Fantasia: Synthesising Recursive Functions Without Trace-complete Examples

Zac Garby
University of Nottingham
psyzg5@nottingham.ac.uk

Graham Hutton
University of Nottingham
graham.hutton@nottingham.ac.uk

ABSTRACT

Program synthesis is the art of deriving a program, automatically, from a specification. Often, this specficiation is a set of input-output examples.

Many existing example-directed synthesis techniques, however, struggle to synthesise recursive functions without "strategically provided", or *trace complete*, examples fully describing recursive behaviour.

In this paper, we introduce a novel approach to analytical example-based synthesis, wherein the synthesiser uses thunks to approximate function behaviour during synthesis. We find that this substantially reduces the required input specification size for many synthesis problems.

Keywords

Program Synthesis, Functional Programming, Example-directed Synthesis, Thunks.

1. INTRODUCTION

Programming is often thought of as a creative activity, but in a lot of cases it doesn't feel this way. In any program, there will inevitably be a scattering of simple but tedious functions, requiring little thought but monotonous data manipulation. In these situations, program synthesis becomes a valuable tool.

Similarly, program synthesis can be applied when a complex function is required and the programmer—perhaps a beginner, perhaps an expert who has just had a long day—would prefer not to implement the function themselves. Here, it may be easier to give a list of input-output examples, and to simply allow the synthesiser to extrapolate a working program from these. In these cases, it may be useful to synthesise just part of the function, leaving "holes" for the programmer to fill in.

Synthesis based on sets of input-output examples is powerful, because it is a paradigm which is easily understood by both machines and by humans. Example-based techniques provide plenty of expressive power to the humans interacting with the system, while retaining enough structure that the synthesis can proceed in a logical, *analytical*, fashion.

This form of synthesis also, in most implementations, provides "correct" solutions: the program produced by the machine is correct with respect to the examples provided (though not necessarily to the human's intention—we come back to this problem later on). This is opposed to some modern techniques based on language models (see [7], [14], or [15] for examples), where the resulting program can potentially disagree with the specification. This can be dangerous if the discrepancy is unobvious, and so we concern ourselves only with correct techniques.

A common problem with example-directed synthesis techniques is that, to generate recursion functions, they require the examples to be *trace complete*. That is, input examples must be given for each recursive call down to the base case (see [1] or [16] for further discussion). We discuss this problem further in Section 2.

Another, more minor, issue with similar techniques is that they are unable, generally, to produce auxiliary functions. In many cases (e.g. `reverse`), solutions using auxiliary functions can be more efficient (i.e. worker/wrapper [6]). Typically, example-directed analytical synthesis produces a single function, losing these benefits.

In Section 3, we propose a novel approach to synthesis of recursive functional programs which solves these two problems. We implement this in a new system which we name Fantasia (see Section 4).

1.1 Examples

To begin, we consider an example: the function `stutter`, which duplicates each element in a list. For example, `stutter [1, 2]` would give us `[1, 1, 2, 2]`. We can specify this function in terms of a type and two examples.

```
stutter : List a → List a
  { [] → []
  , [1, 2] → [1, 1, 2, 2] }
```

The techniques we describe in the remainder of this paper can take this specification and automatically produce the following function[1]:

```
stutter xs = case xs of
  Nil → []
  Cons x xs' → Cons x (Cons x (stutter xs'))
```

It's interesting to note here that the system has synthesised a recursive function without being provided with every recursive case down to the base case, meaning that it was able to "guess" the behaviour of the function while simultaneously synthesising it.

As another example, consider the specification:

[1]In the code examples in this paper, as well as for the implementation later on, we use the Fugue programming language. Fugue is not published, but please see e.g. https://github.com/zac-garby/diss/blob/master/fugue/eg.fugue for more information.

```
head : List a → a
  { [1, 2, 3] → 1 }
```

The examples don't give the synthesiser a clue as to what should be done for the empty list, and so the output is a *partial* program:

```
head xs = case xs of
  Nil → ?
  Cons x xs' → x
```

The synthesiser has used a *hole*, indicating that it "doesn't know", and prompting the user to fill in this part themselves.

1.2 Overview

In this paper, we give a brief overview of the current work in this area (Section 2), then we discuss our new techniques to solve some existing problems (Section 3). We discuss the practical considerations of our technique as demonstrated in our reference implementation, titled FANTASIA, in Section 4. In Section 5, we analyse our results, and then in Section 5.1 we fantasise about potential uses of these synthesis techniques, as well as possible improvements and future work.

1.3 Contributions

There is significant work already on example-directed synthesis of functional programs, which we discuss in Section 2. This paper builds upon this body of work with the following main contributions:

1. We identify—and propose solutions to—two major problems with certain otherwise powerful and elegant techniques in the literature, specifically:
   (a) the synthesis of recursive programs, and
   (b) the synthesis of auxiliary functions.
2. We extend these solutions with the ability to synthesise *partial programs* (i.e. programs which may contain holes).
3. We also provide support for polymorphic functions—something which many of these existing techniques do not, but which provides many tangible benefits.

2. RELATED WORK

Hofmann [9] identifies two broad approaches to program synthesis which are applicable to functional programming: *analytical* and *enumerative*. These are neither mutually exclusive nor are they exhaustive (most notably they do not cover machine-learning techniques), but they are a useful taxonomy of synthesis techniques nonetheless

2.1 Enumerative

In enumerative approaches to program synthesis, an infinite stream of possible programs is constructed by some means, and the emitted programs are checked in order until a program is found which matches the specification.

An example of this is SNIPPY ([5], [4]), which uses a fairly straightforward *"generate-and-test"* synthesiser. The synthesiser enumerates all possible programs generated by the language's grammar, and as a result can only synthesise programs "of up to height 3 (zero-based)" within a timeout of seven seconds. "The astronomical size of the search space" is the primary challenge of all synthesisers, and especially the enumerative kind. SNIPPY attempts to improve performance using *observational equivalence* ([1]) to narrow the search.

Not all enumerative techniques are so straightforward. For example, [8] describes a complex system using reachability in a network of types to either accept or refute candidate solutions. They use this to implement a system, HOOGLE+, which aims to suggest compositions of functions which meet some specification, similar to its namesake HOOGLE, [13].

A further example of a purely enumerative approach to program synthesis is MAGICHASKELLER, a web-based [12] synthesis engine for Haskell. MAGICHASKELLER is based on techniques described in [3] for systematically constructing a stream of lambda expressions, as well as [11] which discusses how to remove options which are equivalent.

In general, enumerative techniques are powerful, as they can trivially make use of existing components in the environment (e.g. pre-defined functions, or user-provided chunks of code). Additionally, they are easy to understand and, for simpler enumerative models, relatively straightforward to implement, which makes them suitable for projects like SNIPPY ([5], [4]) which are focussed less on the synthesis itself but more on human-synthesiser interaction.

Enumerative techniques, however, tend to be quite slow—especially relative to analytical techniques. This is because, in general, they are not moving "towards" a solution, but rather trying many possibilities with the hopes that they will "stumble upon" their goal (to anthropomorphise slightly).

For many applications, speed is not of the utmost importance, but real-time synthesis (i.e. results provided essentially *instantaneously* to the programmer) can lead to many interesting interaction paradigms.

2.2 Analytical

The other main form of program synthesis we consider, as per Hofmann's taxonomy, is the *analytical* approach. Here, the synthesiser takes a more regimented approach, analysing the examples and building up, piece by piece, programs which we *know* will agree with the examples. The main difference is that no evaluation or testing of solutions takes place, as we never even consider "incorrect" programs.

