# Hole Refinements for Polymorphic Type-and-Example Driven Synthesis

Niek Mulleners
Utrecht University
Utrecht, Netherlands
n.mulleners@uu.nl

Johan Jeuring
Utrecht University
Utrecht, Netherlands
j.t.jeuring@uu.nl

Wouter Swierstra
Utrecht University
Utrecht, Netherlands
w.s.swierstra@uu.nl

## Abstract

Many synthesizers implicitly benefit from using polymorphic types, since parametric polymorphism reduces the search space. Additional synthesis constraints may interfere with *parametricity*. In particular, a polymorphic type may cause otherwise feasible input-output examples to contradict each other. We present Taxi (**t**ype-**a**nd-e**x**ample based **i**nferencer), a tool for efficiently reasoning about the feasibility of polymorphic programs specified by input-output examples, and Driver, a tactic language for top-down program synthesis that uses feasibility reasoning to prune the search space. Taxi guarantees that every search state corresponds to a correct (albeit possibly partial) implementation. In addition, it allows for shortcutting the synthesis when a subspecification covers all cases. We show that these techniques have the potential to speed up top-down enumerative type-and-example driven synthesizers.

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

***Keywords:*** parametricity, container functors, feasibility, program synthesis, example propagation

**ACM Reference Format:**
Niek Mulleners, Johan Jeuring, and Wouter Swierstra. 2026. Hole Refinements for Polymorphic Type-and-Example Driven Synthesis. In *Proceedings of the 2026 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM '26), January 11–17, 2026, Rennes, France*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3779209.3779535

## 1 Introduction

In statically typed functional languages, programmers often rely on the compiler as an assistant, both to *check* their progress and to *guide* them along [Lubin and Chasins 2021]. Program sketching [Solar-Lezama 2009], or programming with holes, allows the programmer to be explicit about which parts of the program are unfinished. The compiler can then check that the program is type correct as well as infer the types of the holes, which can be used to guide the programmer further, for example by giving suggestions of valid next steps [Gissurarson 2018].

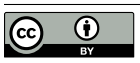
Figure 1 shows some examples of possible suggestions to refine a hole ① of type $[a] \to [a]$. Following the suggestions ensures that the program remains type-correct. However, there are no guarantees that the resulting program is semantically correct. In fact, many of the suggestions are not very sensible at all: due to the polymorphic type, map ② can only ever be refined to map id; const ② to const Nil; and filter ② to either filter (const True) or filter (const False).[1] Most likely, none of these refinements are what the programmer intended. If only the programmer could express their intent to the compiler, we would be able to give more appropriate suggestions. Input-output examples are arguably the easiest and most straight-forward way to do so. For example, the programmer might intend to write a function removing the first element from a list. With just a few input-output examples and some elbow grease, all suggestions but pattern matching can be discarded.

Once only valid hole refinements are allowed, the programmer may repeatedly ask for more refinements, to the point where the program practically writes itself. In this process of stepwise refinement, the program space is narrowed down, while ensuring that the programs within never contradict the type or the input-output examples.

In this paper, we explore how to automatically check whether a hole refinement is valid with respect to a type and input-output examples, and how this technique can benefit program synthesis. To do so, we first define Taxi, a tool for reasoning about the feasibility of programs specified by a polymorphic type and a set of input-output examples. Next, we define Driver, a program synthesizer aimed at customizability (through a language of tactics) and correctness (by employing feasibility reasoning to ensure only correct refinements are introduced).

***Contributions.*** In this paper, we extend our previous work on reasoning about the feasibility of polymorphic programs with monomorphic input-output examples [Mulleners et al. 2024].

[1] We assume here the programmer is only interested in a total implementation.




*PEPM '26, Rennes, France*

ACM ISBN 979-8-4007-2357-5/2026/01
https://doi.org/10.1145/3779209.3779535

```
skip : ∀a. [a] → [a]
skip = ①

① ⊑ reverse
① ⊑ map ②
① ⊑ filter ②
① ⊑ const ②
① ⊑ foldr ② ③
① ⊑ λxs. case xs of
       Nil → ②
       Cons y ys → ③
```

**Figure 1.** Possible suggestions of hole refinements based on the type of a hole.

In particular, we

- define a general form for polymorphic examples (Section 3);
- show how partial programs can be extracted from these specifications (Section 3.6);
- define a notion of coverage checking to check the exhaustiveness of polymorphic examples (Section 3.7);
- provide support for algebraic datatypes (Section 3.3), as well as limited support for ad hoc polymorphism (Section 3.8); and
- implement these techniques in a tool called Taxi and show that it outperforms prior work.

In addition, to show the potential of Taxi in program synthesis, we implement Driver, a customizable, tactics based synthesizer that employs the aforementioned techniques to prune the search space while ensuring that every state corresponds to a partial implementation (Section 4).

## 2 Motivating Examples

In this section we highlight how the techniques described in this paper can be used to reason about the validity of hole refinements and to prune the search space during program synthesis.

### 2.1 Valid Hole Refinements

Suppose the programmer intends to implement the skip function, which removes the first element from a list (if it has any). Based on the type alone, the compiler may give the suggested refinements shown in Figure 1. Uncertain about which of these suggestions is the correct one, the programmer may write some input-output examples:

$$\begin{aligned}&\mathsf{skip}\ [1,2,3] \equiv [2,3]\\ &\mathsf{skip}\ [4,5] \equiv [5]\end{aligned}$$

We will show how, using just these input-output examples, most of the suggestions can be discarded.

For refinements that do not introduce holes, such as reverse, we can evaluate the expression against the input-output examples. The call reverse $[1,2,3]$ evaluates to $[3,2,1]$, rather than the expected result $[2,3]$. Thus reverse is not a valid refinement.

For refinements introducing holes, the correctness relies on whether the holes can still be filled to get the correct result. In the case of map, the first input-output example states that $\exists f.\ \mathsf{map}\ f\ [1,2,3] \equiv [2,3]$. Using equational reasoning, we show that this is incorrect, since map preserves the length of the input list.

$$\mathsf{map}\ f\ [1,2,3] \equiv [f\ 1, f\ 2, f\ 3] \not\equiv [2,3]$$

We cannot get the expected output, regardless of how $f$ is instantiated, so map is not a valid refinement.

For filter, we get $\exists p.\ \mathsf{filter}\ p\ [1,2,3] \equiv [2,3]$. From this we can derive the following constraint on $p$:

$$p\ 1 \equiv \mathsf{False} \ \wedge\ p\ 2 \equiv \mathsf{True} \ \wedge\ p\ 3 \equiv \mathsf{True}$$

It may seem that this constraint is satisfiable. However, we can use *parametricity* [Reynolds 1983; Wadler 1989] to show that it is not. Simply put, parametricity states that we cannot *inspect* or *produce* values of a polymorphic type $a$, only pass them around. Since the type of $p$ is $a \to \mathsf{Bool}$, we cannot distinguish between the inputs 1, 2, and 3. This means that the predicate $p$ cannot return different outputs, hence filter is *not* a valid refinement.

The function const is also not a valid refinement, since it throws away the input, which is relevant for computing the output. We can once again use equational reasoning and parametricity. From the equation $\exists c.\ \mathsf{const}\ c\ [1,2,3] \equiv [2,3]$, we derive that $c$ is constrained to the constant $[2,3]$. Parametricity states, however, that we cannot produce 2 or 3, since $c$ has type $[a]$.

Reasoning about recursion schemes, such as foldr, is a bit more involved. Given the polymorphic type of skip, the input-output example $\mathsf{skip}\ [4,5] \equiv [5]$ implies $\mathsf{skip}\ [2,3] \equiv [3]$. This leads to the following equation:

$$\exists f\ e.\ \mathsf{foldr}\ f\ e\ [1,2,3] \equiv [2,3] \ \wedge\ \mathsf{foldr}\ f\ e\ [2,3] \equiv [3]$$

We can see how these two input-output examples are related by expanding the call to $\mathsf{foldr}\ f\ e\ [1,2,3]$ once:

$$f\ 1\ (\mathsf{foldr}\ f\ e\ [2,3]) \equiv [2,3]$$

Using $\mathsf{foldr}\ f\ e\ [2,3] \equiv [3]$, we end up with the input-output example $f\ 1\ [3] \equiv [2,3]$. Since $f$ has type $[a] \to [a]$, this example is contradictory. As such, foldr is *not* a valid refinement.

The last suggested refinement pattern matches on the input. This refinement is always valid, regardless of the input-output examples.

After checking each possible refinement from Figure 1 for validity, we can conclude that only the last suggestion should be given. If the programmer accepts the suggestion to pattern match on the input, suggestions for the remaining holes lead to the following correct implementation:

$$\mathsf{skip}\ xs = \mathbf{case}\ xs\ \mathbf{of}\ \mathsf{Nil} \to \mathsf{Nil}\ ;\ \mathsf{Cons}\ y\ ys \to ys$$

With the help of the suggestions, the program practically wrote itself. In the next section, we take this a step further, showing how the synthesis of a program can be automated by performing repeated refinements.

