Higher-order Polymorphic Testing

Robert Atkey and Patricia Johann

September 1, 2015

Abstract

ABSTRACT GOES HERE

1 Notes from conversation with Bob 8/26

Positioning: Combines Theorems for Free with Monads, appropriate for Wadlerfest.

Consider programs parametric in the monad (extends BJC) by quantifying over type operators.

Overall approach: Adapt BJC to functor categories, and consider type constructors as fixed points of higher-order functors.

We can think of the monad operators as constructors of a data type (e.g., the monadPlus operators (++) and [] are constructors for binary trees). We can think of (higher-order) constraints as refining the resulting data types to give monads (e.g., imposing the associativity constraint on binary trees gives lists).

It is hard to see how to generate test monads, but we can at least use (higher-order) constraints to give a specification that all test monads must satisfy. For example, we can justify the use of List as the test monad for msum by showing that it satisfies the monadPlus constraints.

So, by tidying up BJC, we can say what the user has to prove in order to know that their test is sound. In the example above, if the user chose binary trees as their test monad, then the free structure on these constructors — i.e., the monad of binary trees — would be a monad supporting operations of the required types, but those operations would fail to satisfy the monadPlus constraints. This would show that binary trees give traces for operations that can be applied, but cannot be taken as the test monad for the higher-order property under consideration.

Of course, the user need not choose the "least" structure satisfying the constraints as their test structure. They can choose any structure satisfying those constraints because, even if the structure they choose satisfies other constraints as well, the structure will definitely satisfy the required constraints, and those other constraints will not have been amongst the required constraints.

Some examples we might investigate are:

- lambda terms [no constraints]
- ptrees [no constraints]
- monads (study these as a special case — note that these do not take full advantage of the higher-order functionality because information at the data level never changes) [monadPlus constraints]

- imperative queues [no constraints]
- other structures as in Peter et al.'s polymorphic testing paper [??]

There may also be examples for which the least monad satisfying the constraints is not expressible in Haskell. Such a situation may arise when this monad arises by, e.g, quotienting, because Haskell cannot express this.

With regard to the special case of monadic constraints, there are two approaches. One approach is to specialize the above general techniques to monads, as described above. Another approach is to generate the free monads, rather than fixed points of higher-order functors, as in the first approach. But are these not actually the same thing?

2 First-order Polymorphic Testing

The
first-order

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:
>
> $$\forall\alpha : *.\forall x : \sigma[\alpha].\ h_1\ \alpha\ x = h_2\ \alpha\ x \tag{1}$$

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 has been considered by Bernardy, Jansson, and Claessen [4], whose 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:

$$\forall p : \Pi_{i\in I}(G_i(\mu F) \to O_i).\ h_1\ (\mu F)\ (p, \mathrm{in}) = h_2\ (\mu F)\ (p, \mathrm{in}) \tag{2}$$

We will say that a type of the form of σ is in *BJC canonical form*. The proof that (2) 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*

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

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

$$\begin{array}{l}\forall \alpha_1, \alpha_2 : *.\ \forall r : \alpha_1 \to \alpha_2. \\ \quad \forall p' : \Pi_{i \in I}(G_i \alpha_1 \to O_i), p : \Pi_{i \in I}(G_i \alpha_2 \to O_i). \\ \quad\quad (\forall i \in I.\ p'_i = p_i \circ G_i r) \Rightarrow \\ \quad\quad\quad \forall f_1 : F\alpha_1 \to \alpha_1, f_2 : F\alpha_2 \to \alpha_2. \\ \quad\quad\quad\quad r \circ f_1 = f_2 \circ Fr \Rightarrow \\ \quad\quad\quad\quad\quad H\ r\ (h\ \alpha_1\ (p', f_1)) = h\ \alpha_2\ (p, f_2)\end{array} \tag{3}$$

Given $\alpha : *$, $p : \Pi_{i \in I}(G_i \alpha \to O_i)$ and $f : F\alpha \to \alpha$, we instantiate (3) 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 (2) is a solution to the polymorphic testing problem.