As a result, analytical synthesis techniques tend to be significantly faster than enumerative techniques. This is because they "seek out" their goals rather than hope they stumble upon them (as enumerative techniques are forced to). Performance of synthesis is important in and of itself—the user would like to not wait too long if at all possible—but additionally, if synthesis can be done in real-time (i.e. essentially instantly as the user enters examples) that unlocks a conversational interaction paradigm which is clunky or impossible otherwise.

Furthermore, the performance benefits and "seeking" behaviour of analytical synthesisers allow them to generally produce larger programs, while enumerative techniques are often more amenable to generating snippets ([5]) or small compositions of existing functions ([8]).

An example of an analytical synthesiser is described in [16]. Similar to our technique, they synthesise programs by iteratively applying *rules*. Their synthesiser takes a goal type, a set of examples, and any required auxiliary—*component*—functions. This differs to our technique in that

we do not allow any external library functions to be used, a point which we discuss further in Section 5.

Another analytical technique which shares similarities to ours is IGOR 2 [10]. The technique described in this paper, in fact, combines analytical "seeking" techniques with enumerative "generate-and-test", leading to strong results. This paper also proposes the use of auxiliary functions during synthesis, something which we explore further. Our approach differs in how we synthesise recursion; also, IGOR 2 does not make use of type information.

It is worth noting that analytical techniques, while very effective for languages with strong type systems (like Haskell or FUGUE), are often less effective than enumerative techniques for languages such as Python with weaker type systems. This is because the "seeking" nature of analytical algorithms often relies on type information to guide its path through the search space.

We choose to take a primarily analytical approach to synthesis in this paper.

2.3 Machine Learning

The final class of synthesis techniques we consider are those based on machine learning. While we do not use any machine learning in our approach, it is worth discussing the reasons why we choose not to.

Machine learning systems aim to generalise a set of input examples into a model which can produce related output for unseen input examples. This sounds a lot like what we're aiming for with program synthesis, so we would be remiss if we didn't consider them. Furthermore, machine learning systems have recently ([7], [15], [14]) demonstrated impressive capabilities for exactly that: synthesising programs.

These systems have the benefit that generally they synthesise based on natural language specifications (e.g. "write a function which generates prime numbers") rather than examples. This, in many ways, leads to a more natural and intuitive interaction with users. These systems can also learn patterns and idioms from existing code, and as a result can generally synthesise a wider range of programs.

Machine learning systems, however, have their drawbacks. Firstly, and perhaps most worryingly, they provide no guarantee of correctness. Systems such as COPILOT [7] show seemingly impressive outputs, but if a user is subsequently forced to verify the generated program manually, the time gained by not writing it is negated.

Additionally, machine learning algorithms tend to be extremely resource intensive, due to their implementation as—nowadays—typically *large language models*. This means that a user has to either rely on an external server (leading to privacy concerns, latency, and sometimes subscription costs) or run the systems locally and accept that the synthesis will be slow or use up a large portion of their computer's memory.

3. OUR TECHNIQUE

In this section, we present our novel techniques for example-directed synthesis of functional programs. This is the main contribution of this paper.

As discussed, we take an analytical approach: our algorithm gradually builds up a working program by analysing the relevant examples at each point and considering all possible continuations of the program to satisfy these examples.

3.1 Worked Example

To make things less abstract, we begin by running through a complete example, from start to finish, explaining the algorithm as we go.

For this, we return to the de facto example of the `stutter` function, with two examples:

```
stutter : List a → List a
  { [] → []
  , [1, 2] → [1, 1, 2, 2] }
```

Synthesis proceeds by considering each of a set of possible *rules* to apply. We cover each of these rules in detail in Section 3.2, but for now we mention them as they apply.

The applicable rule here is R-RECCASE: a rule which introduces a case-split where each case may or may not make a recursive call before proceeding. R-RECCASE produces the following definition:

```
stutter xs = case xs of
  Nil → f xs
  Cons y ys → let h = stutter ys
              in g xs y ys h
```

Here is the first notable difference between our technique and others: we, at each synthesis step, invariably emit an *entire* well-defined function (in this case `stutter`). Instead of synthesising deeper and deeper expressions, we move the logic of i.e. synthesising case branches to auxiliary functions (here `f` and `g`).

To each of the auxiliary functions, we provide a full list of all visible variable names, making this in some ways equivalent to similar techniques (e.g. [16]), but with the added benefit of synthesising auxiliary functions—something not possible in most similar approaches—becoming a triviality.

The second case, `Cons`, utilises a recursive call to `stutter`. The arguments for this recursive call are selected from the pattern-matched constructor, in this case just `ys`.

Synthesis proceeds by completing the definitions of each now-required function. We start with `f`:

```
f : List a → a
  { [] → [] }
```

The `case`-split in the definition of `stutter` has narrowed down the set of examples, so that `f` now has just one. We can now make use of the R-TRIVIAL rule, which synthesises a function when there is some argument which, for each example, is trivially the return-value we want. In this case:

```
f xs = xs
```

No additional auxiliary functions were introduced by this definition, and so we proceed to synthesising `g`.

```
g : List a → a → List a → List a → List a
  { [1, 2], 1, [2], <stutter [2]> → [1, 1, 2, 2] }
```

The other notable contribution of this work is shown here in the fourth argument to the solitary example: `<stutter [2]>`. The `< ... >` notation signifies that this argument's value is a *thunk*: an expression which hasn't yet been fully evaluated. In this case, this comes from the recursive call—since we are still synthesising the function, we cannot fully evaluate this argument yet.

Thunks allow us to synthesise recursive functions without worrying about that. We "kick the problem down the road",

so to speak, and just keep the argument evaluated as far as it can go. The details can come later.

Before proceeding with synthesis, we note that this argument can in fact be evaluated further, since we do in fact have a definition of *stutter*. This is an application of the R-ThunkEval rule.

```
stutter [2]
  (by def. stutter)
= let h = stutter [] in g [2] 2 [] h
  (by def. stutter)
= let h = f [] in g [2] 2 [] h
  (by def. f)
= let h = [] in g [2] 2 [] h
  (by beta-reduction of let)
= g [2] 2 [] [].
```

We can update the examples given to `g` thus:

```
g : List a → a → List a → List a → List a
  { [1, 2], 1, [2], <g [2] 2 [] []>
  → [1, 1, 2, 2] }
```

Synthesis proceeds now by a new rule, R-Constr. This rule applies whenever all examples' outputs are, at the top-level, an instance of the same constructor (in this case `Cons`).

```
g xs y ys h = Cons (i xs y ys h) (j xs y ys h)
```

As before, we consider the newly introduced auxiliary functions, `i` and `j`, in order. But first, now `g` is fully defined, we can advance our thunk argument one step further:

```
g [2] 2 [] []
  (by def. g)
= Cons (i [2] 2 [] []) (j [2] 2 [] []).
```

Now, we can move on to synthesising the two functions, firstly `i`:

```
i : List a → a → List a → List a → List a
  { [1, 2], 1, [2],
    <Cons (i [2] 2 [] []) (j [2] 2 [] [])>
  → 1 }
```

The example here was produced by keeping the inputs the same, while deconstructing the output value `[1, 1, 2, 2]` into its constituents `1` and `1, 2, 2`. These constituent parts, or *constructor arguments*, are passed to their respective auxiliary functions as the new desired output.