### 2.2 Program Synthesis

By repeatedly applying valid refinements, we end up with a program that is correct by construction. For example, consider the function nub, which removes all duplicates from a list, keeping only the first occurrences. We can specify nub using the following type signature and input-output examples.

$$\begin{array}{l}\mathsf{nub} : \forall a.\ \mathsf{Eq}\ a \Rightarrow [a] \to [a] \\ \mathsf{nub}\ xs = ①\end{array} \qquad ① \vDash \begin{cases} xs & & \\ [] & \mapsto & [] \\ [1] & \mapsto & [1] \\ [1,1] & \mapsto & [1] \\ [2,1] & \mapsto & [2,1] \\ [1,1,1] & \mapsto & [1] \\ [1,2,1] & \mapsto & [1,2] \end{cases}$$

The synthesis tactic described in Section 4 will find the following hole refinements, leading to the desired solution.

$$\begin{array}{l} ① \sqsubseteq \mathsf{foldr}\ (\lambda x\ r.\ ②)\ ③\ xs \\ ② \sqsubseteq \mathsf{Cons}\ ④\ ⑤ \\ ③ \sqsubseteq \mathsf{Nil} \\ ④ \sqsubseteq x \\ ⑤ \sqsubseteq \mathsf{filter}\ (\lambda y.\ ⑥)\ r \\ ⑥ \sqsubseteq x \neq y \end{array}$$

① We try map and filter *before* trying foldr, as described in Section 4.5. Since map and filter fail, the first valid refinement for hole ① uses foldr. This results in the following constraints on the holes.

$$② \vDash \begin{cases} x & r & & \\ 1 & [] & \mapsto & [1] \\ 1 & [1] & \mapsto & [1] \\ 2 & [1] & \mapsto & [2,1] \\ 1 & [2,1] & \mapsto & [1,2] \end{cases} \qquad ③ \vDash []$$

② All examples have an output list with at least one element. So we can introduce the Cons constructor. This has precedence over introducing functions.

$$④ \vDash \begin{cases} x & r & & \\ 1 & [] & \mapsto & 1 \\ 1 & [1] & \mapsto & 1 \\ 2 & [1] & \mapsto & 2 \\ 1 & [2,1] & \mapsto & 1 \end{cases} \qquad ⑤ \vDash \begin{cases} x & r & & \\ 1 & [] & \mapsto & [] \\ 1 & [1] & \mapsto & [] \\ 2 & [1] & \mapsto & [1] \\ 1 & [2,1] & \mapsto & [2] \end{cases}$$

③ This hole is constrained to the constant Nil, so no need to try other refinements.

④ Before trying any other tactics, we check if a variable in scope matches the examples. In this case $x$ is the right value.

⑤ The refinement map fails, but filter succeeds. Since filter can be defined in terms of foldr, we do not try a refinement with foldr.

$$⑥ \vDash \begin{cases} x & y & & \\ 1 & 2 & \mapsto & \mathsf{True} \\ 1 & 1 & \mapsto & \mathsf{False} \end{cases}$$

⑥ At this point, the input-output examples along with polymorphic type cover all possible cases. We can recognize this automatically by means of example coverage checking (Section 3.7). Further refinements can no longer influence the semantics of the program, so we can use program extraction to fill hole ⑥ (Section 3.6).

After this final step, the synthesis is finished and the final program follows from the refinements:

$$\mathsf{nub}\ xs = \mathsf{foldr}\ (\lambda x\ r.\ \mathsf{Cons}\ x\ (\mathsf{filter}\ (\lambda y.\ x \neq y)\ r))\ \mathsf{Nil}\ xs$$

Not only is this the first solution found by our synthesizer, but every refinement step is the first valid refinement considered for its hole. This means that no backtracking or unnecessary branching occurred during the synthesis of this program. Note, however, that this particular example works especially well with the synth tactic defined in Section 4. In practice, the user may find it necessary to write their own tactic to achieve the best results.

## 3 Polymorphic Examples

It may seem that a single input-output example, such as $f\ [1,2,3] \equiv [2,3]$, only restricts $f$ on a single input. However, when the constrained function has a polymorphic type, such as $f : \forall a.\ [a] \to [a]$, parametricity constrains many more inputs based on this single input-output example. In particular, $f$ is restricted on every input list of length three, regardless of the type and values of the list elements. In a way, our single input-output actually represents a set of input-outputs, described by the following equation:

$$\forall x\ y\ z.\ f\ [x,y,z] \equiv [y,z]$$

We refer to such an equation as a *polymorphic example*, and to the original input-output example as a *monomorphic example*. Informally, to compute a polymorphic example, we traverse the elements in the monomorphic example, assigning a unique variable to each. Since polymorphic functions work for any input, we universally quantify over the input elements. The output elements of a polymorphic function should come from the input, so we constrain them to the inputs that have the same value. When the input contains duplicate elements, outputs can be chosen arbitrarily among them in the corresponding polymorphic example. For example, for the monomorphic example $f\ [1,1] \equiv 1$:

$$\forall x\ y.\ \exists z \in \{x,y\}.\ f\ [x,y] \equiv [z]$$

Sometimes, an input-output example and a polymorphic type contradict each other. For example, the monomorphic example $f\ [1,2] \equiv [2,3]$ contradicts the type $\forall a.\ [a] \to [a]$, since

the output 3 does not occur in the input. This contradiction is visible in the corresponding polymorphic example:

$$\forall x\ y.\ \exists z \in \{\}.\ f\ [x, y] \equiv [y, z]$$

In our previous work, we have shown how to automate such reasoning about the feasibility of polymorphic functions with monomorphic input-output examples [Mulleners et al. 2024]. Polymorphic functions are interpreted as morphisms between finitary container functors. Translating input-output examples to the setting of container functors results in more manageable constraints that can be solved by an SMT solver. Note how finitary containers are exactly traversable functors [Jaskelioff and O'Connor 2015], matching our intuition of polymorphic examples being computed by *traversing* the input-output examples.

In this section, we give an overview of our previous technique and extend upon it in several ways in our tool Taxi, allowing for faster reasoning, more transparent feedback, as well as support for algebraic datatypes and limited ad hoc polymorphism.

### 3.1 Container Morphisms

Abbott et al. [2003, 2005] show how some datatypes that *contain* other datatypes can be represented in terms of their *structure* and their *contents*. A *container* $S \triangleright P$ has a shape of type $S$, representing its structure and, for every shape $s : S$, a set of positions of type $P_s$, specifying the locations within that structure where each element is located. For example, the structure of a list is its length (a natural number) and a list of length $n$ has exactly $n$ elements. As such, the list container is defined as $\mathbb{N} \triangleright \mathsf{Fin}$, where $\mathsf{Fin}\ n = \{0, \ldots, n-1\}$, the type containing exactly $n$ elements. The *extension* of a container $S \triangleright P$ (written $[\![S \triangleright P]\!]$) is a functor defined using a dependent pair as $[\![S \triangleright P]\!]\ X \triangleq \Sigma_{(s:S)}\ (P_s \to X)$.

Values of type $[\![S \triangleright P]\!]\ X$ consist of a shape $s : S$ and a function mapping each position $p : P_s$ to the corresponding element of type $X$. For example, the list $[x_0, \ldots, x_{n-1}]$ is represented in the list container extension as $(n, \lambda i.\ x_i)$.

When reversing a list, its elements are reordered from highest to lowest index. This is easily done in the container representation: the reverse of $(n, \lambda i.\ x_i)$ is $(n, \lambda i.\ x_{n-i-1})$. Note how this reversal can be expressed in terms of only the indices of elements, rather than their values. Such functions, that only refer to the indices of elements, are exactly *container morphisms*.

A container morphism between the containers $S \triangleright P$ and $T \triangleright Q$ is a pair $(u, g)$, where $u : S \to T$ is a function from input shapes to output shapes and $g : \Pi_{(s:S)}\ (Q_{(u\ s)} \to P_s)$ is a function mapping, for each input shape, output positions to input positions. In other words, $g$ describes where each element in the output comes from. Container morphisms can be applied using their *extension*:

$$\langle\!\langle u, g \rangle\!\rangle\ :\ \forall a.\ [\![S \triangleright P]\!]\ a \to [\![T \triangleright Q]\!]\ a$$
$$\langle\!\langle u, g \rangle\!\rangle\ \triangleq \lambda(s, p).\ (u\ s, p \circ g_s)$$

We can define list reversal as the extension of a container morphism as $\langle\!\langle (\lambda n.\ n), (\lambda n\ i.\ n - i - 1) \rangle\!\rangle$.

