# Proving Properties About Functions on Lists Involving Element Tests

Daniel Seidel* and Janis Voigtländer

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik
Römerstraße 164
53117 Bonn, Germany
{ds,jv}@iai.uni-bonn.de

**Abstract.** Bundy and Richardson [4] developed a method for reasoning about functions manipulating lists which is based on separating shape from content, and then exploiting a mathematically convenient representation for expressing shape-only manipulations. Later, Prince et al. [7] extended the technique to other data structures, and gave it a more formal basis via the theory of containers. All these results are restricted to fully polymorphic functions. For example, functions using equality tests on list elements are out of reach. We remedy this situation by developing new abstractions and representations for less polymorphic functions. In Haskell speak, we extend the earlier approach to be applicable in the presence of (certain) type class constraints.

## 1 Introduction

Abstraction is a useful strategy to get a clear view on the things that matter. Regarding proofs about program equivalences, it is beneficial to have an abstract representation of data structures and functions, holding exactly the information necessary for the intended reasoning in an easily accessible form. For lists, Bundy and Richardson [4] introduced a higher-order formulation in which a list is a pair $(n, f)$ where $n$ is a natural number representing the length of the list, i.e., its shape, and $f$ is a content function taking each position in the list to its corresponding element. Bundy and Richardson's motivation was that reasoning about such representations can be easier than reasoning about standard lists. In a more precise and more general form, the idea later recurred as reasoning via *container representations* [1,7].

The usefulness of the abstraction from the actual elements stored in a list is made apparent by the fact that certain *container morphisms*, taking a list (in this case) to another one, do not inspect or alter the image of $f$. An example for such a container morphism is the function $reverse^c$, the container version of the usual function reversing a list. The application of this container morphism is given as follows:

$$reverse^c\ (n, f) = (n, \lambda i \to f\ (n - i - 1))$$

* This author was supported by the DFG under grant VO 1512/1-1.

In general, a morphism shuffles positions (here by composing $f$ with the function $\lambda i \to n - i - 1$) and can alter the length of the list, remove elements, duplicate others. It cannot modify the elements themselves or add completely new elements.

The advantage of the container representation, which led Bundy and Richardson to using that representation, is that proofs about programs expressible as the composition of container morphisms become (simple) arithmetic proofs. For example, the proof that reversing a (finite) list twice is the identity is obtained very easily as follows:

$$\begin{aligned} \mathit{reverse}^c\ (\mathit{reverse}^c\ (n, f)) &= \mathit{reverse}^c\ (n, \lambda i \to f\ (n - i - 1)) \\ &= (n, \lambda i \to f\ (n - (n - i - 1) - 1)) \\ &= (n, \lambda i \to f\ i) \\ &= (n, f) \end{aligned}$$

Prince et al. [7] use, from Abbott et al. [1,2], that container morphisms correspond to parametrically polymorphic functions (or, *natural transformations*). Such polymorphic functions act independently of the concrete input type and hence, necessarily, independently of concrete elements of a type. Particularly, a fully polymorphic function from lists to lists, expressed via the type $[\alpha] \to [\alpha]$, maps for every type $\tau$ input lists of type $[\tau]$ to output lists of type $[\tau]$ without using any specifics of the type $\tau$. For example, a possible definition of *reverse* in Haskell [6] is:

$$\begin{aligned} &\mathit{reverse} && :: [\alpha] \to [\alpha] \\ &\mathit{reverse}\ [\,] && = [\,] \\ &\mathit{reverse}\ (x : xs) && = (\mathit{reverse}\ xs) +\!\!+ [x] \end{aligned}$$

Using category theoretic notions, Prince et al. observe that such polymorphic functions from lists to lists are isomorphic to the list container morphisms. The correspondence also generalises to other, strictly positive, data types.

What both Bundy and Richardson [4] and Prince et al. [7] fail to do is to reason about functions that are not fully polymorphic. An example, discussed in both papers, is a function *member* that checks whether a given value is an element of a given list. In Haskell:

$$\begin{aligned} &\mathit{member} && :: \mathit{Eq}\ \alpha \Rightarrow \alpha \to [\alpha] \to \mathbf{Bool} \\ &\mathit{member}\ x\ [\,] && = \mathbf{False} \\ &\mathit{member}\ x\ (y : ys) && = (x == y)\ ||\ (\mathit{member}\ x\ ys) \end{aligned}$$

Since programmed equivalence (the binary **Bool**-valued function $(==)$) depends on the type at which it is used, *member* cannot be given the fully polymorphic type $\alpha \to [\alpha] \to \mathbf{Bool}$. It instead comes with the constraint "$\mathit{Eq}\ \alpha \Rightarrow$", using Haskell's type class mechanism [10]. In the discussions of both Bundy and Richardson [4] and Prince et al. [7], the outcome is that the proposed reasoning method is not effective for *member*. Similarly, reasoning would not work for the function *nub* that eliminates duplicates from a list. These kind of functions also fall outside the realm of *shapely operations* in the calculus of Jay [5].

While Bundy and Richardson only identified the problematic case, and Prince et al. went a step further by observing that the problem can be explained by a lack

of polymorphism, we do provide a solution. In retrospect, at least the basic idea behind our solution may seem obvious: if a function is not polymorphic enough, then exploit information about to what actual extent it loses its polymorphism. In Haskell, that information is provided exactly by type class constraints like "$Eq\ \alpha \Rightarrow$". One example is the type of *member* seen above, another is that the already mentioned function *nub* will naturally be given the type $Eq\ \alpha \Rightarrow [\alpha] \to [\alpha]$. Of course, there is no reason at all to expect that the latter corresponds to an ordinary container morphism, because those were shown to be isomorphic to functions of the more general type $[\alpha] \to [\alpha]$ instead. But we can investigate refined notions of container representations and container morphisms, so that effective reasoning in the spirit of the earlier method becomes possible again. That is what we do in this paper.

In Section 2 we reconsider the connection between fully polymorphic functions and container morphisms. This sets the stage for our original development in later parts of the paper. In particular, it explains the use of *free theorems* [9], which in the guise of category-theoretic *naturality* is also at the heart of the isomorphism Prince et al. [7] use, and which in the form of free theorems for functions with type class constraints (also called ad-hoc polymorphic functions) will also pave the way to our results. In contrast to Prince et al., we do not use dependent types and therefore have slightly different formalisations of container values and container morphisms. Our reason for abstaining from using dependent types is notational convenience. Already by comparing the formulations of otherwise equivalent results and examples by Bundy and Richardson [4] and Prince et al. [7], it becomes clear that the former is lighter on notation. For the treatment of ad-hoc/type class polymorphism we found that the overhead of keeping exact dependent typing is even more cumbersome. However, there is something to lose by using less exact typing: we will not have an exact isomorphism as that employed by Prince et al. [7]. But we show in Section 2 that our setup is nevertheless sufficient for doing the kind of reasoning the overall method is aiming for. Moreover, it is perfectly possible to add all the dependent types back in, both in Section 2 and for our extensions to handle *Eq*-polymorphism, as presented in Section 3. Our approach is not limited to the type class *Eq*. In a similar way, container values, container morphisms, and the reasoning method can be extended to handle other type classes. We demonstrate this, still in Section 3, for the type class *Ord*, and offer some further perspective in Section 4.

