Example-Based Reasoning about the Realizability of Polymorphic Programs

NIEK MULLENERS, Utrecht University, Netherlands
JOHAN JEURING, Utrecht University, Netherlands
BASTIAAN HEEREN, Open University of the Netherlands, Netherlands

Parametricity states that polymorphic functions behave the same regardless of how they are instantiated. When developing polymorphic programs, Wadler's free theorems can serve as *free specifications*, which can turn otherwise partial specifications into total ones, and can make otherwise realizable specifications *unrealizable*. This is of particular interest to the field of program synthesis, where the unrealizability of a specification can be used to prune the search space. In this paper, we focus on the interaction between parametricity, input-output examples, and sketches. Unfortunately, free theorems introduce universally quantified functions that make automated reasoning difficult. Container morphisms provide an alternative representation for polymorphic functions that captures parametricity in a more manageable way. By using a translation to the container setting, we show how reasoning about the realizability of polymorphic programs with input-output examples can be automated.

CCS Concepts: • **Software and its engineering** → **Programming by example**; • **Theory of computation** → **Type theory**; *Automated reasoning*.

Additional Key Words and Phrases: parametricity, container functors, unrealizability, program synthesis, example propagation

ACM Reference Format:
Niek Mulleners, Johan Jeuring, and Bastiaan Heeren. 2024. Example-Based Reasoning about the Realizability of Polymorphic Programs. *Proc. ACM Program. Lang.* 8, ICFP, Article 247 (August 2024), 21 pages. https://doi.org/10.1145/3674636

1 Introduction

The design of a program typically starts by specifying knowledge about the problem [Felleisen et al. 2018]. Often, this knowledge is in the form of *types*, *input-output examples*, and *sketches* (incomplete programs containing holes). An example of each of these specifications for the function *reverse* is shown in Figure 1. Types, examples, and sketches can help a programmer formulate a mental model of the problem at hand. Many modern languages also have native support for specifying types, examples, and sketches through type annotations, assertions, and holes. These specifications allow the programmer to express their intent to a programming environment and, in return, the environment could check whether the programmer's mental model is correct and catch possible mistakes early. To do so, the environment has to reason about the *realizability* of a program as specified by the programmer, i.e. whether a program adhering to this specification exists.

Authors' Contact Information: Niek Mulleners, Utrecht University, Utrecht, Netherlands, n.mulleners@uu.nl; Johan Jeuring, Utrecht University, Utrecht, Netherlands, j.t.jeuring@uu.nl; Bastiaan Heeren, Open University of the Netherlands, Heerlen, Netherlands, bastiaan.heeren@ou.nl.

ACM 2475-1421/2024/8-ART247
https://doi.org/10.1145/3674636

$$\mathit{reverse} : \forall a.\ [a] \rightarrow [a]$$

(a) Type

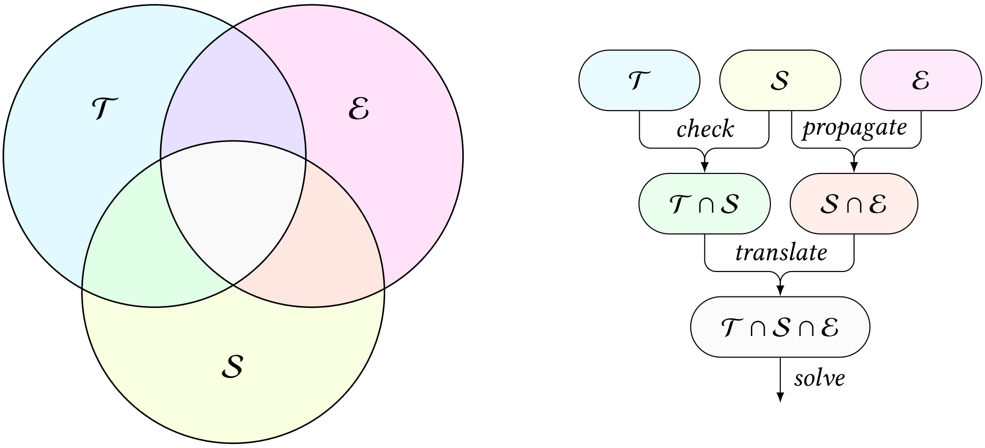
$$\mathit{reverse}\ [1, 2, 3] \equiv [3, 2, 1]$$

(b) Input-output example

$$\exists f\ e.\ \mathit{reverse} = \mathit{foldr}\ f\ e$$

(c) Sketch

Fig. 1. A specification for the function *reverse*.


$\mathcal{T}$
$\mathcal{E}$
$\mathcal{S}$
$\mathcal{T}$
$\mathcal{S}$
$\mathcal{E}$
check
propagate
$\mathcal{T} \cap \mathcal{S}$
$\mathcal{S} \cap \mathcal{E}$
translate
$\mathcal{T} \cap \mathcal{S} \cap \mathcal{E}$
solve

(a) The intersections of programs of type τ, implementing the set of examples ε, and refining the sketch ς.

(b) The pipeline for computing the realizability of a type τ, a set of examples ε, and a sketch ς.

Fig. 2. $\mathcal{T}$ is the set of programs of type τ. $\mathcal{E}$ is the set of programs implementing the examples ε. $\mathcal{S}$ is the set of programs refining the sketch ς.

A type is realizable exactly if it is inhabited [Urzyczyn 1997]. If a type is uninhabited, such as the type in Figure 3a, no program of that type exists. A set of input-output examples is realizable exactly if the examples do not contradict each other. If a set of input-output examples is contradictory, such as the examples in Figure 3b, no program implementing those examples exists. Realizability of a type and realizability of a set of examples do not imply realizability of the combination! We are interested in exploring whether the intersections of program spaces as specified by types, examples, and sketches are empty. These intersections are visualized in Figure 2a.

Types and Examples. The interaction between types and examples can be fairly subtle. Take for example the specification in Figure 3c. Both the type and example are individually realizable, but *parametricity* tells us that the only function with this type is the identity function [Reynolds 1983; Wadler 1989], contradicting the input-output example. We are not aware of prior work that investigates the realizability of polymorphic programs with input-output examples. One reason for this could be that types and examples (as opposed to sketches) are typically assumed to be ground truth. This assumption does not hold when those types and examples are generated automatically. In particular, we will consider types and examples as propagated through a sketch.

Types and Sketches. Types and program sketching go hand in hand. Typed holes have recently been embraced by statically typed functional programming languages such as Haskell [GHC Team 2023; Gissurarson 2018], Agda [Norell 2009], and Hazel [Omar et al. 2019]. Holes allow partial programs to type check, while providing clear subgoals (hole types) for finishing those programs.

$f : \forall a\, b.\ a \to b$

(a) An uninhabited type

$f\ 1 \equiv 3\ \wedge\ f\ 1 \equiv 5$

(b) Contradicting examples

$f : \forall a.\ a \to a\ \wedge\ f\ 1 \equiv 3$

(c) Parametricity at work

Fig. 3. Specifications that are unrealizable.

Examples and Sketches. Whereas examples are typically used as unit tests for complete programs, recent work has explored how and when incomplete programs (sketches) can be checked against input-output examples, resulting in new input-output examples on the holes [Feser et al. 2015; Lubin et al. 2020; Mulleners et al. 2023; Omar et al. 2019]. We refer to this process as *example propagation*.

Types, Examples, and Sketches. When inferring types and input-output examples for the holes of a sketch, reasoning about the realizability of those types and examples becomes a worthy avenue, since the realizability of a sketch relies on the realizability of its holes. For example, assume a programmer specifies *reverse* as in Figure 1, but instead incorrectly writes the sketch *reverse* = *map* f. To finish the program, there has to exist a function f, i.e. the realizability of *reverse* with this sketch implies the realizability of f. The sketch can be evaluated against the input-output example by inlining the definitions of *map* and equality on lists:

$$\mathit{map}\ f\ [1,2,3] \equiv [3,2,1] \implies [f\ 1, f\ 2, f\ 3] \equiv [3,2,1] \implies f\ 1 \equiv 3 \wedge f\ 2 \equiv 2 \wedge f\ 3 \equiv 1$$

From the type of *reverse*, the type of f is inferred to be $a \to a$, for some fixed a. As we saw before, the only function of this type is the identity function, which contradicts the input-output examples. The unrealizability of f allows us to correctly conclude that the sketch *reverse* = *map* f is unrealizable.