### 3.2 Reasoning about Feasibility

Container morphisms are exactly natural transformations between container functors: if a function $f$ is a natural transformation between container functors, there exists a container morphism $(u, g)$ such that $f \cong \langle\!\langle u, g \rangle\!\rangle$ [Abbott et al. 2005]. In our previous work, we used this insight to reason about the feasibility of input-output examples on polymorphic functions, by translating those examples to the container setting, where they can easily be solved by an SMT solver [Mulleners et al. 2024].

More concretely, we asked the following question: given container functors $F$ and $G$, and a set of input-output pairs, does there exist a function $f\ :\ \forall a.\ F\,a \to G\,a$ that implements the input-output pairs? To answer this question, we then defined the following procedure:

1. find a container $S \triangleright P$ such that $F \cong [\![S \triangleright P]\!]$;
2. find a container $T \triangleright Q$ such that $G \cong [\![T \triangleright Q]\!]$;
3. assert that there exists a container morphism $(u, g)$ such that $f \cong \langle\!\langle u, g \rangle\!\rangle$;
4. for every input-output example $f\ i \equiv o$ instantiating $f$ at type $\tau$
   a. compute $(s, p)\ :\ [\![S \triangleright P]\!]\ \tau$ such that $i \cong (s, p)$;
   b. compute $(t, q)\ :\ [\![T \triangleright Q]\!]\ \tau$ such that $o \cong (t, q)$;
   c. note that $\langle\!\langle u, g \rangle\!\rangle(s, p) \equiv (t, q)$;
   d. check the *shape* component $u\ s \equiv t$;
   e. check the *position* component $p \circ g \equiv q$.

Finding a contradiction in step 4d or 4e shows that no such function $f$ exists, and solutions for $u$ and $g$ correspond to a correct definition of $f$.

***Challenges of SMT-based Feasibility Reasoning.*** In our previous work, to try and find solutions for $u$ and $g$, we used an SMT solver. In steps 1, 2, 4a, and 4b, we used handcrafted translations, translating common functors to their canonical container representations.

The effectiveness of this approach on list functions relies heavily on the fact that the shape and position types of the canonical list container ($\mathbb{N}$ and Fin respectively) can be represented as integers with lower and upper bounds, which SMT solvers are optimized for. In contrast, consider the following binary Tree datatype:

$$\textbf{data}\ \mathsf{Tree}\ a = \mathsf{Node}\ (\mathsf{Tree}\ a)\ a\ (\mathsf{Tree}\ a) \mid \mathsf{Leaf}$$

We can represent it as a container whose shape is a bare tree (without values in the nodes) and its positions are indices in that tree. To encode these in an SMT solver, we have to represent the bare tree as an algebraic datatype and check

that the indices do not fall outside that tree. While this is possible, it leads to a large bump in complexity compared to lists. We recognize the following downsides of SMT-based feasibility reasoning, especially in the context of program synthesis:

**Flexibility** Container translations have to be constructed manually for every datatype.

**Efficiency** Shape and position types are dependent types: to model them in an SMT solver one has to make sure only correct values can be represented. While this is easy for simple types like $\mathbb{N}$ and Fin, more complex datatypes are difficult to represent efficiently.

**Transparency** When the specification is infeasible, there is no explanation as to which constraint failed. When it is feasible, the solver returns arbitrary implementations of $u$ and $g$, which prove feasibility, but cannot easily be translated back into a program in the target language. Additionally, there is no way to inspect or reason about the polymorphic examples.

**Reliability** There are no clear guarantees as to when or whether the SMT solver terminates.

***How We Address These Challenges.*** In this work, we use the same general procedure for reasoning about feasibility, but use a generic container translation and forgo the use of a solver.

For the container translation, we use the observation that $F \cong [\![F\top \triangleright \mathsf{Size}]\!]$, where Size $s$ has exactly one element for each unit in $s : F\top$. In other words, we use a translation where the shape of a functor $F$ is that functor filled with units, and the type of positions has exactly one element for each such unit. By using this generic translation, we do not need specialized container translations for every type, as it works for any strictly positive type.

To solve the constraints in steps 4d and 4e, rather than relying on an SMT solver, we define a terminating procedure for computing a normal form for polymorphic examples. This allows us to make better use of the structure that is present in these constraints, for more efficient, transparent, and reliable reasoning.

### 3.3 A Generic Container Translation

In Figure 2, we describe the syntax of simple values, types, and input-output examples. A datatype $D$ is defined as a (possibly recursive) sum of products. A strictly positive type $\pi$ consists of products, datatypes, and free variables. A value $v$ consists of tuples and constructors.

We translate values to their container representation, a shape $s$ and a position map $p$, using the inference rules in Figure 3. We write $v : \pi \downarrow_n^m s \triangleright p$ to denote that the value $v$ checked against the type $\pi$ has shape $s$ and a position map $p$ that maps the positions $[m, n)$ to values. We may leave out position bounds ($m$ and/or $n$) when they are not relevant. The translation of a value to its container representation

| | | | |
|---|---|---|---|
| **Constructor** | $C$ | $\in$ | *Constructors* |
| **Type Variable** | $a$ | $\in$ | *Variables* |
| **Position** | $n$ | $\in$ | $\mathbb{N}$ |
| **Datatype** | $D$ | $::=$ | $\overline{C\,\pi}$ |
| **Positive Type** | $\pi$ | $::=$ | $\top \mid \pi_1 \times \pi_2 \mid D \mid a$ |
| **Value** | $v$ | $::=$ | $() \mid (v_1, v_2) \mid C\,v$ |
| **Shape** | $s$ | $::=$ | $() \mid (s_1, s_2) \mid C\,s \mid \boxed{n}$ |
| **Position Map** | $p$ | $::=$ | $\{\overline{n \mapsto v : a}\}$ |
| **Origin Map** | $\Omega$ | $::=$ | $\{\overline{n \mapsto \{n_1, \ldots, n_i\}}\}$ |

**Figure 2.** Syntax for type-and-example specifications.

$$\boxed{v : \pi \downarrow_n^m s \triangleright p}$$

$$\frac{}{() : \top \downarrow_n^n () \triangleright -}\ \textsc{unit}$$

$$\frac{v_1 : \pi_1 \downarrow_m^l s_1 \triangleright p_1 \qquad v_2 : \pi_2 \downarrow_n^m s_2 \triangleright p_2}{(v_1, v_2) : (\pi_1 \times \pi_2) \downarrow_n^l (s_1, s_2) \triangleright (p_1 \cup p_2)}\ \textsc{tuple}$$

$$\frac{v : \pi \downarrow_n^m s \triangleright p \qquad C\,\pi \in \mathsf{constructors}(D)}{C\,v : D \downarrow_n^m C\,s \triangleright p}\ \textsc{constructor}$$

$$\frac{}{v : a \downarrow_{n+1}^n \boxed{n} \triangleright \{n \mapsto v : a\}}\ \textsc{free}$$

**Figure 3.** Inference rules for translating values to their container representation.

replaces each value at type $a$ with a uniquely labeled box $\boxed{n}$ and stores the value in the position map at position $n$. Intuitively, this translation makes the separation of *structure* and *content* explicit.

To translate a tuple, both parts are translated separately such that their position bounds are consecutive, ensuring that each position is uniquely and consistently labeled. Their shapes are then tupled and their position maps unioned. The FREE rule is the most interesting: any value $v$ at type $a$ is stored in the position map at position $n$, and the value $v$ is replaced by $\boxed{n}$.[2] Note that this is the only rule that strictly increases the range of positions.

The translation can easily be reverted by substituting the boxes in a shape $s$ with the values found in the position map $p$. We write $\hat{p}(s)$ to substitute the boxes in $s$ for values according to $p$. If $v : \pi \downarrow s \triangleright p$, then $\hat{p}(s) = v$.

[2]Note that there is no ambiguity as to which rule applies; $a$ is a free type variable and distinct from other types.

### 3.4 Computing Polymorphic Examples

In this section, we follow the steps from Section 3.2 to turn monomorphic examples into polymorphic examples and check their feasibility. Consider once again the function $f : \forall a.\ F\ a \to G\ a$. Following steps 1 and 2, we start by picking containers for $F$ and $G$. We assume that $F\ a$ and $G\ a$ are strictly positive, i.e. $F\ a = \pi_1$ and $G\ a = \pi_2$. As such, we can use the generic container translation described in Section 3.3.

