Testing Polymorphic Properties of Programs Parameterized Over Abstract Interfaces for Data Types

Robert Atkey and Patricia Johann

February 28, 2015

Abstract

ABSTRACT GOES HERE

1 Introduction

Containers should be mentioned here and are not. Do these eliminate the need for e-p pairs? Depends on whether or not any data type that can be embedded in a container is (isomorphic to) a container.

Given a formally specified property of a program, there have traditionally been two ways to verify that property. One is to provide a formal proof of the property and the other is to amass evidence for it by testing. At first glance, testing and proving appear to be orthogonal techniques for verifying programs: if a program property can be proved then there appears to be no need to test it, and if no proof is forthcoming then testing appears to be the only viable option. Recently, however, testing and proving have actually been used *in conjunction* to verify programs with particularly good effect.

One way to combine testing and proving is to use testing to debug programs before attempting to prove properties of them. This is the approach taken, for example, by Dybjer, Haiyan, and Takeyama [6], who have extended the proof assistant Agda/Alfa [1, 7] with a tool similar to QuickCheck [4] for randomly testing Haskell programs.

Another way to combine testing and proving is by first proving theorems that justify restricting attention to test data for programs that are "representative" in some sense, and then testing those programs only on those data. Determining suitable collections of representative test data can be tricky even for monomorphic programs [], but when polymorphic programs are involved, the situation becomes even more complicated. Indeed, the verification of polymorphic properties presents two orthogonal problems: First, testing can only be performed on specific monomorphic instances of polymorphic properties, and there are infinitely many monomorphic *test types* to choose from. Second, even once a monomorphic test type for a polymorphic property is chosen, there are typically infinitely many potential *test data* of that monomorphic type to choose from. The problem of infinitely many test data is already present for monomorphic properties; fortunately, tools such as QuickCheck and the aforementioned extension of Agda/Alfa

CHECK THIS!

can efficiently test specific monomorphic instances of polymorphic properties. However, the problem of infinitely many test types is specific to polymorphic properties, and it is not at all obvious that there is a collection of monomorphic instances that is sufficiently small to be computationally viable and yet whose verification implies verification of the original property. In particular, given an arbitrary polymorphic property, it is not at all obvious that there wil always be a *single* monomorphic type instance of that property that is representative of *all* such instances.

Bernardy, Jansson, and Claessen [3] have shown that, for a certain class of polymorphic properties, such representative monomorphic instances do exist and, moreover, that restricting attention to them can improve the performance of modern testing tools. The approach Bernardy, Jansson, and Claessen take to managing the testing complexity of polymorphic properties relies on *parametric polymorphism*, sometimes called just *polymorphism* or just *parametricity*. A *(parametric) polymorphic* function is, intuitively speaking, a polymorphic function that is defined at every type by the same type-uniform algorithm. Because parametric polymorphic functions cannot perform type-specific operations, their behavior is constrained in ways that allow the derivation of powerful reasoning principles for them solely from their types. Bernardy, Jansson, and Claessen use this observation to show that a representative monomorphic instance of a polymorphic property p

Not an arbitrary property — they all look like equalities! (Yes — see (3) on page 6.) Do we need the equalities to be computable? Categorically, pullbacks arise whenever equalities arise

whose type is in what they call *canonical form* — i.e., that uses an algebra for a functor F to construct data of the type over which it is parametrized and allows only base type observations of these data — can be taken to be (the carrier of) the initial F-algebra. Intuitively, this is plausible because initiality of this F-algebra entails that it is large enough to contain sufficient data to construct a counterexample to p if one exists, but small enough not to be burdened with data that is extraneous to that enterprise; formally, it is proved by instantiating the instance for p of Reynolds' parametricity theorem [14]. This result allows a significant reduction in testing for any polymorphic property whose type is in canonical form and has already been successfully integrated into QuickCheck.

Compare this result with what you get with known treatment of parametricity for existentials (data types). Standard treatment doesn't handle equations.

test vs verify

In this paper, we show that representative monomorphic instances exist for a significantly larger class of polymorphic program properties than that considered in [3]. Like the results of Bernardy, Jansson, and Claessen, all of the results in this paper are derived from standard results about parametric polymorphism. However, unlike theirs, our results are not restricted to properties whose types are (actually, embed) in the canonical form given in [3]. Our main result is the derivation of monomorphic instances of polymorphic properties in the presence of constraints that can be used to efficiently test such properties. In particular, we show how the monomorphic instances we derive in the presence of constraints can be used to efficiently test properties of programs that are parameterized over abstract interfaces for data types. We further show how the monomorphic instances we derive can be used to obtain, from the abstract interfaces of data types alone, implementations of those data types that are sufficient for efficiently testing polymorphic properties of programs parameterized over them.

Early work on testing polymorphic functions that constructs such representative test data goes

back to the classical result of Knuth [11] that verifying a sorting network only has to done on booleans. A more recent result by Voigtländer [15] similarly shows how parametricity can be used to reduce the space of test data for a function with the polymorphic type of a scan function and efficiently test whether or not the function is indeed a scan function. Knuth's and Voigtländer's results are both instances of those of [3], and hence of those obtained here.

2 Testing Dijkstra's Algorithm: An Example

To illustrate how the results of this paper can be used to verify programs parameterized over abstract interfaces for data type, and can thus be used to derive implementations of data types sufficient for efficient testing of properties of such programs, consider the problem of testing Dijkstra's shortest path algorithm [5]. This algorithm takes as input a graph whose edges are labelled with nonnegative "distances" between their end nodes, together with a specified start node, and finds the path with the overall shortest distance between the start node and every other node in the graph. This is done by assigning each node a potential distance from the start node (usually 0 for the start node and ∞ for all others) and improving these step-by-step on each iteration of the algorithm. Initally, the start node is the algorithm's current node. On each iteration, the unvisited node with the smallest potential distance from the current node becomes the new current node, and the potential distance from the start node to each neighbor of the current node is updated to the smallest one known thus far. After all of its neighbors have been considered, the current node is marked as visited. The algorithm finishes when either the new current node has already been visited or the potential distance of each unvisited node is ∞.

Since a priority queue is usually used to keep track of the unvisited nodes in a graph, let's suppose we want to perform random testing on an implementation of Dijkstra's algorithm that is parameterized over an abstract interface for this data type. This kind of testing is useful in situations where we either cannot, or do not want to, commit *a priori* to any particular implementation of a data type. The abstract interface for the priority queue data type over which Dijkstra's algorithm is parameterized could, for example, comprise the following minimal set of operations needed to implement the algorithm:

$$\begin{array}{lcl} \mathit{empty} & : & t \\ \mathit{insert} & : & (a, \mathbb{N}+1) \to t \to t \\ \mathit{getmin} & : & t \to \mathit{Maybe}\ (a, \mathbb{N}+1, t) \end{array}$$

Here, the operation *empty* constructs an empty priority queue. The operation *insert* takes as input a pair of type $(a, \mathbb{N}+1)$ representing a node in the input graph to Dijkstra's algorithm, together with its distance — which can be either a natural number or ∞ — from the algorithm's start node, and an existing priority queue. It returns a new priority queue in which that node has been properly inserted. Finally, the results of a successful application of the operation *getmin* to its priority queue argument are the item with the shortest distance from the start node, that shortest distance itself, and the priority queue that remains when the node with the shortest associated distance is removed from that priority queue. The presence of the *Maybe* type reflects the possibility that removal fails, as will be the case, e.g., if the queue is empty. As expected, the above abstract interface is polymorphic over the type t that implements priority queues.

As is customary, we require any correct implementation of the above abstract interface for priority queues to satisfy the following constraints, which are also polymorphic in t:

$$
\begin{array}{lcl}
\mathit{getmin}\ \mathit{empty} & = & \mathit{Nothing} \\
\mathit{getmin}\ (\mathit{insert}\ (a,d)\ t) & = & \mathit{case}\ \mathit{getmin}\ t\ \mathit{of} \\
 & & \quad \mathit{Nothing} \longrightarrow \mathit{Just}\ (a,d,t) \\
 & & \quad \mathit{Just}\ (a',d',t') \longrightarrow \mathit{if}\ d < d' \\
 & & \qquad\qquad \mathit{then}\ \mathit{Just}\ (a,d,\mathit{insert}\ (a',d')\ t') \\
 & & \qquad\qquad \mathit{else}\ \mathit{Just}\ (a',d',\mathit{insert}\ (a,d)\ t')
\end{array}
$$

These constraints give us some handle on the behavior of *getmin*, but they do not actually constitute *definition* of *getmin*. Indeed, *empty* and *insert* cannot actually be constructors of an arbitrary data type t, so while the above constraints may *appear* to define *getmin* by pattern matching, in fact they do not. Moreover, the above constraints specify *getmin*'s behavior *only* for data of type t that are obtained as results of calls to the *empty* and *insert* operations, rather than for arbitrary data of type t.

Can there be other such data?

Since, for any t, there can be many triples of operations with the specified behavior, our implementation of Dijkstra's algorithm must be tested not only for randomly generated graphs, but for randomly generated operations having that behavior as well.

Or must it?

Properties vs types.

The properties in BJC and ours are all equalities...

In [3], Bernardy, Jansson, and Claessen show that any polymorphic property whose type can be embedded in a type of *canonical form* — i.e., any property whose arguments can be split into *constructors* of data of the abstracted type and base-type *observations* on data of that type — can be completely tested by testing only the *monomorphic* instance of that property at the carrier of the initial algebra for the functor derived from the type of its constructors. (This result is reproduced as Theorem 1 below.) In practice, it is often necessary to first rewrite the type of a program property of interest so that it is in canonical form. For example, to determine whether or not two implementations *filter1* and *filter2* of the polymorphic function that filters lists according to its predicate argument are the same, we would test the following polymorphic property:

introduce notation for lists

$$\forall\alpha.\,\forall p : \alpha \to \mathrm{Bool}.\,\forall xs : [\alpha].\,\mathit{filter}_1\ \alpha\ p\ xs = \mathit{filter}_2\ \alpha\ p\ xs \tag{1}$$

For this, Bernardy, Jansson, and Claesson first rewrite the type

$$\forall\alpha.\,(\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha]$$

of *filter* as

$$\forall\alpha.\,(\alpha \to \mathrm{Bool}) \to \mathbb{N} \to (\mathbb{N} \to \alpha) \to [\alpha]$$

Here, the argument type $[\alpha]$ of *filter* is decomposed into a natural number observation representing the length of the input list, and a constructor that maps each natural number i to the i^{th} element (of type α) of that list. The predicate argument, which consumes data of type α rather than producing it, is a second observation. Since the carrier of the initial algebra for the functor $F\alpha = \mathbb{N}$ derived

from the type $\mathbb{N} \to \alpha$ of the (one and only) constructor argument is $\mathbb{N}$, the above property can be completely tested by testing it only for data of type $\mathbb{N}$. The structure map $in : \mathbb{N} \to \mathbb{N}$ of the initial F-algebra maps each natural number i to the i^{th} element (now of type $\mathbb{N}$) of that list; without loss of generality we may take in to be id. Of course, the sameness of the two implementations must still be tested on all observations $p : \mathbb{N} \to \text{Bool}$. As shown more formally in Example 1 below, the results of [3] guarantee, overall, that (1) can be completely tested by testing only this monomorphic instance:

$$\forall n : \mathbb{N}.\forall p : \mathbb{N} \to \text{Bool}.\ \mathit{filter}_1\ \mathbb{N}\ p\ [1..n] = \mathit{filter}_2\ \mathbb{N}\ p\ [1..n] \tag{2}$$

Intuitively this makes sense: the $\forall$-quantifier in the type of the property (1) entails that it cannot use any information about the specific type α over which it is parameterized. This ensures that if the property has a counterexample for some list of elements of some type α, then it has a counterexample for some list of elements of type $\mathbb{N}$. And since the elements in the list must all be treated in a type-independent manner, if the property has a counterexample for some list of elements of type $\mathbb{N}$, then it has a counterexample for the list of elements $[1..n]$.