## 2 The Earlier Results on Lists, Rephrased

In what follows, we use Haskell both as the language for writing functions about which we might want to prove properties, and as the specification language for container values and container morphisms, though for the latter use we will stretch Haskell a bit by including general math concepts. Moreover, we do not care about laziness in Haskell, or mixing strict and lazy evaluation using Haskell's *seq*-primitive. In fact, we assume a completely strict dialect of Haskell.

Let us first clarify some notations. The set of natural numbers is denoted by **Nat**. Depending on the context, a natural number $n$ represents either the number $n \in \mathbf{Nat}$ or the set of natural numbers $\{0, \ldots, n-1\}$. Furthermore, the type constructor for lists, already used in Haskell types in the introduction, is defined by

$$[\tau] = \{[x_0, \ldots, x_{n-1}] \mid n \in \mathbf{Nat},\, \forall i \in n.\ x_i :: \tau\}$$

Lists can alternatively be defined as container values, meaning by a shape (the length) and a content function (mapping each position to its entry). An appropriate definition (without using container terminology) was already introduced by Bundy and Richardson [4]. We restate it here by defining the set $\mathcal{C}(\tau)$ of list container values of type $\tau$ as

$$\mathcal{C}(\tau) = \{(n, f) \mid n :: \mathbf{Nat},\ f :: \mathbf{Nat} \to \tau\}$$

where the $f$s need not be totally defined, i.e., can be partial functions. But in every container value $(n, f)$, we require $f$ to be defined at least for all natural numbers less than $n$, i.e., on every position of the represented list.

In the following lemma we give a pair of functions that map back ($\square^{-1}$) and forth ($\square$) between container values and lists and nearly constitute an isomorphism. We continue to use the standard expression syntax of Haskell (while on the type level, $\mathcal{C}(\alpha)$ is "special syntax" that would not be found in actual Haskell). The operator !! takes a list and a position and returns the list entry at that position (counting from 0), and *map* is the usual function that applies its argument function to each element in its input list.

**Lemma 1.** *For each type $\tau$ as instantiation for $\alpha$, the functions $\square$ and $\square^{-1}$ defined as*

$$\begin{array}{ll} \square :: \mathcal{C}(\alpha) \to [\alpha] & \square^{-1} :: [\alpha] \to \mathcal{C}(\alpha) \\ \square\ (n, f) = \mathit{map}\ f\ [0\ ..\ (n-1)] & \square^{-1}\ \mathit{xs} = (\mathit{length}\ \mathit{xs}, \mathit{xs}\ !!) \end{array}$$

*satisfy the following three properties:*

1. $(\square \circ \square^{-1}) = \mathit{id}_{[\tau]}$
2. $(\square^{-1} \circ \square) \subseteq \equiv_{\mathcal{C}(\tau)}$, *where* $\equiv_{\mathcal{C}(\tau)} = \{((n, f), (n, f')) \mid \forall i \in n.\ f\ i = f'\ i\}$
3. $\forall (n, f), (n', f') \in \mathcal{C}(\tau).\ (n, f) \equiv_{\mathcal{C}(\tau)} (n', f')$ *iff* $\square\ (n, f) = \square\ (n', f')$

*Proof.* First, we show that $(\square \circ \square^{-1})\ \mathit{xs} = \mathit{xs}$ holds for every $\tau$ and $\mathit{xs} :: [\tau]$, by a straightforward induction on the length of $\mathit{xs}$. Second, we prove that $(\square^{-1} \circ \square) \subseteq \equiv_{\mathcal{C}(\tau)}$. We can reason as follows, for every container value:

$$\begin{aligned} \square^{-1}\ (\square\ (n, f)) &= (\mathit{length}\ (\mathit{map}\ f\ [0\ ..\ (n-1)]), (\mathit{map}\ f\ [0\ ..\ (n-1)])\ !!) \\ &= (n, f \circ ([0\ ..\ (n-1)]\ !!)) \\ &= (n, f|_n) \end{aligned}$$

where $g|_n$ means the restriction of a function $g$ to the domain $n$. The calculation steps are all by definitions and obvious properties of *length*, *map*, and (!!). Finally, the property $((n, f), (n', f')) \in \equiv_{\mathcal{C}(\tau)}$ iff $\square\ (n, f) = \square\ (n', f')$ follows from the definition of $\square$.

Note that $\square$ and $\square^{-1}$ indeed only *nearly* constitute an isomorphism. For example, let $\tau = \mathbf{Char}$ and let $f_1 :: \mathbf{Nat} \to \mathbf{Char}$ be the partial function with graph $\{(0, \text{'a'}), (1, \text{'b'})\}$ and $f_2 :: \mathbf{Nat} \to \mathbf{Char}$ the one whose graph additionally contains $(2, \text{'c'})$. Then $(2, f_1)$ and $(2, f_2)$ are two different elements of $\mathcal{C}(\mathbf{Char})$, but $\square\ (2, f_1) = [\text{'a'}, \text{'b'}] = \square\ (2, f_2)$.

We define a *container morphism* as a family $(s_n, P_n)_{n \in \mathbf{Nat}}$ of pairs comprising a natural number $s_n$, intuitively the output list length for any input list of length $n$, and a function $P_n :: \mathbf{Nat} \to \mathbf{Nat}$, intuitively mapping positions in the output list to positions in the input list when the latter has length $n$. For each container morphism and each $n \in \mathbf{Nat}$, we allow $P_n$ to be a partial function, but require that for every $i \in s_n$, we have $(P_n\ i) \in n$. The latter guarantees that all output positions are covered and that we never map an output position to a non-existing input position. We often abbreviate $(s_n, P_n)_{n \in \mathbf{Nat}}$ as $(s, P)$. The application of a container morphism to a container value is defined as

$$(s, P)\ (n, f) = (s_n, f \circ P_n)$$

Here are some container morphisms that intuitively correspond to well-known Haskell functions of type $[\alpha] \to [\alpha]$:

$$\begin{aligned} \mathit{reverse}^c &= (n, \lambda i \to n - i - 1)_{n \in \mathbf{Nat}} \\ \mathit{init}^c &= (n - 1, \mathit{id})_{n \in \mathbf{Nat}} \\ \mathit{tail}^c &= (n - 1, \lambda i \to i + 1)_{n \in \mathbf{Nat}} \end{aligned}$$

Before we can prove a systematic connection between fully polymorphic functions (in strict Haskell) and container morphisms, and function composition in either world, we need to say a few words on free theorems. Such theorems are statements about functions only dependent on the function type, relying on a formalisation of parametricity [8] for the functional language at hand. For example, in strict Haskell, the free theorem for the type $[\alpha] \to \mathbf{Nat}$ states that for every function $f :: \tau_1 \to \tau_2$ with $\tau_1$ and $\tau_2$ arbitrary, every function $g :: [\alpha] \to \mathbf{Nat}$, and every list $\mathit{xs} :: [\tau_1]$, we have $g\ (\mathit{map}\ f\ \mathit{xs}) = g\ \mathit{xs}$ if $f$ is defined for all elements of $\mathit{xs}$. The intuition is that in a purely functional language $g$'s behaviour can clearly only depend on its input argument. Moreover, since $g$ is fully polymorphic in the type $\alpha$ of elements of that input list, $g$ cannot inspect those elements in any way. Hence, $g$'s behaviour, and thus output, can only depend on the *structure* of its input list. Since a general property of *map* is that it does not change structure (and the output list is defined if $f$ is defined on all input list elements), $g\ (\mathit{map}\ f\ \mathit{xs}) = g\ \mathit{xs}$ follows. Reasoning by parametricity/free theorems allows to derive similar statements for a wide variety of types.