Parametricity. It should be possible to prove the unrealizability of *reverse* in terms of *map* directly using Reynolds' abstraction theorem [Reynolds 1983] or Wadler's free theorems [Wadler 1989]. However, such proofs do not generalize easily. More importantly, we are not just interested in proving the realizability of arbitrary specifications. Rather, we want a decision procedure that decidably computes whether a specification is realizable or not. Intuitively, a parametrically polymorphic function cannot *inspect* or *produce* polymorphic elements, only "pass them around". This intuition is perfectly captured by *container functors* and *container morphisms* [Abbott et al. 2005], which provide normal forms for polymorphic datatypes and polymorphic functions respectively.

In this paper, we propose a general procedure for computing the realizability of polymorphic programs: first, *type checking* and *example propagation* are used to infer types and input-output examples for the holes of a sketch; then, the resulting constraints are *translated* to the container setting; lastly, the translated constraints are *solved* using an SMT solver. This procedure is visualized in Figure 2b. More specifically, our contributions are to show how to

- compute the realizability of polymorphic types with monomorphic input-output examples (Section 3);
- use example propagation to extend this reasoning to sketching with higher-order combinators such as *map* (Section 3.4);
- reason about input-output traces to support sketching with recursion schemes such as *foldr* (Section 4);
- generalize the notion of trace completeness to take parametricity into account (Section 4.2).

2 Background: Containers

Any list can be uniquely represented by its length n and a function f assigning a value to every index in $\{0, 1, \ldots, n-1\}$. For example, the list $[A, B, C]$ can be represented by $n = 3$, $f\ 0 = A$, $f\ 1 = B$, and $f\ 2 = C$. We say that a list of length n is a *container* with *shape* n and *positions* $\{0, 1, \ldots, n-1\}$. Many datatypes can similarly be represented in terms of their shape and positions.

Containers [Abbott et al. 2003] present a generic way of modeling datatypes that *contain* elements, making the distinction between the structure and the content of a datatype explicit. A container $S \triangleright P$ (sometimes written $(s : S) \triangleright P_s$) is defined by a type of shapes S, representing the structure of the datatype, and for every shape s a type of positions P_s, describing where the elements are located.

Definition 1. *The list container is defined as* $\mathbb{N} \triangleright \mathit{Fin}$*, where* $\mathit{Fin}\ n = \{0, 1, \ldots, n-1\}$*, i.e. the type of natural numbers smaller than* n*. A list* $[x_0, \ldots, x_{n-1}]$ *is represented using the list container as a pair* $(n, \lambda i.\ x_i)$*.*

Functors that can be defined as containers are called *container functors*. A container functor F is isomorphic to the *extension* of some container $S \triangleright P$, defined as $[\![S \triangleright P]\!]\ a = \Sigma_{(s:S)}.\ (P_s \longrightarrow a)$. In other words, if F is a container functor corresponding to the container $S \triangleright P$, values of type $F\ a$ can be represented by a pair (s, p), where s is a shape of type S and p is a function assigning an element of type a to every position P_s.

2.1 Constructing Containers

The simplest container is the identity container, having only a single shape and a single position.

Definition 2. *The identity container is defined as* $\mathbb{1} \triangleright K\ \mathbb{1}$*, where* $\mathbb{1}$ *is the unit type with a single element* $\star$ *and* K *is the constant function, i.e.* $K\ x\ y = x$*. A value* $x : a$ *is represented using the identity container as a pair* $(\star, K\ x)$*.*

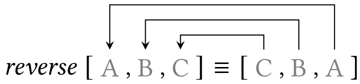
Containers can be combined to create more complex containers. A pair of lists, for example, has two natural numbers n_1 and n_2 as its shape (the lengths of the lists) and exactly $n_1 + n_2$ positions where elements are stored.

Definition 3. *The product of two containers* $S \triangleright P$ *and* $T \triangleright Q$ *is defined as* $((s, t) : S \times T) \triangleright (P_s + Q_t)$*. Given that the container representation of* x *and* y *are* (s, p) *and* (t, q) *respectively, the pair* (x, y) *is represented as* $((s, t), p \oplus q)$*, where* $(p \oplus q)\ (\mathbf{inl}\ z) = p\ z$ *and* $(p \oplus q)\ (\mathbf{inr}\ z) = q\ z$*.*

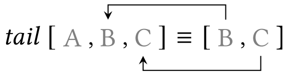
Using combinators like these, containers can be used to construct all *strictly positive types*, which can intuitively be understood as all types that have a tree-like structure. Additionally, the translation between values of inductively defined strictly positive types and their container representation can be automated [Abbott et al. 2005]. As a convention, we use $\hat{\bullet}$ to denote the translated shape component and $\mathring{\bullet}$ to denote the translated position component. For example, the functor F corresponds to the container $\hat{F} \triangleright \mathring{F}$ and a value x of type $F\ a$ is translated to a pair $(\hat{x}, \mathring{x})$.

2.2 Container Morphisms

Given a list $[x_0, x_1, \ldots, x_{n-1}]$, the reverse of that list is $[x_{n-1}, x_{n-2}, \ldots, x_0]$. Using the container representation of lists, the reverse of the list $(n, \lambda i.\ x_i)$ is the list $(n, \lambda i.\ x_{n-i-1})$. Note how we can describe list reversal by specifying, for each index i in the output list, where the element at that index comes from in the input list: the index $n - i - 1$. We can use arrows to visualize where each element comes from:



Similarly, the tail of the list $(n, \lambda i.\ x_i)$ is the list $(n-1, \lambda i.\ x_{i+1})$:



This idea of defining a function in terms of how the shape changes and where each element in the output comes from is formalized using *container morphisms*. A container morphism between the containers $S \triangleright P$ and $T \triangleright Q$ is a pair of functions (u, g), where $u : S \longrightarrow T$ maps input shapes to output shapes and $g : \Pi_{s:S}(Q_{(u\ s)} \longrightarrow P_s)$ maps output positions to input positions [Abbott et al. 2003]. Notice again the explicit separation between structure and content. The *extension* of a container morphism (u, g) (written $\langle\!\langle u, g\rangle\!\rangle$) is a function mapping (s, p) to $(u\ s, p \circ g_s)$. We can define *reverse* and *tail* as the extension of a container morphism as follows:

$$reverse = \langle\!\langle(\lambda n.\ n), (\lambda n\ i.\ n - i - 1)\rangle\!\rangle \qquad tail = \langle\!\langle(\lambda n.\ n - 1), (\lambda n\ i.\ i + 1)\rangle\!\rangle$$

A container morphism represents a natural transformation between container functors. Moreover, every natural transformation between container functors can be defined as the extension of a container morphism [Abbott et al. 2005].

2.3 Small Containers

Small containers (also known as *finitary containers*) are containers whose shapes have a decidable equality and whose positions are finite sets [Prince et al. 2008]. The list container is an example of a small container. Even though a list may contain any number of elements, for any given shape it has a finite number of positions. All the containers in this paper are small, enabling us to enumerate over their positions, which is crucial in decidably computing the realizability of a program.

3 Realizability of Input-Output Examples with Polymorphic Types

To reason about the realizability of an input-output example $f\ x \equiv y$, where f is a natural transformation, we assume that there exists a container morphism $(\hat{f}, \mathring{f})$ satisfying this example. To do so, we have to translate the input x and the output y to the container setting. If we can show that such a container morphism $(\hat{f}, \mathring{f})$ is not realizable, then neither is the natural transformation f.

3.1 Translating a Single Example

An input-output example for $f : \forall a.\ F\ a \longrightarrow G\ a$ is defined by a triple (τ, x, y), where τ is a monomorphic type, x is an input of type $F\ \tau$, and y is an output of type $G\ \tau$, such that $f\ x \equiv y$. Given that F and G are container functors isomorphic to $[\![\hat{F} \triangleright \mathring{F}]\!]$ and $[\![\hat{G} \triangleright \mathring{G}]\!]$ respectively, we can translate x and y to the container setting. The input x is translated to a pair $(\hat{x}, \mathring{x})$, where $\hat{x} : \hat{F}$ and $\mathring{x} : \mathring{F}_{\hat{x}} \longrightarrow \tau$ and the output y is translated to a pair $(\hat{y}, \mathring{y})$, where $\hat{y} : \hat{G}$ and $\mathring{y} : \mathring{G}_{\hat{y}} \longrightarrow \tau$. Since f is a natural transformation between F and G, it corresponds to a container morphism $(\hat{f}, \mathring{f})$ between $\hat{F} \triangleright \mathring{F}$ and $\hat{G} \triangleright \mathring{G}$. There exists a function f such that $f\ x \equiv y$ exactly if there exists a container morphism $(\hat{f}, \mathring{f})$ such that $\langle\!\langle\hat{f}, \mathring{f}\rangle\!\rangle\ (\hat{x}, \mathring{x}) \equiv (\hat{y}, \mathring{y})$:

$$\begin{tikzcd} \exists f.\ \boxed{\forall a.\, f : F\,a \to G\,a} \land \boxed{f\;x \equiv y} \arrow[d] \\ \exists(\hat{f}, \mathring{f}).\ \boxed{\langle\!\langle \hat{f}, \mathring{f} \rangle\!\rangle\ (\hat{x}, \mathring{x}) \equiv (\hat{y}, \mathring{y})} \end{tikzcd}$$