Again, R-Trivial applies here, giving us:

```
i xs y ys h = y
```

Next we move onto `j`, but first, we can update our thunk a little further:

```
Cons (i [2] 2 [] []) (j [2] 2 [] [])
  (by def. i)
= Cons 2 (j [2] 2 [] []).
```

Now,

```
j : List a → a → List a → List a → List a
  { [1, 2], 1, [2], <Cons 2 (j [2] 2 [] [])>
  → [1, 2, 2] }
```

At this point, we can apply R-Constr once more, yielding:

```
j xs y ys h = Cons (k xs y ys h) (l xs y ys h)
```

The synthesis of `k` comes next, and since it is identical to `i`, I will leave the explicit derivation aside for now. It proceeds with R-Trivial, giving:

```
k xs y ys h = y
```

This allows us to advance our thunk again, since:

```
Cons 2 (j [2] 2 [] [])
  (by def. j)
= Cons 2 (Cons (k [2] 2 [] []) (l [2] 2 [] []))
  (by def. k)
= Cons 2 (Cons 2 (l [2] 2 [] [])).
```

The case of `l` is then much more interesting, as the examples are now:

```
l : List a → a → List a → List a → List a
  { [1, 2], 1, [2],
     <Cons 2 (Cons 2 (l [2] 2 [] []))>
  → [2, 2] }
```

Something has happened: our thunk, the final argument in our example, has been evaluated further, and it now looks an awful lot like the desired output from this function.

To make this more precise, we can say that the thunk, `Cons 2 (Cons 2 (l [2] 2 [] []))`, can *unify* with `[2, 2]`, since `[2, 2]` is nothing more than `Cons 2 (Cons 2 Nil)`.

As a result, we can use another rule, R-Unify, to explicitly unify these two expressions, yielding the definition:

```
l xs y ys h = m xs y ys h
```

Along with a new auxiliary function `m`, specified as:

```
m : List a → a → List a → List a → List a
  { [1, 2], 1, [2], [2, 2] → [2, 2]
  , [2],    2, [],  []     → [] }
```

We can see that `l` has essentially been replaced by `m`, but with an extra example appended. This extra example is referred to as the *unifying example*, and provides the means to fully evaluate the thunk:

```
Cons 2 (Cons 2 (l [2] 2 [] []))
  (by def. l)
= Cons 2 (Cons 2 (m [2] 2 [] []))
  (by unifying example of m)
= Cons 2 (Cons 2 []).
```

Finally, synthesis can proceed trivially, using R-Trivial on the fourth argument `h` (which was originally our recursive binding), giving us a closed-form definition of `m` without introducing any further functions to synthesise.

```
m xs y ys h = h
```

Synthesis of `stutter` can terminate here, since there is no more work to be done. In the process, we have accumulated a library of small functions—the building blocks of our program:

```
stutter xs = case xs of
  Nil → f xs
  Cons y ys → let h = stutter ys
              in g xs y ys h
f xs = xs
```

```
g xs y ys h = Cons (i xs y ys h) (j xs y ys h)
i xs y ys h = y
j xs y ys h = Cons (k xs y ys h) (l xs y ys h)
k xs y ys h = y
l xs y ys h = m xs y ys h
m xs y ys h = h
```

We could, of course, leave it here and call the program done: indeed, entering these eight functions into the FUGUE REPL would give us a working implementation of `stutter`, but this is clearly an unsatisfying conclusion.

The final step of synthesis is clean-up. We begin by folding as many functions as we can into one, starting from the root. This gives us the single function,

```
stutter xs = case xs of
  Nil → xs
  Cons y ys → let h = stutter ys
              in Cons y (Cons y h)
```

We can make further small transformations, such as β-reduction on the `let` expression, which results in our final synthesised function:

```
stutter : forall a . List a → List a
stutter xs = case xs of
  Nil → xs
  Cons y ys → Cons y (Cons y (stutter ys))
```

And this is exactly what we were looking for! This algorithm has therefore successfully generalised the two initial examples into a function which works correctly for any list given to it. Furthermore, it's polymorphic, so it will work on lists of types it has never even seen.

3.2 Synthesis Rules

As discussed in the previous section, our technique is based on a set of rules which may or may not be applicable at any given moment during synthesis.

Each of these rules analyses a set of examples and produces an entire function along with a set of new auxiliary functions on which it depends, and which will need to be synthesised subsequently.

Our algorithm can therefore be explained in terms of these rules, leaving the specifics of when—and in what order—they should be applied abstract for now. We come onto these details in Section 4. For now, we consider the theoretical setting where "all" applicable rules are applied instantaneously at all times, as a formal system rather than something which can be directly implemented.

We present these rules in terms of the FUGUE programming language, but, it being a relatively standard example of an ML-style functional programming language (if we ignore *holes*), these techniques may equally apply to most similar languages with minimal changes.

In general, a synthesis problem looks like this:

$$\begin{aligned} f \;:\; & \tau_1 \to \tau_2 \to \cdots \to \tau_n \\ & \{\; x_{1,1},\; x_{1,2},\; \cdots,\; x_{1,n-1} \quad \Rightarrow y_1 \\ & ;\; x_{2,1},\; x_{2,2},\; \cdots,\; x_{2,n-1} \quad \Rightarrow y_2 \\ & \vdots \\ & ;\; x_{m,1},\; x_{m,2},\; \cdots,\; x_{m,n-1} \Rightarrow y_m \;\} \end{aligned}$$

In other words, we are looking for a function, f, of the type $\tau_1 \to \tau_2 \to \cdots \to \tau_n$. The synthesised function must satisfy the m given examples; each example i has $n-1$ inputs and a desired output denoted y_i.

Rule 1: R-TRIVIAL.

Perhaps the simplest rule is R-TRIVIAL. It concerns the case when all of the provided examples already contain, as one of their input values, the desired output value. When multiple examples are present, the matching argument must be in the same position across each example.

If applicable, this rule produces a function which simply returns the value of one of its arguments, unmodified. No further functions are introduced.

In general, R-TRIVIAL can be written using the following notation:

$$\frac{\exists 1 \le i < n, \quad \forall j, x_{j,i} = y_j}{f\ a_1\ a_2\ \cdots\ a_{n-1} = a_i} \qquad \text{(R-TRIVIAL)}$$

This states that, if all of the arguments at position i agree with the desired output, we synthesise the function below, named f.

Note that here, and for the rest of this section, $x_{j,i}$, y_j, and τ_i refer to the general synthesis problem above.

Rule 2: R-CONSTR.

The second rule is R-CONSTR, which deals with the case where all examples' outputs are formed by the same data constructor. In this case, we would like to apply this constructor, synthesising an auxiliary function for each constructor argument.

In general, we can define R-CONSTR for some generalised constructor $C : \tau_1^C \to \tau_2^C \to \cdots \to \tau_k^C$ as:

$$\frac{\begin{array}{l} \forall\, i \in [1,m], y_i = C\ c_{i,1}\ c_{i,2}\ \cdots\ c_{i,k}, \\ \tau_k^C \sqsubseteq \tau_n, \\ f_i' \;:\; \tau_1 \to \tau_2 \to \cdots \to \tau_i^C \\ \quad \{\; x_{1,1},\; \cdots,\; x_{1,n-1} \Rightarrow c_{1,i} \\ \quad ;\; x_{2,1},\; \cdots,\; x_{2,n-1} \Rightarrow c_{2,i} \\ \quad \vdots \\ \quad ;\; x_{m,1},\; \cdots,\; x_{m,n-1} \Rightarrow c_{m,i} \;\} \end{array}}{\begin{array}{l} f\ a_1\ \cdots\ a_{n-1} = C\ (f_1'\ a_1\ \cdots\ a_{n-1}) \\ \qquad\qquad\qquad\quad (f_2'\ a_1\ \cdots\ a_{n-1}) \\ \qquad\qquad\qquad\quad \vdots \\ \qquad\qquad\qquad\quad (f_k'\ a_1\ \cdots\ a_{n-1}) \end{array}} \qquad \text{(R-CONSTR)}$$

A little harder to read than R-TRIVIAL, this rule says that if we have:

- for each example, the output is some instantiation of the constructor C,
- the type which the constructor constructs—τ_k^C—is the same as (or more general than) the return type of f, and finally,
- we can synthesise an auxiliary function, f_i', for each argument i of the constructor, mapping the existing examples' inputs to the respective constructor arguments,

then we can synthesise f as the application of C to these auxiliary functions. This has the effect of "opening up" a constructor application.