Incidentally, since the function *length* itself has exactly the above mentioned polymorphic type, one of the "obvious properties" (actually two, another one for (!!)) in the proof of Lemma 1 could have been deduced without considering the concrete function *length*, just its type. But the real value of free theorems is when we really do not know what concrete function we deal with, such as when we want to prove that *every* strict-Haskell function of type $[\alpha] \to [\alpha]$ corresponds to some container morphism.

Since free theorems are available for free, i.e., can be automatically generated, we will use them as given, without considering further formal background here. Let us note, though, in preparation for Section 3, that free theorems in the presence of type class polymorphism can be established by an indirection via types (and functions) obtained through the dictionary translation method of Wadler and Blott [10].

**Theorem 1.** *For every function $g :: [\alpha] \to [\alpha]$, there exists a container morphism $(s, P)$ such that $g \circ \Box = \Box \circ (s, P)$.*

*Proof.* Let $g :: [\alpha] \to [\alpha]$. Then the free theorem for $g$'s type tells us that $g\ (map\ h\ l) = map\ h\ (g\ l)$ for every choice of types $\tau_1$, $\tau_2$, function $h :: \tau_1 \to \tau_2$, and list $l :: [\tau_1]$ if $h$ is defined for all elements of $l$. Hence, we can reason as follows, for every container value:

$$\begin{aligned} g\ (\Box\ (n, f)) &= g\ (map\ f\ [0\ ..\ (n-1)]) \\ &= map\ f\ (g\ [0\ ..\ (n-1)]) \\ &= \Box\ (\Box^{-1}\ (map\ f\ (g\ [0\ ..\ (n-1)]))) \\ &= \Box\ (length\ (g\ [0\ ..\ (n-1)]), f \circ ((g\ [0\ ..\ (n-1)])\ !!)) \\ &= \Box\ ((length\ (g\ [0\ ..\ (n-1)]), (g\ [0\ ..\ (n-1)])\ !!)_{n \in \mathbf{Nat}}\ (n, f)) \end{aligned}$$

where the second step is by the free theorem, the third by Lemma 1(1), the fourth by the definition of $\Box^{-1}$ and properties of *length*, *map*, and (!!), and the last step by the definition of the application of a container morphism to a container value.

Note that there is not a unique container morphism corresponding, in the sense of Theorem 1, to a function $g :: [\alpha] \to [\alpha]$. For example, for the standard Haskell definition of $init :: [\alpha] \to [\alpha]$, both $init^c = (n-1, id)_{n \in \mathbf{Nat}}$ and $init^c = (n-1, id|_{n-1})_{n \in \mathbf{Nat}}$ satisfy $init \circ \Box = \Box \circ init^c$. This (direction of) non-uniqueness does no harm to our reasoning application, though. Together with Lemma 1, Theorem 1 allows the calculation with container morphisms instead of polymorphic functions. The required results are stated in the following corollary and lemma.

**Corollary 1.** *For every function $g :: [\alpha] \to [\alpha]$, there exists a container morphism $(s, P)$ such that $g = \Box \circ (s, P) \circ \Box^{-1}$.*

*Proof.* By Theorem 1 and Lemma 1(1).

**Lemma 2.** *Let $g, g' :: [\alpha] \to [\alpha]$. Let $(s, P)$, $(s', P')$ be container morphisms such that $g = \Box \circ (s, P) \circ \Box^{-1}$ and $g' = \Box \circ (s', P') \circ \Box^{-1}$. Then we have $g \circ g' = \Box \circ (s, P) \circ (s', P') \circ \Box^{-1}$.*

*Proof.* By the assumptions, we have $g \circ g' = \Box \circ (s, P) \circ \Box^{-1} \circ \Box \circ (s', P') \circ \Box^{-1}$, so it would suffice to show that $\Box \circ (s, P) \circ \Box^{-1} \circ \Box = \Box \circ (s, P)$. By Lemma 1(3), this is equivalent to, for every type $\tau$ and $(n, f) \in \mathcal{C}(\tau)$,

$$(s, P)\ (\Box^{-1}\ (\Box\ (n, f))) \equiv_{\mathcal{C}(\tau)} (s, P)\ (n, f)$$

But by Lemma 1(2), we have $\Box^{-1}\ (\Box\ (n, f)) \equiv_{\mathcal{C}(\tau)} (n, f)$, and it is easy to show from the definitions that for every $(n, f), (n', f')$ with $(n, f) \equiv_{\mathcal{C}(\tau)} (n', f')$, it holds that $(s, P)\ (n, f) \equiv_{\mathcal{C}(\tau)} (s, P)\ (n', f')$.

Let us manifest the usefulness of our formal material by an example. Assume we want to prove that $\mathit{reverse} \circ \mathit{tail} = \mathit{init} \circ \mathit{reverse}$ holds. We have

$$\begin{aligned} \mathit{reverse} &= \Box \circ \mathit{reverse}^c \circ \Box^{-1} \\ \mathit{init} &= \Box \circ \mathit{init}^c \circ \Box^{-1} \\ \mathit{tail} &= \Box \circ \mathit{tail}^c \circ \Box^{-1} \end{aligned}$$

for standard Haskell definitions of the list functions and $\mathit{reverse}^c$, $\mathit{init}^c$, and $\mathit{tail}^c$ as given above Theorem 1.[1] By Lemma 2, it suffices to prove that

$$\Box \circ \mathit{reverse}^c \circ \mathit{tail}^c \circ \Box^{-1} = \Box \circ \mathit{init}^c \circ \mathit{reverse}^c \circ \Box^{-1}$$

and by Lemma 1(3) indeed to prove that for every type $\tau$ and $(n, f) \in \mathcal{C}(\tau)$,

$$(\mathit{reverse}^c \circ \mathit{tail}^c)\ (n, f) \equiv_{\mathcal{C}(\tau)} (\mathit{init}^c \circ \mathit{reverse}^c)\ (n, f)$$

We can calculate for the left-hand side

$$\begin{aligned} (\mathit{reverse}^c \circ \mathit{tail}^c)\ (n, f) &= \mathit{reverse}^c\ (n - 1, \lambda i \to f\ (i + 1)) \\ &= (n - 1, \lambda i \to f\ (((n - 1) - i - 1) + 1)) \\ &= (n - 1, \lambda i \to f\ (n - 1 - i)) \end{aligned}$$

and for the right-hand side

$$\begin{aligned} (\mathit{init}^c \circ \mathit{reverse}^c)\ (n, f) &= \mathit{init}^c\ (n, \lambda i \to f\ (n - i - 1)) \\ &= (n - 1, \lambda i \to f\ (n - i - 1)) \end{aligned}$$

to see that the claim holds.

