial shape. The subshape operation $\downarrow$ works as follows: $s \downarrow p$ gives back a subshape $s' = (9, 2)$ and $s \downarrow (p \oplus p')$ gives $s'' = (2, 9)$.

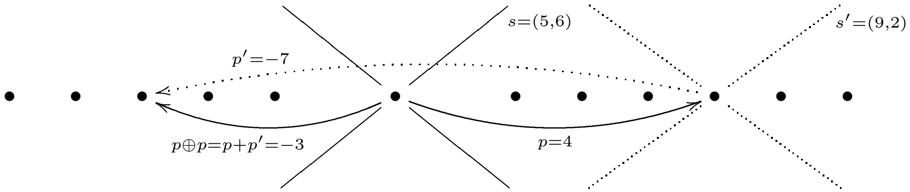

Fig. 4. The template for non-empty lists of length 12 focussed at position 5

4 Containers ∩ Comonads = Directed Containers

Since not every functor can be represented by a container, there is no point in asking whether every comonad can be represented as a directed container. An example of a natural comonad that is not a directed container is the cofree comonad on the finite powerset functor $\mathcal{P}_{\mathrm{f}}$ (node-labelled nonwellfounded strongly-extensional trees) where the carrier of this comonad is not a container

($\mathcal{P}_\mathrm{f}$ is also not a container). But, what about those comonads whose underlying functor is an interpretation of a container? It turns out that any such comonad does indeed define a directed container that is obtained as follows.

Given a comonad (D, ε, δ) and a container $C = S \lhd P$ such that $D = [\![C]\!]^\mathrm{c}$, the counit ε and comultiplication δ induce container morphisms

$$\begin{aligned}
&h^\varepsilon : C \to \mathsf{Id}^\mathrm{c} \\
&h^\varepsilon = t^\varepsilon \lhd q^\varepsilon = \ulcorner \mathsf{e} \circ \varepsilon \urcorner^\mathrm{c} \\
&h^\delta : C \to C \cdot^\mathrm{c} C \\
&h^\delta = t^\delta \lhd q^\delta = \ulcorner \mathsf{m}\,\{C\}\,\{C\} \circ \delta \urcorner^\mathrm{c}
\end{aligned}$$

using that $[\![-]\!]^\mathrm{c}$ is fully faithful. From (D, ε, δ) satisfying the laws of a comonad we can prove that $(C, h^\varepsilon, h^\delta)$ satisfies the laws of a comonoid in **Cont**. Further, we can define

$$\begin{aligned}
&s \downarrow p = \mathsf{snd}\,(t^\delta\ s)\,p \\
&\mathsf{o}\,\{s\} = q^\varepsilon\{s\} * \\
&p \oplus \{s\}\,p' = q^\delta\,\{s\}\,(p, p')
\end{aligned}$$

and the comonoid laws further enforce the laws of the directed container for $(C, \downarrow, \mathsf{o}, \oplus)$.

It may seem that the maps t^ε and $\mathsf{fst} \circ t^\delta$ are not used in the directed container structure, but $t^\varepsilon : S \to 1$ contains no information $(\forall\{s\}.\, t^\varepsilon\ s = *)$ and the comonad/comonoid right unit law forces that $\forall\{s\}.\, \mathsf{fst}\,(t^\delta\ s) = s$, which gets used in the proof of each of the five directed container laws. The latter fact is quite significant. It tells us that the comultiplication δ of any comonad whose underlying functor is the interpretation of a container preserves the shape of a given datastructure as the outer shape of the datastructure returned.

The situation is summarized as follows.

Proposition 9. *Any comonad (D, ε, δ) and container $C = S \lhd P$ such that $D = [\![C]\!]^\mathrm{c}$ determine a directed container $\lceil (D, \varepsilon, \delta), C \rceil$.*

Proposition 10. $\lceil [\![C, \downarrow, \mathsf{o}, \oplus]\!]^\mathrm{dc}, C \rceil = (C, \downarrow, \mathsf{o}, \oplus)$.

Proposition 11. $[\![\lceil (D, \varepsilon, \delta), C \rceil]\!]^\mathrm{dc} = (D, \varepsilon, \delta)$.

These observations suggest the following theorem.

Proposition 12. *The following is a pullback in* **CAT***:*

$$\begin{array}{ccc}
\mathbf{DCont} & \xrightarrow{U} & \mathbf{Cont} \\
{\scriptstyle [\![-]\!]^\mathrm{dc}}\downarrow{\scriptstyle \text{f.f.}} & & {\scriptstyle [\![-]\!]^\mathrm{c}}\downarrow{\scriptstyle \text{f.f.}} \\
\mathbf{Comonads}(\mathbf{Set}) & \xrightarrow{U} & [\mathbf{Set}, \mathbf{Set}]
\end{array}$$

It is proved by first noting that a pullback is provided by **Comonoids**(**Cont**) and then verifying that **Comonoids**(**Cont**) is isomorphic to **DCont**.