Rule 3: R-Case.

The third rule is in a sense the opposite of R-Constr, as it allows us to deconstruct constructor values in example inputs and delegate to a different auxiliary function depending on which constructor we find. It is one of the two branching rules—along with R-RecCase—which introduce conditional execution in the form of case splits.

To define it, first suppose without loss of generality that we are considering just one polymorphic data-type, T, defined by $||T||$ constructors and z type parameters. Each constructor C_t has k_t arguments, and can therefore be written as:

$$C_t : \forall a_1\ \cdots\ a_z\ .\ \tau^T_{t,1} \to \cdots \to \tau^T_{t,k_t} \to T\ \alpha_1\ \cdots\ \alpha_z$$

Here, $\tau^T_{t,ki}$ is the type of the ith argument of the constructor C_t, and each constructor returns the same type: $T\ \alpha_1\ \cdots\ \alpha_z$.

We can now define the rule R-Case for this type, T, as shown below.

$$\frac{\begin{array}{l}\exists 1 \le i < n,\\ C_t : \forall a_1\ \cdots\ a_z\ .\ \tau^T_{t,1} \to \cdots \to \tau^T_{t,k_t} \to T\ \alpha_1\ \cdots\ \alpha_z,\\ T\ \alpha_1\ \cdots\ \alpha_z \sqsubseteq \tau_i,\\ \sigma(T\ \alpha_1\ \cdots\ \alpha_z) = \tau_i,\\ f'_j\ :\ \tau_1 \to \cdots \to \tau_{n-1} \to \sigma(\tau^T_{j,1}) \to \cdots \to \sigma(\tau^T_{j,k_j}) \to \tau_n\\ \quad \{\ x_{l,1},\ \cdots,\ x_{l,n-1},\ c_1,\ \cdots,\ c_{k_j} \Rightarrow y_l\\ \quad \mid\ x_{l,i} = C_j\ c_1\ \cdots\ c_{k_j} \text{ for } 1 \le l \le m\ \}\end{array}}{\begin{array}{l}f\ a_1\ a_2\ \cdots\ a_{n-1} = \textbf{case}\ a_i\ \textbf{of}\\ \quad C_1\ b_1\ \cdots\ b_{k_1} \to f'_1\ a_1\ \cdots\ a_{n-1}\ b_1\ \cdots\ b_{k_1},\\ \quad C_2\ b_1\ \cdots\ b_{k_2} \to f'_2\ a_1\ \cdots\ a_{n-1}\ b_1\ \cdots\ b_{k_2},\\ \quad \vdots \qquad\qquad \vdots\\ \quad C_{||T||}\ b_1\ \cdots\ b_{k_{||T||}} \to f'_{||T||}\ a_1\ \cdots\ a_{n-1}\ b_1\ \cdots\ b_{k_{||T||}}\end{array}}\ \text{(R-Case)}$$

Here, a_i is the argument to split on. σ is a substitution that unifies the data-type T with the argument being split on, τ_i. This is important in the presence of polymorphism, since otherwise the subsequent auxiliary functions f'_j will be too general, which could lead to incorrectly typed programs. We know that such a substitution must exist, because we make the assertion that this data-type is at least as general as the scrutinee's type, τ_i.

After a case-split, the subsequence functions are provided with all previous function arguments, plus all of the arguments from their respective constructor.

The examples given to these functions for further synthesis are taken as a subset of the examples of f: all those which have, as the ith input, an instance of some constructor of the type T. Each example is augmented with extra input arguments—the corresponding constructor's arguments.

Rule 4: R-RecCase.

The fourth rule, R-RecCase, is similar to R-Case. In fact, the latter can be seen as a special case of the former, but we keep both as R-Case leads to easier reasoning and simpler derivations for some problems.

In our formulation, R-RecCase is the only way to introduce recursion into functions. It constructs a case analysis against one argument, but differs from R-Case in that the bodies of some cases may introduce very specific recursive bindings. These bindings consist of a `let` expression, binding a call to the function being synthesised—f—applied to some combination of the case constructor's arguments. The value of this call, at this point a *thunk*, is then given as a further argument to the auxiliary function for that case.

We can define the rule R-RecCase similarly to R-Case:

$$\frac{\begin{array}{l}\exists 1 \le i < n,\\ C_t : \forall a_1\ \cdots\ a_z\ .\ \tau^T_{t,1} \to \cdots \to \tau^T_{t,k_t} \to T\ \alpha_1\ \cdots\ \alpha_z,\\ T\ \alpha_1\ \cdots\ \alpha_z \sqsubseteq \tau_i,\quad \sigma(T\ \alpha_1\ \cdots\ \alpha_z) = \tau_i,\\ k_t = 0,\quad \forall t \le \epsilon,\quad \epsilon \ge 0,\\ f'_j\ :\ \tau_1 \to \cdots \to \tau_{n-1} \to \tau_n\\ \quad \{\ x_{l,1},\ \cdots,\ x_{l,n-1} \Rightarrow y_l\\ \quad \mid\ x_{l,i} = C_j \text{ for } 1 \le l \le m\ \},\ \text{if } j \le \epsilon\\ g'_j\ :\ \tau_1 \to \cdots \to \tau_{n-1} \to \sigma(\tau^T_{j,1}) \to \cdots \to \sigma(\tau^T_{j,k_j}) \to \tau_n \to \tau_n\\ \quad \{\ x_{l,1},\ \cdots,\ x_{l,n-1},\ c_1,\ \cdots,\ c_{k_j},\ \langle f\ a'_{j,1}\ \cdots\ a'_{j,n-1} | f\rangle \Rightarrow y_l\\ \quad \mid\ x_{l,i} = C_j\ c_1\ \cdots\ c_{k_j} \text{ for } 1 \le l \le m,\\ \quad \text{where } a'_{j,p} \in c_{1..k_j} : \tau_p \text{ for } 1 \le p < n\ \},\ \text{if } j > \epsilon\end{array}}{\begin{array}{ll}f\ a_1\ a_2\ \cdots\ a_{n-1} = \textbf{case}\ a_i\ \textbf{of}\\ C_1 & \to f'_1\ a_1\ \cdots\ a_{n-1},\\ \vdots & \vdots\\ C_\epsilon & \to f'_\epsilon\ a_1\ \cdots\ a_{n-1},\\ C_{\epsilon+1}\ b_1\ \cdots\ b_{k_{\epsilon+1}} & \to \textbf{let}\ r = f\ a'_{\epsilon+1,1}\ \cdots\ a'_{\epsilon+1,n-1}\\ & \quad \textbf{in}\ g'_{\epsilon+1}\ a_1\ \cdots\ a_{n-1}\ b_1\ \cdots\ b_{k_{\epsilon+1}}\ r,\\ \vdots & \vdots\\ C_{||T||}\ b_1\ \cdots\ b_{k_{||T||}} & \to \textbf{let}\ r = f\ a'_{||T||,1}\ \cdots\ a'_{||T||,n-1}\\ & \quad \textbf{in}\ g'_{||T||}\ a_1\ \cdots\ a_{n-1}\ b_1\ \cdots\ b_{k_{||T||}}\ r,\end{array}}\ \text{(R-RecCase)}$$

The idea here is that we assume that the set of constructors $C_1, C_2, \cdots C_{||T||}$ of T are sorted into two disjoint groups. The first ϵ constructors in this group are the nullary constructors (i.e. those with no arguments, for example `Nil`). The remaining constructors have each at least one argument. We can assume, without loss of generality[2], that all constructors C_i with $1 \le i \le \epsilon$ are those with no arguments, and the rest are those with at least one, forming these disjoint sets.