Proof. (Theorem 1) We want to prove that (2) implies the following specialization of (1) 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:

$$\begin{array}{ll} & h_1\ \alpha\ (p, f) \\ = & \qquad \{\text{Lemma 1 applied to } h_1\} \\ & H\ (\!|f|\!)\ (h_1\ \mu F\ (\langle p_i \circ G_i (\!|f|\!) \rangle_{i \in I}, \mathrm{in})) \\ = & \qquad \{\text{Application of (2)}\} \\ & H\ (\!|f|\!)\ (h_2\ \mu F\ (\langle p_i \circ G_i (\!|f|\!) \rangle_{i \in I}, \mathrm{in})) \\ = & \qquad \{\text{Lemma 1 applied to } h_2\} \\ & h_2\ \alpha\ (p, f) \end{array}$$

□

2.1 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* BJC canonical form as stated in Theorem 1, but rather that the type of h_1 and h_2 be *embeddable* into a type of BJC 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 types of BJC canonical form to solutions to the polymorphic testing problem for functions *actually* of such types. 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 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* $(\mu F, \text{in} : F(\mu F) \to \mu F)$*, 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*

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

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

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

□

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

$$\forall\alpha : *.\ (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha] \tag{4}$$

In this instance of the polymorphic testing problem, we have (up to currying) $\sigma[\alpha] = (\alpha \to \text{Bool}) \times [\alpha]$ *and* $H\alpha = [\alpha]$*. In this case,* [*sigma*[α] *is not actually* of *BJC canonical form, but it does* embed *in the BJC canonical form type*

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

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

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

of (2) is a solution to the polymorphic testing problem for the type in (4). More concretely, this instance is

$$\forall n : \mathbb{N}. \forall p : \mathbb{N} \to \text{Bool}.\ \mathit{filter}_1\ \mathbb{N}\ p\ [1..n] = \mathit{filter}_2\ \mathbb{N}\ p\ [1..n] \tag{5}$$

Is p actually quantified over, or arbitrary-but-fixed? Same question throughout. Compare with BJC.

In Example 1 of [4], 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 the type in Equation 4. Interestingly, this property still requires a fixed function X that maps from all of $\mathbb{N}$ into α to construct its test list, as well as a predicate of type $\alpha \to \text{Bool}$. By contrast, the sufficient monomorphic condition derived by applying Theorem 5 in [4] (i.e., Theorem 2 above)

actually justifies requiring only testing for the identity function on μF (i.e., $\mathbb{N}$ here) and predicates of type $\mathbb{N} \to \text{Bool}$. This is reflected in Equation 5 above. The weaker result (i.e., the more complex test property) derived in [4] 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 allows us to derive additional prunings of the test space. A similar comment applies to all of the examples in [4].

Note that although p will be applied only to the list $[1..n]$, it must still 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 subsection, where we formally prove that the domain of p can be restricted to avoid this overgeneration of test data.

2.2 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 the polymorphic testing problem for its type. In particular, if a property involves a container [1] (i.e., a dependent pair), then they must embed that container into a non-dependent pair before proceeding as in [4]. (See Appendix B of [4].) 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 show how working directly with containers prevents this, and 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 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 polymorphic 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 F_s for all $s : S$, and types $\{O_i\}_{i\in I}$ such that*

$$\sigma[\alpha] = (\Pi_{i\in I}(G_i\alpha \to O_i)) \times \Sigma s : S.\ (F_s\alpha \to \alpha) \tag{6}$$

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

$$\forall s : S.\forall p : \Pi_{i\in I}(G_i(\mu F_s) \to O_i).\ h_1\ \mu F_s\ (p, s, \text{in}_s) = h_2\ \mu F_s\ (p, s, \text{in}_s) \tag{7}$$

The proof that (20) 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*

$$\forall\alpha : *.\ (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha] \tag{8}$$

It is not difficult to see that $\sigma[\alpha]$ is isomorphic *to the 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 (6) 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 (20) is a solution to the polymorphic testing problem for the type (4). More concretely, this instance is

$$\forall n : \mathbb{N}.\forall p : \text{Fin } n \to \text{Bool}.\ \mathit{filter}_1\ (\text{Fin } n)\ p\ [1..n] = \mathit{filter}_2\ (\text{Fin } n)\ p\ [1..n] \tag{9}$$

Note that the condition in Equation 9 is much easier to test than the corresponding one given in Example 1 of [4]. 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 [4]. Moreover, each such map is itself finite, and is thus simpler to represent, generate, and work with than its "corresponding" infinite domain maps. Finally, Bernardy, Jansson, and Claessen discuss after Theorem 6 in [4] 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 [1], 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 6 is in *container canonical form* (CCF).