For every input-output example $f\ i \equiv o$, following steps 4a and 4b, we translate the input and output using the inference rules in Figure 3:

$$i : \pi_1 \downarrow s \triangleright p \qquad\qquad o : \pi_2 \downarrow t \triangleright q$$

We can then reformulate the input-output example in terms of $s$, $t$, $p$, and $q$:

$$f\ \hat{p}(s) \equiv \hat{q}(t)$$

As we did in the introduction of Section 3, we want to generalize this input-output example to constrain inputs of any type $\tau$, containing any elements. To do so, we have to show how $p$ and $q$ constrain $f$. Together, the position maps $p$ and $q$ define, for every output position in $t$, which input positions in $s$ are valid origins. Concretely, each output position corresponds to a set of input positions, given by the *origin map* $\Omega$:

$$\Omega = \{\ n \mapsto \{\ m \mid (m \mapsto v : a) \in p\ \} \mid (n \mapsto v : a) \in q\ \}$$

To give a general form for polymorphic examples in terms of the triple $\langle s, t, \Omega \rangle$, we generalize the input-output example by quantifying over $p$ and $q$:

$$f \vDash \langle s, t, \Omega \rangle \ \triangleq\ \forall p.\ \exists q.\ f\ \hat{p}(s) \equiv \hat{q}(t) \wedge p \ltimes_\Omega q \qquad (1)$$

where $p \ltimes_\Omega q =$

$$\forall (n \mapsto v : a) \in q.\ \exists (m \mapsto v : a) \in p.\ \exists (n \mapsto o) \in \Omega.\ m \in o$$

***An Example Translation.*** Consider the input-output example $f\ [1, 1] \equiv [1]$ for the function $f : \forall a.\ [a] \to [a]$. After translating, we end up with a polymorphic example $f \vDash \langle s, t, \Omega \rangle$, where

$$s = [\boxed{0}, \boxed{1}], \quad t = [\boxed{0}], \quad \text{and} \quad \Omega = \{\ 0 \mapsto \{0, 1\}\ \}.$$

Equation 1 quantifies over the position maps $p$ and $q$. Since we know the number of positions in $s$ and $t$, we can make $p$ and $q$ explicit. They are of the form $\{\ 0 \mapsto x : a, 1 \mapsto y : a\ \}$ and $\{\ 0 \mapsto z : a\ \}$ respectively, so we can quantify over $x$, $y$, and $z$ instead. In doing so, the relation $p \ltimes_\Omega q$ simplifies to $z \in \{x, y\}$:

$$\forall x\ y.\ \exists z \in \{x, y\}.\ f\ [x, y] \equiv [z]$$

Typically, when the shapes of polymorphic examples are known, we will write them in this form, by quantifying over the elements rather than the position maps.

### 3.5 Feasibility Conflicts

We have seen how to translate input-output examples into polymorphic examples. For a set of examples, we get

$$\overline{f\ i_k \equiv o_k} \qquad \Longrightarrow \qquad \overline{f \vDash \langle s_k, t_k, \Omega_k \rangle}$$

We are interested to know whether a program $f$ satisfying the input-output examples exists, i.e. whether the polymorphic examples are feasible. As shown in our previous work, if polymorphic examples are infeasible, this is due to one of three conflicts [Mulleners et al. 2024]. We can easily check for these conflicts by inspecting the polymorphic examples.

**Magic Output** An example has an output element that does not occur in the input. When this is the case, $\Omega$ is empty at the corresponding output position.

**Shape Conflict** Two examples have the same input shape, but not the same output shape.

**Position Conflict** Two or more examples have the same input and output shape, but they disagree on the origins of the output elements. To check for position conflicts, we merge examples with the same input shape by taking the pointwise intersection of their origin maps:

$$\left.\begin{array}{l} f \vDash \langle s, t, \Omega_1 \rangle \\ \ldots \\ f \vDash \langle s, t, \Omega_n \rangle \end{array}\right\} \quad f \vDash \langle s, t, \bigcap_i \Omega_i \rangle$$

A position conflict occurs exactly if there is a magic output in the merged example.

### 3.6 Program Extraction

After checking and merging the constraints, we end up with a set of polymorphic examples with unique input shapes and origin maps that are non-empty at every position. We can show that a partial program implementing these constraints exists. For every position $n$ in every output shape $t$, we pick an input position from the origin map, denoted $\omega(t, n)$. We can then define a program $f$ by pattern matching on the input shapes. For every triple $\langle s, t, \Omega \rangle$, we create a pattern by filling the positions in the input shape with fresh variables. All other cases are caught by a wildcard pattern, whose right-hand side is simply a hole.

$$\begin{array}{l} f = \lambda\textbf{case} \\ \quad \hat{p}(s) \to \hat{q}(t) \quad \textbf{where} \quad p = \{\overline{m \mapsto x_m}\} \\ \qquad\qquad\qquad\qquad\ \ q = \{\overline{n \mapsto x_{\omega(t,n)}}\} \\ \quad \_ \to \bigcirc \end{array}$$

This implementation is guaranteed to satisfy all the input-output examples, but may be partial, if not all patterns are covered.

### 3.7 Example Coverage Checking

We can perform a basic form of pattern match coverage checking to see which cases are missing from a set of polymorphic examples. Note that we only have to check which input *shapes* are missing, by enumerating all possible shapes

of the input container. We are particularly interested in cases where all shapes are covered, as this means a program found by program extraction is guaranteed to be total. For example:

$$\begin{aligned}&\mathsf{swap} : \forall a\ b.\ \mathsf{Either}\ a\ b \to \mathsf{Either}\ b\ a\\&\mathsf{swap}\ (\mathsf{Left}\ 3) \equiv \mathsf{Right}\ 3\\&\mathsf{swap}\ (\mathsf{Right}\ ()) \equiv \mathsf{Left}\ ()\end{aligned}$$

$$\xrightarrow{\textsc{Extract}} \quad \lambda\mathbf{case}\ \mathsf{Left}\ x \to \mathsf{Right}\ x\,;\ \mathsf{Right}\ x \to \mathsf{Left}\ x$$

### 3.8 Ad Hoc Polymorphism

To reason about ad hoc polymorphic functions, we adopt the approach by Seidel and Voigtländer [2010] for extending containers with ad hoc polymorphism. When the ad hoc polymorphism describes a relation (such as an equality or an ordering), it can be used to indirectly inspect elements by checking if the relation holds between them. Consider, for example, the following specification for the max function:

$$\begin{aligned}&\mathsf{max} : \forall a.\ \mathsf{Ord}\ a \Rightarrow a \to a \to a\\&\mathsf{max}\ 1\ 2 \equiv 2\\&\mathsf{max}\ 1\ 1 \equiv 1\\&\mathsf{max}\ 3\ 2 \equiv 3\end{aligned}$$

We would like to compute conditional polymorphic examples, as follows:

$$\forall x\ y.\ \exists z \in \{x, y\}.\ \begin{cases}\mathsf{max}\ x\ y \equiv y & \textbf{if}\ \ x < y\\ \mathsf{max}\ x\ y \equiv z & \textbf{if}\ \ x \equiv y\\ \mathsf{max}\ x\ y \equiv x & \textbf{if}\ \ x > y\end{cases}$$

We can, however, avoid extending our formalization by reformulating the problem. One way to define an ordering is in terms of the Ordering datatype and the accompanying compare function.

$$\begin{aligned}&\textbf{data}\ \mathsf{Ordering} = \mathsf{LT} \mid \mathsf{EQ} \mid \mathsf{GT}\\&\mathsf{compare} : \forall a.\ \mathsf{Ord}\ a \Rightarrow a \to a \to \mathsf{Ordering}\end{aligned}$$

We can capture the ordering of the inputs using the compare function and pass it to the fully polymorphic helper function max′. The feasibility of max and max′ are equivalent, as they can be defined in terms of each other.

$$\begin{aligned}&\mathsf{max}\ x\ y = \mathsf{max}'\ (\mathsf{compare}\ x\ y)\ x\ y\\&\quad \textbf{where}\ \mathsf{max}' : \forall a.\ \mathsf{Ordering} \to a \to a \to a\end{aligned}$$

We can now compute the following polymorphic examples, showing that max′ (and thus max) is feasible. Better yet, we can use example coverage checking and program extraction to find a total solution for max.

$$\forall x\ y.\ \exists z \in \{x, y\}.\ \begin{cases}\mathsf{max}'\ \mathsf{LT}\ \ x\ y \equiv y\\ \mathsf{max}'\ \mathsf{EQ}\ x\ y \equiv z\\ \mathsf{max}'\ \mathsf{GT}\ x\ y \equiv x\end{cases}$$

Effectively, the Ord constraint is replaced by the Ordering argument. Every function with an Ord constraint can be automatically rewritten in a similar manner, in terms of a helper function that replaces the constraint with an argument that carries the ordering information. We automate this process for dealing with Ord constraints, and do the same for Eq constraints.

## 4 Sketching

We have seen how to use program extraction (Section 3.6) to generate a partial program implementing a feasible specification. Such programs do not, however, generalize beyond the given input-output examples. Instead, we want to introduce common abstractions that generalize beyond the input-output examples in a natural way. To do so, we will use *program sketching*, a technique whereby a program is written incrementally, leaving holes for the parts that are yet to be defined. At each increment, we make sure that the program is still feasible and compute which specification should hold on each hole to guarantee a correct solution.