Curiously, in Example 1 of their paper, BJC derive

$$\forall n : \mathbb{N}.\forall p : \alpha \to \text{Bool}.\ \mathit{filter}_1\ \mathbb{N}\ p\ [X_1..X_n] = \mathit{filter}_2\ \mathbb{N}\ p\ [X_1..X_n]$$

as their final monomorphic property. This is strange, because their Theorem 5 (our Theorem 2 below) actually supports derivation of the monomorphic property in (2) above. See Example 1 below.

Unfortunately, however, the type of our abstract interface for priority queues — and hence the type of any property of any implementation of Dijkstra's algorithm that is parameterized over this interface — is not quite so easy to work with. For one thing, the constraints on *getmin* must be taken into account, and although polymorphic testing in the presence of constraints was treated intuitively Example 10 of [3], no general methodology for incorporting them was developed there. For another thing, the return type of *getmin* is neither the parameterized variable t itself, as would be needed for *getmin* to be a constructor, nor a base type, as would be needed for *getmin* to be an observation. The type of *getmin* thus does not appear to embed in a type of canonical form. The techniques of Bernardy, Jansson, and Claessen therefore are not immediately applicable to the testing of Dijkstra's algorithm.

revise below... 1. We can incorporate constraints w/no coalgebraic parts. This is Theorem 5. This applies to Dijkstra's algorithm using Lans, but not nicely because algebras are hard to construct. 2. We can even handle coalgebraic constraints when they are in extended canonical form and thus make definitions on initial algebra of regular constraints. This is Theorem 6. This applies nicely to Dijkstra. An orthogonal positioning issue: deriving impls from specs? Or polytesting with coalgebraic constraints?

However, as we show in Section 4 below, constraints can indeed be incorporated into the polymorphic testing problem. A categorical technique that has already proved useful in the study of nested types and GADTs [9, 10] guarantees that the results of [3] can be extended to a provably correct technique for testing polymorphic properties of programs whose arguments can be *observing constructors*

Below called destructors...? Need a formal definition of observing constructor, and better terminology. In some sense they are redundant, because the constraints specify their behavior entirely by what they do on data constructed by the other constructors. Look for a maximal set of constructors (maybe none exists) for which the constraints specify the behavior of the other constructors in terms of the ones in that set?

, in addition to (regular) constructors and observations. Intuitively, an observing constructor is an operation, like *getmin*, whose argument type involves the abstracted variable, but whose type can be rewritten so that its return type is just the abstracted variable. Moreover, as we see in Section 5, if an observing constructor satisfies *extended canonical form constraints* — i.e., constraints that prescribe the behavior of the operation on all data constructed by the property's (regular) constructors — then those constraints actually give *definitions* of the observing constructors on the carrier of the initial algebra for the types of the type's (regular) constructors, i.e., on the monomorphic testing type derived using the techniques of [3] for the type comprising just the types of the constructor and observation arguments to the property. Solutions to polymorphic testing problems thus allow us to turn the logical content of abstract specifications of observing constructors into the computational content of partial implementations of those operators. As a result, they help advance progress toward truly executable specifications.

Is "executable specifications" really the right point of view? We simply require that the coalgebraic constraints DEFINE the observing constructors on the initial (F, Q)-algebra.

In the specific case of our abstract interface for priority queues, *getmin* is an observing constructor because, as discussed in Section 5, its type can be rewritten as

$$\forall b.\ (Eql\ (Maybe\ (a, \mathbb{N}+1, b))\ t, t) \to t$$

and this is indeed of the form $Ft \to t$ for the functor

Check that this is a functor in t.

$Ft = \forall b.\ (Eql\ (Maybe\ (a, \mathbb{N}+1, b))\ t, t)$.

So Lans can be used to allow us to apply Theorem 5. But this requires building difficult algebras. A second approach uses Theorem 6 and is nicer when it is applicable because the algebras are easier to construct. But it is not always applicable even when the first method is. Discuss that for Dijkstra now.

Moreover, the constraints satisfied by *getmin* are canonical form constraints because they specify the behavior of *getmin* on all data constructed using *empty* and *insert*. Direct application of Theorem 5 then shows that we can take t to be the carrier $[(a, \mathbb{N}+1)]$ of the initial algebra for the functor $Ft = 1 + ((a, \mathbb{N}+1), t)$ derived from the types of *empty* and *insert*. That is, we can set

$$\begin{array}{lcl} t & = & [(a, \mathbb{N}+1)] \\ empty & = & [] \\ insert\ (a, d)\ t & = & cons\ (a, d)\ t \end{array}$$

Theorem 5 also ensures that *getmin* satisfies the specialization of its constraints when t is $[(a, \mathbb{N}+1)]$. This, in turn, guarantees that if our implementation of Dijkstra's algorithm is correct with respect to an implementation of priority queues as lists of pairs of type $(a, \mathbb{N}+1)$ then it is correct with respect to *all* possible implementations of the above abstract interface for priority queues. Moreover, the constraints on *getmin* actually constitute a *definition* of this operation on the data in the initial

algebra $[(a, \mathbb{N} + 1)]$ for the constructors *empty* and *insert*, i.e., on the derived monomorphic test type. We have therefore moved from a collection of constraints on *getmin* that must be satisfied by any correct implementation of the abstract interface for priority queues to an actual *definition* of *getmin* — by pattern matching on data of type $[(a, \mathbb{N} + 1)]$, since initiality ensures that all data of this type is constructed by *empty* and *insert*.

As we see in Section 5, this means that getmin needn't be quantified over when testing. And we only need to test on initial algebra for algebraic part. Much nicer than first approach!

Overall, then, we see that it is not necessary to test properties of Dijkstra's algorithm against every implementation of priority queues, as originally feared. We need only test against abstract interfaces of data types whose constructors interact in constrained ways with the other operations of those data types. In the case of Dijkstra's algorithm, this means that we need only test against the naive implementation given in terms of lists, since if a property holds for Dijkstra's algorithm with that naive implementation of priority queues, then it will hold for Dijkstra's algorithm with *any* implementation of priority queues. Moreover, the definition of *getmin* on lists can be used for efficient polymorphic testing, so all we need in order to generate to test properties of Dijkstra's algorithm is a random graph to test on. This is the best we can hope for, since any graph algorithm will need to be shown to work for all graphs (of the kind it is intended to processs). More generally, the observations above justify an approach to testing where only the primary parts of the property need to be tested, rather than *both* the primary parts of the property *and* all the auxiliary data structures used by algorithms parameterized by them. This has the potential to dramatically reduce the random test data that needs to be generated. After all, the testing of a property parametrized on an abstact interface for a data type requires *some* actual implementation of that data type — just a specification will not suffice. What this paper shows is that parametricity allows us to generate data and test a polymorphic property relative to the "simplest" implementation of that abstract interface we can construct, and thereby to conclude correctness of that property relative to all implementations.

Of course in practice we will want to use efficient implementations of abstract interfaces for data types for testing, such as those found in Okasaki [13].

3 The Polymorphic Testing Problem

The *polymorphic testing problem* can be formally stated as follows:

> Let $\sigma[\alpha]$ be a type expression with a free type variable α, and let H be a definable functor. Given a pair of functions h_1, h_2 of type $\forall \alpha : *. \sigma[\alpha] \to H\alpha$, find a *monomorphic* sufficient condition that implies the following property:
>
> $$\forall \alpha : *. \forall x : \sigma[\alpha]. \; h_1 \; \alpha \; x = h_2 \; \alpha \; x \tag{3}$$

The idea here is that h_1, say, is a polymorphic function to be tested against a reference implementation h_2 that serves as a specification of the computational behavior h_1 should have. This problem was previously considered by Bernardy, Jansson, and Claessen in [3]. Their main result is:

Theorem 1. *Let $\sigma[\alpha]$, H, h_1 and h_2 be as in the description of the polymorphic testing problem. If there exist functors $\{G_i\}_{i \in I}$ and F, and types $\{O_i\}_{i \in I}$ such that*

$$\sigma[\alpha] = (\Pi_{i \in I}(G_i \alpha \to O_i)) \times (F\alpha \to \alpha)$$

and if there exists an initial F-algebra $(\mu F, \mathrm{in} : F(\mu F) \to \mu F)$*, then the following condition is a solution for this instance of the polymorphic testing problem:*

$$\forall p : \Pi_{i\in I}(G_i(\mu F) \to O_i).\ h_1\ (\mu F)\ (p, \mathrm{in}) = h_2\ (\mu F)\ (p, \mathrm{in}) \tag{4}$$

We will say that a type of the form of σ is in *BJC canonical form*. The proof that (4) is a solution to the polymorphic testing problem relies on the following lemma. It is a consequence of the parametricity property for polymorphic functions whose domains are in BJC canonical form.

Lemma 1. *Let* $h : \forall\alpha : *.\ (\Pi_{i\in I}(G_i\alpha \to O_i)) \times (F\alpha \to \alpha) \to H\alpha$*. Assume that there is an initial F-algebra* $(\mu F, \mathrm{in} : F(\mu F) \to \mu F)$*. If* $(\!|f|\!)$ *is the unique F-algebra homomorphism from* $(\mu F, \mathrm{in})$ *to* (α, f)*, then*

$$\begin{array}{l}\forall\alpha : *.\forall p : \Pi_{i\in I}(G_i\alpha \to O_i).\forall f : F\alpha \to \alpha.\\ \qquad h\ \alpha\ (p, f) = H\ (\!|f|\!)\ (h\ \mu F\ (\langle p_i \circ G_i(\!|f|\!)\rangle_{i\in I}, \mathrm{in}))\end{array}$$

Proof. We invoke the parametricity property for h (specialised to the case of functional relations):

$$\begin{array}{l}\forall\alpha_1, \alpha_2 : *.\ \forall r : \alpha_1 \to \alpha_2.\\ \quad \forall p' : \Pi_{i\in I}(G_i\alpha_1 \to O_i), p : \Pi_{i\in I}(G_i\alpha_2 \to O_i).\\ \qquad (\forall i \in I.\ p'_i = p_i \circ G_i r) \Rightarrow\\ \qquad\quad \forall f_1 : F\alpha_1 \to \alpha_1, f_2 : F\alpha_2 \to \alpha_2.\\ \qquad\qquad r \circ f_1 = f_2 \circ Fr \Rightarrow\\ \qquad\qquad\quad H\ r\ (h\ \alpha_1\ (p', f_1)) = h\ \alpha_2\ (p, f_2)\end{array} \tag{5}$$

Given $\alpha : *$, $p : \Pi_{i\in I}(G_i\alpha \to O_i)$ and $f : F\alpha \to \alpha$, we instantiate (5) with $\alpha_1 = \mu F$, $\alpha_2 = \alpha$, $r = (\!|f|\!)$, $p' = \langle p_i \circ G_i(\!|f|\!)\rangle_{i\in I}$, $p = p$, $f_1 = \mathrm{in}$ and $f_2 = f$. The condition on p' and p in the third line holds by definition, and the condition on f_1 and f_2 in the fourth and fifth lines holds by the fact that $(\!|f|\!)$ is an F-algebra homomorphism. □

We can use Lemma 1 to prove that (4) is a solution to the polymorphic testing problem.