To solve the equation $\langle\!\langle \hat{f}, \mathring{f} \rangle\!\rangle\ (\hat{x}, \mathring{x}) \equiv (\hat{y}, \mathring{y})$, we first rewrite it by applying the container morphism extension, resulting in $(\hat{f}\ \hat{x}, \mathring{x} \circ \mathring{f}) \equiv (\hat{y}, \mathring{y})$.[1] Using equivalence on containers, as well as function extensionality, we separate the *shape* constraints on $\hat{f}$ from the *position* constraints on $\mathring{f}$:

$$\underbrace{\hat{f}\ \hat{x} \equiv \hat{y}}_{\textit{shape}} \land \underbrace{\forall q.\ \mathring{x}\ (\mathring{f}\ q) \equiv \mathring{y}\ q}_{\textit{position}}$$

Next, we make the intermediate argument returned by $\mathring{f}$ explicit by introducing an existential quantification:

$$\underbrace{\hat{f}\ \hat{x} \equiv \hat{y}}_{\text{(a)}} \land \forall q.\ \exists p.\ \underbrace{\mathring{f}\ q \equiv p}_{\text{(b)}} \land \underbrace{\mathring{x}\ p \equiv \mathring{y}\ q}_{\text{(c)}} \tag{1}$$

The resulting equation consists of three components:

(a) An input-output example for $\hat{f}$, describing the realizability of the shape morphism.
(b) A set of input-output examples for $\mathring{f}$, describing the realizability of the position morphism.
(c) A consistency check with respect to parametricity, ensuring that each element in the output also occurs in the input. Note how $\forall q.\ \exists p.\ \mathring{x}\ p \equiv \mathring{y}\ q$ can be rewritten as $\textit{codomain}(\mathring{y}) \subseteq \textit{codomain}(\mathring{x})$.

Similar to how the consistency of input-output examples with respect to a sketch can be computed using example propagation, resulting in input-output examples for the subcomponents of the program (the holes), our translation to the container setting checks the consistency of input-output examples with respect to a polymorphic type, resulting in input-output examples for the abstract subcomponents of the program (the shape and position morphisms).

To figure out if f is realizable, we have to solve the satisfiability of Equation 1, but this is complicated by the universal and existential quantifications. If we assume, however, that $\hat{F} \triangleright \mathring{F}$ and $\hat{G} \triangleright \mathring{G}$ are *small containers*, we can get rid of any quantifications by enumerating over the positions:

$$\hat{f}\ \hat{x} \equiv \hat{y} \land \bigwedge_{q} \bigvee_{p} \mathring{f}\ q \equiv p \land \mathring{x}\ p \equiv \mathring{y}\ q \tag{2}$$

When we know $\mathring{x}$ and $\mathring{y}$, the satisfiability of this formula is trivially determined by an SMT solver.

3.2 Example: Reverse as Map

To see how example consistency can be used to reason about the realizability of a program, we will turn our attention once more to the example of *reverse* in terms of *map*:

$$\boxed{\forall a.\ \textit{reverse} : [a] \to [a]} \land \boxed{\exists f.\ \textit{reverse} = \textit{map}\ f} \land \boxed{\textit{reverse}\ [A, B, C] \equiv [C, B, A]}$$

The first step in checking the realizability of *reverse* is to propagate the type and input-output example through the sketch. Type checking gives $f : a \to a$, for a fixed a. To propagate the

[1] For the sake of readability, we leave the dependent argument to $\mathring{f}$ implicit, as it can be inferred from the type of its argument.

example, we evaluate the sketch applied to the input $[A, B, C]$ as far as possible, by inlining the definition of *map*:

$$\mathit{map}\, f\ [A, B, C] \quad \rightsquigarrow \quad [f\ A, f\ B, f\ C]$$

Then, we simplify the resulting equation using equality on lists:

$$[f\ A, f\ B, f\ C] \equiv [C, B, A] \qquad \Longrightarrow \qquad f\ A \equiv C\ \wedge\ f\ B \equiv B\ \wedge\ f\ C \equiv A$$

We will focus on the first conjunct (highlighted): intuitively, $f\ A \equiv C$ is inconsistent, because the C in the output does not occur in the input of f. This implies that f is unrealizable, which we can prove formally by translating the example $f\ A \equiv C$ to the container setting. We interpret f as a natural transformation $(\hat{f}, \mathring{f})$ between the identity functor and translate the values A and C accordingly:

$$\begin{tikzcd} \exists f.\ \boxed{\forall a.\ f : a \longrightarrow a} \wedge \boxed{f\ A \equiv C} \arrow[d] \\ \exists(\hat{f}, \mathring{f}).\ \boxed{\langle\!\langle \hat{f}, \mathring{f} \rangle\!\rangle\ (\star, \mathrm{K}\, A) \equiv (\star, \mathrm{K}\, C)} \end{tikzcd}$$

We write out the resulting constraint on $(\hat{f}, \mathring{f})$ according to Equation 1 and simplify:

$$\underbrace{\hat{f}\ \star \equiv \star}_{\text{(a)}} \ \wedge\ \underbrace{\mathring{f}\ \star \equiv \star}_{\text{(b)}} \ \wedge\ \underbrace{A \equiv C}_{\text{(c)}}$$

The constraints on $\hat{f}$ and $\mathring{f}$ are trivially satisfied, but the consistency check (c) fails: the codomain of $\mathrm{K}\, C$ is not a subset of the codomain of $\mathrm{K}\, A$, i.e. $\{C\} \nsubseteq \{A\}$. We conclude that f, and, by extension, *reverse* as a *map*, is unrealizable.

3.3 Translating Multiple Examples

The only way to show that a single input-output example is unrealizable is by showing that it fails the consistency check (c). However, when considering multiple input-output examples, their realizability additionally relies on the realizability of their respective shape morphisms (a) and position morphisms (b).

A polymorphic function $f : \forall a.\ [a] \longrightarrow [a]$ cannot *inspect* the elements in the input list and can therefore not differentiate between different input lists that have the same length (i.e. the same *shape*). When a set of input-output examples for f requires inspecting the input elements, this will result in either a shape conflict on $\hat{f}$ or a position conflict on $\mathring{f}$. For example, assume that f is supposed to sort the input list and remove any duplicates. This requires inspecting the elements in the list, so we expect to find a contradiction:

- The examples $f\ [A, A] \equiv [A] \wedge f\ [A, B] \equiv [A, B]$ imply the contradictory shape constraint $\hat{f}\ 2 \equiv 1 \wedge \hat{f}\ 2 \equiv 2$, leading to a *shape conflict*.
- The examples $f\ [A, B] \equiv [A, B] \wedge f\ [B, A] \equiv [A, B]$ imply the contradictory position constraint $\mathring{f}\ 0 \equiv 0 \wedge \mathring{f}\ 0 \equiv 1$, leading to a *position conflict*.

3.4 Sketching with Map

Now that we have seen how to reason about the realizability of sets of input-output examples, it is straightforward to extend this reasoning to any program p specified with the sketch $\exists f.\ p = \mathit{map}\, f$, by recognizing that an input-output example for p represents a set of input-output examples for f. To be precise, an input-output example for p with an input list of length n corresponds to exactly

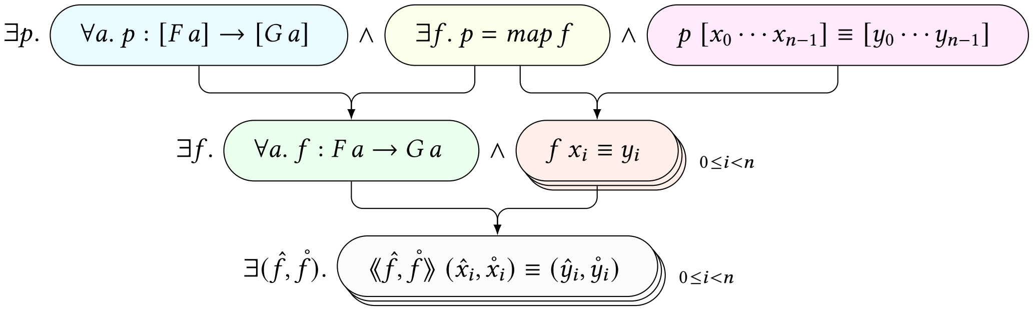

$\exists p.$
$\forall a.\ p : [F\ a] \to [G\ a]$
$\wedge$
$\exists f.\ p = map\ f$
$\wedge$
$p\ [x_0 \cdots x_{n-1}] \equiv [y_0 \cdots y_{n-1}]$
$\exists f.$
$\forall a.\ f : F\ a \to G\ a$
$\wedge$
$f\ x_i \equiv y_i$
$0 \le i < n$
$\exists (\hat{f}, \mathring{f}).$
$\langle\!\langle \hat{f}, \mathring{f} \rangle\!\rangle\ (\hat{x}_i, \mathring{x}_i) \equiv (\hat{y}_i, \mathring{y}_i)$
$0 \le i < n$

Fig. 4. The pipeline for computing the realizability of a polymorphic program defined as a *map* with a single input-output example.

n input-output examples for f. The complete pipeline for programs defined as a *map* is shown in Figure 4. Note how each step in the pipeline changes the focus: first the whole program p; then the hole f; and finally the container morphism $(\hat{f}, \mathring{f})$.