Let us contrast the above proof with an attempt at directly proving $\mathit{reverse} \circ \mathit{tail} = \mathit{init} \circ \mathit{reverse}$ using the Haskell definition of *reverse* from the introduction as well as some suitable definitions of *tail* and *init*. The interesting case is the one of a non-empty list: $\mathit{reverse}\ (\mathit{tail}\ (x : xs)) = \mathit{init}\ (\mathit{reverse}\ (x : xs))$, which reduces to the proof obligation $\mathit{reverse}\ xs = \mathit{init}\ ((\mathit{reverse}\ xs) +\!\!+ [x])$. Now an *inductive* proof using the defining equations of *init* would be required, where first the given proof obligation would have to be *generalised* to an actually suitable induction hypothesis (since simply performing induction on $xs$ in $\mathit{reverse}\ xs = \mathit{init}\ ((\mathit{reverse}\ xs) +\!\!+ [x])$ leads nowhere). In contrast, the above proof requires neither induction nor inventing a generalisation. It just performs simple arithmetics.

[1] Clearly, neither Theorem 1 nor Corollary 1 prove the equivalence $\mathit{reverse} = \Box \circ \mathit{reverse}^c \circ \Box^{-1}$ for the specific syntactic definitions of *reverse* and $\mathit{reverse}^c$ given in the introduction (and likewise for *init* and *tail*). The theorem and corollary provide, for every $g$, *one* suitable definition for $g^c$. It might not be the one we find useful for reasoning. Finding such a useful syntactic representation, like $\mathit{reverse}^c = (n, \lambda i \to n - i - 1)_{n \in \mathbf{Nat}}$, must be done on a case-by-case basis, but is often very natural, like in all cases here.

## 3 Refining the Container-Related Notions

The results in the previous section can be extended in two directions. One is to consider not only functions from lists to lists, but also functions between other data structures that can be viewed as container values. That direction is already explored by Prince et al. [7]. The extension that we consider is orthogonal to that first one. In particular, while we focus on functions from lists to lists here, we are confident that our results could be easily combined with the results of Prince et al. [7] to handle functions (involving element tests like equivalence and ordering) between arbitrary container structures.

Considering functions like *nub*, removing all duplicates from a list, or *sort*, sorting a list's elements, it is clear that they are not fully polymorphic in their list element type. The functions require the availability of an equivalence test or an order defined on elements of the input list. Hence, Theorem 1 is not applicable anymore. Our aim now is to appropriately adapt the notions of container value and container morphism to get equally useful results for functions of types $Eq\ \alpha \Rightarrow [\alpha] \to [\alpha]$ and $Ord\ \alpha \Rightarrow [\alpha] \to [\alpha]$ as the earlier works provide for functions of type $[\alpha] \to [\alpha]$.

It is important to note that our view on type classes is that they really hold what they pretend to provide. In the case of *Eq*, that means that every type in *Eq* indeed carries an *equivalence* relation. In real Haskell, the implemented relations can be arbitrary (no reflexivity, transitivity, or symmetry are guaranteed or checked). In the same spirit, in Section 3.2 we expect types in the type class *Ord* to carry an actual total preorder (a reflexive, transitive, and total relation).

### 3.1 The Type Class *Eq*

To capture what happens if elements in a list are testable for equivalence, the container notions have to be adjusted. We use $\mathcal{E}(M)$ to denote the class of all (decidable) equivalence relations over a set $M$. For simplicity of notation, we freely regard an equivalence relation $\cong$ on a subset of **Nat** as the equivalence relation $\cong \cup\, id_{\mathbf{Nat}}$ on **Nat** when appropriate. For a type $\tau$ that is an instance of *Eq*, we denote by $\cong_\tau$ the corresponding fixed (in a given program) equivalence relation. Since in Haskell, it is actually accessible via the binary **Bool**-valued function $(==)$, we set:

$$\cong_\tau = \{(x, y) \mid x :: \tau,\, y :: \tau,\, (x == y) = \mathbf{True}\}$$

**Definition 1.** *Let $\tau$ be some type that is an instance of Eq. An Eq*-container value *of type $\tau$ is a triple $(n, \cong, f)$ with $n :: \mathbf{Nat}$, $\cong \in \mathcal{E}(\mathbf{Nat})$, and $f :: \mathbf{Nat} \to \tau$ a partial function such that*

$$\forall i, j \in n.\ i \cong j \Leftrightarrow (f\ i) \cong_\tau (f\ j)$$

*or, equivalently,*[2]

$$(\ker_{\cong_\tau}\ f|_n) = (\cong \cap\, (n \times n)) \tag{1}$$

[2] The kernel of a function over a relation is defined as $(\ker_\cong\ f) = \{(i, j) \mid (f\ i) \cong (f\ j)\}$.

*The set of all such container values is denoted by* $\mathcal{C}^{Eq}(\tau)$.

The roles of $n$ and $f$ in the above definition are as before in the case of ordinary list containers $\mathcal{C}(\tau)$. Condition (1) implies the previous side condition that the function $f$ is defined at least for all natural numbers less than $n$. But condition (1) is stronger than that. It involves the key new ingredient of *Eq*-container values, namely the second component $\cong$. The role of that equivalence relation is to capture information, in terms of list positions, about equivalence tests between elements accessible via $f$. For a concrete example, assume $\tau = \mathbf{Char}$ and that the equivalence relation $\cong_{\mathbf{Char}}$ were such that upper- and lowercase of the same letter were considered equivalent, while different letters were considered inequivalent. Then the list ['a', 'A', 'b'] :: [**Char**] could be represented as an *Eq*-container value as $(3, \cong, f)$, where $\cong = \{(0,0), (0,1), (1,0), (1,1), (2,2)\}$ and $f$ maps 0 to 'a', 1 to 'A', and 2 to 'b'.

Some further explanations seem in order, to avoid possible misconceptions. First, from Definition 1, in particular from the presence of $\cong$ (though not $\cong_\tau$) in container value triples, it might seem that each *Eq*-container value somehow stores its own completely private equivalence relation so that, within the same program, two members $(n_1, \cong_1, f_1)$ and $(n_2, \cong_2, f_2)$ of the same $\mathcal{C}^{Eq}(\tau)$ can interpret equivalence between elements of type $\tau$ in two different ways. Under that perception, for example, some $(2, \cong_1, f_1), (2, \cong_2, f_2) \in \mathcal{C}^{Eq}(\mathbf{Char})$ could represent the same list ['a', 'A'] :: [**Char**] while somehow $\cong_1$ and $\cong_2$ could be chosen in such a way that in one case when ['a', 'A'] is passed to some function $g :: Eq\ \alpha \Rightarrow [\alpha] \to [\alpha]$ (or its "container version") the two list elements are considered equivalent, while in the other case they are not.

But that is *not* the case! Actually, by condition (1) we have that $n$, $f$, and $\tau$ (through $\cong_\tau$) uniquely determine $\cong$ (or at least its relevant part, on $n \times n$). So why, then, do we include the $\cong$ in $(n, \cong, f) \in \mathcal{C}^{Eq}(\tau)$ at all? The point is that we will be able (in Definitions 2 and 3 below) to describe the behaviour of (a container analogue of) a function $g :: Eq\ \alpha \Rightarrow [\alpha] \to [\alpha]$ on $(n, \cong, f)$ solely by relying on $n$ and $\cong$, rather than looking into $f$. That is the key abstraction/enabler for exploiting the type class polymorphism when reasoning about such functions: that the behaviour of such a function $g$ can be understood by just considering relative equivalences between list elements (as captured via $\cong$), rather than the concrete list elements themselves (as still accessible via $f$, but deliberately not used in determining $g$'s behaviour). So explicitly representing and (while preserving the invariant (1); see Lemma 4) manipulating $\cong$ is crucial to effectively "let the symbols do the work".