Proof. (Theorem 1) We want to prove that (4) implies the following specialization of (3) to the current situation:

$$\forall\alpha : *.\forall p : \Pi_{i\in I}(G_i\alpha \to O_i).\forall f : F\alpha \to \alpha.\ h_1\ \alpha\ (p, f) = h_2\ \alpha\ (p, f)$$

Given $\alpha : *$, $p : \Pi_{i\in I}(G_i\alpha \to O_i)$ and $f : F\alpha \to \alpha$, we reason as follows:

$$\begin{array}{ll} & h_1\ \alpha\ (p, f)\\ = & \qquad\qquad \{\text{Lemma 1 applied to } h_1\}\\ & H\ (\!|f|\!)\ (h_1\ \mu F\ (\langle p_i \circ G_i(\!|f|\!)\rangle_{i\in I}, \mathrm{in}))\\ = & \qquad\qquad \{\text{Application of (4)}\}\\ & H\ (\!|f|\!)\ (h_2\ \mu F\ (\langle p_i \circ G_i(\!|f|\!)\rangle_{i\in I}, \mathrm{in}))\\ = & \qquad\qquad \{\text{Lemma 1 applied to } h_2\}\\ & h_2\ \alpha\ (p, f)\end{array}$$

□

3.1 Solution Reduction via Embedding-Projection Pairs

In fact, Bernardy, Jansson, and Claessen do not require that the type of h_1 and h_2 is actually *of* canonical form as stated in Theorem 1, but rather that the type of h_1 and h_2 be *embeddable* into a type of canonical form. Indeed, they prove the generalization in Theorem 2 below, which reduces solutions to the polymorphic testing problem for functions of types embeddable into canonical types to solutions to the polymorphic testing problem for functions *actually* of canonical type. We require the following definition in order to state the generalization precisely:

Definition 1. *Let $\sigma[\alpha]$ and $\tau[\alpha]$ be type expressions with a single free type variable. An* embedding-projection pair $(e, p) : \sigma[\alpha] \hookrightarrow \tau[\alpha]$ *consists of a pair of polymorphic functions $e : \forall\alpha.\ \sigma[\alpha] \to \tau[\alpha]$ and $p : \forall\alpha.\ \tau[\alpha] \to \sigma[\alpha]$ such that $\forall\alpha : *.\ (p\ \alpha) \circ (e\ \alpha) = \mathrm{id}_{\sigma[\alpha]}$.*

We then have:

Theorem 2. *If $\tau[\alpha] = (\Pi_{i\in I}(G_i\alpha \to O_i)) \times (F\alpha \to \alpha)$ is a type with a free type variable α, if there exists an initial F-algebra $(\mu F, \mathrm{in} : F(\mu F) \to \mu F)$, if $(e, p) : \sigma[\alpha] \hookrightarrow \tau[\alpha]$ is an embedding-projection pair, if H is a definable functor, and if h_1 and h_2 are functions of type $\forall\alpha : *.\ \sigma[\alpha] \to H\alpha$, then*

$$\begin{array}{ll} & \forall s : \Pi_{i\in I}(G_i(\mu F) \to O_i).\ h_1\ \mu F\ (s, \mathrm{in}) = h_2\ \mu F\ (s, \mathrm{in}) \\ \Rightarrow & \\ & \forall\alpha : *.\forall x : \sigma[\alpha].\ h_1\ \alpha\ x = h_2\ \alpha\ x \end{array}$$

Proof. Given $\alpha : *$ and $x : \sigma[\alpha]$, we can apply Theorem 1 to $h_1 \circ p$ and $h_2 \circ p$, which both have type $\forall\alpha.\ \tau[\alpha] \to H\alpha$ to get that

$$\begin{array}{ll} & \forall s : \Pi_{i\in I}(G_i(\mu F) \to O_i).\ (h_1 \circ p)\ \mu F\ (s, \mathrm{in}) = (h_2 \circ p)\ \mu F\ (s, \mathrm{in}) \\ \Rightarrow & \qquad \{\text{by Theorem 1}\} \\ & \forall\alpha : *.\ \forall y : \tau[\alpha].\ (h_1 \circ p)\ \alpha\ y = (h_2 \circ p)\ \alpha\ y \\ \Rightarrow & \qquad \{\text{take } q = e\,\alpha\,x\} \\ & \forall\alpha : *.\ \forall x : \sigma[\alpha].\ (h_1 \circ p)\ \alpha\ (e\,\alpha\,x) = (h_2 \circ p)\ \alpha\ (e\,\alpha\,x) \\ \Leftrightarrow & \qquad \{(e, p) \text{ is an embedding-projection pair}\} \\ & \forall\alpha : *.\ \forall x : \sigma[\alpha].\ h_1\ \alpha\ x = h_2\ \alpha\ x \end{array}$$

□

This representation of lists is used by Bundy's "ellipses".

Example 1. *Suppose we want to compare two functions — suggestively named $filter_1$ and $filter_2$ — of type*

$$\forall\alpha : *.\ (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha] \tag{6}$$

In this instance of the polymorphic testing problem, we have (up to currying) $\sigma[\alpha] = (\alpha \to \mathrm{Bool}) \times [\alpha]$ and $H\alpha = [\alpha]$. It is not difficult to see that $\sigma[\alpha]$ embeds in (but is not isomorphic to) the canonical form type

$$\forall\alpha : *.\ (\alpha \to \mathrm{Bool}) \times \mathbb{N} \times (\mathbb{N} \to \alpha) \to [\alpha]$$

This type is the instance of the type schema from Theorem 1 for $G_1\alpha = \alpha$, $O_1 = \mathrm{Bool}$, $G_2\alpha = 1$, $O_2 = \mathbb{N}$, and $F\alpha = \mathbb{N}$, so $(\mu F, \mathrm{in}) = (\mathbb{N}, id)$, and, by Theorem 2, the corresponding monomorphic instance

$$\forall n : \mathbb{N}.\ \forall p : \mathbb{N} \to \mathrm{Bool}.\ filter_1\ \mathbb{N}\ (n, p, id) = filter_2\ \mathbb{N}\ (n, p, id)$$

of (4) is a solution to the polymorphic testing problem for the type in (6). More concretely, this instance is

$$\forall n : \mathbb{N}.\forall p : \mathbb{N} \to \text{Bool}.\ \mathit{filter}_1\ \mathbb{N}\ p\ [1..n] = \mathit{filter}_2\ \mathbb{N}\ p\ [1..n] \tag{7}$$

As promised, Theorem 2 formally justifies the sufficiency of the condition in Equation (2) of the introduction.

In Example 1 of their paper, Bernardy, Jansson, and Claessen derive the sufficient monomorphic test property

$$\forall n : \mathbb{N}.\forall p : \alpha \to \text{Bool}.\ \mathit{filter}_1\ \mathbb{N}\ p\ [X_1..X_n] = \mathit{filter}_2\ \mathbb{N}\ p\ [X_1..X_n]$$

for *filter*. Interestingly, this property still requires a fixed function X that maps into α to construct its test list, as well as a predicate of type $\alpha \to$ Bool. By contrast, the sufficient monomorphic condition derived by applying Theorem 5 in [3] (i.e., Theorem 2 above) actually justifies requiring only testing for the identity function on μF (which is $\mathbb{N}$ here) and predicates of type $\mathbb{N} \to$ Bool (see Equation 7). The weaker result (i.e., the more complex test property) derived in [3] appears to be a consequence of the informal, intuitive approach Bernardy, Jansson, and Claessen take to applying their theorem in practice. Our more formal application of the theorem here gets additional prunings of the testing search space out of it. A similar comment applies to all of the examples in [3].

Note that although p will be applied only to the list $[1..n]$, it must be defined for all natural numbers. This means that many more maps p will be generated for testing than are either used or necessary. We will revisit this example in the next section, where we will formally prove that the domain of p can be restricted to avoid this overgeneration of test data.

3.2 Polymorphic Testing for Containers: Eliminating the Need for e-p Pairs

Since Bernardy, Jansson, and Claessen prove their results only for types that embed in types of BJC canonical form, they must embed any type of any property of interest into such a type before solving its poltmorphic testing problem. In particular, if a property involves a dependent container [], then they must embed the dependent container into a non-dependent container before proceeding as usual. When the embedding is not an isomorphism, this has the effect of enlarging the position type for the container by erasing the type index, which in turn causes the overabundance of testing observed at the end of the last section to be performed. In this section, we prove that working directly with containers supports even more pruning of the testing search space. In effect, we show that position types need never be taken to be "too big" simply to make embedding and projection possible, and thus ensure that no more testing is done for properties involving dependent containers than is absolutely necessary. The upshot is that while embedding-projection pairs do prune the testing search space considerably, we can derive even greater pruning by containerizing the domain types of polymorhic functions to be tested instead.

Our main theorem for containers is:

Theorem 3. *Let $\sigma[\alpha]$, H, h_1 and h_2 be as in the description of the polymorphic testing problem. If there exist functors $\{G_i\}_{i\in I}$, a type S, functors Fs for all $s : S$, and types $\{O_i\}_{i\in I}$ such that*

$$\sigma[\alpha] = (\Pi_{i\in I}(G_i\alpha \to O_i)) \times \Sigma s : S.\ (Fs\alpha \to \alpha) \tag{8}$$

and for all $s : S$ there exists an initial Fs-algebra $(\mu(Fs), \text{in} : Fs(\mu(Fs)) \to \mu(Fs))$, then the following condition is a solution for this instance of the polymorphic testing problem:

$$\forall s : S.\forall p : \Pi_{i\in I}(G_i(\mu(Fs)) \to O_i).\ h_1\ \mu(Fs)\ (p, s, \text{in}) = h_2\ \mu(Fs)\ (p, s, \text{in}) \tag{9}$$

The proof that (9) is a solution to the polymorphic testing problem is almost identical to the proof of Theorem 1 above.

Example 2. *Suppose again that we want to compare two functions $filter_1$ and $filter_2$ of type*

$$\forall\alpha : *.\ (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha] \tag{10}$$

It is not difficult to see that $\sigma[\alpha]$ is isomorphic *to the dependent container type*

$$\forall\alpha : *\ (\alpha \to \text{Bool}) \times \Sigma n : \mathbb{N}.\ (\text{Fin}\ n \to \alpha)$$

where Fin n *is the type of natural numbers less than n. This type is the instance of the type schema (8) for $G\alpha = \alpha$, $S = \mathbb{N}$ and $Fn\alpha =$ Fin n, so Theorem 3 ensures that the corresponding monomorphic instance*

$$\forall n : \mathbb{N}.\forall p : \text{Fin}\ n \to \text{Bool}.\ \mathit{filter}_1\ (\text{Fin}\ n)\ (p, n, \mathit{id}) = \mathit{filter}_2\ (\text{Fin}\ n)\ (p, n, \mathit{id})$$

of (9) is a solution to the polymorphic testing problem for the type (6). More concretely, this instance is

$$\forall n : \mathbb{N}.\forall p : \text{Fin}\ n \to \text{Bool}.\ \mathit{filter}_1\ (\text{Fin}\ n)\ p\ [1..n] = \mathit{filter}_2\ (\text{Fin}\ n)\ p\ [1..n] \tag{11}$$

Note that the condition in Equation 11 is much easier to test than the corresponding one given in Example 1 of [3]. Because the domain of p is restricted from $\mathbb{N}$ to Fin n, only finitely many maps p need to be considered here, rather than the infinitely many requiring testing by [3]. Moreover, each such map is itself finite, and is thus simpler to represent, generate, and work with than its "corresponding" infinite domain map. Finally, Bernardy, Jansson, and Claessen discuss after Theorem 6 in [3] how to most efficiently generate test values for sufficient monomorphic properties for noncontainerized types — by generating results of applying projections to observations directly, rather than first generating observations and the calculating their projections. But since embeddings and projections are both identity functions when types are containerized, and since a type that can be embedded in a container type must itself be representable as a container type

Check this.

, this issue simply disappears with containerization.

We say that a type as in Theorem 3 whose domain type $\sigma[\alpha]$ is of the containerized form in Equation 8 is in *AJ canonical form*.

3.3 Polymorphic Testing with Multiple Type Variables

All of the results appearing in this section so far generalize to properties whose types are parameterized over multiple type variables. When there is no dependency between the type variables in the type of a property to test, then those variables can, in effect, be monomorphized in parallel. This makes sense intuitively, and is precisely the approach taken in Example 7 of [3]. We now show how Theorem 3 justifies this formally for a containerized version of that example.