### 4.1 An Example: delete in terms of filter

Consider the following specification for the delete function, along with a simple sketch.

$$\begin{aligned}&\mathsf{delete} : \forall a.\ \mathsf{Eq}\ a \Rightarrow a \to [a] \to [a]\\&\mathsf{delete}\ 0\ [] \equiv\ []\\&\mathsf{delete}\ 1\ [1] \equiv\ []\\&\mathsf{delete}\ 2\ [2, 2] \equiv\ []\\&\mathsf{delete}\ 3\ [1, 3] \equiv\ [1]\end{aligned} \qquad \mathsf{delete}\ a\ xs = ⓪$$

After confirming that the specification of delete is feasible, we may attempt to implement it using filter, by introducing the following hole filling.

$$⓪ \mapsto \mathsf{filter}\ (\mathsf{predicate}\ a)\ xs \quad \textbf{where}\ \mathsf{predicate}\ x\ y = ①$$

Using type checking and equational reasoning, we can infer that predicate should satisfy the following specification:

$$\begin{aligned}&\mathsf{predicate} : \forall a.\ \mathsf{Eq}\ a \Rightarrow a \to a \to \mathsf{Bool}\\&\mathsf{predicate}\ 1\ 1 \equiv \mathsf{False}\\&\mathsf{predicate}\ 2\ 2 \equiv \mathsf{False}\\&\mathsf{predicate}\ 3\ 1 \equiv \mathsf{True}\end{aligned}$$

We can compute the polymorphic examples as described in Section 3 to show that predicate is feasible.

$$\forall x\ y.\ \begin{cases}\mathsf{predicate}\ x\ y \equiv \mathsf{False} & \textbf{if}\ \ x \equiv y\\ \mathsf{predicate}\ x\ y \equiv \mathsf{True} & \textbf{if}\ \ x \not\equiv y\end{cases}$$

The resulting constraint is not only feasible, but also exhaustive and unique. This means that there is a single, total solution for predicate, which we can find using program extraction. After inlining predicate and normalizing, we end up with the following implementation of delete:

$$\mathsf{delete}\ x\ xs = \mathsf{filter}\ (\lambda y.\ x \not\equiv y)\ xs$$

### 4.2 A Sketching Framework

We define Driver, a domain specific language in Haskell for sketching. The main datatypes are defined in Figure 4.

```
data Sketch hole = ... | Hole hole
data Signature = Signature { args : Args, out : Type }
type Args = [(String, Type)]
data Example = Example { args : [Value], out : Value }
data Spec = Spec { sig : Signature, exs : [Example] }
```

**Figure 4.** The datatypes used in sketching.

A specification consists of a type signature and a list of input-output examples. A sketch is an expression from the lambda-calculus extended with data (de)constructors and holes. A sketch is parameterized over the type of values stored in the holes. For example, we may store numbers in holes to refer to them. More interestingly, we can store specifications in holes to describe the subgoals.

***Tactics.*** Starting from a specification, we may want to introduce a sketch and compute the subgoals. Imagine a function propagate that can compute subgoals by propagating a specification through a sketch.

```
propagate : Sketch a → Spec → Maybe (Sketch Spec)
```

Unfortunately, we cannot define this function in general. The process of inferring the input-output examples that should hold on the holes of an expression, known as *example propagation* [Osera and Zdancewic 2015], is not well-defined for all sketches. For example, when implementing delete, if we try to introduce the hole filling ⓪ ↦ filter ① ②, there is an infinite number of possible input-output examples we could infer for ②. As a solution, we only allow sketching through tactics, i.e. functions that attempt to introduce a sketch for which example propagation is well-defined. For example, we can describe a tactic that applies filter to a variable in scope, introducing the following hole filling:

```
filter (predicate a) xs   where predicate x y = ◯
```

In addition to failing when the sketch does not fit the specification, the tactic should be able to (1) arbitrarily choose a variable *xs* to apply filter to (possibly leading to multiple correct sketches), and (2) generate fresh variable names for predicate, *x*, and *y*. To this end, we define tactics in terms of a Synth monad, which allows failure, branching, and generating fresh variables.

```
type Tactic = Spec → Synth (Sketch Spec)
(‖) : Tactic → Tactic → Tactic
fail : Tactic
```

We define a tactic introFilter that applies filter to a specific variable in scope, passes all the other variables in scope along to the predicate, computes the specification that should hold on the predicate, ensures that this specification is feasible, and applies program extraction if all cases are covered.

```
introFilter : Var → Tactic
```

To apply this function to any variable in scope, we define the tactic combinator anywhere.

```
anywhere : (Var → Tactic) → Tactic
anywhere t spec = foldr (‖) fail ts
  where ts = map t (variables spec)
```

To find an implementation for the function delete, we simply apply the tactic anywhere introFilter to the specification and enumerate the values in the resulting Synth monad. If we find a sketch without holes, it is guaranteed to be a total program that implements the specification.

We define many such tactics. For example, exact introduces a sketch without holes; introCase introduces a case distinction on a variable in scope (provided it is an algebraic datatype); and introConstructor introduces a constructor if all input-output examples agree. Additionally, we define tactics introducing higher-order combinators such as filter, map, and foldr. While introMap and introFilter only work on lists, introFold works on any inductively defined datatype.

```
exact : Sketch Void → Tactic
introCase : Var → Tactic
introConstructor : Tactic
introMap : Var → Tactic
introFilter : Var → Tactic
introFold : Var → Tactic
```

***Recursion Schemes.*** The tactic introFold is particularly important, for defining recursive programs. Unfortunately, example propagation on folds results in subgoals in terms of *input-output traces*, rather than input-output examples. Consider the following example for reverse:

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

If we try to implement reverse as foldr $f$ [], this leaves us with the following input-output trace:

$$f\ 1\ (f\ 2\ (f\ 3\ [])) \equiv [3,2,1]$$

It is possible to reason about the feasibility of such input-output traces by encoding the constraints in an SMT solver, as we did in our previous work [Mulleners et al. 2024]. Taxi can, however, only reason about constraints that are just input-output pairs. We can remedy this by requiring that the input-output examples are *trace complete* [Osera and Zdancewic 2015]. A set of input-output examples is trace complete if any recursive call is also represented in the input-output examples. In the case of reverse, we additionally require the following input-output examples:

$$\text{reverse}\ [2,3] \equiv [3,2]$$
$$\text{reverse}\ [3] \equiv [3]$$
$$\text{reverse}\ [] \equiv []$$

With these examples, the constraint on $f$ becomes tractable:

$$f\ 1\ [3,2] \equiv [3,2,1]$$
$$f\ 2\ [3] \equiv [3,2]$$
$$f\ 3\ [] \equiv [3]$$

The tactic introFold only succeeds on specifications that are trace complete.

### 4.3 Composing Tactics

Typically, we want to apply tactics one after the other. Consider the function intersect : $\forall a.\ [a] \to [a] \to [a]$, which removes all elements from the first list *xs* that do not occur in the second list *ys*. Applying the tactic introFilter to a specification for intersect results in a sketch with a single hole containing a specification for $(\in ys)$. Then, we traverse the sketch, applying introFold to the hole. This leaves us with a sketch of sketches (of type Sketch (Sketch Spec)), which we flatten by *accepting* the hole filling.[3]

```
accept : Sketch (Sketch hole) → Sketch hole
```

We can define tactic composition in terms of accept.

```
compose : Tactic → Tactic → Tactic
compose t u spec = do
  sketch ← t spec
  nested ← traverse u sketch
  return (accept nested)
```

We can implement intersect using the tactic

```
compose (anywhere introFilter) (anywhere introFold)
```

After introducing filter and foldr, coverage checking kicks in, leading to the following solution:

```
intersect xs ys = filter (λx. foldr (f x) False ys) xs
  where f x y False = x ≡ y
        f x y True  = True
```

### 4.4 Program Synthesis

Note that the tactics we defined so far describe only the high-level structure of the functions they implement. The details, such as which variables to apply functions to and how to solve the subgoals, are handled by the anywhere combinator and by example coverage checking. This means that these tactics can also be used to implement various other functions that use the same structure. The right tactic can synthesize programs for a variety of different specifications. Often, however, we cannot rely so much on example coverage checking, and require many more tactics to be used in a specific order to find the definition we are looking for. We will define a tactic that repeatedly tries many different tactics. First, we define the tactic combinator repeat.

```
repeat : Tactic → Tactic
repeat tactic = compose tactic (repeat tactic)
```

Next, we define a step tactic that tries a variety of different tactics in parallel.

```
step : Tactic
step = anywhere assume ‖ anywhere eliminate
  ‖ introTuple ‖ introConstructor
  ‖ anywhere₂ compare ‖ anywhere₂ equate
eliminate x = introMap x ‖ introFilter x
  ‖ introFold x ‖ introCase x
```

The tactic assume tries to fill a hole with a variable in scope. The tactics compare and equate introduce a pattern match on the ordering and equality of pairs of inputs respectively. To apply them we use a variant of anywhere, called $\text{anywhere}_2$, that applies its argument to each unique pair of variables in scope. We define a synthesis tactic as synth = repeat step. This tactic is general enough to synthesize delete, intersect, reverse, as well as many other functions.