Soundness. Realizability reasoning is sound if we cannot draw false conclusions from its results. This is the case exactly if the initial equation in the pipeline is equivalent to the final equation. That way, a solution to the final equation implies that a correct program exists; and a contradiction implies that no such program exists. We argue that reasoning about the realizability of a program as a *map* is sound, as all the steps shown in Figure 4 are valid rewritings.

Completeness. Realizability reasoning is complete exactly if it is a decision procedure, i.e. it always returns either a solution or a contradiction. This is true if the final equation in the pipeline is in a decidable logic. Reasoning about the realizability of a program as a *map* is complete as long as F and G are small container functors. That way, the final equation in Figure 4 is in a quantifier free logic, where each component is an instantiation of Equation 2.

4 Reasoning About Recursion Schemes

One of the most ubiquitous functions in functional programming is *foldr*, a recursion scheme that embodies structural recursion over inductively defined datastructures. Gibbons et al. [2001] ask the question “When is a function a fold?” and show how to prove whether a function is indeed a fold. Intuitively speaking, and restricting ourselves to folds over lists, we say that a function h is a fold if, for any element x and any list xs, the result of $h\ (x : xs)$ is uniquely determined by x and $h\ xs$. For example, the function *reverse* is a fold, since

$$reverse\ (x : xs) = reverse\ xs +\!\!+ [x].$$

The function $tail : \forall a.\ [a] \to [a]$ (taken from [Gibbons et al. 2001]), however, given by the equations

$$\begin{aligned} tail\ [\,] &= [\,] \\ tail\ (x : xs) &= xs \end{aligned}$$

is not a fold, because $tail\ xs$ throws away the first element of xs, which is needed in the tail of $x : xs$. In Haskell terms, $tail$ is not a fold because there exist no f and e such that $tail = foldr\ f\ e$.

4.1 Realizability of Tail

Hofmann [2010] shows how to detect whether a set of input-output examples specifies a fold. Similar results can be achieved using example propagation [Feser et al. 2015; Lubin et al. 2020]. In this section, we will use a combination of example propagation and parametric reasoning to show that *tail* is *not* a fold using a minimal number of input-output examples.

To prove automatically that *tail* is not a fold, we will prove that $tail : \forall a.\ [a] \rightarrow [a]$ with the sketch $\exists f.\ tail = foldr\ f\ []$ is unrealizable:[2]

$$\boxed{\forall a.\ tail : [a] \rightarrow [a]} \wedge \boxed{\exists f.\ tail = foldr\ f\ []} \wedge \boxed{\cdots}$$

Whether and how we can prove the unrealizability of this specification depends on the exact input-output examples. For example, take the following input-output examples for *tail*:

$$\boxed{tail\ [A, B, C] \equiv [B, C]\ \wedge\ tail\ [D, E] \equiv [E]}$$

Since *foldr* uses structural recursion, the call $tail\ [A, B, C]$ relies on recursive calls to $[B, C]$, $[C]$, and $[]$. Because we cannot inspect the elements, we cannot distinguish the call to $[B, C]$ from the call to $[D, E]$. Since the call to $[D, E]$ returns $[E]$, throwing away the first element, the element B is also thrown away in the call to $[B, C]$. This implies that the result of tail $[A, B, C]$ cannot contain the value B, contradicting our example.

Formally, the proof starts with propagating the type and input-output examples through the sketch. Given the types of *tail* and *foldr*, the type of f is inferred to be $a \times [a] \rightarrow [a]$, for some fixed a. To propagate the example $tail\ [A, B, C] \equiv [B, C]$, we evaluate the sketch applied to the input $[A, B, C]$, by inlining the definition of *foldr*:

$$foldr\ f\ []\ [A, B, C] \quad \rightsquigarrow \quad f\ (A, f\ (B, f\ (C, [])))$$

Unfortunately, the resulting equation $f\ (A, f\ (B, f\ (C, []))) \equiv [B, C]$ is *not* an input-output example for f, but rather an input-output *trace*, specifying the trace of calls made to f resulting in the output $[B, C]$. We will make the intermediate results returned by f explicit by introducing two existentially quantified variables, revealing that an input-output trace represents a set of related input-output examples:

$$\boxed{\exists x\ y.\ f\ (A, x) \equiv [B, C] \wedge f\ (B, y) \equiv x \wedge f\ (C, []) \equiv y}$$

To translate these examples, we interpret f as a natural transformation $(\hat{f}, \mathring{f})$ between the non-empty list container (i.e. the product of the identity container and the list container) and the list container. For each existentially quantified variable x, we introduce two existentially quantified variables $\hat{x}$ and $\mathring{x}$:

$$\begin{aligned} \exists (\hat{x}, \mathring{x})\ (\hat{y}, \mathring{y}).\ & \langle\!\langle \hat{f}, \mathring{f} \rangle\!\rangle\ ((\star, \hat{x}), \mathrm{K}\ A \oplus \mathring{x}) \equiv (2, \{0 \mapsto B, 1 \mapsto C\}) \\ \wedge\ & \langle\!\langle \hat{f}, \mathring{f} \rangle\!\rangle\ ((\star, \hat{y}), \mathrm{K}\ B \oplus \mathring{y}) \equiv (\hat{x}, \mathring{x}) \\ \wedge\ & \langle\!\langle \hat{f}, \mathring{f} \rangle\!\rangle\ ((\star, 0), \mathrm{K}\ C) \qquad \equiv (\hat{y}, \mathring{y}) \end{aligned}$$

We first focus on the position component of $f\ (A, x) \equiv [B, C]$. After simplifying:

$$(\mathrm{K}\ A \oplus \mathring{x})(\mathring{f}\ 0) \equiv B \wedge (\mathrm{K}\ A \oplus \mathring{x})(\mathring{f}\ 1) \equiv C$$

The constant function $\mathrm{K}\ A$ cannot return B or C, implying that both B and C are in the codomain of $\mathring{x}$:

$$\{B, C\} \subseteq codomain(\mathring{x}) \tag{3}$$

In other words, both B and C are returned by the recursive call $tail\ [B, C]$. Given the polymorphic type of *tail*, this should conflict the constraint $tail\ [D, E] \equiv [E]$. To prove this, we make the input-output trace incurred by $tail\ [D, E] \equiv [E]$ explicit:

[2] Note that this defines $tail\ [] = []$.

$$\boxed{\exists z.\ f\ (D,z) \equiv [E] \land f\ (E,[\,]) \equiv [\,]}$$

Next, we will match up the shape components of the different calls to f: the input shapes of $f\ (C,[\,])$ and $f\ (E,[\,])$ are equal, so we can unify their output shapes $\hat{y}$ and $\hat{z}$; in doing so, the input shapes of $f\ (B,y)$ and $f\ (D,z)$ become equal, showing that $\hat{x}$ is equal to 1:

$$\left.\begin{array}{l} \left.\begin{array}{lcl} f\ (C,[\,]) \equiv y & \Longrightarrow & \hat{f}\ (\star,0) \equiv \hat{y} \\ f\ (E,[\,]) \equiv z & \Longrightarrow & \hat{f}\ (\star,0) \equiv \hat{z} \end{array}\right\}\ \hat{y} \equiv \hat{z} \\ \begin{array}{lcl} f\ (B,y) \equiv x & \Longrightarrow & \hat{f}\ (\star,\hat{y}) \equiv \hat{x} \\ f\ (D,z) \equiv [E] & \Longrightarrow & \hat{f}\ (\star,\hat{z}) \equiv 1 \end{array} \end{array}\right\}\ \hat{x} \equiv 1$$

This implies that the domain of $\mathring{x} : \mathit{Fin}\ \hat{x} \to \tau$ is equal to $\mathit{Fin}\ 1$. In other words, we used the shape component of $\mathit{tail}\ [D,E] \equiv [E]$ to show that the recursive call to $\mathit{tail}\ [B,C]$ returns a list with a single element. Naturally, this list cannot contain both B and C, contradicting (3) and thereby proving that *tail* is not a fold. Finding this contradiction by hand requires some effort, but this is trivial for an SMT solver.