Example 3. *Suppose we want to test two implementations $isPrefixOf_1$ and $isPrefixOf_2$ of*

$$\mathit{isPrefixOf} : \forall\alpha\,\beta : *.\ (\alpha \to \beta \to \text{Bool}) \to [\alpha] \to [\beta] \to \text{Bool}$$

This type is isomorphic to the dependent container type

$$\forall\alpha\,\beta : *.\ (\alpha \to \beta \to \text{Bool}) \to \Sigma n : \mathbb{N}.\ (\text{Fin}\ n \to \alpha) \to \Sigma m : \mathbb{N}.\ (\text{Fin}\ m \to \beta) \to \text{Bool}$$

Because — and only because — the type arguments α and β are completely independent of one another, this type can in turn be embedded into the type

$$\forall(\alpha,\beta):*.\ ((\alpha,\beta)\to \text{Bool})\ \times\ \Sigma(n,m):(\mathbb{N},\mathbb{N}).\ ((\text{Fin}\ n,\text{Fin}\ m)\to(\alpha,\beta))\to \text{Bool}$$

The technique of Theorem 3 can be applied to get that the following monomorphic instance of (9) is a solution to the polymorphic testing problem for the type of isPrefixOf:

$$\begin{array}{l}\forall n:\mathbb{N}.\,\forall m:\mathbb{N}.\,\forall p:(\text{Fin}\ n,\text{Fin}\ m)\to\text{Bool}.\\ \quad \mathit{isPrefixOf}_1\ (\text{Fin}\ n,\text{Fin}\ m)\ (p,n,m,\mathit{id}) = \mathit{isPrefixOf}_2\ (\text{Fin}\ n,\text{Fin}\ m)\ (p,n,m,\mathit{id})\end{array}$$

This instance can be rewritten as

$$\begin{array}{l}\forall n:\mathbb{N}.\,\forall m:\mathbb{N}.\,\forall p:\text{Fin}\ n\to\text{Fin}\ m\to\text{Bool}.\\ \quad \mathit{isPrefixOf}_1\ (\text{Fin}\ n)\ (\text{Fin}\ m)\ p\ [1..n]\ [1..m] = \mathit{isPrefixOf}_2\ (\text{Fin}\ n)\ (\text{Fin}\ m)\ p\ [1..n]\ [1..m]\end{array}$$

As above, this condition is an improvement on that given in Example 7 of [3] because we need only test on finite lists of natural numbers, rather than lists over α and β. That is, there is no quantification over functions X and Y that map into α and β, respectively, because attention can be restricted to the concrete lists $[1..n]$ *and* $[1..m]$*. In addition, only test maps p of type* $\text{Fin}\ n\to\text{Fin}\ m\to\text{Bool}$*, rather than* $\alpha\to\beta\to\text{Bool}$*, need to be considered.*

Theorem 3 can also be used to solve polymorphic testing problems when there is a linear dependency among the type variables in the a canonical test type, i.e., provided the type is of the form $\forall\alpha_1\ldots\alpha_n.\ \tau$, where α_1 is independent of all other type variables, and α_2 depends only on α_1, and α_3 depends only on α_2 and α_1, α_4 depends only on α_3 and α_2 and α_1, and so on. In this case, Theorem 3 can be applied serially to monomorphize the type arguments one at a time, starting with α_1. This is (the containerized version of) the approach taken on an intuitive basis in, for instance, Example 6 of [3]. We now show how this is justified by Theorems 1 and 3.

Example 4. *The type arguments to the function*

$$\mathit{map}:\forall\alpha\,\beta:*.\ (\alpha\to\beta)\to[\alpha]\to[\beta]$$

are not independent of one another because β depends on α in the type of map's first term argument. But this dependency is linear, because α is independent of all other type variables and β depends only on α. Treating β as a constant, we therefore first construct the following isomorphic type for map:

$$\forall\alpha\,\beta:*.\ (\alpha\to\beta)\to\Sigma n:\mathbb{N}.\,(\text{Fin}\ n\to\alpha)\to[\beta]$$

Then we apply Theorem 3 to get the instance

$$\forall n:\mathbb{N}.\,\forall\beta:*.\forall p:\text{Fin}\ n\to\beta.\ \mathit{map}_1\ (\text{Fin}\ n)\ (p,\mathit{id}) = \mathit{map}_2\ (\text{Fin}\ n)\ (p,\mathit{id})$$

of (9), i.e.,

$$\forall n:\mathbb{N}.\,\forall\beta:*.\forall p:\text{Fin}\ n\to\beta.\ \mathit{map}_1\ (\text{Fin}\ n)\ p\ [1..n] = \mathit{map}_2\ (\text{Fin}\ n)\ p\ [1..n]$$

to be tested, in which α has been monomorphized. We can now apply Theorem 3 to this new property to monomorphize β as well. The type of the functions $\mathit{map}_1\ (\text{Fin}\ n)\ [1..n]$ *and* $\mathit{map}_2\ (\text{Fin}\ n)\ [1..n]$ *that we are interested in testing is, up to isomorphism,*

$$\forall\beta.\forall n:\mathbb{N}.\,(\text{Fin}\ n\to\beta)\to[\beta]$$

Theorem 1 thus gives the derived monomorphic instance

$$\forall n : \mathbb{N}.\forall p : 1.\, map_1\ (\text{Fin}\ n)\ (\text{Fin}\ n)\ p\ [1..n] = map_2\ (\text{Fin}\ n)\ (\text{Fin}\ n)\ p\ [1..n]$$

i.e.,

$$\forall n : \mathbb{N}.\, map_1\ (\text{Fin}\ n)\ (\text{Fin}\ n)\ id\ [1..n] = map_2\ (\text{Fin}\ n)\ (\text{Fin}\ n)\ id\ [1..n]$$

of (4) to be tested, where 1 *is the unit type. This is a significant improvement over the test condition*

$$\forall n : \mathbb{N}.\, map_1\ \alpha\ \beta\ f\ [X_1 ... X_n] = map_2\ \alpha\ \beta\ f\ [X_1 ... X_n]$$

from Example 6 of [3]. Indeed, it shows that it not only suffices to test for lists of various lengths (the function $f : \alpha \to \beta$ is fixed), as noted in [3], but that it also suffices to test for finite lists of natural numbers and the specific fixed function that is the identity on all singletons.

In general, however, the type variables in a canonical form type are mutually dependent. This gives rise to a variant of the polymorphic testing problem that subsumes both the type independence exhibited in Example 3 and the linear dependency exhibited in Example 4 above. This variant can be formalized as:

> Let $\sigma[\alpha_1, \ldots, \alpha_n]$ be a type expression with free type variables $\alpha_1, \ldots, \alpha_n$, and let H be a definable functor with n arguments. Given a pair of functions h_1, h_2 of type $\forall \alpha_1 : *, \ldots, \alpha_n : *.\ \sigma[\alpha_1, ..., \alpha_n] \to H\alpha_1 \ldots \alpha_n$, find a *monomorphic* sufficient condition that implies the following property:
>
> $$\forall \alpha_1 : *, \ldots, \alpha_n : *.\forall x : \sigma[\alpha_1, \ldots, \alpha_n].\ h_1\ \alpha_1\ \ldots\ \alpha_n\ x = h_2\ \alpha_1\ \ldots\ \alpha_n\ x \qquad (12)$$

Note that the type variables $\alpha_1, ..., \alpha_n$ can indeed be mutually dependent here.

The following generalization of Theorem 1 provides a solution to this variant. We first give an auxiliary definition necessary to state our result.

Definition 2. *If $F_1, \ldots, F_n$ are n-ary functors, then an* $\langle F_1, \ldots, F_n \rangle$-algebra *comprises a carrier $\langle \alpha_1, ..., \alpha_n \rangle$ and maps $\langle f_1, ..., f_n \rangle$ such that for all $k = 1, ..., n$, $f_k : F_k\, \alpha_1 ... \alpha_n \to \alpha_k$. An* $\langle F_1, \ldots, F_n \rangle$-algebra homomorphism *from $(\langle \alpha_1, ..., \alpha_n \rangle, \langle f_1, ..., f_n \rangle)$ to $(\langle \beta_1, ..., \beta_n \rangle, \langle g_1, ..., g_n \rangle)$ comprises maps $h_k : \alpha_k \to \beta_k$ such that $g_k \circ F_k \langle h_1, ..., h_n \rangle = h_k \circ f_k$ for $k = 1, ..., n$. An $\langle F_1, \ldots, F_n \rangle$-algebra is* initial *if there is a unique $\langle F_1, \ldots, F_n \rangle$-algebra homomorphism from it to any other $\langle F_1, \ldots, F_n \rangle$-algebra. We write $(\mu\langle F_1 \ldots F_n \rangle, \text{in}_1, ..., \text{in}_n)$ for the initial $\langle F_1, \ldots, F_n \rangle$-algebra.*

Theorem 4. *Let $\sigma[\alpha_1, ..., \alpha_n]$ be a type with n free type variables $\alpha_1, ..., \alpha_n$, and let H be a definable n-ary functor. Given a pair of functions $h_1, h_2 : \sigma[\alpha_1, ..., \alpha_n] \to H\alpha_1 ... \alpha_n$. If there exist n-ary functors $\{G_i\}_{i \in I}$, types $\{O_i\}_{i \in I}$, and n-ary functors $F_1, \ldots, F_n$ such that*

$$\sigma[\alpha_1, ..., \alpha_n] = (\Pi_{i \in I}(G_i \alpha_1 \ldots \alpha_n \to O_i)) \times \Pi_{1 \le k \le n}(F_k \alpha_1 \ldots \alpha_n \to \alpha_k)$$

and there exists an initial $\langle F_1, \ldots, F_n \rangle$-algebra $(\mu\langle F_1 \ldots F_n \rangle, \text{in}_1 \ldots, \text{in}_n)$, then letting $\mu_k \langle F_1 \ldots F_n \rangle$ abbreviate $\pi_k(\mu\langle F_1 \ldots F_n \rangle)$, the following condition is a solution for this instance of the polymorphic testing problem with multiple type parameters:

$$\begin{aligned} &\forall p : \Pi_{i \in I}\, (G_i\, (\mu_1 \langle F_1 \ldots F_n \rangle) \ldots (\mu_n \langle F_1 \ldots F_n \rangle) \to O_i). \\ &\quad h_1\ (\mu_1 \langle F_1 \ldots F_n \rangle) \ldots (\mu_n \langle F_1 \ldots F_n \rangle)\ (p, \text{in}_1, \ldots, \text{in}_n) = \\ &\qquad h_2\ (\mu_1 \langle F_1 \ldots F_n \rangle) \ldots (\mu_n \langle F_1 \ldots F_n \rangle)\ (p, \text{in}_1, \ldots, \text{in}_n) \end{aligned} \qquad (13)$$

The proof is similar to the proof of Theorem 1 above.

Example 5. *The type arguments to the function*

$$isListIso : \forall \alpha\, \beta : *.\ (\alpha \to \beta) \to (\beta \to \alpha) \to \text{Bool}$$

are mutually dependent. Here, $F_1\, \alpha\, \beta = \beta$ *and* $F_2\, \alpha\, \beta = \alpha$*, so if there exists an initial* $\langle F_1, F_2 \rangle$*-algebra* $(\mu\langle F_1, F_2\rangle, \langle \text{in}_1, \text{in}_2\rangle)$*, then it suffices to test*

$$isListIso_1\ (\mu_1\langle F_1, F_2\rangle)\ (\mu_2\langle F_1, F_2\rangle)\ \text{in}_1\ \text{in}_2 = isListIso_2\ (\mu_1\langle F_1, F_2\rangle)\ (\mu_2\langle F_1, F_2\rangle)\ \text{in}_1\ \text{in}_2$$

This example cannot be handled by Bernardy, Jansson, and Claessen. It is not hard to see that (the containerized version of) Theorem 4 also subsumes, and gives the expected results for, canonical types in which the α_i are independent of one another (as in Example 3) or are linearly dependent (as in Example 4).

4 The Generalized Polymorphic Testing Problem with Constraints

The function *filter* places no constraints at all on its constructors. Sometimes, however, we want to test polymorphic properties whose constructors satisfy the laws of some algebraic theory. Bernardy, Jansson, and Claessen consider precisely such properties in Section 4.2 of [3]. For instance, in their Example 10 they discuss how to test properties of a sorting network generator *sort* of polymorphic type

$$\forall \alpha.\ ((\alpha, \alpha) \to (\alpha, \alpha)) \to [\alpha] \to [\alpha]$$

This type is similar to the type of *filter*, but instead of taking a predicate as its first term argument, *sort* takes a comparison function that itself takes a pair of data elements as input and returns the pair consisting of those elements appropriately ordered. Note, however, that the type-uniformity entailed by parametricity means that a generator of this type cannot determine whether or not the comparison function *actually* swaps the order of its arguments or just promises to.