Sam Staton pointed it out to us that the proof of the first part only hinges on **Cont** and [**Set**, **Set**] being monoidal categories and $[\![-]\!]^\mathrm{c} : \mathbf{Cont} \to [\mathbf{Set}, \mathbf{Set}]$

being a fully faithful monoidal functor. Thus we actually establish a more general fact, viz., that for any two monoidal categories $\mathcal{C}$ and $\mathcal{D}$ and a fully-faithful monoidal functor $F : \mathcal{C} \to \mathcal{D}$, the pullback of F along the forgetful functor $U : \mathbf{Comonoids}(\mathcal{D}) \to \mathcal{D}$ is $\mathbf{Comonoids}(\mathcal{C})$.

In summary, we have seen that the interpretation of a container carries the structure of a comonad exactly when it extends to a directed container.

5 Containers ∩ Monads = ?

Given that comonads whose underlying functor is the interpretation of a container are the same as directed containers, it is natural to ask whether a similar characterization is possible for monads whose underlying functor can be represented as a container. The answer is "yes", but the additional structure is more involved than that of directed containers.

Given a container $C = S \lhd P$, the structure (η, μ) of a monad on the functor $T = [\![C]\!]^{\mathrm{c}}$ is interdefinable with the following structure on C

- $\mathsf{e} : S$ (for the shape map for η),
- $\bullet : \Pi s : S.(P\,s \to S) \to S$ (for the shape map for μ),
- $\nwarrow : \Pi\{s : S\}.\,\Pi v : P\,s \to S.P\,(s \bullet v) \to P\,s$ and
- $\nearrow : \Pi\{s : S\}.\,\Pi v : P\,s \to S.\Pi p : P\,(s \bullet v).\,P\,(v\,(v \nwarrow \{s\}p))$ (both for the position map for μ)

subject to three shape equations and five position equations. Perhaps not unexpectedly, this amounts to having a monoid structure on C.

To get some intuition, consider the monad structure on the datatype of lists. The unit is given by singleton lists and multiplication is flattening a list of lists by concatenation. For the list container $S = \mathsf{Nat}$, $P\,s = \mathsf{Fin}\,s$, we get that $\mathsf{e} = 1$, $s \bullet v = \sum_{p:\mathsf{Fin}\,s} v\,p$, $v \nwarrow \{s\}\,p =$ [greatest p' : $\mathsf{Fin}\,s$ such that $\sum_{p'':\mathsf{Fin}\,p'} v\,p'' \leq p$] and $v \nearrow \{s\}\,p = p - \sum_{p'':\mathsf{Fin}\,(v \nwarrow \{s\}\,p)} v\,p''$. The reason is that the shape of singleton lists is e while flattening a list of lists with outer shape s and inner shape $v\,p$ for every position p in s results in a list of shape $s \bullet v$. For a position p in the shape of the flattened list, the corresponding positions in the outer and inner shapes of the given list of lists are $v \nwarrow \{s\}\,p$ and $v \nearrow \{s\}\,p$.

For lack of space, we refrain from a more detailed discussion of this variation of the concept of containers.

6 Related Work

We build on the theory of containers as developed by Abbott, Altenkirch and Ghani [1, 4] to analyze strictly positive datatypes. Some generalizations of the concept of containers are the indexed containers of Altenkirch and Morris [5, 17] and the quotient containers of Abbott et al. [2]. In our work we look at a specialization of containers rather than a generalization. Simple/indexed containers are intimately related to strongly positive datatypes/families and simple/dependent

polynomial functors as appearing in the works of Dybjer [8], Moerdijk and Palmgren [16], Gambino and Hyland [9], Kock [15]. Girard's normal functors [11] and Joyal's analytic functors [14] functors are similar to containers resp. quotient containers, but only allow for finitely many positions in a shape.

Gambino and Kock [10] treat polynomial monads.

Abbott, Altenkirch, Ghani and McBride [3] have investigated derivatives of datatypes that provide a systematic way to explain Huet's zipper type [13].

Brookes and Geva [6] and later Uustalu with coauthors [20, 21, 12, 7] have used comonads to analyze notions of context-dependent computation such as dataflow computation, attribute grammars, tree transduction and cellular automata. Uustalu and Vene's [22] observation of a connection between bottom-up tree relabellings and containers with extra structure started our investigation into directed containers.

7 Conclusions and Future Work

We introduced directed containers as a specialization of containers for describing a certain class of datatypes (datastructures where every position determines a sub-datastructure) that occur very naturally in programming. It was a pleasant discovery for us that directed containers are an entirely natural concept also from the mathematical point of view: they are the same as containers whose interpretation carries the structure of a comonad.