2.3 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 [4]. 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* $\mathit{isPrefixOf}_1$ *and* $\mathit{isPrefixOf}_2$ *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 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 (20) is a solution to the polymorphic testing problem for the type of isPrefixOf:

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

This instance can be rewritten as

$$\forall n : \mathbb{N}.\, \forall m : \mathbb{N}.\, \forall p : \mathrm{Fin}\ n \to \mathrm{Fin}\ m \to \mathrm{Bool}.$$
$$\mathit{isPrefixOf}_1\ (\mathrm{Fin}\ n)\ (\mathrm{Fin}\ m)\ p\ [1..n]\ [1..m] = \mathit{isPrefixOf}_2\ (\mathrm{Fin}\ n)\ (\mathrm{Fin}\ m)\ p\ [1..n]\ [1..m]$$

As above, this condition is an improvement on that given in Example 7 of [4] 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* $\mathrm{Fin}\ n \to \mathrm{Fin}\ m \to \mathrm{Bool}$*, rather than* $\alpha \to \beta \to \mathrm{Bool}$*, need to be considered.*

Theorem 3 can also be used to solve polymorphic testing problems when there is only a linear dependency among the type variables in the a canonical test type, i.e., provided the type is of the form $\forall \alpha_1 ... \alpha_n.\ \tau$, where $\alpha_1, ..., \alpha_n$ is an ordering of the variables over which the type $\sigma[\alpha] \to H\alpha$ is parameterized, and this ordering is such that α_1 is independent of all other type variables and, for all $i > 1$, α_i depends only on $\alpha_1, ..., \alpha_{i-1}$. 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 [4]. 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, so we cannot monomorphize in parallel. But this dependency is linear, because α is independent of all other type variables and β depends only on α, so we can monomorphize serially. Treating β as a constant, we therefore first construct the following isomorphic type for map:

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

Then we apply Theorem 3 to get the instance

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

of (20), i.e.,

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

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

$$\forall \beta : *.\forall p : \mathrm{Fin}\ n \to \beta.\ [\beta]$$

Theorem 1 thus gives the derived monomorphic instance

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

of (2) to be tested. This is an improvement over the test condition

$$\forall \alpha\, \beta : *.\ \forall n : \mathbb{N}.\ \forall f : \alpha \to \beta.\ \forall X : \mathbb{N} \to \alpha.\ \mathit{map}_1\ \alpha\ \beta\ f\ [X_1 ... X_n] = \mathit{map}_2\ \alpha\ \beta\ f\ [X_1 ... X_n]$$

from Example 6 of [4]. Indeed, it shows that it not only suffices to test for lists of various lengths, as noted in [4], 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:
>
> $$\forall \alpha_1 : *, \ldots, \alpha_n : *. \forall x : \sigma[\alpha_1, \ldots, \alpha_n].\ h_1\ \alpha_1\ \ldots\ \alpha_n\ x = h_2\ \alpha_1\ \ldots\ \alpha_n\ x \tag{10}$$

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 $k = 1, ..., n$, $f_k : F_k\, \alpha_1 ... \alpha_n \to \alpha_k$. That is, an $\langle F_1, \ldots, F_n \rangle$-algebra is simultaneously an F_k-algebra for each $k = 1, ..., n$. 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, i.e, if it is simultaneously an initial F_k-algebra for each $k = 1, ..., n$. We write $(\mu \langle F_1 \ldots F_n \rangle, \mathrm{in}_1, ..., \mathrm{in}_n)$ for the initial $\langle F_1, \ldots, F_n \rangle$-algebra.*

We then have

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 $(\mu \langle F_1 \ldots F_n \rangle, \mathrm{in}_1 \ldots, \mathrm{in}_n)$, 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:

$$\begin{aligned} &\forall p : \Pi_{i \in I} \left(G_i \left(\mu_1 \langle F_1 \ldots F_n \rangle\right) \ldots \left(\mu_n \langle F_1 \ldots F_n \rangle\right) \to O_i\right). \\ &\quad h_1 \left(\mu_1 \langle F_1 \ldots F_n \rangle\right) \ldots \left(\mu_n \langle F_1 \ldots F_n \rangle\right) (p, \mathrm{in}_1, \ldots, \mathrm{in}_n) = \\ &\qquad h_2 \left(\mu_1 \langle F_1 \ldots F_n \rangle\right) \ldots \left(\mu_n \langle F_1 \ldots F_n \rangle\right) (p, \mathrm{in}_1, \ldots, \mathrm{in}_n) \end{aligned} \tag{11}$$

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

Example 5. *The type arguments to the function*

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

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

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

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).