To test an implementation $sort_1$ against a reference implementation *sort*, Bernardy, Jansson, and Claessen first embed their common type into the following canonical form type:

$$\forall \alpha.\ ((\alpha, \alpha) \to \alpha) \to ((\alpha, \alpha) \to \alpha) \to \mathbb{N} \to (\mathbb{N} \to \alpha) \to [\alpha]$$

In [3] the three constructors of this type are called $Max : (\alpha, \alpha) \to \alpha$, $Min : (\alpha, \alpha) \to \alpha$, and $X : \mathbb{N} \to \alpha$; here, *Min x y* represents the minimum of *x* and *y* and *Max x y* represents their maximum. Bernardy, Jansson, and Claessen then argue informally that testing implementations of $sort_1$ can be reduced to testing implementations of

$$\lambda n : \mathbb{N}.\ sort_1\ \ \mathbb{N}\ Min\ Max\ [X1\ ..\ Xn]\ :\ \mathbb{N} \to [\alpha]$$

against *sort*. (But as in the above examples, their Theorem 3 actually justifies testing on lists of the form $[1..n]$ rather than $[X1..Xn]$) And using our Theorem 3 above we can actually get the following more specific sufficient testing property:

$$\forall n : \mathbb{N}.\ sort_1\ (\text{Fin}\ n)\ Min\ Max\ [1..n] = sort\ (\text{Fin}\ n)\ Min\ Max\ [1..n]$$

where *Min* and *Max* are comparison functions on Fin *n* that take as input a pair (x, y) of values in Fin *n* as input and return their minimum and maximum, respectively.) Note that *sort* and $sort_1$ each

return a list, each of whose elements is a *Min-Max* comparison tree describing how to compute that element by taking minima and maxima of various elements of the input list. However, as observed by Bernardy, Jansson, and Claessen, to ensure that those trees actually correspond to a correct sorting function, the carrier of the initial algebra must be understood as the free distributive lattice generated by the data elements *Xi*, and *Min* and *Max* must be understood as meet and join, respectively. Put differently, the initial algebra for each $F\alpha = (\alpha, \alpha)+(\alpha, \alpha)+\mathrm{Fin}\, n$ must be quotiented by the constraint imposed by a free distributive lattice.

This observation makes sense intuitively, but how can we turn the intuition that let us solve the polymorphic testing problem for implementations of *sort* into a general technique for handling constraints on the algebra components of canonical form types? These are precisely the kinds of constraints we need to solve in order to test Dijkstra's algorithm as discussed in Section 2. The next theorem gives a general method for solving the polymorphic testing problem in the presence of algebraic constraints on the algebra part of a canonical form type of a property

function? The type is the type of a function being tested, not of the property itself!

to be tested. That is, it solves what we dub the *polymorphic testing problem with constraints*. This problem can be formalized as follows:

> Let $\sigma[\alpha]$ be a type expression with a free type variable α, and let H be a definable functor. Given a pair of functions h_1, h_2 of type $\forall\alpha : *.\ \sigma[\alpha] \to H\alpha$ and a predicate $P[\tau, x]$ on types τ and values $x : \sigma[\tau]$, the polymorphic testing problem with constraints is to find a *monomorphic* sufficient condition that implies the following property:
>
> $$\forall\alpha : *.\forall x : \sigma[\alpha].\ P[\alpha, x] \Rightarrow h_1\ \alpha\ x = h_2\ \alpha\ x \tag{14}$$

To solve the polymorphic testing problem with constraints, we need two auxiliary definitions:

Definition 3. *Let F be a functor and $Q[\tau, f]$ be a predicate on types τ and F-algebras $f : F\tau \to \tau$. The* category of (F, Q)-algebras *has pairs $(\tau, f : F\tau \to \tau)$ such that $Q[\tau, f]$ as its objects, and normal F-algebra homomorphisms as its morphisms.*

Definition 4. *An* initial (F, Q)-algebra *is an initial object in the category of (F, Q)-algebras.*

Of course for any choice of F and Q, an initial (F, Q)-algebra need not exist, but is unique up to isomorphism when it does. We then have

Theorem 5. *Let $\sigma[\alpha]$, P, H, h_1, and h_2 be as in the description of the polymorphic testing problem with constraints. If there exist functors $\{G_i\}_{i\in I}$ and F, and types $\{O_i\}_{i\in I}$ such that*

$$\sigma[\alpha] = (\Pi_{i\in I}(G_i\alpha \to O_i)) \times (F\alpha \to \alpha) \tag{15}$$

if $P[\tau, (p, f)]$ is equivalent to a predicate $Q[\tau, f]$ on just the F-algebra component f of $\sigma[\tau]$, and if an initial (F, Q)-algebra $\mathrm{in} : F(\mu(F, Q)) \to \mu(F, Q)$ *exists, then the following condition is a solution for this instance of the polymorphic testing problem with constraints:*

$$\forall p : \Pi_{i\in I}(G_i(\mu(F, Q)) \to O_i).\ h_1\ (\mu(F, Q))\ (p, \mathrm{in}) = h_2\ (\mu(F, Q))\ (p, \mathrm{in}) \tag{16}$$

The proof that (16) is a solution to the polymorphic testing problem with constraints relies on the following analogue of Lemma 1, which also makes use of the free theorem for the type of h_1 and h_2.

Lemma 2. *Let* $h : \forall\alpha : *. (\Pi_{i\in I}(G_i\alpha \to O_i)) \times (F\alpha \to \alpha) \to H\alpha$. *If there exists an initial* (F, Q)*-algebra* $(\mu(F, Q), \text{in} : F(\mu(F, Q)) \to \mu(F, Q))$, *then*

$$\begin{array}{l}\forall\alpha : *.\forall p : \Pi_{i\in I}(G_i\alpha \to O_i).\forall f : F\alpha \to \alpha.\\ \quad Q[\alpha, f] \Rightarrow\\ \quad\quad h\ \alpha\ (p, f) = H\ (\!|f|\!)\ (h\ (\mu(F, Q))\ (\langle p_i \circ G_i(\!|f|\!)\rangle_{i\in I}, \text{in}))\end{array}$$

where $(\!|f|\!)$ *is the unique* (F, Q)*-algebra homomorphism from* $(\mu(F, Q), \text{in})$ *to* (α, f).

Proof. The proof is almost identical to the proof of Lemma 1. To construct the (F, Q)-algebra homomorphism from $\mu(F, Q)$ to α we need to know that $Q[\alpha, f]$ holds, but this is satisfied by assumption. □

Once we know $Q[\alpha, f]$ holds, the proof of Theorem 5 is virtually identical to the proof of Theorem 1, which is its analogue in the absence of constraints. However, the proof of Theorem 5 uses Lemma 2, rather than Lemma 1.

The class of predicates on algebras considered in Theorem 5 does not include *all* of the constraints we might like to impose on polymorphic properties to be tested. But it does capture interesting and useful classes of constraints, such as constraints on the constructors of a property to be tested like those on *getmin* in Dijkstra's algorithm. Similar constraints given by predicates can ensure, for example, that a program is parameterized over a monad or an applicative functor. This could ultimately give an effective way to incorporate effects into polymorphic testing.

4.1 Solution Reduction via Embedding-Projection Pairs

Definition 5. *Let* $\sigma[\alpha]$ *and* $\tau[\alpha]$ *be type expressions with a single free type variable, and let* $P[\alpha, x]$, $Q[\alpha, x]$ *be predicates on* α *and on* $x : \sigma[\alpha]$ *and* $y : \tau[\alpha]$, *respectively. A* constraint-preserving embedding-projection pair $(e, p) : (\sigma[\alpha], P) \hookrightarrow (\tau[\alpha], Q)$ *consists of a pair of polymorphic functions* $e : \forall\alpha.\ \sigma[\alpha] \to \tau[\alpha]$ *and* $p : \forall\alpha.\ \tau[\alpha] \to \sigma[\alpha]$ *such that*

$$\begin{array}{rl} & \forall\alpha : *.\ \forall x : \sigma[\alpha].\ P[\alpha, x] \Rightarrow Q[\alpha, e\ \alpha\ x]\\ \textit{and} & \forall\alpha : *.\ (p\ \alpha) \circ (e\ \alpha) = id_{\sigma[\alpha]}\end{array}$$

Of course, we could also consider the condition $\forall\alpha : *.\forall y : \tau[\alpha].\ Q[\alpha, y] \Rightarrow P[\alpha, p\ \alpha\ x]$ in addition to the two conditions on embedding-projection pairs in Definition 5. But since it is not needed for the proof below, we omit it.

Lemma 3. *If* $(e, p) : (\sigma[\alpha], P) \hookrightarrow (\tau[\alpha], Q)$ *is a constraint-preserving embedding-projection pair,* H *is a definable functor, and* h_1 *and* h_2 *are functions of type* $\forall\alpha : *.\ \sigma[\alpha] \to H\alpha$, *then*

$$\begin{array}{l}\quad (\forall\alpha : *.\forall x : \tau[\alpha].\ Q[\alpha, x] \Rightarrow h_1\ \alpha\ (p\ \alpha\ x) = h_2\ \alpha\ (p\ \alpha\ x))\\ \Rightarrow\\ \quad (\forall\alpha : *.\forall x : \sigma[\alpha].\ P[\alpha, x] \Rightarrow h_1\ \alpha\ x = h_2\ \alpha\ x)\end{array}$$

Proof. Given $\alpha : *$ and $x : \sigma[\alpha]$ such that $P[\alpha, x]$, we have:

$$\begin{array}{ll} & h_1\ \alpha\ x \\ = & \qquad \{(e,p) \text{ is an embedding-projection pair}\} \\ & h_1\ \alpha\ (p\ \alpha\ (e\ \alpha\ x)) \\ = & \qquad \{\text{by hypothesis, since } P[\alpha,x] \Rightarrow Q[\alpha, e\ \alpha\ x]\} \\ & h_2\ \alpha\ (p\ \alpha\ (e\ \alpha\ x)) \\ = & \qquad \{(e,p) \text{ is an embedding-projection pair}\} \\ & h_2\ \alpha\ x \end{array}$$

□

Example 6. *Properties of a sorting network generator of the kind discussed at the start of this section can be tested using the obvious theorem generalising both Theorem 3 and Theorem 5. Bernardy, Jansson, and Claessen assume that the comparator argument to sort is built out of Min and Max, and* post facto *impose the free distributive lattice contraint on the initial algebra for the type of sort.*

Decide on a notation for pairs: (α, β) or $\alpha \times \beta$. Be consistent throughout.

By contrast, we impose the free distributive lattice constraint on algebras for $F\alpha = (\alpha, \alpha) + (\alpha, \alpha) + \text{Fin}\ n$ a priori. *That is, we require all F-algebras we consider to satisfy the constraint Q requiring that, for all x, y, and z, Min x (Max y z) = Max (Min x y) (Min x z). Since the initial such algebra — i.e., the initial (F, Q)-algebra) certainly exists — and since it is in fact just the free distributive lattice D generated by* Fin *n and having operations Min : (D, D) → D and Max : (D, D) → D, we can formally derive that a solution to the polymorphic testing problem for the sorting network generator is given by the monomorphic test condition*

$$\forall n : \mathbb{N}.\ sort_1\ D\ Min\ Max\ [1..n] = sort\ D\ Min\ Max\ [1..n]$$

In this way we can make the informal analysis in [3] precise.

In the next section we will see how constraints on algebras in canonical form types can be used to derive definitions of operators from their specifications.

5 Polymorphic Testing with Coalgebraic Constraints

Make intro consistent with this section.