Hopefully having accepted $\cong$ as an explicit component of *Eq*-container values, note further that we use $f$ as a function from list positions into $\tau$. We could have been tempted to instead define $f$ as a function from *equivalence classes* of positions, with respect to $\cong$, into $\tau$, rather than from the positions themselves. While these choices may appear to be interchangeable, there is actually a crucial difference. With our choice we can distinguish elements that are equivalent with respect to $\cong_\tau$, but not equal. For example, consider the list ['a', 'A'] :: [**Char**] and assume that the equivalence relation $\cong_{\mathbf{Char}}$ were again the one mentioned

in the paragraph directly following Definition 1. Then a container representation working with a function from equivalence classes of positions would, at length two, only be able to represent lists with two equal elements (['a', 'a'], ['b', 'b'], ['A', 'A'], ...) and lists with different letters (['a', 'b'], ['b', 'a'], ['a', 'B'], ...), but not the list ['a', 'A'] as distinguishable from ['a', 'a'] and ['A', 'A']. One might be willing to accept this limited expressiveness, as indeed when the equivalence provided by the type class instance for **Char** is "same letter"-ness, then all of ['a', 'A'], ['a', 'a'], and ['A', 'a'] ought to be considered equivalent with respect to the inferred type class instance for [**Char**]. But after all, equivalent with respect to a type class instance is not the same as semantically equal, and we want to keep that distinction in our reasoning. For example, we want to still be able to observe that applying (the container morphism corresponding to) *reverse* to ['a', 'A'] gives ['A', 'a'], and not ['a', 'a'] or ['A', 'A'].

After having made and justified these important decisions, we can set up a pair of functions between *Eq*-container values and lists satisfying similar properties as the pair of functions $\square$ and $\square^{-1}$ defined in Lemma 1.

**Lemma 3.** *For each type $\tau$ that is an instance of Eq, the instantiations of the functions $\square^{Eq}$ and $(\square^{Eq})^{-1}$ defined as*

$$\begin{array}{ll} \square^{Eq} & :: Eq\ \alpha \Rightarrow \mathcal{C}^{Eq}(\alpha) \to [\alpha] \\ \square^{Eq}\ (n, \cong, f) & = map\ f\ [0\ ..\ (n-1)] \\ (\square^{Eq})^{-1} & :: Eq\ \alpha \Rightarrow [\alpha] \to \mathcal{C}^{Eq}(\alpha) \\ (\square^{Eq})^{-1}\ xs & = (length\ xs, \ker_{\cong_\alpha}\ (xs\ !!), xs\ !!) \end{array}$$

*satisfy the following three properties:*

1. $(\square^{Eq} \circ (\square^{Eq})^{-1}) = id_{[\tau]}$
2. $((\square^{Eq})^{-1} \circ \square^{Eq}) \subseteq \equiv_{\mathcal{C}^{Eq}(\tau)}$,
   *where* $\equiv_{\mathcal{C}^{Eq}(\tau)} = \{((n, \cong, f), (n, \cong', f')) \mid \forall i \in n.\ f\ i = f'\ i\}$
3. $\forall (n, \cong, f), (n', \cong', f') \in \mathcal{C}^{Eq}(\tau)$.
   $(n, \cong, f) \equiv_{\mathcal{C}^{Eq}(\tau)} (n', \cong', f')$ *iff* $\square^{Eq}\ (n, \cong, f) = \square^{Eq}\ (n', \cong', f')$

*Proof.* The proofs of properties (1)–(3) are similar to the proof of Lemma 1. An important aspect to show is that indeed $((\square^{Eq})^{-1}\ xs) \in \mathcal{C}^{Eq}(\tau)$ for every $xs :: [\tau]$, particularly so for condition (1) from Definition 1. But the required statement is obtained relatively directly from the definition of $(\square^{Eq})^{-1}$.

Now, appropriate morphisms between *Eq*-container values, and their application, are defined as follows.

**Definition 2.** *An* *Eq*-container morphism $(s, P)$ *is a family of pairs* $(s_n^{\cong}, P_n^{\cong})_{n \in \mathbf{Nat}, \cong \in \mathcal{E}(\mathbf{Nat})}$ *such that* $s_n^{\cong} :: \mathbf{Nat}$ *and* $P_n^{\cong} :: \mathbf{Nat} \to \mathbf{Nat}$ *a partial function with* $(P_n^{\cong}\ i) \in n$ *for every* $i \in s_n^{\cong}$.

The intuitions for $s_n^{\cong}$ and $P_n^{\cong}$ are as for ordinary container morphisms before, except that now both can depend on the new parameter $\cong$ in addition to $n$. After all, we need to be prepared for the fact that the behaviour (i.e., determining

the length of the output list and the distribution of elements in it) of a function involving equivalence tests cannot anymore be described by just inspecting the input list length. In addition, information about such equivalence tests may have to be accessed.

**Definition 3.** *Let $(n, \cong, f)$ be an Eq-container value and $(s, P)$ an Eq-container morphism. The application of $(s, P)$ to $(n, \cong, f)$ is defined as*

$$(s, P)\ (n, \cong, f) = (s_n^{\cong}, \ker_{\cong}\ P_n^{\cong}, f \circ P_n^{\cong})$$

The first and last components of the output triple are analogous to the ordinary case without element tests. For the middle component, we capture the position-wise equivalence of output list elements in terms of the mapping to input positions and what we know about, again position-wise, equivalence of input list elements.

The following lemma states the well-behavedness of the notions defined above.

**Lemma 4.** *Let $\tau$ be a type that is an instance of Eq. Let $c \in \mathcal{C}^{Eq}(\tau)$ and let $m$ be an Eq-container morphism. Then we have $(m\ c) \in \mathcal{C}^{Eq}(\tau)$.*

*Proof.* The critical point to prove is the condition (1) from Definition 1 on the content function of the resulting container value. Let $c = (n, \cong, f)$ and $m = (s, P)$. We have $(m\ c) = (s_n^{\cong}, \ker_{\cong}\ P_n^{\cong}, f \circ P_n^{\cong})$ and hence need to show that

$$(\ker_{\cong_\tau}\ (f \circ P_n^{\cong})|_{s_n^{\cong}}) = ((\ker_{\cong}\ P_n^{\cong}) \cap (s_n^{\cong} \times s_n^{\cong}))$$

is satisfied. But that is an easy exercise, using $(\ker_{\cong_\tau}\ f|_n) = (\cong \cap\ (n \times n))$.

Comparing the definitions of morphisms on ordinary container values and on *Eq*-container values, we can easily translate the former ones into the latter ones.

*Note 1.* Every (ordinary) container morphism $(s_n, P_n)_{n \in \mathbf{Nat}}$ can be viewed as the *Eq*-container morphism $(s_n, P_n)_{n \in \mathbf{Nat}, \cong \in \mathcal{E}(\mathbf{Nat})}$.

To verify that our definitions of *Eq*-container values and *Eq*-container morphisms are useful when reasoning about strict-Haskell functions of type $Eq\ \alpha \Rightarrow [\alpha] \to [\alpha]$, we need results similar to Theorem 1, Corollary 1, and Lemma 2. Indeed, such results are possible and given below.