Synthesis for the nullary constructors is identical to the approach given in R-Case—we synthesise an auxiliary function, giving it all variables in scope (in this case just the function's arguments, as there are no constructor arguments to add). The new examples are produced by selecting exactly the examples for which the ith argument is the constructor in question.

Synthesis for the other constructors—those with some non-zero number of arguments—proceeds differently. We synthesise a **let** expression, which first calls f—the top-level function being synthesised—on some arguments $a'_{\ldots}$. These arguments are selected from the constructor's arguments $c_{1..k_j}$.

Introducing recursion in such a regimented fashion is advantageous for a number of reasons. Firstly, compared to the alternative of allowing arbitrary recursive calls, it allows us to reason more effectively about which examples will be required, and what the recursive arguments might be. Secondly, limiting recursive arguments strictly to constructor arguments allows us to make certain guarantees, most im-

[2]This is without loss of generality because we can freely reorder the constructors of a data-type.

portantly that infinite recursion will never occur (because we only ever call recursively on "smaller" arguments).

The **in** part of the **let** expression calls another auxiliary function, g', which is provided—as in R-Case—all of the functions arguments along with the constructor arguments. Additionally, it gets a final argument, r: the result of the recursive call we bound in the first part of the **let**.

The most interesting, and distinctly novel, part of the R-RecCase rule, is that the value of the recursive call of f in the new examples is a *thunk*: $\langle f\ a'_{j,1}\ a'_{j,2}\ \cdots\ a'_{j,n-1} | f \rangle$. Note here the thunk syntax, $\langle t|D\rangle$, means an expression—potentially not fully evaluated—which is "blocked" by some set of functions D.

The use of a thunk here is necessary since, as we are in the process of synthesising f, we cannot possibly evaluate the recursive call all the way to a value. Thus, the thunk depends on f.

The idea here is that, by use of R-EvalThunk, we can iteratively evaluate these thunks further and further as more functions become available. Eventually, they will become "similar enough" to what we want, and we can use R-Unify to "collapse" the thunks down into real values. This lets us "guess", in a sense, what the examples intended, and so we can get away with fewer provided examples.

Rule 5: R-Unify.

While R-RecCase introduces thunks into examples, we need a way to remove them. A thunk is an abstract thing, and if we want to apply most other rules, such as R-Trivial, we need concrete—*closed form*—arguments.

Just as R-RecCase is a generalisation of R-Case, R-Unify is a generalisation of R-Trivial. It considers one argument to the function being synthesised and, if this argument is a thunk (for at least one given example), attempts to consolidate this thunk with the corresponding output of the example.

We cover the details of this consolidation, or *unification*, of thunks in Section 3.4, but essentially it aims to find structural similarity between a thunk and a term, identifying a substitution that could be made to unify the two.

To this end, R-Unify rule looks to replace the function call blocking evaluation of the thunk (i.e. the function call which *makes* the thunk a thunk) with a value. This rule, therefore, only applies when there is just one dependency.

To keep the synthesis problem equivalent after reducing the thunk, we are forced to add a further example—the *unifying example*—which captures the idea of making this substitution. Since the blocking function will always be (after exhaustively applying R-EvalThunk) f—the function being synthesised—the unifying example for each thunk can be appended onto the end of the existing example list.

We begin by defining a notion of *unification*, a three-way relation between closed-form (i.e. fully evaluated) terms, thunks, and sets of examples:

$$x \sim \langle t|D\rangle \Rightarrow E$$

This is read as "the term x unifies with the thunk $\langle t|D\rangle$, producing a (potentially empty) set of unifying examples". For example, the term `[1, 2]` unifies with the thunk `<2 :: (2 :: f x)>`, yielding the unifying example `f x = []`. The relation is defined straightforwardly as:

$$\frac{x \equiv y}{x \sim \langle y|\emptyset\rangle \Rightarrow \emptyset} \qquad \frac{}{x \sim \langle f\ y_1\ \cdots\ y_k | f\rangle \Rightarrow \{f\ y_1\ \cdots\ y_k \to x\}}$$

$$\frac{1 \le i \le k \quad x_i \sim \langle y_i | d \subseteq D\rangle \Rightarrow E_i}{C\ x_1\ x_2\ \cdots\ x_k \sim \langle C\ y_1\ y_2\ \cdots\ y_k | D\rangle \Rightarrow \{e \in E_{1,2,\cdots,k}\}}$$

Most notably, an instance of a constructor C can unify with a thunk-instance of the same constructor, as long as the arguments can unify pairwise.

Additionally, a term unifies with a thunk if the thunk is identical (i.e. already evaluated to the same term, but still held in a thunk).

Finally, any term can unify with a function-call thunk, since we can simply add an example asserting that the function, when applied to its arguments, yields the term we're looking for.

With this, we can define the R-Unify rule as follows:

$$\frac{\begin{array}{l} 1 \le i < n, \\ \forall j \in [1,m], y_j \sim \langle x_{j,i} | f\rangle \Rightarrow \{E_j\}, \\ f\ :\ \tau_1 \to \tau_2 \to \cdots \to \tau_n \\ \quad \{\ x_{1,1},\ \cdots,\ \langle x_{1,i}|f\rangle,\ \cdots,\ x_{1,n-1} \quad \Rightarrow y_1 \\ \quad \vdots \\ \quad ;\ x_{m,1},\ \cdots,\ \langle x_{m,i}|f\rangle,\ \cdots,\ x_{m,n-1} \Rightarrow y_1\ \} \end{array}}{\begin{array}{l} f\ :\ \tau_1 \to \tau_2 \to \cdots \to \tau_n \\ \quad \{\ x_{1,1},\ \cdots,\ x_{1,i-1},\ y_1,\ \cdots,\ x_{1,n-1} \quad \Rightarrow y_1 \\ \quad \vdots \\ \quad ;\ x_{m,1},\ \cdots,\ x_{m,i-1},\ y_m,\ \cdots,\ x_{m,n-1} \Rightarrow y_m\ \} \\ \quad \cup\ E_{1,2,\cdots,m} \end{array}} \text{(R-Unify)}$$

This rule lets us unify a single argument in each example—argument i—against the desired output for the example. The condition that the argument is of the form $\langle x_{m,i}|f\rangle$ means that $x_{m,i}$ is a thunk which depends only on f.

If this is the case, we can replace each example k with a new example where the ith argument is replaced by the desired output, y_k. This is the result of the unification: we can think of this as replacing the function calls in the ith argument with some value which would make them unify with the output value, but this—by definition of unification—will have the same result as just swapping out the whole term.