3 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 [4]. 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 $sort_2$, 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 [4] 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

$$\lambda n : \mathbb{N}.\ sort_1\ \mathbb{N}\ Min\ Max\ [X1\ ..\ Xn]\ =\ sort_2\ \mathbb{N}\ Min\ Max\ [X1\ ..\ Xn]$$

(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:

$$\forall n : \mathbb{N}.\ sort_1\ (\text{Fin}\ n)\ Min\ Max\ [1..n] = sort_2\ (\text{Fin}\ n)\ Min\ Max\ [1..n]$$

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 $sort_1$ and $sort_2$ 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 $F\alpha = (\alpha,\alpha)+(\alpha,\alpha)+\text{Fin}\ n$ must be quotiented by the constraints 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 genuine, jutifiable, 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 **??**. 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 function 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:

$$\forall\alpha : *.\forall x : \sigma[\alpha].\ P[\alpha, x] \Rightarrow h_1\ \alpha\ x = h_2\ \alpha\ x \tag{12}$$

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*

$$\sigma[\alpha] = (\Pi_{i\in I}(G_i\alpha \to O_i)) \times (F\alpha \to \alpha) \tag{13}$$

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:

$$\forall p : \Pi_{i\in I}(G_i(\mu(F, Q)) \to O_i).\ h_1\ (\mu(F, Q))\ (p, \mathrm{in}) = h_2\ (\mu(F, Q))\ (p, \mathrm{in}) \tag{14}$$

The proof that (14) 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 $(\mu(F, Q), \mathrm{in} : F(\mu(F, Q)) \to \mu(F, Q))$, then*

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

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

3.1 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*

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

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 : $(D, D) \to D$ and Max : $(D, D) \to D$, we can formally derive that a*

solution to the polymorphic testing problem for the sorting network generator is given by the monomorphic test condition

$$\forall n : \mathbb{N}.\ sort_1\ D\ Min\ Max\ [1..n] = sort_2\ D\ Min\ Max\ [1..n]$$

In this way we can make the informal analysis in [4] 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.

4 Polymorphic Testing with Coalgebraic Constraints

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 a 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

$$\forall\alpha : *.\ \Pi_{i\in I}(G_i\alpha \to O_i) \times \Pi_{j\in J}(F_j\alpha \to \alpha) \times \Pi_{l\in L}(\alpha \to H_l\alpha)\ \to\ H\alpha \tag{15}$$

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 (15) 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 (15) 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 or co-constructor or...?

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 (15) 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 **??** — 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* **??**.

Move example from old intro to here?

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 **??** *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:*

$$\forall\overline{p_i : G_i(\mu(F, Q')) \to O_i}.\ \ h_1\ (\mu(F, Q'))\ (\overline{p_i}, \text{in}, \overline{h_l}) = h_2\ (\mu(F, Q'))\ (\overline{p_i}, \text{in}, \overline{h_l}) \tag{16}$$

Theorem 5 is thus the special case of Theorem 10 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*

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

where $(\!|\overline{f_j}|\!)$ *is the unique* (F, Q')*-algebra homomorphism from* $(\mu(F, Q'), \mathrm{in})$ *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 $Q''[\mu(F, Q'), (\mathrm{in}, \overline{h_l})]$ is satisfied, but this is guaranteed by the second clause of Definition 6. □

The proof of Theorem 10 is now immediate. To see how Theorem 10 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* $[(a, \mathbb{N} + 1)]$*:*

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

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 10 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 10 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.

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.
Relate to Section 4.2 of BJC by doing their Example 8?

5 Higher-order Polymorphic Testing

It is implicit that there is a category of Haskell types and a category of Haskell type constructors here. Do things semantically or syntactically?