The results of the previous two sections can be used to give a sufficient monomorphic condition for the polymorphic testing problem with constraints even for types $\sigma[\alpha]$ with coalgebraic, as well as algebraic, parts. To see how, consider an algebraic specification for a data type $H\alpha$ with operations of the following types:

$$\begin{array}{l} g_1 : G_1\alpha \to O_1,\ \ldots,\ g_i : G_i\alpha \to O_i \\ f_1 : F_1\alpha \to \alpha,\ \ldots,\ f_j : F_j\alpha \to \alpha, \\ h_1 : \alpha \to H_1\alpha,\ \ldots,\ h_l : \alpha \to H_l\alpha, \end{array}$$

The type of such a specification is of the form

$$\forall\alpha : *.\ \Pi_{i\in I}(G_i\alpha \to O_i) \times \Pi_{j\in J}(F_j\alpha \to \alpha) \times \Pi_{l\in L}(\alpha \to H_l\alpha) \ \to\ H\alpha \tag{17}$$

We call $\Pi_{j\in J}(F_j\alpha \to \alpha)$ and $\Pi_{l\in L}(\alpha \to H_l\alpha)$ the *algebra part* and the *coalgebra part* of (17) of $\sigma[\alpha]$, respectively. A term whose type is given by an algebra $F_j\alpha \to \alpha$ is a constructor for α, and a term whose type is given by an observation $G_i\alpha \to O_i$ is an observation on α, just as before. But (17) is not in BJC canonical form because the H_ls need be neither constant nor identity functors. We say that an $h_l : \alpha \to H_l\alpha$ is a *destructor*

or observing constructor

for α. Of course, some operations, such as those of type $\alpha \to O_i$, can be considered either observations or destructors; we leave it to the programmer to apportion the operations of $H\alpha$ into observations, constructors, and destructors as they see fit. If there is at least one $l \in L$ such that H_l is not a constant or identity functor, then α has at least one destructor, and we say that a type $\sigma[\alpha]$ of the form in (17) is in *extended canonical form*. The type of the specification for priority queues in Dijkstra's algorithm — with operations *empty*, *insert*, and *getmin*, as in Section 2 — is in extended canonical form because the type of *getmin* is of the form $\alpha \to H_i\alpha$ for $H_i\alpha = Maybe(a, \mathbb{N}+1, \alpha)$.

Explain rewriting by left Kan extensions.

If $\sigma[\alpha]$ is in extended canonical form, and if we can rewrite the coalgebras $\alpha \to H_l\,\alpha$ using left Kan extensions, then there can be no constraints on the coalgebra part of $\sigma[\alpha]$. We can therefore apply Theorem 5 to solve the polymorphic testing problem with constraints. For example:

Example 7. *Consider again the abstract specification of priority queues used in Dijkstra's algorithm in Section 2. After rewriting with left Kan extensions, the type of this specification is in BJC canonical form. Since there are no observating constructors in this specification, and since the predicate Q representing the constraint on getmin introduced in Section 2 is equivalent to a predicate on the algebra component of the argument type of this specification, we can use Theorem 5 to solve the polymorphic testing problem for the abstract specification for priority queues provided there exists an initial (F, Q)-algebra* in *for the functor $Ft = 1 + (a, \mathbb{N}+1, t) + Lan_{Maybe\ (a,\mathbb{N}+1,\_)}\ Id\ t$. In this case, we know that in order to test two implementations of the above abstract specification, it suffices to test them only on the initial (F, Q)-algebra.*

The difficulty with the above approach is that it can be nontrivial to construct appropriate left Kan extensions, and to construct initial (F, Q)-algebras after the types of observing constructors have been rewritten with those left Kan extensions to become part of the algebra F. But, fortunately, there is another approach to solving the polymorphic testing problem in the presence of certain kinds of constraints that mitigates these problems completely. If the constraint satisfied by the observing constructor is a so-called *canonical constraint* then, assuming it exists, we can take the initial algebra for the simpler functor describing just the types of the standard constructors as the testing type, and use the canonical constraint to define the observing constructors as derived operations on all data of that testing type. This allows us to avoid the complexity of computing polymorphic test types without losing the ability to test efficiently.

To see how this works, we first define the notion of a canonical constraint.

Definition 6. *Let $\sigma[\alpha]$ be a type with one free type variable α, let H be a definable functor, and let $h_1, h_2 : \forall\alpha : *.\, \sigma[\alpha] \to H\alpha$. If there exist functors $\{G_i\}_{i\in I}$, $\{F_j\}_{j\in J}$, and $\{H_l\}_{l\in L}$ and types $\{O_i\}_{i\in I}$ such that*

$$\sigma[\alpha] = \Pi_{i\in I}(G_i\alpha \to O_i)\ \times\ \Pi_{j\in J}(F_j\alpha \to \alpha)\ \times\ \Pi_{l\in L}(\alpha \to H_l\alpha)$$

where $F = \Sigma_{j\in J}F_j$, then we say that a predicate $Q[\tau, (\overline{f_j}, \overline{h_l})]$ on types τ, algebras $\overline{f_j}$, and coalgebras $\overline{h_l}$ is a canonical constraint *if it comprises*

- *a predicate* $Q'[\tau, \overline{f_j}]$ *whose initial* (F, Q')*-algebra* $\text{in} : F(\mu(F, Q')) \to \mu(F, Q')$ *exists, and*
- *a predicate* $Q''[\tau, (\overline{f_j}, \overline{h_l})]$ *that defines the coalgebras* $\overline{h_l}$ *on* $\mu(F, Q')$*, i.e., a set of Horn clauses* $Q''[\tau, (\overline{f_j}, \overline{h_l})]$ *that have the* h_l*s as their heads and that completely determine the values of* $h_l\, z$ *for every* $l \in L$ *and* $z : \mu(F, Q')$*.*

The second clause of Definition 6 entails that $Q''[\mu(F, Q'), (\text{in}, \overline{h_l})]$ holds. Note that $Q''[\tau, (\overline{f_j}, \overline{h_l})]$ need not define the $\overline{h_l}$s on values of types other than $\mu(F, Q')$.

The intuition underlying Definition 6 is that we should not need to test the constraint Q'' on every data type since the algebra part of $\sigma[\tau]$ entails that we are interested only in the initial (F, Q')-algebra $(\mu(F, Q'), \text{in})$. The requirement in the second clause of the definition that the predicate Q'' in a canonical constraint Q actually defines each h_l on data of type $\mu(F, Q')$ ensures the existence of a $\sum_{l \in L} H_l$-coalgebra structure on $\mu(F, Q')$ because $Q'[\mu(F, Q'), \text{in}]$ is satisfied, and this in turn ensures that $Q[\mu(F, Q'), (\text{in}, \overline{h_l})]$ is actually equivalent to the constraint $Q'[\mu(F, Q'), \text{in}]$ on just the algebra part of $\sigma[\mu(F, Q')]$.

h_1 and h_2 are functions to test and h_ls are coalgebras. Change notation of one of these to avoid clashes.

Theorem 6. *Let* $\sigma[\alpha]$*,* $\{G_i\}_{i \in I}$*,* $\{F_j\}_{j \in J}$*,* $\{H_l\}_{l \in L}$*,* $\{O_i\}_{i \in I}$*,* F*,* H*,* h_1*,* h_2*,* Q*, and* Q' *be as in Definition 6. If* $P[\tau, (p, \overline{f_j}, \overline{h_l})]$ *is equivalent to a canonical constraint* $Q[\tau, (\overline{f_j}, \overline{h_l})]$ *then the following condition is a solution for this instance of the polymorphic testing problem with constraints:*

$$\forall \overline{p_i : G_i(\mu(F, Q')) \to O_i}.\ \ h_1\ (\mu(F, Q'))\ (\overline{p_i}, \text{in}, \overline{h_l}) = h_2\ (\mu(F, Q'))\ (\overline{p_i}, \text{in}, \overline{h_l}) \tag{18}$$

Theorem 5 is thus the special case of Theorem 6 for the polymorphic testing problem with constraints when the canonical form type has no coalgebraic part.

Lemma 4. *Let* $h : \forall \alpha : *.\ \Pi_{i \in I}(G_i \alpha \to O_i) \times \Pi_{j \in J}(F_j \alpha \to \alpha) \times \Pi_{l \in L}(\alpha \to H_l \alpha) \ \to\ H\alpha$ *and* $F = \Sigma_{j \in J} F_j$*. If* $Q[\tau, (\overline{f_j}, \overline{h_l})]$ *is a canonical constraint, then*

$$\begin{aligned}
&\forall \alpha : *.\ \forall \overline{p_i : G_i \alpha \to O_i}.\ \forall \overline{f_j : F_j \alpha \to \alpha}.\ \forall \overline{h_l : \alpha \to H_l \alpha}.\\
&\quad Q[\alpha, (\overline{f_j}, \overline{h_l})] \Rightarrow\\
&\qquad h\, \alpha\, (\overline{p_i}, \overline{f_j}, \overline{h_l}) = H\, (\!|\overline{f_j}|\!)\, (h\, (\mu(F, Q'))\, (\langle p_i \circ G_i (\!|\overline{f_j}|\!) \rangle_{i \in I}, \text{in}, \overline{h_l})
\end{aligned}$$

where $(\!|\overline{f_j}|\!)$ *is the unique* (F, Q')*-algebra homomorphism from* $(\mu(F, Q'), \text{in})$ *to* $(\alpha, \overline{f_j})$*.*

Proof. The proof is essentially that of Lemma 2. To construct the (F, Q')-algebra homomorphism from $\mu(F, Q')$ to α we need to know that $Q'[\alpha, \overline{f_j}]$ holds, but this is satisfied by the assumption that Q is canonical. We also need to know that $Q''[\mu(F, Q'), (\text{in}, \overline{h_l})]$ is satisfied, but this is guaranteed by the second clause of Definition 6. □

The proof of Theorem 6 is now immediate. To see how Theorem 6 more easily solves the polymorphic testing problem in the presence of coalgebraic constraints that define destructors on $\mu(F, Q')$ we once again consider Dijkstra's algorithm.

Hidden foundations?

Example 8. *The constraints satisfied by the observing constructor getmin in the abstract specification for priority queues are canonical constraints since they define getmin on the initial algebra* $[(a, \mathbb{N}+1)]$ *for the functor* $Ft = 1 + (a, \mathbb{N}+1, t) = 1 + (a, \mathbb{N}) \times t$ *describing the types of the standard constructors empty and insert for priority queues. Indeed, these constraints give the following pattern-matching definition of getmin on all terms of type* $[(a, \mathbb{N}+1)]$*:*

$$
\begin{array}{lcl}
\mathit{getmin}\ \mathit{empty} & = & \mathit{Nothing} \\
\mathit{getmin}\ (\mathit{insert}\ (a,d)\ t) & = & \mathit{case}\ \mathit{getmin}\ t\ \mathit{of} \\
 & & \quad \mathit{Nothing} \longrightarrow \mathit{Just}\ (a,d,t) \\
 & & \quad \mathit{Just}\ (a',d',t') \longrightarrow \mathit{if}\ d < d' \\
 & & \qquad\qquad \mathit{then}\ \mathit{Just}\ (a,d,\mathit{insert}\ (a',d')\ t') \\
 & & \qquad\qquad \mathit{else}\ \mathit{Just}\ (a',d',\mathit{insert}\ (a,d)\ t')
\end{array}
$$

The definition shows that getmin is a redundant constructor for priority queues; indeed, for any value $x : [(a, \mathbb{N}+1)]$*, getmin x is defined entirely in terms of empty and insert. By Theorem 6 we thus need only test polymorphic properties of programs involving priority queues on the type* $[(a, \mathbb{N}+1)]$*, provided we still parameterize over functions getmin satisfying the given constraints. In particular, this means that we can fully test Dijkstra's algorithm just by testing on the simplest implementation of priority queues as lists.*

One way to read Theorem 6 is as reducing the testing of a function defined using *ad hoc* polymorphism — as embodied, for example, in an abstract specification for a data type — to the testing of a parametric polymorphic function on a "simplest" implementation of that specification.

The same reasoning used above for observing constructors applies to standard constructors, too. As a result, we can always restrict attention to a functor describing the types of a "basis set" of standard and observing constructors for a polymorphic type in extended canonical form — i.e., a set of standard and observing constructors with the property that all other standard and observing constructors can be completely defined on the initial algebra of the functor given by the constructors in the set — and take the initial algebra of this functor as our polymorphic testing type for properties of the extended type. The price we have to pay is that, in the simpler property to be tested, we must still explicitly quantify over those non-basis constructors not satisfying canonical constraints relative to basis constructors. In effect, then, we are permitted to decide for every standard or observing constructor whether to include it in the algebra determining the polymorphic testing type or to quantify over explicitly it when testing. The examples in the subsections below show how this choice can impact testing.