An important note is that this rule only applies when one single example, E_j, is "produced" by the unification (for each existing example). This is not an intrinsic limitation of the method, but we find in our testing that it is sufficient. Essentially this means that we can only unify thunks with a single function call in them, and while this could be generalised in future revisions, it is enough for all of the functions that we are interested in.

Finally, we append each of the unifying examples, $E_{1,2,\cdots,m}$, to the set of examples. This justifies the substitution we made earlier in the existing examples—replacing $x_{m,i}$ with y_m—as these additional examples assert that this substitution really is valid.

Rule 6: R-Void.

The sixth rule covers one very specific case—the case where there are *no* examples. Such a situation can occur after a case-split (either from R-Case or R-RecCase), if a constructor has no associated examples.

In this situation, there are essentially two possible courses of action. We could "fail", and say that the synthesis was not possible at this point; here, we'd backtrack and try another tactic. However, as we are interested in interactive synthesis

and hence incomplete programs, it makes sense instead to produce a hole. This way, the programmer is informed that the synthesis was "not successful", but they are given the option to complete it themself.

This rule is defined very simply as:

$$\frac{m = 0}{f\ a_1\ a_2\ \cdots\ a_{n-1} = ?} \qquad \text{(R-Void)}$$

Here m, as before, is the number of examples; also, ? is used in the Fugue language's syntax to denote a hole.

A hole is an interactive language feature through which the programmer can explore possible fill options, and the Fugue interactive REPL will suggest relevant options.

We have to be careful about introducing holes, since in a sense they could "always" apply (in Fugue, $\forall\tau.\ ?\ :\ \tau$). Only allowing them when there are no examples is a step towards this, but they would still dominate. We discuss tactics to temper the use of holes in Section 4.

Rule 7: R-Homo.

Our final rule, R-Homo, provides a route for synthesis in the case that none of the other rules were applicable. Sometimes, all of the examples lead to the same output, but it isn't clear how their inputs lead to this value. In these cases, it may be useful to have a way of saying "ignore the inputs; we just want that output".

To that end, this rule is applicable whenever all of the outputs are identical. Notably, this then applies for *any* synthesis problem with just one example[3]. This leads to potential problems which we return to in Section 4, as this rule can easily become too "powerful" and dominate.

This rule is defined as:

$$\frac{\begin{array}{l} m > 0 \\ y_1 = y_2 = \cdots = y_m \end{array}}{f\ a_1\ a_2\ \cdots\ a_{n-1} =\ y_1} \qquad \text{(R-Homo)}$$

This is useful for example when synthesising the `length` function, which requires the base-case input `[]` to produce the apparently unrelated length `0`.

3.3 Other Rules

On top of the synthesis rules defined above, we need some additional rules to transform examples into other examples of a slightly different shape so that the synthesis rules apply. We discuss these rules in this section.

Rule 8: R-EvalThunk.

Firstly, we need a rule for evaluating thunks when new functions become available during synthesis. In the worked example (Section 3.1) this was implicitly done between applying the other synthesis rules, but for completeness we include an explicit rule to do this.

$$\frac{\begin{array}{l} f\ :\ \tau_1 \to \tau_2 \to \cdots \to \tau_n \\ \quad \{\ x_{1,1},\ \cdots,\ \langle x_{1,j} \mid D_1\rangle,\ \cdots,\ x_{1,n-1} \quad \Rightarrow y_1 \\ \quad ;\ x_{2,1},\ \cdots,\ \langle x_{2,j} \mid D_2\rangle,\ \cdots,\ x_{2,n-1} \quad \Rightarrow y_2 \\ \quad \vdots \\ \quad ;\ x_{m,1},\ \cdots,\ \langle x_{m,j} \mid D_m\rangle,\ \cdots,\ x_{m,n-1} \Rightarrow y_m\ \} \\ f' : \tau'_1 \to \tau'_2 \to \cdots \to \tau'_{n'} \\ \langle x_{i,j} \mid D_i\rangle[f'] = \langle x'_{i,j} \mid D'_i\rangle,\ \forall 1 \le i \le m \end{array}}{\begin{array}{l} f\ :\ \tau_1 \to \tau_2 \to \cdots \to \tau_n \\ \quad \{\ x_{1,1},\ \cdots,\ \langle x'_{1,j} \mid D'_1\rangle,\ \cdots,\ x_{1,n-1} \quad \Rightarrow y_1 \\ \quad ;\ x_{2,1},\ \cdots,\ \langle x'_{2,j} \mid D'_2\rangle,\ \cdots,\ x_{2,n-1} \quad \Rightarrow y_2 \\ \quad \vdots \\ \quad ;\ x_{m,1},\ \cdots,\ \langle x'_{m,j} \mid D'_m\rangle,\ \cdots,\ x_{m,n-1} \Rightarrow y_m\ \} \end{array}} \qquad \text{(R-EvalThunk)}$$

This rule evaluates, for some argument index i, the respective argument for each example in a function specification. These arguments must be thunks to be evaluated, but if some arguments are fully-evaluated values (i.e. not thunks), they can be temporarily transformed into thunks using R-ThunkIntro and R-ThunkErase—evaluating a fully-evaluated value as if it were a thunk is always possible, and makes no change to the value itself.

The evaluation of these thunks may be done against any function, f', defined earlier during synthesis. We place the restriction, however, that all thunks are evaluated against the same function. This is for practical purposes, though in theory there is nothing stopping an alternate formation being devised where each thunk can be evaluated against a different function. Indeed, we can approximate this behaviour with multiple uses of R-EvalThunk anyway.

Rule 9: R-ThunkIntro.

This rule, R-ThunkIntro, along with its "inverse" R-ThunkErase, are utilities which allow fully-evaluated arguments to be transformed into thunks and fully-evaluated thunks to be transformed back into regular values. These are useful not only for applying certain rules which require all arguments to be thunks (e.g. R-EvalThunk), but also R-ThunkIntro is one of the two methods (the other being R-Unify) by which a thunk can be removed.

Firstly, R-ThunkIntro is defined simply as:

$$\frac{x_{i,j} = t,\ t \text{ is a closed term}}{x_{i,j} = \langle t \mid \emptyset\rangle} \qquad \text{(R-ThunkIntro)}$$

This rule acts upon one argument (j) of one example (i), and transforms it—if it is a fully-evaluated term—into a thunk with no dependents.

Rule 10: R-ThunkErase.

R-ThunkErase, similarly, is defined as:

$$\frac{x_{i,j} = \langle t \mid \emptyset\rangle}{x_{i,j} = t} \qquad \text{(R-ThunkErase)}$$

This rule allows us to collapse a thunk which has no dependents into a regular fully-evaluated term. This is possible because we know that if a thunk has no dependents that it must be fully evaluated.

[3]Though, it is not applicable in the case of zero examples, for obvious reasons.

3.4 Thunks

Thunks are a core component of our synthesis technique, and so we set aside this section to deal with them in more detail.

Definition.

A thunk, th, is a pair:

$$th ::= \langle t \mid D \rangle$$

The pair consists of a *thunk body* t and a set of functions on which the evaluation of t depends.

We can exhaustively define thunk bodies as one of four possible, very specific, cases. A thunk body, t, is defined as:

$$\begin{aligned} t ::= & \; x \\ & \mid C \; t_1 \; t_2 \; \cdots \; t_k \\ & \mid f \; x_1 \; x_2 \; \cdots \; x_n \\ & \mid \textbf{let } y = \langle t' \mid D' \rangle \textbf{ in } f \; x_1 \; x_2 \; \cdots \; x_n \; y \end{aligned}$$

Here, the variables $t, t_1, t_2, \cdots$ range over thunk bodies and x, y, etc range over terms of the underlying language, in this case FUGUE.