Generalizing Beyond Tail. Adding a constant, monomorphic input to a set of examples does not affect realizability. Hence, our proof readily extends to functions that generalize over *tail* using an additional argument:

- the function $\mathit{drop} : \mathbb{N} \to [a] \to [a]$, which drops the first n elements from a list, specializes to *tail* by setting n to 1;
- the function $\mathit{removeAt} : \mathbb{N} \to [a] \to [a]$, which removes an element from a list at index i, specializes to *tail* by setting i to 0.

Neither *drop* nor *removeAt* can be implemented using the sketch $\lambda n.\ \mathit{foldr}\ (f\ n)\ [\,]$. There is, however, a way to implement *drop* and *removeAt* as a fold, using the sketch $\lambda n\ \mathit{xs}.\ \mathit{foldr}\ f\ e\ \mathit{xs}\ n$, which instantiates *foldr* such that the output type is $\mathbb{N} \to [a]$. See Section 5.1 for a short discussion of these different interpretations.

4.2 Sketching with Foldr

To reason about program traces generated by *foldr*, we look at how *foldr* iteratively builds up a result. When evaluating $\mathit{foldr}\ (+)\ 1\ [2,3,4]$, two intermediate results are computed ($4+1=5$ and $3+5=8$) before the final result ($2+8=10$) is computed.[3] Note how each intermediate result is computed by combining the previous result with an element from the input list, going through the list from right to left (hence the name *foldr*, which stands for *right fold*). To reflect this, we write an input-output example for *foldr* as follows, numbering the elements in the list from right to left:

$$\mathit{foldr}\ f\ y_0\ [x_{n-1},\ldots,x_0] \equiv y_n$$

The argument y_0 is the initial result and the output y_n is the final result. There are $n-1$ intermediate results $(y_1,\ldots,y_{n-1})$, which we can make explicit:

$$\begin{array}{rl} \exists y_1 \cdots y_{n-1}. & y_1 \equiv f\ (x_0,y_0) \\ \land & y_2 \equiv f\ (x_1,y_1) \\ & \vdots \\ \land & y_n \equiv f\ (x_{n-1},y_{n-1}) \end{array}$$

[3] The exact order of evaluation depends on the evaluation strategy, but we can still reason about the intermediate results.

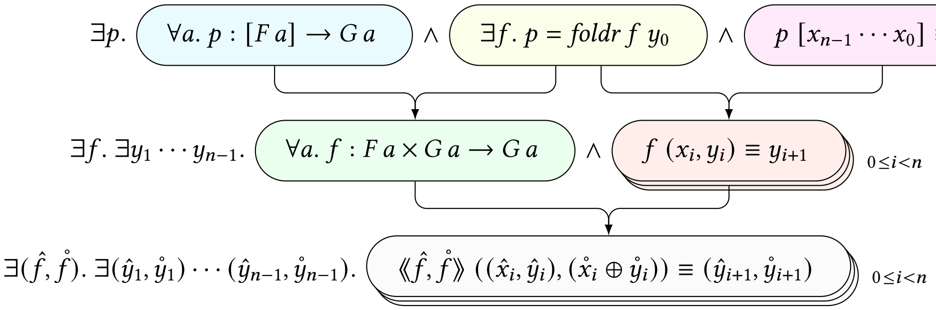

$\exists p.$
$\forall a.\ p : [F\,a] \to G\,a$
$\wedge$
$\exists f.\ p = \mathit{foldr}\ f\ y_0$
$\wedge$
$p\ [x_{n-1} \cdots x_0] \equiv y_n$
$\exists f.\ \exists y_1 \cdots y_{n-1}.$
$\forall a.\ f : F\,a \times G\,a \to G\,a$
$\wedge$
$f\ (x_i, y_i) \equiv y_{i+1}$
$0 \leq i < n$
$\exists(\hat{f}, \mathring{f}).\ \exists(\hat{y}_1, \mathring{y}_1) \cdots (\hat{y}_{n-1}, \mathring{y}_{n-1}).$
$\langle\!\langle \hat{f}, \mathring{f} \rangle\!\rangle\ ((\hat{x}_i, \hat{y}_i), (\mathring{x}_i \oplus \mathring{y}_i)) \equiv (\hat{y}_{i+1}, \mathring{y}_{i+1})$
$0 \leq i < n$

Fig. 5. The pipeline for computing the realizability of a polymorphic program defined as a fold with a single input-output example.

This allows us to concisely define the input-output example on our fold as a set of input-output examples on f:

$$\exists y_1 \cdots y_{n-1}.\ \bigwedge_{0 \leq i < n} f\ (x_i, y_i) \equiv y_{i+1}$$

Along with the inferred type of f, these input-output examples are translated to the container setting. The resulting pipeline for programs defined as a fold is shown in Figure 5.

Soundness. Using the same argument as in Section 3.4, reasoning about the realizability of a program as a fold is sound, as all the steps shown in Figure 5 are valid rewritings.

Completeness. When considering only a single input-output example, reasoning about the realizability of a program as a fold is *not* complete. To see why this is the case, let us write out the final equation of Figure 5 to separate the *shape* and *position* constraints:

$$\bigwedge_{0 \leq i < n} \underbrace{\hat{f}\ (\hat{x}_i, \hat{y}_i) \equiv \hat{y}_{i+1}}_{\mathit{shape}} \ \wedge\ \underbrace{\forall q.\ (\mathring{x}_i \oplus \mathring{y}_i)\ (\mathring{f}\ q) \equiv \mathring{y}_{i+1}\ q}_{\mathit{position}} \tag{4}$$

Note that the values of the intermediate results $y_1, \ldots, y_{n-1}$ (and therefore their shapes $\hat{y}_1, \ldots, \hat{y}_{n-1}$) are not known. This means that we do not know the exact types of q and cannot simply eliminate the universal quantifications by enumeration as we did in Equation 2. In other words, the realizability of an input-output trace cannot be expressed in a quantifier-free logic.

If, however, all recursive calls are represented in the set of input-output examples (a property known as *trace completeness* [Osera and Zdancewic 2015]), the values of the intermediate results would be known and all universal quantifiers could be eliminated. Moreover, since the types of universally quantified variables are only determined by the *shapes* of intermediate results, we can relax trace completeness to only require that the shape component of each recursive call is represented in the example set. Such an example set is called *trace complete modulo parametricity* (or *shape complete*). For example, the following set of input-output examples is shape complete:

$$\mathit{tail}\ [A, B, C] \equiv [B, C]\ \wedge\ \mathit{tail}\ [1, 2] \equiv [2]\ \wedge\ \mathit{tail}\ [\mathit{true}] \equiv []$$

To conclude, reasoning about the realizability of a program as a fold is complete if F and G are small container functors and the set of input-output examples is shape complete.

4.3 A Note on Scoping

In the previous sections, we have shown how to rephrase the question of whether a function is a fold as a realizability problem. We have been very careful to state that a fold can be defined as *foldr f e*, and not just defined “in terms of *foldr*”. As it turns out, any function on lists can be defined in terms of *foldr*! Take, for example, the following implementation for *tail*:

$$\begin{aligned}
&\mathit{tail} :: \forall a.\ [a] \to [a] \\
&\mathit{tail}\ \mathit{xs} = \mathit{foldr}\ (\lambda x\ r.\ \mathit{tail'}\ \mathit{xs})\ []\ \mathit{xs} \\
&\quad \textbf{where}\ \mathit{tail'}\ [] \qquad\quad = [] \\
&\qquad\qquad\ \mathit{tail'}\ (y : \mathit{ys}) = \mathit{ys}
\end{aligned}$$

More generally, any function *h* on lists for which an implementation *h’* exists can be implemented in terms of *foldr* in at least two different ways:

$$h\ \mathit{xs} = \mathit{foldr}\ (\lambda x\ r.\ r)\ (h'\ \mathit{xs})\ \mathit{xs} \tag{5}$$

$$h\ \mathit{xs} = \mathit{foldr}\ (\lambda x\ r.\ h'\ \mathit{xs})\ (h'\ [])\ \mathit{xs} \tag{6}$$

The problem is that the algebra of a fold (represented in the function *foldr* by the first two arguments) should be independent of the input list. Hence, to figure out if a function is a fold, we should make sure that *xs* is not in scope of the arguments *f* and *e* when computing the realizability of *foldr f e xs*.