***Enumeration Strategy.*** Synthesis proceeds by building up a tactic, applying it to a specification, and then enumerating the results. We use Dijkstra's algorithm [Dijkstra 1959] to enumerate the results, allowing the programmer to add weights to tactics. For synth, we want to prioritize simple tactics over recursion schemes, and especially discourage pattern matching, because it leads to overfitting. As such, we give simple tactics a weight of 1, recursion schemes a weight of 4, and pattern matching a weight of 7.

### 4.5 Biased Choice

The step tactic performs a lot of branching, to cover a large space of programs. Some of this branching, however, is redundant. For example, if the tactic assume x succeeds, there is no need to attempt any other tactic, since a correct solution is found. We introduce a new tactic combinator $\triangleleft$ for biased choice, where the right hand tactic is only tried when the left hand tactic fails. Together with a biased version of the anywhere tactic, we redefine the step tactic so that no other tactic is tried if introducing any variable is successful.

```
step = anywhereBiased assume ◁ ...
```

It may be tempting to replace any occurrence of (‖) with ($\triangleleft$), but this will lead to some programs never being reached. For example, if we write introCase $x \triangleleft$ introMap $x$, the tactic introMap $x$ will never be reached, because pattern matching on lists always succeeds.

The biased choice operator only makes sense if we can be sure that if the left hand tactic succeeds, the resulting subgoals are feasible. This means that proper feasibility reasoning is a prerequisite for using a biased choice operator.[4] We should, however, only use $t \triangleleft u$ instead of $t \parallel u$ if we always prefer a solution in terms of $t$ over a solution in terms of $u$ (when both are applicable). This implies that we can

[3] Note that Sketch is a monad, with join = accept and return = Hole.

[4] Except if the left hand tactic does not introduce holes.

always use $t \triangleleft u$ or $u \triangleleft t$ interchangeably when $t$ and $u$ are distinct (i.e. they never both succeed on the same specification). We use this principle to adjust the definition of eliminate:

- If both introMap $x$ and introFilter $x$ apply, the specification describes the identity function on lists, which would have already been caught by assume $x$. Hence, we can consider them distinct.
- If introFilter $x$ applies, introFold $x$ will also apply, since filter can be defined in terms of foldr. We prefer a definition in terms of filter, to avoid reimplementing filter in terms of foldr.
- Since introCase $x$ will always apply (provided that $x$ is an algebraic datatype), we have to decide whether we prefer defining a function as a fold whenever possible. While there are cases where pattern matching leads to a simpler solution, we accept the risk of finding some convoluted solutions.[5]

This leads to the following redefinition of eliminate:

$$\lambda x.\ \text{introMap}\ x \triangleleft \text{introFilter}\ x \triangleleft\ \text{introFold}\ x \triangleleft \text{introCase}\ x$$

## 5 Evaluation

We have implemented both Taxi and Driver in Haskell. The full implementation is available on Zenodo [Mulleners 2025] for reproduction and on GitHub[6] for reuse. In this section, we evaluate our implementation on two benchmarks: one for automatically testing whether a function is a fold, and one for program synthesis. Both benchmarks are run on an HP Elitebook 850 G6 with an Intel® Core™ i7-8565U CPU (1.80 GHz) and 32 GB of RAM.

### 5.1 Benchmark: Fold Detection

We evaluated our previous work by using feasibility reasoning to automatically test whether a series of functions from the Haskell prelude can be implemented as a fold [Mulleners et al. 2024]. We can similarly check whether a function is a fold by applying the tactic introFold. To evaluate Taxi, we test it on the same benchmark of functions. Whereas our previous work distinguished between trace complete and trace incomplete example sets, Taxi only works for trace complete example sets. For simplicity and for a more equal comparison, we ran both tools on trace complete examples sets. We made the following minor changes to the benchmark:

- The examples for append and prepend were not trace complete, so we added the missing case.
- We replaced integer arguments with natural numbers.
- For zip and unzip we use the more generic type as seen in the Haskell prelude, using two separate type variables rather than a single one.

As seen in Table 1, we improve significantly on this benchmark, achieving speedups of more than a hundredfold. This paves the way for applications where feasibility reasoning is called repeatedly, for example to prune the search space during program synthesis. In addition, we extend the benchmark with various functions on trees, to show how Taxi can reason about folds over inductive datatypes other than lists, as well as ad hoc polymorphic functions using Eq and Ord constraints.

### 5.2 Benchmark: Program Synthesis

We evaluate Driver on the same benchmark of programs as Taxi, by applying the synth tactic defined in Section 4.4. To see the effect of the techniques described in this paper, we test synthesis using various different options: we perform synthesis without feasibility reasoning (N); with feasibility reasoning (F); with example coverage checking (C), using program extraction to shortcut the search when a specification is exhaustive; with branching control (B), using the eliminate tactic from Section 4.5; and with both coverage checking and branching control (C+B).

This benchmark suite, shown in Table 1, shows that we can use synth to synthesize a large variety of functions at reasonable speed. While feasibility reasoning only seldomly leads to significant speedups, it allows for coverage checking and branching control, leading to additional speedups with minimal overhead.

Coverage checking works particularly well for programs with Eq and Ord constraints. This can be explained by the additional equate and compare tactics, introduced by Eq and Ord constraints respectively, that can be circumvented by coverage checking.

Branching control mostly affects programs working on lists, since introMap and introFilter are specialized to lists. In a way, branching control using a biased choice operator allows us to add helper functions like map and filter to the search space without increasing the branching factor.

Several functions still fail to synthesize, either due to a timeout or overfitting. Some functions fail because they cannot be implemented using the synth tactic. Typically because they require a stronger recursion scheme.

The functions insert, sorted, sort, and ordNub, are most naturally defined using paramorphisms [Meertens 1992], rather than a fold. We can define a tactic introPara, which generalizes introFold. Both insert and sorted are synthesized correctly by composing introPara and synth. The functions sort and ordNub are synthesized by composing introFold, introPara, and synth.

Interestingly, the function sort does synthesize correctly when branching control (B) is turned off. On closer inspection, we see that this is an implementation of insertion sort, with the following curious, inefficient solution for insert:

$$\text{insert}\ x = \text{foldr}\ (\lambda y\ r.\ \text{min}\ x\ y\ :\ \text{map}\ (\text{max}\ y)\ r)\ [x]$$

[5] In general, it is up to the programmer to decide whether this is a risk worth taking when defining a synthesis tactic. Luckily, one can always try a different tactic when the result is not satisfactory.

[6] https://github.com/NiekM/taxi-driver

**Table 1.** Benchmark testing fold detection and synthesis on a series of common functions. Times in milliseconds. The fold detection benchmark shows how fast we can compute whether each function can be implemented as a fold. We compare our current work (Curr.) to our previous work (Prev.) [Mulleners et al. 2024] where possible. We use ✓ and ✗ respectively to denote whether a function is or is not a fold. Note that union is a fold over its second, but not its first argument. The synthesis benchmark shows how fast each of the functions can be synthesized using various levels of feasibility reasoning: no feasibility reasoning (N); feasibility reasoning (F); feasibility reasoning with coverage checking (C); feasibility reasoning with branching control (B); and feasibility reasoning with coverage checking and branching control (C+B). Timeouts (taking longer than 1 second) are denoted by ⊥. Overfitted synthesis results are shown in gray. Noticeable improvements (≥ 50%) are shown in **bold**.