In this paper, we could not discuss the equivalents of distributive laws between comonads, the composition of comonads, strict comonads and the product of (strict) comonads in the directed container world. We have already done some work around these concepts and constructions and plan to report our results in an extended version of this paper and elsewhere.

Acknowledgments. We are indebted to Thorsten Altenkirch, Jeremy Gibbons, Peter Morris, and Sam Staton for comments and suggestions. This work was supported by the European Regional Development Fund (ERDF) through Estonian Centre of Excellence in Computer Science (EXCS) project.

References

[1] Abbott, M., Altenkirch, T., Ghani, N.: Containers: Constructing strictly positive types. Theor. Comput. Sci. 342(1), 3–27 (2005)
[2] Abbott, M., Altenkirch, T., Ghani, N., McBride, C.: Constructing Polymorphic Programs with Quotient Types. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 2–15. Springer, Heidelberg (2004)
[3] Abbott, M., Altenkirch, T., Ghani, N., McBride, C.: δ for data: Differentiating data structures. Fund. Inform. 65(1-2), 1–28 (2005)
[4] Abbott, M.: Categories of Containers. Ph.D. thesis, University of Leicester (2003)
[5] Altenkirch, T., Morris, P.: Indexed containers. In: Proc. of 24th Ann. IEEE Symp. on Logic in Computer Science, LICS 2009, pp. 277–285. IEEE CS Press (2009)

[6] Brookes, S., Geva, S.: Computational comonads and intensional semantics. In: Fourman, M.P., Johnstone, P.T., Pitts, A.M. (eds.) Applications of Categories in Computer Science, London. Math. Society Lect. Note Series, vol. 77, pp. 1–44. Cambridge Univ. Press (1992)
[7] Capobianco, S., Uustalu, T.: A categorical outlook on cellular automata. In: Kari, J. (ed.) Proc. of 2nd Symp. on Cellular Automata, JAC 2010. TUCS Lecture Note Series, vol. 13, pp. 88–89. Turku Centre for Comput. Sci. (2011)
[8] Dybjer, P.: Representing inductively defined sets by wellorderings in Martin-Löf's type theory. Theor. Comput. Sci. 176(1-2), 329–335 (1997)
[9] Gambino, N., Hyland, M.: Wellfounded Trees and Dependent Polynomial Functors. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 210–225. Springer, Heidelberg (2004)
[10] Gambino, N., Kock, J.: Polynomial functors and polynomial monads. Tech. Rep. 867, Centre de Recerca Matemàtica, Barcelona (2009)
[11] Girard, J.Y.: Normal functors, power series and lambda-calculus. Ann. of Pure and Appl. Logic 37(2), 129–177 (1988)
[12] Hasuo, I., Jacobs, B., Uustalu, T.: Categorical Views on Computations on Trees. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 619–630. Springer, Heidelberg (2007)
[13] Huet, G.: The zipper. J. of Funct. Program. 7, 549–554 (1997)
[14] Joyal, A.: Foncteurs analytiques et espèces de structures. In: Labelle, G., Leroux, P. (eds.) Combinatoire énumerative. Lect. Notes in Math., vol. 1234, pp. 126–159. Springer, Heidelberg (1987)
[15] Kock, J.: Polynomial functors and trees. Int. Math. Research Notices 2011(3), 609–673 (2011)
[16] Moerdijk, I., Palmgren, E.: Wellfounded trees in categories. Ann. of Pure and Appl. Logic 104(1-3), 189–218 (2000)
[17] Morris, P.: Constructing Universes for Generic Programming. Ph.D. thesis, University of Nottingham (2007)
[18] Norell, U.: Towards a Practical Programming Language Based on Dependent type Theory. Ph.D. thesis, Chalmers University of Technology (2007)
[19] Prince, R., Ghani, N., McBride, C.: Proving Properties about Lists using Containers. In: Garrigue, J., Hermenegildo, M. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 97–112. Springer, Heidelberg (2008)
[20] Uustalu, T., Vene, V.: The Essence of Dataflow Programming. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 2–18. Springer, Heidelberg (2005)
[21] Uustalu, T., Vene, V.: Attribute evaluation is comonadic. In: van Eekelen, M. (ed.) Trends in Functional Programming, vol. 6, pp. 145–162. Intellect (2007)
[22] Uustalu, T., Vene, V.: Comonadic notions of computation. In: Adámek, J., Kupke, C. (eds.) Proc. of 9th Int. Wksh. on Coalgebraic Methods in Computer Science, CMCS 2008. Electron. Notes in Theor. Comput. Sci., vol. 203(5), pp. 263–284. Elsevier (2008)