5 Evaluation

To test how well our technique is able to check for realizability, we will evaluate it on a benchmark of polymorphic Haskell functions, checking for each whether it can be implemented as a fold. To create this benchmark, we selected all functions from the Haskell prelude[4] that are natural transformations of type $\forall a.\ H\, a \times [F\, a] \to G\, a$, i.e. any total polymorphic function taking at least a list as input. We made small changes to some functions to make them fit these criteria:

- All functions with a `Foldable` constraint are specialized to `[]` (`concat`, `null`, `length`).
- Partial functions that return a list return the empty list for invalid inputs (`tail`, `init`).
- Other partial functions have their return type wrapped in `Maybe` (`head`, `last`, `index`).
- Since there are two ways to make the function `(++)` fit this scheme (i.e. we can try to fold over either of its input lists), we made both ways explicit in `append` and `prepend`.
- Functions with multiple type parameters are specialized to the same type parameter (`zip`, `unzip`).

The functions are listed in Table 1. For each function *p* in the benchmark, we check if the sketch $\exists f.\ p\ (x, \mathit{ys}) = \mathit{foldr}\ (f\ x)\ (e\ x)\ \mathit{ys}$ is realizable:

$$\boxed{\forall a.\ p : H\, a \times [F\, a] \to G\, a} \wedge \boxed{\exists f.\ p\ (x, \mathit{ys}) = \mathit{foldr}\ (f\ x)\ (e\ x)\ \mathit{ys}} \wedge \boxed{\cdots}$$

Note how we use an extended version of the pipeline in Figure 5 that takes an additional argument of type *H a*. This argument is passed to the function *f* and allows us to check the realizability of functions taking additional arguments (beyond an input list). While there is no technical limitation that stops us from reasoning about multiple holes, for simplicity we assume that the base case *e x* is given. As such, *e* is not existentially quantified.

[4] https://hackage.haskell.org/package/base-4.19.1.0/docs/Prelude.html

Table 1. Benchmark computing the realizability of Haskell prelude functions as folds. For each function, its type is given and whether it is a fold. The running time with shape complete (SC) and shape incomplete (SI) example sets is shown in milliseconds.

name	::	type	fold?	SC	SI
null	::	[a] -> Bool	✓	106	98
length	::	[a] -> Int	✓	110	99
head	::	[a] -> Maybe a	✓	123	107
last	::	[a] -> Maybe a	✓	121	107
tail	::	[a] -> [a]	✗	120	130
init	::	[a] -> [a]	✗	117	123
reverse	::	[a] -> [a]	✓	157	144
index	::	Int -> [a] -> Maybe a	✗	131	132
drop	::	Int -> [a] -> [a]	✗	140	149
take	::	Int -> [a] -> [a]	✓	197	217
splitAt	::	Int -> [a] -> ([a], [a])	✓	566	545
append	::	[a] -> [a] -> [a]	✓	238	275
prepend	::	[a] -> [a] -> [a]	✓	237	240
zip	::	[a] -> [a] -> [(a, a)]	✓	223†	234†
unzip	::	[(a, a)] -> ([a], [a])	✓	228†	⊥
concat	::	[[a]] -> [a]	✓	247†	318†

5.1 Additional Arguments

The functions `index`, `drop`, `take`, and `splitAt` take an integer as an argument in addition to the input list. There are multiple ways to interpret these functions as natural transformations between container functors. For example, for the function `drop`, we can either

- choose $H\,a = \mathit{Int}$, $F\,a = a$, and $G\,a = [a]$, keeping the integer argument constant throughout the fold;
- or choose $H\,a = ()$, $F\,a = a$, and $G\,a = \mathit{Int} \rightarrow [a]$, allowing a different integer to be passed to recursive calls.

In the first interpretation, `drop` is *not* a fold, but in the second interpretation it is:

```
drop :: [a] -> Int -> [a]
drop = foldr f (const [])
  where f x r i = if i > 0 then r (i - 1) else x : r i
```

We only consider the first interpretation in our benchmark, since the functor $\mathit{Int} \rightarrow [-]$ is *not* a small container, as there are an infinite number of positions for every shape. The same approach is taken for all functions that take an extra argument. This includes `index`, `take`, and `splitAt`, but also `append`, `prepend`, and `zip`. In general, any additional arguments can either be kept constant or not. Unless the argument type has a finite number of values, the resulting container functors are *not* small.

5.2 Input-Output Examples

Each function is tested on a shape complete example set curated by the author (each consisting of 4 to 10 input-output examples), as well as a shape incomplete example set constructed by removing every other example from the shape complete example set.

5.3 Running Times

For each function, we automatically prove its (un)realizability as a fold with shape complete (SC) and shape incomplete (SI) example sets. Each proof is performed 10 times and the average results are shown in Table 1.

Interestingly, apart from `unzip`, there is no significant difference in runtime between proofs with shape complete and shape incomplete example sets. Some proofs perform slightly faster with shape incomplete example sets. This could be explained by a naive choice we made in the implementation, which seems to prevent the solver from effectively making use of the guarantees that shape completeness provides (see Section 6.3 for a more in-depth discussion). As a result, those shape incomplete example sets seem to perform better simply because they have fewer input-output examples, and thus fewer constraints.

With the exception of `unzip` with shape incomplete examples, all automated proofs finish in less than a second. These running times are fast enough for automated program analysis and feedback, but not yet for extensive pruning during synthesis. However, our tool could be used for top-level pruning, where realizability is only checked at the start of a synthesis problem to figure out the right recursion scheme.

5.4 Naive Container Translations

The return type of the functions `zip`, `unzip`, and `splitAt` uses a product. When using a naive translation to container functors using Definition 3, computing the realizability of these functions results in a timeout (marked †). Our tool uses a more efficient translation described in more depth in Section 6.3.

6 Implementation

We have implemented a tool for reasoning about the realizability of polymorphic programs. It includes the pipelines from Figures 4 and 5 and provides the basics for defining more realizability problems. The tool is implemented in Haskell, as it allows us to reason about actual Haskell functions and values, rather than describe an intermediate language to reason about. For translating to SMT-LIB, we use the SBV library,[5] a well-documented library with many features that provides a statically typed API into SMT-LIB. We use Z3 [de Moura and Bjørner 2008] as the underlying solver.

6.1 Encoding Containers

To encode a functor `f`, we have to associate it to its shape and position types. To do so, we define two type families `Shape` and `Position`.

```
type family Shape    (f :: Type -> Type) :: Type
type family Position (f :: Type -> Type) :: Type
```

For example, to associate the list functor `[]` to a shape and position (see Definition 1) we define:

```
newtype Fin = Fin Nat deriving (Eq, Ord, Num)
type instance Shape    [] = Nat
type instance Position [] = Fin
```

Note that Haskell does not support dependent types, so we overapproximate `Fin` as a newtype over natural numbers. Additionally, SMT-LIB has no native support for $\mathbb{N}$ and *Fin*, so we employ another type family `Sym` (for *sym*bolic representation) to associate our Haskell datatypes to types

[5] https://leventerkok.github.io/sbv/

supported by SMT-LIB, along with a type class `Encode` for encoding values. Natively supported types are exactly those for which the constraint `SymVal` holds, which is exported by the SBV library.

```
type family Sym (a :: Type) :: Type
class SymVal (Sym a) => Encode a where
  encode :: a -> SMT a
```

Both `Nat` and `Fin` are represented in the SMT solver as integers:

```
type instance Sym Nat = Integer
type instance Sym Fin = Integer
instance Encode Nat where encode = fromIntegral
instance Encode Fin where encode = fromIntegral
```

Of course, these are overapproximations, which we will resolve by describing how the symbolic representations should be constrained in the SMT solver. A value of type `Nat` is encoded as an integer n constrained by $n \geq 0$ and an associated value of type `Fin` as an integer m constrained by $m \geq 0 \land m < n$, for some natural number n. We refer to `Nat` (and other shape types) as refinement types and to `Fin` (and other position types) as dependent types.