| | | | Fold Detection | | | | Synthesis | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| name | : | Type | Fold? | Curr. | Prev. | Gain | N | F | C | B | C+B |
| append | : | $\forall a.\ [a] \to [a] \to [a]$ | ✓ | 0.455 | 159 | 349× | 1.28 | 1.59 | 1.57 | 1.36 | 1.35 |
| concat | : | $\forall a.\ [[a]] \to [a]$ | ✓ | 0.568 | 126 | 222× | 3.66 | 3.90 | 2.92 | 2.39 | **1.74** |
| drop | : | $\forall a.\ \text{Nat} \to [a] \to [a]$ | ✗ | 0.263 | 45 | 171× | 4.15 | 2.76 | 2.77 | 2.21 | 2.20 |
| head | : | $\forall a.\ [a] \to \text{Maybe}\ a$ | ✓ | 0.240 | 33 | 138× | 0.305 | 0.362 | 0.288 | 0.321 | 0.295 |
| index | : | $\forall a.\ \text{Nat} \to [a] \to \text{Maybe}\ a$ | ✗ | 0.235 | 39 | 166× | 14.0 | 11.6 | 11.7 | **5.45** | 5.42 |
| init | : | $\forall a.\ [a] \to [a]$ | ✗ | 0.260 | 31 | 119× | 23.4 | **9.40** | 8.30 | **4.51** | 4.21 |
| last | : | $\forall a.\ [a] \to \text{Maybe}\ a$ | ✓ | 0.314 | 31 | 99× | 1.60 | 1.30 | **0.363** | 0.706 | 0.371 |
| length | : | $\forall a.\ [a] \to \text{Nat}$ | ✓ | 0.182 | 23 | 126× | 0.303 | 0.350 | 0.306 | 0.312 | 0.313 |
| null | : | $\forall a.\ [a] \to \text{Bool}$ | ✓ | 0.203 | 20 | 99× | 0.237 | 0.271 | 0.242 | 0.281 | 0.248 |
| prepend | : | $\forall a.\ [a] \to [a] \to [a]$ | ✓ | 0.460 | 149 | 324× | 1.20 | 1.57 | 1.58 | 1.37 | 1.38 |
| reverse | : | $\forall a.\ [a] \to [a]$ | ✓ | 0.272 | 57 | 210× | 3.26 | 3.27 | **1.63** | **1.58** | 1.36 |
| splitAt | : | $\forall a.\ \text{Nat} \to [a] \to ([a], [a])$ | ✓ | 0.396 | 384 | 970× | 223 | 221 | 226 | 190 | 192 |
| tail | : | $\forall a.\ [a] \to [a]$ | ✗ | 0.160 | 31 | 194× | 0.494 | 0.303 | 0.311 | 0.308 | 0.304 |
| take | : | $\forall a.\ \text{Nat} \to [a] \to [a]$ | ✓ | 0.423 | 83 | 196× | 122 | 114 | 113 | **29.9** | 30.2 |
| unzip | : | $\forall a\ b.\ [(a, b)] \to ([a], [b])$ | ✓ | 0.350 | 126 | 360× | 3.69 | 3.89 | 3.42 | 1.90 | **1.45** |
| zip | : | $\forall a\ b.\ [a] \to [b] \to [(a, b)]$ | ✓ | 0.324 | 128 | 395× | 21.6 | 23.0 | 23.2 | **8.54** | 8.79 |
| bfe | : | $\forall a.\ \text{Tree}\ a \to [a]$ | ✗ | 0.887 | – | – | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| depth | : | $\forall a.\ \text{Tree}\ a \to \text{Nat}$ | ✓ | 0.536 | – | – | 4.77 | 4.74 | 4.36 | 2.67 | 2.64 |
| inorder | : | $\forall a.\ \text{Tree}\ a \to [a]$ | ✓ | 1.21 | – | – | 8.74 | 8.79 | 8.66 | 5.43 | 5.42 |
| levels | : | $\forall a.\ \text{Tree}\ a \to [[a]]$ | ✓ | 1.15 | – | – | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| mirror | : | $\forall a.\ \text{Tree}\ a \to \text{Tree}\ a$ | ✓ | 0.554 | – | – | 0.713 | 0.966 | 0.905 | 0.866 | 0.807 |
| size | : | $\forall a.\ \text{Tree}\ a \to \text{Nat}$ | ✓ | 0.624 | – | – | 3.76 | 3.76 | 3.04 | 2.38 | 2.34 |
| compress | : | $\forall a.\ \text{Eq}\ a \Rightarrow [a] \to [a]$ | ✓ | 0.745 | – | – | 105 | **28.9** | 16.9 | 23.0 | **12.8** |
| group | : | $\forall a.\ \text{Eq}\ a \Rightarrow [a] \to [[a]]$ | ✓ | 0.673 | – | – | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| elem | : | $\forall a.\ \text{Eq}\ a \Rightarrow a \to [a] \to \text{Bool}$ | ✓ | 0.723 | – | – | 0.836 | 0.979 | 0.852 | 0.961 | 0.845 |
| elemIndex | : | $\forall a.\ \text{Eq}\ a \Rightarrow a \to [a] \to \text{Maybe Nat}$ | ✓ | 0.482 | – | – | 631 | 603 | **212** | **28.4** | 18.2 |
| nub | : | $\forall a.\ \text{Eq}\ a \Rightarrow [a] \to [a]$ | ✓ | 0.313 | – | – | 6.47 | 6.46 | **1.52** | 3.81 | 1.27 |
| union | : | $\forall a.\ \text{Eq}\ a \Rightarrow [a] \to [a] \to [a]$ | ✗/✓ | 1.07 | – | – | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| insert | : | $\forall a.\ \text{Ord}\ a \Rightarrow a \to [a] \to [a]$ | ✓ | 0.569 | – | – | 108 | 102 | **37.8** | **45.5** | 21.8 |
| maximum | : | $\forall a.\ \text{Ord}\ a \Rightarrow [a] \to \text{Maybe}\ a$ | ✓ | 0.608 | – | – | 5.73 | 5.29 | **0.710** | **1.84** | **0.697** |
| ordNub | : | $\forall a.\ \text{Ord}\ a \Rightarrow [a] \to [a]$ | ✓ | 0.475 | – | – | 72.5 | 68.9 | **20.7** | **24.1** | **8.63** |
| pivot | : | $\forall a.\ \text{Ord}\ a \Rightarrow a \to [a] \to ([a], [a])$ | ✓ | 0.732 | – | – | 95.4 | 94.6 | **5.62** | **46.5** | 4.02 |
| sort | : | $\forall a.\ \text{Ord}\ a \Rightarrow [a] \to [a]$ | ✓ | 0.566 | – | – | 216 | 201 | **55.1** | **66.7** | **20.4** |
| sorted | : | $\forall a.\ \text{Ord}\ a \Rightarrow [a] \to \text{Bool}$ | ✗ | 0.542 | – | – | 20.7 | 20.1 | 20.5 | 19.9 | 20.2 |

Proving the correctness of this implementation is left as an exercise to the reader. It may seem that branching control has made synthesis worse, but this is only accidental: the synth tactic finds three solutions at the same depth, of which only one is correct. In the benchmark we only consider the first of these solutions, and the order of solutions is arbitrary, but influenced by the use of branching control. This example highlights two ways in which the programmer can interact with the synthesizer: (1) the programmer can enumerate the first $n$ results returned by the synthesizer, to see if any

of them have the intended behavior; and (2) the programmer can test the returned solution, find a counterexample,[7] and update the specification accordingly. Both of these approaches would help the programmer find a correct (albeit very inefficient) solution for sort. Unsurprisingly, we can use the same approach to synthesize insert directly.

The functions drop, index, splitAt, take, and zip all are most naturally defined by recursing over both inputs simultaneously. This can be done as a fold over one of the inputs, by passing the other input as an additional argument to the fold. Following our previous work, however, the introFold tactic does not allow passing additional arguments [Mulleners et al. 2024]. As such, these functions all fail to synthesize.

The expected definition for depth requires computing the maximum of two natural numbers, requiring structural recursion over both numbers at the same time.[8] Similarly, the function levels can be implemented as a fold in terms of longZip, which traverses two lists at the same time.

Breadth first enumeration (bfe) can be implemented in many ways, but typically using an intermediate result. For example, bfe can be implemented by first computing levels, and then concatenating the result. The synth tactic provides no way to compute or reason about intermediate results, so naturally it fails to implement bfe.

# 6 Related Work

## 6.1 Programming with Holes

When writing a program, there are many points at which the state of the program is not syntactically correct, in particular when parts of the program are still unfinished or missing. This makes it difficult to test the program. A common practice is to have unfinished parts of a program raise an exception (such as `NotImplemented`). This allows the program to still compile and type check, as well as be executed (exceptionally). A more principled approach, however, is to use *holes*. Holes explicitly tell the compiler which parts of the program are yet to be finished, allowing it to give helpful feedback, for example inferring the types of the holes.

There have been many efforts to make programming with holes (also known as *sketching* [Solar-Lezama 2009]) an integral part of the development process. Gissurarson [2018] uses the type of a hole to give suggestions on how to fill it. Omar et al. [2017]; Yuan et al. [2023]; Zhao et al. [2024] use holes in the structure editor Hazel, in order to make as many editor states as possible meaningful. Omar et al. [2019] define *live evaluation*, which proceeds around the holes rather than aborting when a hole is encountered during execution. Lubin et al. [2020] compare the result of live evaluation against an expected output to turn top-level assertions into assertions on the holes of a program. Taxi brings types, assertions (in the form of input-output examples), and sketching together to allow reasoning about incomplete programs and possible hole suggestions.