Make sure they do.

START HERE!!!!!!!!!!

Bob says: Better than BJC in that it gives a sharp condition for when we have to quantify over (standard and observing) constructors in testing. BJC quantify over all observations in formal stuff, but in their examples they don't. **But I don't see such an example.** We are able to give a good condition for when we can make that split. We also allow observing constructors at all, which BJC don't do.

5.1 Testing Stack-based Compilation

Suppose we have the following abstract specification for a stack data type:

$$\begin{array}{lcl} \mathit{empty} & : & \mathit{stk} \\ \mathit{push} & : & a \to \mathit{stk} \to \mathit{stk} \\ \mathit{pop} & : & \mathit{stk} \to \mathit{Maybe}\,(a, \mathit{stk}) \end{array}$$

where

$$\begin{array}{lcl} \mathit{pop}\,\mathit{empty} & = & \mathit{Nothing} \\ \mathit{pop}\,(\mathit{push}\,i\,s) & = & \mathit{Just}\,(i, s) \end{array}$$

Consider the following small expression language [8]

$$\mathit{data}\,\mathit{Exp} \;=\; \mathit{Const}\,\mathit{Int} \mid \mathit{Add}\,\mathit{Exp}\,\mathit{Exp}$$

and the corresponding stack-based language

$$\mathit{data}\,\mathit{Instr} \;=\; \mathit{Push}\,\mathit{Int} \mid \mathit{AddI}$$

Suppose now we want to test a function *compile* that compiles expressions to stacks of instructions to make sure that evaluating an expression with an evaluation function *eval* and executing the corresponding instructions on the stack using a function *run* give the same results. That is, suppose we have functions as described, with the following type signatures:

$$\begin{array}{lcl} \mathit{compiler} & : & \mathit{Exp} \to \mathit{List}\,\mathit{Instr} \\ \mathit{eval} & : & \mathit{Exp} \to \mathit{Int} \\ \mathit{run} & : & \forall \mathit{stk}.\,\mathit{stk} \to (\mathit{Int} \to \mathit{stk} \to \mathit{stk}) \to (\mathit{stk} \to \mathit{Maybe}(\mathit{Int}, \mathit{stk})) \to \mathit{List}\,\mathit{Instr} \to \mathit{stk} \end{array}$$

Then we can formalize the property we wish to prove relative to our abstract specification for stacks as

$$\begin{array}{l} \forall e : \mathit{Exp}. \\ \forall \mathit{empty} : \mathit{stk}. \\ \forall \mathit{push} : \mathit{Int} \to \mathit{stk} \to \mathit{stk}. \\ \forall \mathit{pop} : \mathit{stk} \to \mathit{Maybe}\,(\mathit{Int}, \mathit{stk}). \\ \mathit{pop}\,(\mathit{run}\,\mathit{empty}\,\mathit{push}\,\mathit{pop}\,(\mathit{compiler}\,e)) = \mathit{Maybe}\,(\mathit{eval}\,e, \mathit{empty}) \end{array}$$

We can take the algebra part of the type of the property to be proved to be derermined by the types of *empty* and *push*, and the initial algebra for their associated functor to be $\mathit{List}\,(\mathit{Fin}\,n)$. Provided *empty* is [], the definition of *pop* is completely specified on $\mathit{List}\,(\mathit{Fin}\,n)$, so Theorem 5 ensures that we need only test the following property:

$$\forall e : \mathit{Exp}.\, \mathit{pop}\,(\mathit{run}\,[]\,\mathit{cons}\,\mathit{pop}\,(\mathit{compiler}\,e)) = \mathit{Maybe}\,(\mathit{eval}\,e, [])$$

That is, we need only test our polymorphic property against a naive implementation of stacks as lists.

Of course, if *empty* is not [], then *pop* is underspecified. In this case, we need to provide a specification of *pop*'s action on the start stack, along with a randomly generated expression, in order to test our property. That is, we can just test

$$\begin{array}{l} \forall e : \mathit{Exp}. \\ \forall \mathit{pop} : \mathit{List}\,(\mathit{Fin}\,n) \to \mathit{Maybe}\,(\mathit{IntList}\,(\mathit{Fin}\,n)). \\ \mathit{pop}\,(\mathit{run}\,[]\,\mathit{cons}\,\mathit{pop}\,(\mathit{compiler}\,e)) = \mathit{Maybe}\,(\mathit{eval}\,e, []) \end{array}$$

5.2 Testing an Abstract Association List Implementation

Suppose we have the following abstract specification for association lists:

$$\begin{array}{lcl} \mathit{initial} & : & t \\ \mathit{insert} & : & a \to t \to t \\ \mathit{member} & : & t \to a \to \mathrm{Bool} \end{array}$$

where

$$\begin{array}{lcl} \mathit{member}\ (\mathit{insert}\ s\ x)\ x & = & \mathit{True} \\ \mathit{member}\ (\mathit{insert}\ s\ y)\ x & = & \mathit{True}\ \ \text{if}\ \mathit{member}\ s\ x = \mathit{True} \end{array}$$

and consider the polymorphic function

$$\mathit{insertAll} : (a \to t \to t) \to [a] \to t \to t$$

Intuitively, t is a set-like type, the first argument to *insertAll* is an insertion operation, and its third argument is initial set, which need not necessarily be empty. Suppose we want to test the following property of *insertAll*:

$$\begin{array}{l} \forall t : *.\forall a : *.\ \forall \mathit{initial} : t.\forall \mathit{insert} : a \to t \to t.\forall \mathit{member} : t \to a \to \mathrm{Bool}.\forall \mathit{xs} : \mathit{List}\ a. \\ \quad (\forall x : a.\forall s' : t.\ \mathit{member}\ (\mathit{insert}\ x\ s')\ x) = \mathit{True} \to \\ \quad\quad (\forall x : a.\forall y : a.\forall s' : t.\ \mathit{member}\ x\ s' = \mathit{True} \to \mathit{member}\ (\mathit{insert}\ y\ s')\ x = \mathit{True}) \to \\ \quad\quad\quad \mathit{all}\ (\mathit{member}\ (\mathit{insertAll}\ \mathit{insert}\ \mathit{xs}\ \mathit{initial}))\ \mathit{xs} = \mathit{True} \end{array}$$

Here, *all* is the function that checks whether or not all elements of the list that is its second argument satisfy the property that is its first.

Since the initial algebra for the functor given by the types of *initial* and *insert* is *List* (*Fin n*), and since the constraints partially specify *member* on *List* (*Fin n*), Theorem 5 ensures that we need only test the following property:

$$\begin{array}{l} \forall n : \mathit{Nat}.\ \forall \mathit{member} : \mathit{Fin}\ n \to \mathit{List}\ (\mathit{Fin}\ n) \to \mathrm{Bool}. \\ \quad (\forall x : \mathit{Fin}\ n.\forall s' : \mathit{List}\ (\mathit{Fin}\ n).\ \mathit{member}\ (\mathit{insert}\ x\ s')\ x) = \mathit{True} \to \\ \quad\quad (\forall x : \mathit{Fin}\ n.\forall y : \mathit{Fin}\ n.\forall s' : \mathit{List}\ (\mathit{Fin}\ n).\ \mathit{member}\ x\ s' = \mathit{True} \to \mathit{member}\ (\mathit{insert}\ y\ s')\ x = \mathit{True}) \to \\ \quad\quad\quad \mathit{all}\ (\mathit{member}\ (\mathit{insertAll}\ \mathit{cons}\ [1..n-1]\ [\,])\ [0..n-1] = \mathit{True} \end{array}$$

That is, we need only test against a naive implementation of association lists as actual lists (since *initial* can be taken to be the empty list and *insert* can be taken to be *cons*) of numbers representing the positions in the original input lists *xs*, provided *member* is instantiated with a function satisfying the constraints. Note that there is only one non-specified case — for the action of *member* on *insert* — in what would otherwise be a complete definition of *member* on *List* (*Fin n*), and we can test by parameterizing over the different choices for the missing clause of the would-be definition. To give the missing clause we first observe that x has type *Fin n*, where n is the cardinality of the set-like data structure *initial*. For each choice of n there are thus 2^n possible specifications of *member*'s action on *initial*, each corresponding to one of the 2^n membership specifications for *initial*, against which the abstract specification for association lists must be tested. This gives more than a single specification against which to test, but it also gives far fewer such specifications than the infinitely many possible "member" functions that are defined for all types t and satisfy *member*'s two Horn clause constraints for the type *List a*.

5.3 Testing a Strongly Connected Components Algorithm

Consider once again the abstract specification of stacks from Section 5.1, and consider the polymorphic function

$$scc : ((a, a)) \to \text{Bool}) \to a \to List\,(List\,a)$$

for computing strongly connected components of a graph. The first argument to *scc* is the representation of an input graph with n vertices as an adjacency matrix, and the second is a start vertex in the input graph. The function *scc* returns a list of lists of vertices, with each "inner" list representing the vertices in one strongly connected component.

Now suppose we have a function

$$connected : ((a, a) \to \text{Bool}) \to List\,a \to \text{Bool}$$

that determines whether or not the list of vertices that is its second argument represents a connected component in the graph that is its first argument. And suppose that we want to test the following property of *scc*, which specifies that each list of vertices *scc* returns does indeed represent a connected component in its input graph:

$$\begin{aligned}
&\forall stk : *.\ \forall empty : stk.\\
&\qquad \forall push : a \to stk \to stk.\\
&\qquad \forall pop : stk \to Maybe\,(a, stk).\\
&\qquad \forall graph : (a, a) \to \text{Bool}.\\
&\qquad \forall sv : a.\\
&\qquad \forall connected : ((a, a) \to \text{Bool}) \to List\,a \to \text{Bool}.\\
&\qquad all\ (connected\ graph)\ (scc\ empty\ push\ pop\ graph\ sv)
\end{aligned}$$

Note that the start stack *empty* need not actually be empty.

We can take the algebra part of the type of the property to be determined by the types of the constructors *empty* and *push*. Since the initial algebra for their associated functor is $List\,(Fin\,n)$, and since the constraints partially specify *pop* on $List\,(Fin\,n)$, Theorem 5 ensures that only the following property need be tested:

$$\begin{aligned}
&\forall n : Nat.\\
&\forall pop : List\,(Fin\,n) \to Maybe\,(Fin\,n, List\,(Fin\,n)).\\
&\forall graph : \Sigma n : Nat.\,(Fin\,n, Fin\,n) \to \text{Bool}.\\
&\forall sv : Fin\,n.\\
&\forall connected : (\Sigma n : Nat.\,(Fin\,n, Fin\,n) \to \text{Bool}) \to List\,(Fin\,n) \to \text{Bool}.\\
&all\ (connected\ graph)\ (scc\ [\,]\ cons\ pop\ graph\ sv)
\end{aligned}$$

That is, we need only test against a naive implementation of stacks as lists and all specifications of *pop*'s action on the start stack *empty*. If *empty* is [], then *pop* is completely specificed, as already discussed in Section 5.1, and there is no need to quantify over it in the above test property.

5.4 Traces, like our LICS Trees?

6 Higher-kinded Observations

Put in MonadPlus stuff from other draft so that we have constraints on the constructors. Extend the theory part to handle the higher-orderness.

An example without observing constructors. Monads? Applicative functors? These tie to effects.

How does this fit with "real" Haskell? Say just restricting to total subset here?

Suppose you want to verify a property of a monadic polymorphic program. For example, you might want to verify that Haskell's standard library function

```
msum :: MonadPlus m => [m a] -> m a
```

has the simple property that applying `msum` to each of two lists and then combining the results with the sum of the monad `m` is the same as first concatenating the two lists and then applying `msum` to the result. Or you might want to verify another, more sophisticated property, of `msum`. One way to do this is to call on an automated testing tool, such as QuickCheck [], to randomly generate test cases for the property. This seems reasonable, until we remember that `msum` is (implicitly) parameterized over all data `m` of the higher kind `* -> *`, as well as all types `a`, that satisfy the `MonadPlus` constraints, and that it is far from clear how to randomly generate monads — or, indeed, data of other higher-kinded types, especially those that satisfy auxilliary constraints. So how can a program such as `msum` be tested, and tested efficiently?