The type class `Ref` describes how the symbolic representation of a refinement type is constrained in terms of a function `refine` returning a symbolic boolean `SBool`. Note that when calling `refine`, we have to disambiguate between multiple refinement types that have the same symbolic representation. To do so, we use visible dependent quantification (denoted `forall a ->`) as introduced by the `RequiredTypeArguments` extension,[6] to avoid having to use proxy arguments or ambiguous types.

```
class Encode a => Ref a where
  refine :: forall a -> Sym a -> SBool
```

By convention, operators that act on symbolic values start with a dot (`.`).

```
instance Ref Nat where
  refine _ n = n .>= 0
```

For dependent types, we additionally define a type family `Arg` to associate them with the argument type they depend on. The type class `Dep` describes, for a dependent type, how its symbolic representation is constrained in terms of its argument.

```
type family Arg (a :: Type) :: Type
class (Encode a, Ref (Arg a)) => Dep a where
  depend :: forall a -> Sym (Arg a) -> Sym a -> SBool

type instance Arg Fin = Nat
instance Dep Fin where
  depend _ n m = m .>= 0 .&& m .< n
```

Using `Ref` and `Dep`, we can define a type dependency between a refinement type `t` and a dependent type `u` by stating that the argument type of `u` is equal to `t`.

```
type Dependent :: Type -> Type -> Constraint
type Dependent t u = (Ref t, Dep u, Arg u ~ t)
```

The type class `Container` describes an isomorphism between a container functor `f` and its extension, giving a dependency between its shape and position types.

```
class Dependent (Shape f) (Position f) => Container f where
  toExtension   :: f a -> Extension f a
```

[6] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/required_type_arguments.html

```
    fromExtension :: Extension f a -> f a
```

As described in Section 2, the extension of a container is a shape along with a function mapping positions to elements. In our implementation, we choose to represent this function as a dictionary, to emphasize that we are dealing with *small* containers.

```
  data Extension f a = Extension
    { shape    :: Shape f
    , position :: Map (Position f) a
    }
```

Finally, we define `Container []` by giving a translation to and from the container extension.

```
  instance Container [] where
    toExtension xs = Extension
      { shape    = genericLength xs
      , position = Map.fromList (zip [0..] xs)
      }
    fromExtension (Extension _ p) = Map.elems p
```

6.2 Encoding the Pipelines

For each of the pipelines in Figures 4 and 5, we introduce a function that takes a list of input-output examples and produces a set of constraints to be passed to the SMT solver. All inputs and outputs are translated to their container representation. For each existentially quantified variable in the final equation of the pipeline, an uninterpreted function is introduced. These uninterpreted functions are then constrained as in Equations 1 and 4, making sure to insert calls to `refine` and `depend` so that the functions are only constrained on their true domain. For example, a symbolic function of type $Int \rightarrow Int$ representing a shape morphism $\hat{f} : \mathbb{N} \rightarrow \mathbb{N}$ should not be constrained on negative inputs, and all its results should be nonnegative.

6.3 Efficient Encodings of Position Sets

In Section 4.2 we describe how a shape complete example set leads to a quantifier-free formula because the quantified positions can be enumerated over. In our implementation, however, we have not made proper use of this observation. Rather, we simply describe how the shapes constrain the position types and leave it to the SMT solver to figure out that these constraints imply a finite set of positions. This approach turns out to be too naive, especially for position types that contain unions.[7]

For example, the function `splitAt` returns a pair of lists (i.e. it is a natural transformation to the functor `Product [] []`). Using the standard constructors for containers (as taken from Abbott et al. [2005]), this corresponds to the container

$$((n, m) : \mathbb{N} \times \mathbb{N}) \triangleright Fin\ n + Fin\ m$$

The position type contains a union, and using this container to compute the realizability of `splitAt` in terms of `foldr` results in a timeout. However, an equivalent, more efficient container exists, namely the container

$$((n, m) : \mathbb{N} \times \mathbb{N}) \triangleright Fin\ (n + m)$$

By using this optimized container representation, our tool can efficiently compute the realizability of `splitAt` in terms of `foldr`. Similarly, a list of pairs (the return type of `zip` and `unzip`) can efficiently be represented by the container $(n : \mathbb{N}) \triangleright Fin\ 2n$.

[7] Both product and sum containers use unions in their position types.

In our tool, these efficient containers are defined ad-hoc, but they pave the way for a more general solution: every small container can be represented as $(s : S) \triangleright \mathit{Fin}\ (f\ s)$, where f is some function that returns the number of positions given a shape s of type S. The problem is that f may be difficult to turn into a constraint that is efficiently solved by an SMT solver. When the example set is shape complete, however, all shapes are known, making it possible to compute $f\ s$ before translating to SMT-LIB. This way, both shapes and positions could be represented in SMT-LIB as integers. We leave the exploration of this approach to future work.

7 Related Work

7.1 Typed-Directed Program Synthesis

Many program synthesizers use types to constrain the search space [Feser et al. 2015; Frankle et al. 2016; Gissurarson et al. 2023; Inala et al. 2015; Katayama 2008; Koppel et al. 2022; Lee and Cho 2023; Lubin et al. 2020; Mulleners et al. 2023; Osera 2019; Osera and Zdancewic 2015; Polikarpova et al. 2016]. By inferring synthesis rules from typing rules, synthesized programs are type correct by construction [Inala et al. 2015; Osera and Zdancewic 2015]. Polymorphic types additionally constrain the search space, by restricting the production forms to those that make no assumptions about the types, allowing synthesizers to implicitly benefit from the abstractions provided by parametricity [Reynolds 1983; Wadler 1989]. This is emphasized when polymorphic types are combined with other specifications: often, parametricity can turn an otherwise partial specification into a total specification [Frankle et al. 2016; Osera 2019; Polikarpova et al. 2016]. However, none of these synthesizers explicitly take the interaction between polymorphic types and other specifications into account.

7.2 Reasoning About Polymorphic Functions

Parametrically polymorphic functions behave the same regardless of how they are instantiated. This property is known as *parametricity*. Reynolds [1983] captures parametricity in his abstraction theorem, by giving a relational interpretation to types. Wadler [1989] shows how this interpretation can be used to derive free theorems for functions based solely on their types. Abbott et al. [2003, 2005] define the notion of *containers* and show that any polymorphic function between strictly positive types corresponds to a morphism between containers. Prince et al. [2008] use morphisms between containers to prove properties about polymorphic functions. Seidel and Voigtländer [2010] extend this reasoning to ad-hoc polymorphic functions. Bernardy et al. [2010] show how properties of polymorphic functions can be faithfully tested on a single monomorphic instance. However, these techniques only reason about the correctness of complete programs, rather than the realizability of incomplete programs.

7.3 Example Propagation

Whereas input-output examples are commonly used to test the correctness of complete programs, they can often also be used to reason about the realizability of incomplete programs, a process known as *example propagation*. This is of particular interest to top-down enumerative synthesizers, for pruning the search space. Example propagation is typically implemented using specific deduction rules [Feser et al. 2015; Frankle et al. 2016; Hofmann and Kitzelmann 2010; Osera 2019; Osera and Zdancewic 2015], live evaluation [Lubin et al. 2020; Mulleners et al. 2023; Omar et al. 2019], or function inversion [Lee and Cho 2023; Teegen et al. 2021]. In this paper, example propagation for *map* and *foldr* is defined in an ad-hoc manner, not unlike the deduction rules used by Feser et al. [2015].

7.4 Unrealizability

In recent years, a lot of advancements have been made in reasoning about the unrealizability of programs in the context of syntax-guided synthesis (SyGuS) [Hu et al. 2019, 2020; Hu and D'Antoni 2018] and semantics-guided synthesis (SemGuS) [Kim et al. 2023, 2021]. Typically, these techniques prove that a specification is unrealizable when restricted to a specific grammar. This allows synthesizers to incrementally extend the grammar until a minimal solution (in terms of program size) is found. A similar approach could be possible using unrealizability reasoning as described in this paper, where a synthesizer checks the realizability of a specification against a hierarchy of increasingly expressive recursion schemes.

A different approach to synthesis using unrealizability is taken by Farzan et al. [2022]. Their tool Synduce extends counterexample-guided inductive synthesis (CEGIS) with a dual inductive procedure for generating unrealizability witnesses.

8 Conclusion

We have shown how to automatically compute the realizability of polymorphic functions with input-output examples by interpreting those functions as container morphisms. This approach goes hand in hand with example propagation, a technique to compute the realizability of sketches with input-output examples. We have presented a general schema for computing the realizability of polymorphic functions with input-output examples and sketches, in the form a pipeline that combines type checking, example propagation, translation to containers, and SMT solving. Two concrete instantiations for this pipeline are presented for reasoning about programs defined using *map* and *foldr*. To support recursion schemes such as *foldr*, we have extended our technique to reason not just about input-output examples, but input-output traces. Additionally, we have introduced the notion of shape completeness, a generalization of trace completeness [Osera and Zdancewic 2015] that takes parametricity into account, which is required to show that reasoning about folds is decidable.

8.1 Future Work

The examples in this paper focus primarily on functions acting on lists. It would, however, be interesting to explore more complex datatypes, such as binary trees. Our current technique is restricted to polymorphic functions between strictly positive, unary functors, as this allows an easy translation of inputs and outputs to the container setting as described by Abbott et al. [2005]. We can relax these restrictions by extending the theory of containers. For example, switching from unary to n-ary containers would enable reasoning about multiple type variables. Supporting ad-hoc polymorphism and functors that are not strictly positive would require more complicated extensions to containers, such as described by Seidel and Voigtländer [2010] and Bernardy et al. [2010]. Alternatively, future work could explore the possibility of expressing the realizability of polymorphic programs directly in terms of relational parametricity [Reynolds 1983], rather than relying on the abstraction of container functors.