**Theorem 2.** *For every function $g :: Eq\ \alpha \Rightarrow [\alpha] \to [\alpha]$, there exists an Eq-container morphism $(s, P)$ such that $g \circ \square^{Eq} = \square^{Eq} \circ (s, P)$.*

*Proof.* Let $g :: Eq\ \alpha \Rightarrow [\alpha] \to [\alpha]$. Then the free theorem for $g$'s type tells us that $g\ (map\ h\ l) = map\ h\ (g\ l)$ for every choice of types $\tau_1$, $\tau_2$ that are instances of *Eq*, function $h :: \tau_1 \to \tau_2$, and list $l :: [\tau_1]$, provided that $(\ker_{\cong_{\tau_2}}\ h) = \cong_{\tau_1}$ and that $h$ is defined for all elements of $l$. Now, let $(n, \cong, f) \in \mathcal{C}^{Eq}(\tau)$. By the definition of *Eq*-container values, we know that the function $f$ satisfies $(\ker_{\cong_\tau}\ f|_n) = (\cong \cap (n \times n))$. So for $h = f|_n$, $\tau_1 = n$, $\cong_{\tau_1} = (\cong \cap\ (n \times n))$, and $\tau_2 = \tau$ we can apply the free

theorem above and obtain $g\ (map\ f|_n\ l) = map\ f|_n\ (g\ l)$ for every list $l :: [n]$. Hence, we can reason similarly to the proof of Theorem 1 as follows:[3]

$$\begin{aligned}
g_{\cong_\tau}\ (\square^{Eq}\ (n, \cong, f)) &= g_{\cong_\tau}\ (map\ f\ [0\ ..\ (n-1)]) \\
&= g_{\cong_\tau}\ (map\ f|_n\ [0\ ..\ (n-1)]) \\
&= map\ f|_n\ (g_{\cong\cap(n\times n)}\ [0\ ..\ (n-1)]) \\
&= \square^{Eq}\ ((\square^{Eq})^{-1}\ (map\ f|_n\ (g_{\cong\cap(n\times n)}\ [0\ ..\ (n-1)]))) \\
&= \square^{Eq}\ (length\ (g_{\cong\cap(n\times n)}\ [0\ ..\ (n-1)]), \\
&\qquad \ker_{\cong_\tau}\ (f|_n \circ ((g_{\cong\cap(n\times n)}\ [0\ ..\ (n-1)])\ !!)), \\
&\qquad f|_n \circ ((g_{\cong\cap(n\times n)}\ [0\ ..\ (n-1)])\ !!)) \\
&= \square^{Eq}\ ((s, P)\ (n, \cong, f))
\end{aligned}$$

where we set

$$(s, P) = (length\ (g_{\cong\cap(n\times n)}\ [0\ ..\ (n-1)]), (g_{\cong\cap(n\times n)}\ [0\ ..\ (n-1)])\ !!)_{n\in\mathbf{Nat},\cong\in\mathcal{E}(\mathbf{Nat})}$$

and use

$$f \circ ((g_{\cong\cap(n\times n)}\ [0\ ..\ (n-1)])\ !!) = f|_n \circ ((g_{\cong\cap(n\times n)}\ [0\ ..\ (n-1)])\ !!)$$

as well as

$$\ker_{\cong}\ ((g_{\cong\cap(n\times n)}\ [0\ ..\ (n-1)])\ !!) = \ker_{\cong_\tau}\ (f|_n \circ ((g_{\cong\cap(n\times n)}\ [0\ ..\ (n-1)])\ !!))$$

These two statements used here are true since $g_{\cong\cap(n\times n)}\ [0\ ..\ (n-1)] :: [n]$ contains only elements from 0 to $n-1$ and since, for the second statement, we know that $(\ker_{\cong_\tau}\ f|_n) = (\cong \cap\ (n \times n))$.

**Corollary 2.** *For every function $g :: Eq\ \alpha \Rightarrow [\alpha] \to [\alpha]$, there exists an Eq-container morphism $(s, P)$ such that $g = \square^{Eq} \circ (s, P) \circ (\square^{Eq})^{-1}$.*

*Proof.* By Theorem 2 and Lemma 3(1).

**Lemma 5.** *Let $g, g' :: Eq \Rightarrow [\alpha] \to [\alpha]$. Let $(s, P), (s', P')$ be Eq-container morphisms such that $g = \square^{Eq} \circ (s, P) \circ (\square^{Eq})^{-1}$ and $g' = \square^{Eq} \circ (s', P') \circ (\square^{Eq})^{-1}$. Then we have $g \circ g' = \square^{Eq} \circ (s, P) \circ (s', P') \circ (\square^{Eq})^{-1}$.*

*Proof.* Similarly to the proof of Lemma 2.

We have now established all the formal setup that is required for reasoning about functions of type $Eq \Rightarrow [\alpha] \to [\alpha]$ by instead reasoning about *Eq*-container morphisms. To manifest this with some examples, consider first the following container morphism versions of *nub* and *rmSingles*, where the first of these functions removes duplicates from a list and the second one throws away each element that appears only once in a given list (in both cases, ultimately with respect to an equivalence relation provided via a type class instance for *Eq*, of course):

[3] To highlight the changes of the equivalence relation that $g$ uses, we have throughout subscripted each instance of $g$ with the equivalence relation it actually works with.

$$
\begin{aligned}
&\mathit{nub}^c = (s^{\cong}_n, P^{\cong}_n)_{n\in\mathbf{Nat},\cong\in\mathcal{E}(\mathbf{Nat})}\\
&\quad\text{with}\;\; s^{\cong}_n = |n_{/\cong}| \text{ and}\\
&\qquad\quad P^{\cong}_n = \lambda i \to \min\{j \,:\, |\{[k]_\cong \,:\, k \le j\}| = i+1\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{rmSingles}^c = (s^{\cong}_n, P^{\cong}_n)_{n\in\mathbf{Nat},\cong\in\mathcal{E}(\mathbf{Nat})}\\
&\quad\text{with}\;\; s^{\cong}_n = \textstyle\sum_{e\in n_{/\cong},|e|>1} |e| \text{ and}\\
&\qquad\quad P^{\cong}_n = \lambda i \to \min\{j \,:\, |\{j' \in \textstyle\bigcup_{e\in n_{/\cong},|e|>1} e \,:\, j' \le j\}| = i+1\}
\end{aligned}
$$

Note that we use standard notations $n_{/\cong}$ for factorisation with respect to an equivalence relation and $[k]_\cong$ for building equivalence classes.

As already noticed, we can view "ordinary" container morphisms as *Eq*-container morphisms as well. For an example, we give the application of $\mathit{init}^c$ to an *Eq*-container value. As we use them in the following examples of proofs, we show the result of applying $\mathit{nub}^c$ and $\mathit{rmSingles}^c$, in general, as well.

$$
\begin{aligned}
\mathit{init}^c\ (n,\cong,f) &= (n-1,\cong,f)\\
\mathit{nub}^c\ (n,\cong,f) &= (|n_{/\cong}|, \mathit{id}, \lambda i \to f\ (\min\{j \,:\, |\{[k]_\cong \,:\, k \le j\}| = i+1\}))\\
\mathit{rmSingles}^c\ (n,\cong,f) &= (\textstyle\sum_{e\in n_{/\cong},|e|>1} |e|,\\
&\qquad \ker_\cong\ (\lambda i \to \min\{j \,:\, |\{j' \in \textstyle\bigcup_{e\in n_{/\cong},|e|>1} e \,:\, j' \le j\}|\\
&\qquad\qquad = i+1\}),\\
&\qquad \lambda i \to f\ (\min\{j \,:\, |\{j' \in \textstyle\bigcup_{e\in n_{/\cong},|e|>1} e \,:\, j' \le j\}|\\
&\qquad\qquad = i+1\}))
\end{aligned}
$$

Note that we used algebraic simplifications like that $(\ker_\cong\ \mathit{id})$ is $\cong$ and that the kernel of an (up to the relevant $\cong$) injective function is the identity.