It is natural to model monads and other type constructors that are functors as least fixed points of functors on the category of endofunctors on C, i.e., on the functor category of C. In this category, objects are functors and morphisms are natural transformations. Functors on functor categories are called *higher-order functors*.

Writing $F \to G$ for $\forall \alpha : *.\, F\alpha \to G\alpha$ for (first-order) functors F and G, we can state the *higher-order polymorphic testing problem* as:

> Let $\sigma[L] : (* \to *) \to (* \to *)$ be a map between type constructors with a free type constructor variable L, and let $H : (* \to *) \to (* \to *)$ be a definable higher-order functor. Given a pair of functions $H_1, H_2 : \forall L : * \to *.\, \sigma[L] \to HL$, find a *monomorphic* sufficient condition that implies the following property:
>
> $$\forall L : * \to *. \forall K : \sigma[L].\, H_1\ L\ K = H_2\ L\ K \tag{17}$$

We have the following instantiation of Theorem 1 for higher-order functors:

Theorem 7. *Let $\sigma[L]$, H, H_1 and H_2 be as in the description of the higher-order polymorphic testing problem. If there exist higher-order functors $\{G_i\}_{i\in I}$ and F, and type constructors $\{O_i\}_{i\in I}$ such that*

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

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:

$$\forall p : \Pi_{i\in I}(G_i(\mu F) \to O_i).\, H_1\ \mu F\ (p, \mathrm{in}) = H_2\ \mu F\ (p, \mathrm{in}) \tag{18}$$

We can similarly instantiate Theorems 2 and 3 in the higher-order setting. We first need

Definition 7. *Let $\sigma[L]$ and $\tau[L]$ be maps between type constructors with a single free type constructor variable L. A* higher-order embedding-projection pair *$(e, p) : \sigma[L] \hookrightarrow \tau[L]$ consists of a pair of functions $e : \forall L : * \to *.\ \sigma[L] \to \tau[L]$ and $p : \forall L : * \to *.\ \tau[L] \to \sigma[L]$ such that $\forall L : * \to *.\ (p\ L) \circ (e\ L) = \mathrm{id}_{\sigma[L]}$.*

We then have the following higher-order analogue of Theorem 2:

Theorem 8. *If $\tau[L] = (\Pi_{i \in I}(G_i L \to O_i)) \times (FL \to L)$ is a type constructor with a free type constructor variable L, if there exists an initial F-algebra $(\mu F, \mathrm{in} : F(\mu F) \to \mu F)$, if $(e, p) : \sigma[L] \hookrightarrow \tau[L]$ is a higher-order embedding-projection pair, if H is a definable higher-order functor, and if H_1 and H_2 are functions of type $\forall L : * \to *.\ \sigma[L] \to HL$, then*

$$\begin{array}{rl} & \forall s : \Pi_{i \in I}(G_i(\mu F) \to O_i).\ H_1\ \mu F\ (s, \mathrm{in}) = H_2\ \mu F\ (s, \mathrm{in}) \\ \Rightarrow & \\ & \forall L : * \to *.\ \forall K : \sigma[L].\ H_1\ L\ K = H_2\ L\ K \end{array}$$

Finally, we have the following higher-order analogue of Theorem 3:

Theorem 9. *Let $\sigma[L]$, H, H_1 and H_2 be as in the description of the higher-order polymorphic testing problem. If there exist higher-order functors $\{G_i\}_{i \in I}$, a type S*

Or do we want the shape of a higher-order container to be a type or a functor?

, higher-order functors F_s for all $s : S$, and functors $\{O_i\}_{i \in I}$ such that

$$\sigma[L] = (\Pi_{i \in I}(G_i L \to O_i)) \times \Sigma s : S.\ (F_s L \to L) \tag{19}$$

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

$$\forall s : S. \forall p : \Pi_{i \in I}(G_i(\mu F_s) \to O_i).\ H_1\ \mu F_s\ (p, s, \mathrm{in}_s) = H_2\ \mu F_s\ (p, s, \mathrm{in}_s) \tag{20}$$