Our current approach to sketching is rather ad-hoc, describing only how to reason about examples propagated through *map* and *foldr*. While it is possible to add support for more such combinators, it would be interesting to take a more general approach to sketching, such as live bidirectional evaluation [Lubin et al. 2020] or automated function inversion [Teegen et al. 2021].

Another avenue for future work is to explore the different applications of realizability reasoning, including automated program analysis in IDEs, automated feedback in programming tutors, and pruning in program synthesis.

Data-Availability Statement

All code is available on Zenodo [Mulleners 2024] for reproduction and on GitHub[8] for reuse.

Acknowledgments

We would like to thank Koen Claessen for helping work out the initial idea; Wouter Swierstra for suggesting the connection to container functors; the members of the ST4LT and ST groups for their support and discussions; and anonymous reviewers for many helpful suggestions.

References

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2003. Categories of Containers. In *Proceedings of the 6th International Conference on Foundations of Software Science and Computation Structures and Joint European Conference on Theory and Practice of Software* (Warsaw, Poland) *(FOSSACS'03/ETAPS'03)*. 23–38. https://doi.org/10.1007/3-540-36576-1_2

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing Strictly Positive Types. *Theor. Comput. Sci.* 342, 1 (sep 2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002

Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. 2010. Testing Polymorphic Properties. In *Programming Languages and Systems*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–144. https://doi.org/10.1007/978-3-642-11957-6_8

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion synthesis with unrealizability witnesses. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 244–259. https://doi.org/10.1145/3519939.3523726

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (Aug. 2015), 229–239. https://doi.org/10.1145/2813885.2737977

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-Directed Synthesis: A Type-Theoretic Interpretation. *SIGPLAN Not.* 51, 1 (jan 2016), 802–815. https://doi.org/10.1145/2914770.2837629

GHC Team. 2023. *GHC 9.8.1 User's Guide*. https://downloads.haskell.org/~ghc/9.8.1/docs/users_guide/exts/typed_holes.html

Jeremy Gibbons, Graham Hutton, and Thorsten Altenkirch. 2001. When is a function a fold or an unfold? *Electronic Notes in Theoretical Computer Science* 44, 1 (2001), 146–160. https://doi.org/10.1016/S1571-0661(04)80906-X CMCS 2001, Coalgebraic Methods in Computer Science (a Satellite Event of ETAPS 2001).

Matthías Páll Gissurarson. 2018. Suggesting valid hole fits for typed-holes (experience report). *SIGPLAN Not.* 53, 7 (sep 2018), 179–185. https://doi.org/10.1145/3299711.3242760

Matthías Páll Gissurarson, Diego Roque, and James Koppel. 2023. Spectacular: Finding Laws from 25 Trillion Terms. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*. IEEE, 293–304. https://doi.org/10.1109/ICST57152.2023.00035

Martin Hofmann. 2010. Data-Driven Detection of Catamorphisms - Towards Problem Specific Use of Program Schemes for Inductive Program Synthesis. In *Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages (IFL 2010) ; Alphen aan den Rijn, 1. - 3. Sept. 2010*, Jurriaan Hage (Ed.). Utrecht, 25 – 39. https://fis.uni-bamberg.de/handle/uniba/3953.

Martin Hofmann and Emanuel Kitzelmann. 2010. I/O Guided Detection of List Catamorphisms: Towards Problem Specific Use of Program Templates in IP. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Madrid, Spain) *(PEPM '10)*. Association for Computing Machinery, New York, NY, USA, 93–100. https://doi.org/10.1145/1706356.1706375

Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. 2019. Proving Unrealizability for Syntax-Guided Synthesis. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 335–352. https://doi.org/10.1007/978-3-030-25540-4_18

Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. 2020. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming*

[8] https://github.com/NiekM/parametrickery.haskell

Language Design and Implementation (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1128–1142. https://doi.org/10.1145/3385412.3385979

Qinheping Hu and Loris D'Antoni. 2018. Syntax-Guided Synthesis with Quantitative Syntactic Objectives. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 386–403. https://doi.org/10.1007/978-3-319-96145-3_21

Jeevana Priya Inala, Xiaokang Qiu, Benjamin S. Lerner, and Armando Solar-Lezama. 2015. Type Assisted Synthesis of Recursive Transformers on Algebraic Data Types. *CoRR* abs/1507.05527 (2015). arXiv:1507.05527 http://arxiv.org/abs/1507.05527

Susumu Katayama. 2008. Efficient Exhaustive Generation of Functional Programs Using Monte-Carlo Search with Iterative Deepening. In *PRICAI 2008: Trends in Artificial Intelligence*, Tu-Bao Ho and Zhi-Hua Zhou (Eds.). Springer Berlin Heidelberg, 199–210. https://doi.org/10.1007/978-3-540-89197-0_21

Jinwoo Kim, Loris D'Antoni, and Thomas Reps. 2023. Unrealizability Logic. *Proc. ACM Program. Lang.* 7, POPL, Article 23 (jan 2023), 30 pages. https://doi.org/10.1145/3571216

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. 2021. Semantics-guided synthesis. *Proc. ACM Program. Lang.* 5, POPL, Article 30 (jan 2021), 32 pages. https://doi.org/10.1145/3434311

James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. 2022. Searching Entangled Program Spaces. *Proc. ACM Program. Lang.* 6, ICFP, Article 91 (Aug 2022), 29 pages. https://doi.org/10.1145/3547622

Woosuk Lee and Hangyeol Cho. 2023. Inductive Synthesis of Structurally Recursive Functional Programs from Non-Recursive Expressions. *Proc. ACM Program. Lang.* 7, POPL, Article 70 (jan 2023), 31 pages. https://doi.org/10.1145/3571263

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proc. ACM Program. Lang.* 4, ICFP, Article 109 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408991

Niek Mulleners. 2024. Reproduction Package for the ICFP 2024 Article 'Example-Based Reasoning about the Realizability of Polymorphic Programs'. Zenodo. https://doi.org/10.5281/zenodo.11470781

Niek Mulleners, Johan Jeuring, and Bastiaan Heeren. 2023. Program Synthesis Using Example Propagation. In *Practical Aspects of Declarative Languages: 25th International Symposium, PADL 2023, Boston, MA, USA, January 16–17, 2023, Proceedings* (Boston , MA, USA). Springer-Verlag, Berlin, Heidelberg, 20–36. https://doi.org/10.1007/978-3-031-24841-2_2

Ulf Norell. 2009. *Dependently Typed Programming in Agda*. Springer Berlin Heidelberg, Berlin, Heidelberg, 230–266. https://doi.org/10.1007/978-3-642-04652-0_5

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290327

Peter-Michael Osera. 2019. Constraint-Based Type-Directed Program Synthesis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development* (Berlin, Germany) *(TyDe 2019)*. Association for Computing Machinery, New York, NY, USA, 64–76. https://doi.org/10.1145/3331554.3342608

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

Rawle Prince, Neil Ghani, and Conor McBride. 2008. Proving Properties about Lists Using Containers. In *Proceedings of the 9th International Conference on Functional and Logic Programming* (Ise, Japan) *(FLOPS'08)*. Springer-Verlag, Berlin, Heidelberg, 97–112. https://doi.org/10.1007/978-3-540-78969-7_9

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*.

Daniel Seidel and Janis Voigtländer. 2010. Proving Properties about Functions on Lists Involving Element Tests. In *Proceedings of the 20th International Conference on Recent Trends in Algebraic Development Techniques* (Etelsen, Germany) *(WADT'10)*. Springer-Verlag, Berlin, Heidelberg, 270–286. https://doi.org/10.1007/978-3-642-28412-0_17

Finn Teegen, Kai-Oliver Prott, and Niels Bunkenburg. 2021. Haskell^{-1}: Automatic Function Inversion in Haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell* (Virtual, Republic of Korea) *(Haskell 2021)*. Association for Computing Machinery, New York, NY, USA, 41–55. https://doi.org/10.1145/3471874.3472982

Pawel Urzyczyn. 1997. Inhabitation in typed lambda-calculi (a syntactic approach). In *Typed Lambda Calculi and Applications*, Philippe de Groote and J. Roger Hindley (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 373–389. https://doi.org/10.1007/3-540-62688-3_47

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (Imperial College, London, United Kingdom) *(FPCA '89)*. Association for Computing Machinery, New York, NY, USA, 347–359. https://doi.org/10.1145/99370.99404

Received 2024-02-28; accepted 2024-06-18