Let us now demonstrate the usefulness of reasoning with our extended container notions, based on three examples.

**An example proof with *Eq*-container morphisms.** We wish to show that $\mathit{nub} \circ \mathit{init}$ always returns a prefix of the result of just *nub*. Using our new setup, we can do this by showing that for every *Eq*-container value $c$,

$$
\mathbf{prefix}\ ((\mathit{nub}^c \circ \mathit{init}^c)\ c)\ (\mathit{nub}^c\ c) \tag{2}
$$

holds, where **prefix** is defined by

$$
\mathbf{prefix}\ (n_1,\cong_1,f_1)\ (n_2,\cong_2,f_2) \;\Leftrightarrow\; n_1 \le n_2 \;\wedge\; \forall i \in n_1.\ f_1\ i = f_2\ i
$$

To prove the desired statement, we take an arbitrary *Eq*-container value $c = (n,\cong,f)$ and first calculate both arguments to **prefix** in (2) above. We get

$$
\begin{aligned}
&\mathit{nub}^c\ (\mathit{init}^c\ (n,\cong,f))\\
&= \mathit{nub}^c\ (n-1,\cong,f)\\
&= (|(n-1)_{/\cong}|, \mathit{id}, \lambda i \to f\ (\min\{j \,:\, |\{[k]_\cong \,:\, k \le j\}| = i+1\}))
\end{aligned}
$$

and

$$
\mathit{nub}^c\ (n,\cong,f) = (|n_{/\cong}|, \mathit{id}, \lambda i \to f\ (\min\{j \,:\, |\{[k]_\cong \,:\, k \le j\}| = i+1\}))
$$

To verify the **prefix** property, we then have to establish the following statements:

1. $|(n-1)_{/\cong}| \leq |n_{/\cong}|$
2. $\forall i \in |(n-1)_{/\cong}|.\ f\ (\min\{j : |\{[k]_\cong : k \leq j\}| = i+1\}) = f\ (\min\{j : |\{[k]_\cong : k \leq j\}| = i+1\})$

of which the first is a simple property of factorisation (of a subset, with respect to the same equivalence relation), and of which the second is a syntactic tautology.

**Another example proof.** We wish to show that *rmSingles* $\circ$ *nub* always returns an empty list. Using our new setup, we can do this by showing that for every *Eq*-container value $c$, the container value $(rmSingles^c \circ nub^c)\ c$ has 0 in its length component. So let $c = (n, \cong, f)$ be an *Eq*-container value. Then:

$$\begin{aligned}
& rmSingles^c\ (nub^c\ (n, \cong, f)) \\
= {} & rmSingles^c\ (|n_{/\cong}|, id, \lambda i \to f\ (\min\{j : |\{[k]_\cong : k \leq j\}| = i+1\})) \\
= {} & (\sum_{e \in |n_{/\cong}|_{/id}, |e|>1} |e|, \cdots, \cdots) \\
= {} & (0, \cdots, \cdots)
\end{aligned}$$

**And yet another example proof.** We wish to show that *nub* is idempotent, i.e., $nub \circ nub = nub$. Using our new setup, we can do this by showing that $nub^c \circ nub^c = nub^c$. So let $c = (n, \cong, f)$ be an *Eq*-container value. Then:

$$\begin{aligned}
& nub^c\ (nub^c\ (n, \cong, f)) \\
= {} & nub^c\ (|n_{/\cong}|, id, h) \\
& \quad \text{with } h = \lambda i \to f\ (\min\{j : |\{[k]_\cong : k \leq j\}| = i+1\}) \\
= {} & (||n_{/\cong}|_{/id}|, id, \lambda i \to h\ (\min\{j : |\{[k]_{id} : k \leq j\}| = i+1\})) \\
= {} & (|n_{/\cong}|, id, h) \\
= {} & nub^c\ (n, \cong, f)
\end{aligned}$$

where except for the next-to-last one all steps are simply by applying definitions. That one interesting step is valid by $|m_{/id}| = m$ for every $m \in \mathbf{Nat}$,[4] and by the fact that for every $i \in \mathbf{Nat}$,

$$\begin{aligned}
\min\{j : |\{[k]_{id} : k \leq j\}| = i+1\} &= \min\{j : |\{\{k\} : k \leq j\}| = i+1\} \\
&= \min\{j : |\{\{0\}, \{1\}, \ldots, \{j\}\}| = i+1\} \\
&= i
\end{aligned}$$

### 3.2 The Type Class *Ord*

As a second example for the adjustment of the container notions to type classes, we consider the type class *Ord*. Similarly to the adjustment for the type class *Eq*,

[4] Note that our notation overloading is at work here, according to which $m \in \mathbf{Nat}$ can represent the actual number $m$ in one context and the set of numbers $\{0, \ldots, m-1\}$ in another context.

the container values and container morphisms have to be aware of the operation(s) now available on the formerly completely polymorphic list content. We use an approach analogous to that in Section 3.1, but replace equivalence relations by total preorders. We use $\mathcal{O}(M)$ to denote the class of all total preorders over a set $M$. For simplicity of notation, we freely regard a total preorder $\preceq$ on a subset $n$ of **Nat** as the total preorder $\preceq \cup \{(i,j) \mid (i \in n \wedge n \leq j) \vee (n \leq i \leq j)\}$ on **Nat** when appropriate.

As the remaining definitions, results, and proofs are in close analogy to the ones for the type class *Eq*, we will give them in condensed form only, and omit all proofs. For a type $\tau$ that is an instance of *Ord*, we denote by $\preceq_\tau$ the corresponding fixed (in a given program) total preorder:

$$\preceq_\tau = \{(x,y) \mid x :: \tau,\, y :: \tau,\, (x <= y) = \mathbf{True}\}$$

**Definition 4.** *Let $\tau$ be some type that is an instance of Ord. An Ord*-container value *of type $\tau$ is a triple $(n, \preceq, f)$ with $n :: \mathbf{Nat}$, $\preceq \in \mathcal{O}(\mathbf{Nat})$, and $f :: \mathbf{Nat} \to \tau$ a partial function such that $(\ker_{\preceq_\tau} f|_n) = (\preceq \cap (n \times n))$.*[5] *The set of all such container values is denoted by $\mathcal{C}^{Ord}(\tau)$.*

**Definition 5.** *We define functions $\square^{Ord}$ and $(\square^{Ord})^{-1}$ as*

$$\begin{aligned}
&\square^{Ord} && :: Ord\ \alpha \Rightarrow \mathcal{C}^{Ord}(\alpha) \to [\alpha] \\
&\square^{Ord}\ (n, \preceq, f) && = map\ f\ [0\ ..\ (n-1)] \\
&(\square^{Ord})^{-1} && :: Ord\ \alpha \Rightarrow [\alpha] \to \mathcal{C}^{Ord}(\alpha) \\
&(\square^{Ord})^{-1}\ \ xs && = (length\ xs, \ker_{\preceq_\alpha} (xs\ !!), xs\ !!)
\end{aligned}$$