5.1 Example: Perfect Trees

We consider the data type of *perfect trees* given by

$$PTree\ \alpha = PLeaf\ \alpha + PNode\ (PTree\ (\alpha, \alpha))$$

This data type can be seen as the carrier of the initial algebra for the higher-order functor

$$HPTree\ F\ \alpha = HPLeaf\ \alpha + HPNode\ (HPTree\ F\ (\alpha, \alpha))$$

Example 9. *Suppose we want to compare two functions — suggestively named $pcount_1$ and $pcount_2$ — of type*

$$\forall \alpha : *.\ PTree\ \alpha \to \mathrm{Int} \tag{21}$$

i.e., mapping PTree to K_{Int}. We can instantiate Theorem 9 to solve this instance of the higher-order polymorphic testing problem. We first observe that PTree α is isomorphic to the data type lists over α whose lengths are powers of 2, so that the data constructor PTree can be represented by the "higher-order container"

I don't yet know how to finish this example.

$$\Sigma n : \mathbb{N}.\ (F_n\ L \to L)$$

Here, $F_n = K_{\text{Fin}\,2^n}$??? Δ^n???? K_1????? Thus, the type in (4) is thus isomorphic to

$$\Sigma n : \mathbb{N}.\ (F_n\ L\ \to L) \to K_{\text{Int}}$$

This type is the instance of the type schema from Theorem 9 where H is the higher-order functor mapping each functor L to Int*, i.e., $H = K_{K_{\text{Int}}}$. The monomorphic instance*

$$\forall n : \mathbb{N}.\ pcount_1\ \mu F_n\ (n, \text{in}_n) = pcount_2\ \mu F_n\ (n, \text{in}_n) \tag{22}$$

i.e.,

$$\forall n : \mathbb{N}.\ \forall \alpha : *.\ pcount_1\ (\mu F_n\ \alpha)\ (n, \text{in}_n\ \alpha) = pcount_2\ (\mu F_n\ \alpha)\ (n, \text{in}_n\ \alpha) \tag{23}$$

thus gives a sufficient condition for the higher-order polymorphic testing problem for the (common) type of $pcount_1$ and $pcount_2$. More concretely, this condition expands to testing

$$\forall n : \mathbb{N}.\ \forall \alpha : *.\ pcount_1\ 1\ P = pcount_2\ 1\ P \tag{24}$$

where P is the perfect tree corresponding to $(n, id_{\Delta^n \alpha})$???????, i.e., the perfect tree of depth n with unit data. holds.

5.2 Higher-order Polymorphic Testing with Constraints

Definition 8. *Let $\sigma[L]$ be a type with one free type constructor variable L, let H be a definable higher-order functor, and let $h_1, h_2 : \forall L : * \to *.\ \sigma[L] \to HL$. If there exist higher-order functors $\{G_i\}_{i\in I}$, $\{F_j\}_{j\in J}$, and $\{H_m\}_{m\in M}$ and functors $\{O_i\}_{i\in I}$ such that*

$$\sigma[L] = \Pi_{i\in I}(G_i L \to O_i)\ \times\ \Pi_{j\in J}(F_j L \to L)\ \times\ \Pi_{m\in M}(L \to H_m L)$$

where $F = \Sigma_{j\in J} F_j$, then we say that a predicate $Q[L, (\overline{f_j}, \overline{h_m})]$ on type constructors L, algebras $\overline{f_j}$, and coalgebras $\overline{h_l}$ is a canonical higher-order constraint *if it comprises*

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

Theorem 10. *Let $\sigma[L]$, $\{G_i\}_{i\in I}$, $\{F_j\}_{j\in J}$, $\{H_m\}_{m\in M}$, $\{O_i\}_{i\in I}$, F, H, h_1, h_2, Q, and Q' be as in Definition 8. If $P[L, (p, \overline{f_j}, \overline{h_m})]$ is equivalent to a canonical higher-order constraint $Q[L, (\overline{f_j}, \overline{h_m})]$ then the following condition is a solution for this instance of the polymorphic testing problem with constraints:*

$$\forall \overline{p_i : G_i(\mu(F, Q')) \to O_i}.\ \ h_1\ (\mu(F, Q'))\ (\overline{p_i}, \text{in}, \overline{h_m}) = h_2\ (\mu(F, Q'))\ (\overline{p_i}, \text{in}, \overline{h_m}) \tag{25}$$

5.3 Monadic Constraints

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Categories of Containers. Proceedings, Foundations of Software Science and Computation Structures 2003, pp. 23 – 38.

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

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

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

[5] 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.

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

[7] 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.

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

[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.