First off, a thunk body can trivially be a term, x. In this case, the thunk is fully evaluated, but for pragmatic reasons it is sometimes useful to hold such a value in a thunk anyway.

Secondly, a thunk body can be some constructor applied to k other thunk bodies.

Alternatively, a thunk body can be a function, f, applied to some number of *terms*, not thunks. (In general, since these functions will typically represent recursive calls, it is not useful here to allow other thunks as arguments to function call thunks.)

Finally, a thunk body can be a **let** expression. The bound value in the expression is itself a *thunk* in and of itself (as opposed to a thunk body), and as such has its own dependents. The body is a function call to, again, some number of terms, but the final argument is *always* the variable bound in the **let** expression. This type of thunk body arises from the R-RECLET synthesis rule, and is just general enough to represent all possible thunks we would need.

Evaluation.

Thunks exist to be evaluated further, with the aim to eventually have no dependents and thus be able to transform, via R-THUNKERASE, into regular non-thunk values.

This evaluation is triggered by the R-EVALTHUNK rule, which applies the definition of a single function on which the thunk depends. How exactly this function is applied depends on the nature of the thunk's body, and so we define this evaluation using the following rules:

$$\frac{th = \langle x \mid \emptyset \rangle, \quad f : \tau_1 \to \tau_2 \to \cdots \to \tau_n}{th[f] = th}$$

$$\frac{th = \langle C \; t_1 \; t_2 \; \cdots \; t_k \mid D \rangle, \quad f : \tau_1 \to \tau_2 \to \cdots \to \tau_n}{th[f] = \langle C \; t_1[f] \; t_2[f] \; \cdots \; t_k[f] \mid (D \setminus \{f\}) \rangle}$$

$$\frac{th = \langle f \; \tau_1 \; \cdots \; x_n \mid \{f\} \rangle, \quad f : \tau_1 \to \cdots \to \tau_n, \quad x_i : \tau_i}{th[f] = \langle f \; \tau_1 \; \cdots \; x_n \mid D_f \rangle}$$

$$\frac{th = \langle g \; \tau_1 \; \cdots \; x_n \mid \{f\} \rangle, \quad f : \tau_1 \to \cdots \to \tau_n, \quad f \neq g}{th[f] = th}$$

$$\frac{th = \langle \textbf{let } y = th' \textbf{ in } f \; x_{1 \cdots n} \; y \mid D \rangle, \langle x' | \emptyset \rangle = th'[f]}{th[f] = \langle f \; x_1 \; x_2 \; \cdots \; x_n \; x' \mid \{f\} \rangle}$$

$$\frac{th = \langle \textbf{let } y = th' \textbf{ in } f \; x_{1 \cdots n} \; y \mid D \rangle, \langle x' | D' \neq \emptyset \rangle = th'[f]}{th[f] = \langle \textbf{let } y = \langle x' | D' \rangle \textbf{ in } f \; x_{1 \cdots n} \; y \mid D' \rangle}$$

The first rule here deals with applying a function to a thunk which is already a fully-evaluated term x. In this case, unsurprisingly, nothing changes: $th[f] = th$.

The second rule deals with the application of a function to a thunk whose body is a constructor applied to k arguments. The thunk depends on the set of functions D, which may or may not include f. If, then, we apply the function recursively to each constructor argument, we have our new thunk body. Then, since we know that f has been applied throughout the body, we can remove it from the set of dependencies of the thunk.

The third and fourth rules define application of a function to a thunk which is itself a function call. There are two cases, either the function at hand is the same function as the thunk wants to call; or, it is not. If it is, we can call this function on the thunk's arguments—this, incidentally, is one reason why it's important to keep thunk function-call arguments as fully-evaluated terms: this way, we can always call the function just as defined by the underlying language's function call mechanics.[4]

The final two rules define application of a function to a **let-in** thunk. We begin by applying the function inside the thunk, to the sub-thunk th' which is the bound variable of the **let** expression. There are then two cases: either this thunk no longer has any dependent functions (in which case, this application reduced it to a value), or it has not reached this point yet. If the bound variable is now a value, we can collapse the **let** expression to its body, passing in the now-evaluated value as the final argument to the function call—x'. Otherwise, we are forced to keep the thunk as a **let** form, though we update the set of dependent functions. Either way, we are closer to a value overall.

4. FANTASIA & PRACTICALITIES

We have implemented the techniques described in this paper in an extension of the FUGUE compiler, which we refer to as FANTASIA[5].

In this section, we discuss some technical details relating to the implementation of the FANTASIA algorithm. We try for the most part to discuss these things in a language-agnostic manner (both in terms of the implementation language, and the language which we are synthesising).

4.1 Iterative Deepening

We aim to find the smallest possible solution to synthesis for the given set of examples. As a result, we employ a

[4]In actual fact, our implementation is more complex than this, but the theory is the same. The difference is that we represent functions, during synthesis, as one of a set of possible function bodies. In this case, we are forced to define a notion of function application for these specific body types.

[5]A note on naming: FANTASIA, like FUGUE, is a reference to a common form of music. A fantasia is an imaginative, improvisational piece; similar, therefore, to a program synthesiser. Also, many baroque composers—famously J. S. Bach—wrote a number of pieces titled *Fantasia and Fugue*.

breadth-first search approach by applying, recursively, the rules defined in Section 3.2. In this way, we can see the synthesis as producing a parse tree, of sorts, with the synthesis rules being the grammar's productions.

We first search for solutions of a "depth" 1, meaning the depth of—using the grammar analogy again—our parse tree. We then search for solutions up to a maximum depth 2, then 3, and so on.

This has a number of benefits. Firstly, as explained in [16], the smallest program is likely to be the one which generalises best to unseen inputs. This is because a small program cannot make the trivial solution, which would be a **case** over each given example, and so is forced to find some common structure or pattern in the provided information.

Secondly, this actually allows us to synthesise holes. If we did not use an iterative deepening procedure, most branches would eventually reach a point where no examples were left. This would lead to a solution which, while correct, contains a large number of holes. The problem here is one of overgeneralisation: we want to prioritise solutions where the synthesiser *knew* what to do, and these situations are those where holes are not produced.

Thirdly, and related, this iterative deepening approach gives us an intuitive way to prioritise "good" rules, while penalising "bad" rules. We come back to this in more detail in Section 4.3.

4.2 Multiple Synthesis Results

For most of this paper, we have been discussing synthesis as if it were a process which takes some examples and emits exactly one program. This, however, is not always useful. As much as we would like the first solution to be the best one, this is sometimes not the case. In these situations, we would like to be able to ask the synthesiser for the "next best" solution.

In theory, what we would like to do then is try *all* applicable rules at each point during synthesis, and return all combinations of these. In practice, we implement this using Haskell's list monad. As a result, we can write the program mostly as if it were the single-output form, and the monad handles the specifics for us.

This is elegant, but we do have some issues if we do just this. Most notably, we are given some duplicate outputs. This is for a number of reasons, although most of the duplicates come from the fact that synthesising at depth n will also yield all results from depths $< n$. As a solution, we simply remove any duplicate solutions, although a more elegant potential solution is discussed in Section 5.1.

A final note on the multiple-solution implementation is that we are forced to order the application of our rules in some way. We choose the following order:

1. R-Unify first. If we can unify an argument, it is almost always a good idea to do this before proceeding any further.

2. R-Trivial second. If a solution is trivially found in one of the function's arguments, there is usually no reason not to use it.