**Definition 6.** *An Ord*-container morphism $(s, P)$ *is a family of pairs $(s_n^\preceq, P_n^\preceq)_{n \in \mathbf{Nat}, \preceq \in \mathcal{O}(\mathbf{Nat})}$ such that $s_n^\preceq :: \mathbf{Nat}$ and $P_n^\preceq :: \mathbf{Nat} \to \mathbf{Nat}$ a partial function with $(P_n^\preceq\ i) \in n$ for every $i \in s_n^\cong$.*

**Definition 7.** *Let $(n, \preceq, f)$ be an Ord-container value and $(s, P)$ an Ord-container morphism. The application of $(s, P)$ to $(n, \preceq, f)$ is defined as*

$$(s,P)\ (n, \preceq, f) = (s_n^\preceq, \ker_\preceq P_n^\preceq, f \circ P_n^\preceq)$$

**Theorem 3.** *For every function $g :: Ord\ \alpha \Rightarrow [\alpha] \to [\alpha]$, there exists an Ord-container morphism $(s, P)$ such that $g = \square^{Ord} \circ (s, P) \circ (\square^{Ord})^{-1}$.*

**Lemma 6.** *Let $g, g' :: Ord \Rightarrow [\alpha] \to [\alpha]$. Let $(s, P), (s', P')$ be Ord-container morphisms such that $g = \square^{Ord} \circ (s, P) \circ (\square^{Ord})^{-1}$ and $g' = \square^{Ord} \circ (s', P') \circ (\square^{Ord})^{-1}$. Then we have $g \circ g' = \square^{Ord} \circ (s, P) \circ (s', P') \circ (\square^{Ord})^{-1}$.*

Comparing the definitions of morphisms on ordinary container values and on *Ord*-container values, we can again easily translate the former ones into the latter ones, analogously to Note 1. Moreover, every *Eq*-container morphism can be viewed as an *Ord*-container morphism as well.

[5] Note that the kernel of a function can not only be taken over an equivalence relation.

*Note 2.* Every *Eq*-container morphism $(s_n^{\cong}, P_n^{\cong})_{n\in\mathbf{Nat},\cong\in\mathcal{E}(\mathbf{Nat})}$ can be viewed as the *Ord*-container morphism $(s_n^{\preceq\cap\succeq}, P_n^{\preceq\cap\succeq})_{n\in\mathbf{Nat},\preceq\in\mathcal{O}(\mathbf{Nat})}$.

Clearly, there are also *Ord*-container morphisms that are no ordinary container morphisms and no *Eq*-container morphisms. They correspond exactly to the functions of type $Ord \Rightarrow [\alpha] \to [\alpha]$ that are not of type $Eq\ \alpha \Rightarrow [\alpha] \to [\alpha]$ (or even of type $[\alpha] \to [\alpha]$). For example, the Haskell function

$$\begin{array}{lll} least & :: Ord\ \alpha \Rightarrow [\alpha] \to [\alpha] & \\ least\ [\,] & = [\,] & \\ least\ (x : xs) & = [go\ x\ xs] & \\ & \quad \textbf{where}\ go\ x\ [\,] & = x \\ & \qquad\qquad go\ x\ (y : ys) & = go\ (\textbf{if}\ x <= y\ \textbf{then}\ x\ \textbf{else}\ y)\ ys \end{array}$$

corresponds to:

$$\begin{array}{l} least^c = (s_n^{\preceq}, P_n^{\preceq})_{n\in\mathbf{Nat},\preceq\in\mathcal{O}(\mathbf{Nat})} \\ \quad \text{with}\ \ s_n^{\preceq} = \min\{n, 1\}\ \text{and} \\ \qquad\ P_n^{\preceq} = \lambda i \to \min\{j\ :\ \forall k \in (n \setminus j).\ j \preceq k\} \end{array}$$

## 4 Conclusion and Future Work

We have extended the ellipsis [4] or container [7] technique for reasoning about functions on lists to the case of the presence of element tests. The key insight was to use, in the proofs of Theorems 2 and 3, an extension of free theorems [8,9] to ad-hoc polymorphism à la type classes [10]. An obvious goal for future work is to see what needs to be done to make reasoning with our refined container-related notions, as we have performed on examples by hand, more effective and mechanisable. Just as the techniques of Bundy and Richardson [4] and Prince et al. [7] have to rely on good proof tactics for arithmetics, our method will have to rely on tactics that additionally take properties of equivalence relations and total preorders into account, and that can exploit algebraic notions like the kernel of a function over a relation, etc.

Another issue is that of transforming function definitions we want to reason about into suitable container morphism representations in the first place. As we have seen with examples like $rmSingles^c$, describing a structure change as a result of element tests can be somewhat involved to express by a mathematical formula. Only more practical experience will be able to tell how problematic that really is. Note, though, that container morphism representations need not necessarily be provided by the "customer" of a proof system. Indeed, in Bundy and Richardson's setup the container versions of list functions were used internally only, not exposed to the user.

How about further extensions? We have already mentioned that moving from lists to a broader range of data structures is largely orthogonal to taking element tests into account. A more challenging extension is to treat other type classes than *Eq* and *Ord*. The framework of free theorems is readily available for other type

classes as well. However, finding the right abstractions and morphism notions may appear to require new insights for each new class. For example, while both *Eq* and *Ord* mathematically correspond to relations, or to ways of *observing* elements of an unspecified type, what about type classes that provide ways of *constructing* elements via some operations, say class *Monoid*? Interestingly, recent work by Bernardy et al. [3] could shed some light here. For the purpose of testing (not verification), they essentially characterise polymorphic functions in terms of monomorphic inputs, such as characterising a function of type $[\alpha] \to [\alpha]$ in terms of its action on integer lists of the form $[1 \,..\, n]$. For more complicated types, in particular higher-order ones, they work from a classification of function arguments (typically themselves functions) into observers and constructors, and describe a methodology for finding fixed types and monomorphic inputs that completely determine a function's behaviour. Via the dictionary translation method, type class constraints lead to precisely such different kinds of function arguments, so there is a good chance for leverage here.

**Acknowledgements.** We thank the anonymous reviewers for their comments and suggestions for improving the paper.

## References

1. M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *FoSSaCS, Proc.*, volume 2620 of *LNCS*, pages 23–38. Springer, 2003.
2. M. Abbott, T. Altenkirch, and N. Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.
3. J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In *ESOP, Proc.*, volume 6012 of *LNCS*, pages 125–144. Springer, 2010.
4. A. Bundy and J. Richardson. Proofs about lists using ellipsis. In *LPAR, Proc.*, volume 1705 of *LNCS*, pages 1–12. Springer, 1999.
5. C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.
6. S.L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
7. R. Prince, N. Ghani, and C. McBride. Proving properties about lists using containers. In *FLOPS, Proc.*, volume 4989 of *LNCS*, pages 97–112. Springer, 2008.
8. J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proc.*, pages 513–523. Elsevier, 1983.
9. P. Wadler. Theorems for free! In *FPCA, Proc.*, pages 347–359. ACM, 1989.
10. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL, Proc.*, pages 60–76. ACM, 1989.