To get a handle on this, we can first expand out the `MonadPlus` constraint. This gives the following equivalent type for `msum`:

```
msum :: forall m a. Monad m => (forall b. m b -> m b -> m b) ->
                               (forall b. m b) ->
                               [m a] -> m a
```

Here, the first and second term arguments to `msum` are required to satisfy the monoid laws. That is, if $\oplus$ and $\oslash$ are the operations of an instance of `MonadPlus`, then

```
forall b. forall n, p, q :: m b. (n ⊕ p) ⊕ q = n ⊕ (p ⊕ q)
```

and

```
forall b. forall n :: m b.  ⊘ ⊕ n = n ⊕ ⊘
```

are required to hold. This rewriting of `msum`'s type lets us make the monoid constraints on the monad explicit and lets us express the property of `msum` that we want to verify as:

```
forall m a. Monad m => ⊕ :: (forall b. m b -> m b -> m b).
                       ⊘ :: (forall b. m b).                          (*)
                       l1, l2 :: [m a].
                       (msum [m][a]⊕⊘l1) ⊕ (msum [m][a]⊕⊘l2) =
                           msum [m][a]⊕⊘(l1 ++ l2)
```

Of course, the type of `msum` is still quantified over `m` and `a`, and data of type `m a` must still satisfy the algebraic constraints given by the monoid laws, so although the constraints to which `msum` is

subject and the property of it that we hope to verify are now expressed explicitly, we have made no progress at all toward (efficient) testing of the latter. In particular, QuickCheck cannot be applied here. And even if it could, the efficiency-improving polymorphic property testing techniques of Bernardy, Jansson, and Claessen [3] remain inapplicable, so QuickCheck's testing for functions and properties involving higher-kinded data would be neither as efficient as its testing for data of kind *, nor as efficient as we would ideally like.

To begin to see how to recover from this state of affairs, we can draw some inspiration and insight from the application of Bernardy, Jansson, and Claessen's techniques to a "lower kinded" analogue of our testing problem for `msum`. Consider the Haskell standard library function

```
mconcat:: forall a. (a -> a -> a) -> a -> [a] -> a
```

Here, the first two term arguments to `mconcat` are assumed to satisfy the monoid laws. Although Bernardy, Jansson, and Claessen do not actually give a *general technique* for efficiently testing programs in the presence of constraints, they do consider the specific example of efficiently testing a sorting network generator and do give a solution for this example.

And make observation on page 112...

The essence of their solution is to use their technique for testing in the absence of constraints, but to remember thereafter that the quantified type must ultimately be understood as satisfying the constraints (there, of a distributive lattice). Proceeding in the same manner for `mconcat`, we first note that the type polymorphism of `mconcat` entails that its return element can be obtained only by either selecting an element from the list argument to `mconcat`, choosing the second term argument to `mconcat`, or applying the first term argument of `mconcat` to two similarly obtained data elements that we already have. Decomposing `mconcat`'s list argument into a natural number observation `n` representing the length of the list, and a constructor `X` mapping a natural number `i`

i in $\{1, ..., n\}$ but BJC don't make this tightening/weakening

to the list element at index `i`, the results of [3] show that it suffices to rewrite the type of `mconcat` as

```
forall a. (a -> a -> a) -> a -> Nat -> (Nat -> a) -> a
```

The property for `mconcat` analogous to that above for `msum` is

```
forall a.(⊕ :: forall a. a -> a -> a).(⊘ :: a). l1, l2 :: [a].          (**)
               (mconcat [a]⊕⊘l1)⊕(mconcat [a]⊕⊘l2) =
                   mconcat [a]⊕⊘(l1 ++ l2)
```

From this property we construct the monomorphic test type

```
data A : * where
  X : Nat -> A
  Y : Nat -> A
  oplus : A -> A -> A
  oslash : A
```

To test `(**)`, it thus suffices to test the following monomorphic property corresponding to it:

```
forall a n k. (⊕ :: forall a. a -> a -> a). (⊘ :: a).
                let l1 = [X1 ... Xn]
                    l2 = [Y1 ... Yk]
                (mconcat [a]⊕⊘l1) ⊕ (mconcat [a]⊕⊘l2) =
                    mconcat [a]⊕⊘(l1 ++ l2)
```

However, the constraints on the type of `mconcat` require that testing must be done modulo the monoid laws, so the type `A` must be understood a monoid whose addition and unit operations are `oplus` and `oslash`, respectively, and whose generators are the `Xi`. From here we can either verify `mconcat'` symbolically using the monoid laws, or we can observe that since the functor `List` together with `(++)` and `[]` is the "simplest" data type constructor satisfying the monoid constraints, and since on inputs `n` and `k` we test `mconcat'` on only the `n`-element list `[X1 ... Xn]` and the `k`-element list `[Y1 ... Yk]`, we can in fact take our testing monotype to be `([Fin n], [Fin k], (++), [])`, where for any natural number `i`, `Fin i` is a data type representing the set $\{1, 2, ..., n\}$. Indeed, this type can be seen as an "implementable representation" of `A`. Moreover, using `(++)` builds in the associativity of `oplus`, and using `[]` builds in the unit-ness of `oslash`. This approach gives the following final monomorphic test property:

```
forall a n k. let l1 = [X1 ... Xn]
                   l2 = [Y1 ... Yk]
              (mconcat [a] (++) [] l1) ++ (mconcat [a] (++) [] l2) =
                       mconcat [a] (++) [] (l1 ++ l2)
```

Where did this implementable representation come from? How can we get implementable representations systematically? Perhaps there is scope for formalizing this part of the verification, which seems ad hoc both here and in BJC. This paper answers this question.

Now, extrapolating the above reasoning to `msum`, we first rewrite the type of `msum` as

```
forall m a. Monad m => (forall b. m b -> m b -> m b) ->
                       (forall b. m b) ->
                       Nat -> (Nat -> m a) -> m a
```

and consider the type of the property `(*)` that we want to prove. From this we construct the monomorphic test type

```
data MA : * where
  X : Nat -> MA
  Y : Nat -> MA
  oplus : MA -> MA -> MA
  oslash : MA
  Z : A
```

The clause for Z is form the return of the monad. Why not bind, too?

Since there is no way for `msum` to construct data whose type is a raw `a` (i.e., data of type `a` outside of monadic context `m`), we can take `a` to be the singleton type 1. We then see that to test `(*)`, it suffices to verify the following monomorphic property corresponding to it:

```
forall m n k. Monad m => let l1 = [X1, ..., Xn]
                             l2 = [Y1, ..., Yk]
                         (msum [m][1]⊕⊘l1)⊕(msum [m][1]⊕⊘l2) =
                             msum [m][1]⊕⊘(l1 ++ l2)
```

The only arguments we need to quantify over here are the natural numbers `n` and `k`, and the monad `m`. As for `mconcat` we need to take into account the constraints on the type of `msum`. These require testing to be done modulo the monoid laws *and* the monad laws, so the type `MA` must be understood as a monad whose generators are the `Xi` and `Yj` and that carries monoidal structure via the addition and unit operators `oplus` and `oslash`, respectively. We note that the functor `List` with `(++)` and `[]` is the "simplest" data type constructor supporting all of the `MonadPlus` constraints, so by analogy with the situation for `mconcat`, we can take our monomorphic testing property for `msum` to be

```
forall n k. let l1 = [X1 ... Xn]
                l2 = [Y1 ... Yk]
            (msum [List][1] (++) [] l1) ++ (msum [List][1] (++) [] l2) =
                msum [List][1] (++) [] (l1 ++ l2)
```

Fix when paper is more settled.

Still, the question remains: how can we turn this analogical reasoning into a proper technique for verifying properties of polymorphic functions with higher-kinded parameters and constraints? This paper ultimately answers that question. Before doing so, however, we first formalize what we call the (standard) *polymorphic testing problem* and recap the solution to this problem for container types given in [3]. We then show how moving to dependent containers allows us not only to handle a wider variety of functions, but also lets us improve on the solution of Bernardy, Jansson, and Claessen for non-dependent containers by tightening the range of data from which test data can/need be drawn.

Check that this is right.

We next formalize the solution presented above to what we dub the *polymorphic testing problem with constraints*. Finally, we extend our base kind solutions to support the testing of functions with higher-kinded arguments subject to arbitrary constraints

Check that this is right.

, and as a special case show how to formally solve the *algebraic monadic polymorphic testing problem* by reducing it to polymorphic testing for base kind; the example involving the function `msum` from the introduction, whose higher-kinded argument satisfies the `monadPlus` constraints, illustrates the techniques we employ. A second wisely used class of examples treated by our methods is that of applicative functors.

For readability, we develop our solutions for functions of only one type parameter. However,

each solution we present is easily extended to functions with multiple type parameters.

algebraic constraints — quotient to get test data. BJC do this in sorting example, but don't generalize. We were able to make into a general theorem. Emphasize monads and applicative functors; monads tie into algebraic effects. These don't capture all constraints one might want to consider, but they capture interesting classes. Sweep under rug the fact that we don't have interesting constraints on observations. By stating the problem we make it clear what future work is. Different dimension of the problem that are brought to forefront my stating it.

constraints on observations — observation must be an ordering (see insert into a sorted list example) — but how to turn this trick into a technique?

—

This solution is for properties all of whose parameters are of base kind, so allows us to deal only with observations of base kind. We consider higher-kinded observations

Observations or data type? If the former, then is Monad a helpful example here since it is not the observation, but rather (some of) its operations would be.

in Section 6.

Consider, for example, a polymorphic function with type

$$\forall ma.Monad\, m \Rightarrow (\forall a.ma \to ma \to ma) \to (\forall a.ma) \to [ma] -\!> ma$$

with constraints on operations that make this in to a monoid Can we principledly find m and a that allow us to test this function?

Monad example? P olymorphic set example?

Why poly sets?

empty :: forall a. Set a insert :: forall a. a -¿ Set a -¿ Set a

7 Variations

Dual?

gather :: (a -¿ Boo) -¿ (a -¿ [a]) -¿ a -¿ Int -¿ [a] forall a : *. all good (gather good tree root d)

$A = \nu(1 + \Sigma n : Nat.Fin\, a \to A$

Finite segments of streams are handled properly by a map function on streams?

What if there is interaction between the constructors and destructors?

(Fa -¿ a) x (a -¿ Ga)

References

[1] Agda. Available at tt http://wiki.portal.chalmers.se/agda/pmwiki.php.

[2] J.-P. Bernardy. *A Theory of Parametric Polymorphism and an Application*. PhD Thesis, Chalmers University of Technology, 2011.

[3] J.-P. Bernardy, P. Jansson, and K. Claessen. Testing Polymorphic Properties. Proceedings, European Symposium on Programming 2010, pp. 125 – 144.

[4] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings, International Conference on Functional Programming*, pp. 268 – 279, 2000.

[5] E.W. Dijkstra: A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* 1 (1959), pp. 269 – 271.

[6] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining Testing and Proving in Dependent Type Theory. In *Proceedings, Theorem Proving in Higher Order Logics*, pp. 188 – 203, 2003.

[7] T. Hallgren. Alfa. Available from `http://www.cs.chalmers.se/ hallgren/alfa`.

[8] G. Hutton. Hutton's razor...

[9] P. Johann and N. Ghani. Initial Algebra Semantics is Enough! In *Proceedings, Typed Lambda Calculus and Applications*, pp. 207 – 222, 2007.

[10] P. Johann and N. Ghani. Foundations for Structured Programming with GADTs. Proceedings, Principles of Programming Languages 2008, pp. 297 – 308.

[11] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd edition)*. Addison-Wesley Professional, 1998.

[12] S. MacLane. Categories for the Working Mathematician. Springer-Verlag, 1971.

[13] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[14] J. Reynolds. Types, Abstraction and Parametric Polymorphism. In *Proceedings, Information Processing*, pp. 513 – 523, 1983.

[15] J. Voigtländer. Much Ado About Two (Pearl): A Pearl on Parallel Prefix Computation. In *Procedings, Principles of Programming Languages*, pp. 29 – 35, 2008.