## 6.2 Type-and-Example Directed Program Synthesis

Program synthesis concerns the automatic generation of programs based on a specification. Many synthesizers focus on types and input-output examples as specifications. A common idea for top-down synthesizers is that the space of type-correct programs can be explored by "inverting the typing rules" [Osera 2019]. Some synthesizers focus on efficiently exploring this search space using compact representations of type-correct programs [Botelho Guerra et al. 2023; Guo et al. 2019; Koppel et al. 2022]. Other synthesizers aim to explore fewer programs, narrowing down the search space by only allowing those refinements that preserve the input-output constraints [Feser et al. 2015; Frankle et al. 2016; Lubin et al. 2020; Mulleners et al. 2023; Osera and Zdancewic 2015]. Driver takes a similar approach, but focuses on the interactions between input-output constraints and polymorphic types, and explores how synthesizers can benefit from these interactions to prune and short-circuit the search.

## 6.3 Feasibility of Specifications

When programming from specifications, an important question to ask is whether a program satisfying the specification exists. Morgan [1990] describes specifications for which no implementation exists as *infeasible*. Feasibility reasoning is related to proof search and program synthesis, but focuses more on figuring out that a specification is *not* feasible. How well we can reason about the feasibility of specifications depends on the nature of the specifications, as well as the underlying programming language. Urzyczyn [1997] reasons about the feasibility of types, known to as *type inhabitation*. The term *realizability* is often used instead of feasibility, typically to refer to the feasibility of programs in a restricted grammar [Hu et al. 2019, 2020; Kim et al. 2023]. In our previous work, we reasoned about the feasibility of polymorphic types and input-output examples [Mulleners et al. 2024]. Taxi extends upon this work, allowing for faster and more transparent reasoning about a wider range of types.

## 6.4 Tactic Programming

Tactic programming is a form of metaprogramming, where tactics are functions that refine a goal (i.e. a specification) [Gordon et al. 1979]. Tactics transform a single goal into subgoals (a process referred to as *backwards reasoning*), such that a solution to these subgoals leads to a solution of the original goal. Tactic programming is most commonly seen in proof assistants, such as The Rocq Prover [Coquand and Huet 1985],[9] Lean [de Moura et al. 2015],[10] and Isabelle/HOL [Nipkow

[7] E.g. using property-based testing to find a minimal counterexample.
[8] The compare tactic currently only allows comparing polymorphic variables with an Ord constraint, not natural numbers.

[9] https://rocq-prover.org/docs/metaprogramming-ltac2
[10] https://lean-lang.org/doc/reference/latest/Tactic-Proofs

et al. 2002]. These proof assistants typically use the Curry-Howard correspondence to prove propositions by encoding them in a dependent type system and then implementing a program of that type to serve as a proof witness. Tactics allow one to focus on solving the goals, typically extracting the proof witness only as a byproduct. *Tactic combinators*, or *tacticals*, are used to combine simple tactics into more complex ones, to the point that a single tactic may perform *automated proof search* to solve a large range of goals.

There has been some effort towards introducing tactics outside of proof assistants. In general-purpose languages, the intended behavior of programs is typically not so rigorously defined, and as a result, there are not always clear goals to be discharged by tactics. In addition, we care more about the exact nature of programs, beyond just their existence as proof witnesses. In general-purpose statically typed languages, however, we can still use types as goals, and use tactics to introduce type-correct refinements for a sketch. An example is `refinery`,[11] a framework for tactic programming loosely based on [Sterling and Harper 2017], which used to power the Wingman for Haskell synthesis tool.[12] Wingman allowed the programmer to refine their programs by applying tactics through the language server protocol.

Driver uses a tactic language inspired by `refinery` to provide a structured way of exploring sketches, but extends it in two ways: Driver uses backwards reasoning not just with types, but also with input-output examples (i.e. example propagation); and Driver tactics can return multiple possible sketches, such that tactics build up a program space that can be traversed and enumerated.

## 7 Conclusion

We have shown how automated reasoning about the feasibility of polymorphic functions with input-output examples can be made efficient enough to serve as a pruning tool during type-and-example driven program synthesis, ensuring that every synthesis state corresponds to a correct (albeit partial) implementation. These guarantees allow the use of a biased choice operator, which allows one to use more building blocks during synthesis, without the downside of additional branching. Example coverage checking allows the synthesizer to automatically finish any subgoals that correspond to a total implementation, without the need to further explore the search space.

### 7.1 Limitations and Future Work

Taxi can only deal with conjunctions of input-output examples, as opposed to more complex constraints, such as input-output traces. As such, Driver may get stuck when trying to apply a recursion scheme to trace incomplete input-output examples. This is particularly difficult to avoid when the recursion occurs deeper in the program, as it is unlikely that propagated input-output examples turn out to be trace complete, even if the top-level input-output examples *are* trace complete. An SMT-based approach can represent more complex constraints, and therefore reason about the feasibility of input-output traces, at the cost of efficiency, transparency, and decidability. Future work could explore how such constraints can still be solved efficiently, as well as how example propagation can be adapted to allow propagating such constraints further.

Various functions require more complex recursion patterns than the ones used in this paper. For example, the typical implementation for zip recurses over both input lists in tandem, requiring a specialized tactic. Determining which recursion schemes are worth including and figuring out how their inclusion influences performance requires more experimentation.

[11] https://github.com/TOTBWF/refinery
[12] https://hackage.haskell.org/package/hls-tactics-plugin

## Acknowledgments

We would like to acknowledge the contribution that the late Bastiaan Heeren made to this work before his unfortunate passing. We would also like to thank the members of the ST4LT and ST research groups at Utrecht University for their support and feedback.

## 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. doi: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. doi:10.1016/j.tcs.2005.06.002

Henrique Botelho Guerra, João F. Ferreira, and João Costa Seco. 2023. Hoogle⋆: Constants and $\lambda$-abstractions in Petri-net-based Synthesis using Symbolic Execution. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:28. doi:10.4230/LIPIcs.ECOOP.2023.4

Thierry Coquand and Gérard Huet. 1985. Constructions: A higher order proof system for mechanizing mathematics. In *EUROCAL '85*, Bruno Buchberger (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–184. doi:10.1007/3-540-15983-5_13

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 378–388. doi:10.1007/978-3-319-21401-6_26

Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271. doi:10.1007/BF01386390

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. doi: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. doi:10.1145/2914770.2837629

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

M. Gordon, R. Milner, and C. P. Wadsworth. 1979. *Edinburgh LCF: A Mechanized Logic of Computation.* Number 78 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg. doi:10.1007/3-540-09724-4

Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program Synthesis by Type-Guided Abstraction Refinement. *Proc. ACM Program. Lang.* 4, POPL, Article 12 (Dec. 2019), 28 pages. doi:10.1145/3371080

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

Mauro Jaskelioff and Russell O'Connor. 2015. A representation theorem for second-order functionals. *Journal of Functional Programming* 25 (2015), e13. doi:10.1017/S0956796815000088

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

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. doi:10.1145/3547622

Justin Lubin and Sarah E. Chasins. 2021. How statically-typed functional programmers write code. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 155 (Oct. 2021), 30 pages. doi:10.1145/3485532

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. doi:10.1145/3408991

Lambert Meertens. 1992. Paramorphisms. *Form. Asp. Comput.* 4, 5 (Sept. 1992), 413–424. doi:10.1007/BF01211391

Carroll Morgan. 1990. *Programming from specifications.* Prentice-Hall, Inc., USA.

Niek Mulleners. 2025. *NiekM/taxi-driver: PEPM 2026.* doi:10.5281/zenodo.17900892

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. doi:10.1007/978-3-031-24841-2_2

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 (Aug. 2024), 21 pages. doi:10.1145/3674636

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer-Verlag, Berlin, Heidelberg. doi:10.1007/3-540-45949-9

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. doi:10.1145/3290327

Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. *SIGPLAN Not.* 52, 1 (Jan. 2017), 86–99. doi:10.1145/3093333.3009900

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. doi: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. doi:10.1145/2737924.2738007

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. doi:10.1007/978-3-642-28412-0_17

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems*, Zhenjiang Hu (Ed.). Springer Berlin Heidelberg, 4–13. doi:10.1007/978-3-642-10672-9_3

Jonathan Sterling and Robert Harper. 2017. Algebraic Foundations of Proof Refinement. *ArXiv* (2017). doi:10.48550/arXiv.1703.05215

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. doi: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. doi:10.1145/99370.99404

Yongwei Yuan, Scott Guest, Eric Griffis, Hannah Potter, David Moon, and Cyrus Omar. 2023. Live Pattern Matching with Typed Holes. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 96 (April 2023), 27 pages. doi:10.1145/3586048

Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. 2024. Total Type Error Localization and Recovery with Holes. *Proc. ACM Program. Lang.* 8, POPL, Article 68 (Jan. 2024), 28 pages. doi:10.1145/3632910

Received 2025-10-24; accepted 2025-11-28