3. R-Constr comes before the two **case**-introduction rules, because it leads to simpler solutions and is usually preferable when applicable.

4. R-RecCase is one of the last rules to attempt, since it adds a lot of complexity which may not be needed. We try it before the regular R-Case, though, perhaps unintuitively since R-Case is a special case of it. We do this because recursive solutions are often more useful.

5. R-Case the last of the "normal" rules, we attempt a regular **case**-split.

6. R-Void only applies if there are zero examples, and the other rules all need at least one, so this rule could technically come anywhere without having any effect on the result.

7. R-Homo is attempted, finally, but only if none of the other rules were applicable. The specific requirements for using this rule are discussed in Section 4.3, but we have to be very careful with its use in general because it can lead to nonsense solutions quite easily otherwise.

4.3 Prioritising Good Rules

Most rules are "good", in that we would generally like to use them whenever they're applicable. R-Constr, for example, is never "bad" to apply. R-Void and R-Homo, on the other hand, can be problematic.

Using R-Void to introduce a hole still leads to a valid solution, but in some way this solution is "less useful" than an alternative without a hole, because the user has to do additional work to complete it.

Similarly, a solution which uses R-Homo will tend to be generalise more poorly to unseen inputs than one which doesn't, and can therefore be seen as "bad".

As a result, our implementation of Fantasia explores some methods to avoid these rules unless they are necessary. We take two precautions.

Firstly, we assign these two rules a "cost", in the sense that they require more remaining depth in order to be applied. These costs are adjustable parameters, but we define the cost of R-Void to be 1 and of R-Homo to be 3. The cost of other rules, though we don't explicitly consider this in our implementation, is 0.

When we go to apply one of these rules, we first check that we have "enough depth left". This is related to iterative deepening. If the maximum depth we are willing to go (during this stage of iterative deepening) is, say, 5, and we have reached depth 3, we would not be allowed to use R-Homo. We could, however, use R-Void, since we have 2 remaining depth.

In general, for a current depth d, a cost C, and a maximum depth $d_{\max}$, the following condition must hold for a rule to apply:

$$d + C \leq d_{\max}$$

This solves our problem in a lot of cases, as essentially now the solutions which use these two rules are held back by some amount of depth iterations. As a result, more useful solutions tend to appear earlier on.

Still, R-Homo is too powerful. It is applicable in too many cases, and so we additionally say that it can only apply if no other rule was applicable. These restrictions temper use of R-Homo sufficiently in our testing. Though they are arbitrary, the depth cost is parameterised and so can be customised accordingly.

4.4 Winding & Finalising

The final implementation detail we discuss is the final processing of the synthesised functions, for final presentation to the user.

Winding.

Since our synthesis technique generates excessive auxiliary functions, we need a cleaning-up step to fold these back into as few functions as possible. To prevent confusion between this process and the other meaning of "folding" in functional programming, we refer to it as "winding".

Winding a set of functions starts from a root function—in our case the function which we set out to synthesise—and proceeds recursively down the tree. Throughout, we keep track of a state. The state holds the set of functions, which can be modified whenever necessary.

If, while recursively descending the function body, we find a function call, there are two cases. If the function being called is directly recursive (i.e. contains an *explicit* call to itself), we simply continue down the tree as before, and ignore this call. Otherwise, we inline the function call, and recursively unwind again starting from this new function as the root, before continuing on.

If we find a variable which refers to another function, we begin a new winding process starting from that variable instead. This new winding process uses the same state, so the results are combined.

Finally, we are left with hopefully less functions than we started with. We can remove all functions which are no longer required (i.e. those to which there is no path of calls from our root function). It is also worthwhile at this point removing any unused arguments from auxiliary (i.e. non-root) functions, since our techniques typically introduce many arguments which we don't need.

Simplification.

We then move to simplify the remaining functions, using some simple rules. This, again, isn't strictly necessary, but we find that a few small rules here can help a lot.

Firstly, for any **let** or **let rec** expressions, we consider the three possible cases.

1. The **let** body contains no references to it's bound value. In this case, we replace the entire **let** expression with just its body.
2. The **let** body contains exactly one reference to its binding. In this case, we replace the **let** expression with its body, but we substitute the binding into the body first.
3. The **let** body contains more than one references to its binding. In this case, we do nothing.

We also simplify **case** expressions. If all branches of a **case** analysis expression have the same value, we replace the entire **case** expression with this value.

5. ANALYSIS & CONCLUSION

Fantasia is an analytical synthesis engine. We could also refer to it as "reconstructive", in the sense that (in the vast majority of cases) the solution comes directly from the structure of the examples (as opposed to some other analytical techniques, and most enumerative techniques, which make more arbitrary guesses at expressions).

These two points result in very fast synthesis compared to other techniques, even similar ones. In our naïve implementation in Haskell, with no performance optimisation, we achieve real-time synthesis up to depths of 6 or 7 (depending on the nature of the examples). As a result, we are able to re-run synthesis *immediately* as the user enters each character of their examples, without any perceptible delay.

Contrast this to, for example, SnipPy [5] which can synthesise within seven seconds programs "of up to height 3". Real-time synthesis of such large programs opens up the possibility to a powerful editing paradigm where the programmer can edit, in real-time, their examples, and then provide new examples based on the results.

This is augmented further by two things. Firstly, Fugue's use of holes—and Fantasia's synthesis of them—lets the system give programmers more detailed and informative feedback during synthesis. For example, if we begin by giving the single-example problem of $f : [a] \to [a]$ $\{[] \Rightarrow []\}$, another synthesiser might give back the single program, $f\ a = a$. Fantasia does too, since this is a correct result, but it gives a second result: a **case** split over the input list. The case of the `Cons` constructor is simply given as a hole, since no relevant examples were given. This, however, lets the user know that this is a *possibility*, and serves to prompt them to add the relevant example if they desire.

The use of holes in a synthesis environment like this also shines when we are dealing with partial functions, for example `head`. Many synthesisers would refuse to synthesise such a function, since no example could be given for the empty list; or, they would silently return an `undefined` in this case. Fugue does not have an `undefined` value like Haskell, so we had to take a different approach here. A hole is returned for the "impossible" case of the head of an empty list, which tells the user "you have to come up with a value to put here, either some undefined value or perhaps change the return type to `Maybe a` and return `Nothing`". h Furthermore, Fantasia lends itself to iteratively synthesising as many results as the user wants. This again improves the experience of the real-time synthesis "conversation" the programmer will have with the system, as in general the first result will appear very quickly (or instantaneously), while some larger—potentially still useful—results may take a little longer. The provision of quick results keeps the user from waiting too long, especially when those small programs are often the ones desired, but the nature of the problem is that we can leave the synthesiser running in parallel (in another thread, if we want) and provide new results if and when they are constructed.

To conclude on the performance side of things, a real-time synthesis approach as made possible by Fantasia leads to a conversational paradigm between the user and the system, which is hard to find in other existing synthesis systems. This conversational approach makes the synthesis of small functions more feasible, since it may be faster to give one or two examples than to hand-write even these small programs. It also makes the synthesis of large programs more feasible, since holes can indicate where additional examples might be needed, and the fast feedback loop makes these larger programs possible in the first place.

Fantasia can synthesise polymorphic functions. This is useful firstly because polymorphic functions are useful in general, for example we might want to synthesise a `length` function which works on lists of any type. Polymorphic functions hold a more subtle benefit, though, in the context of

program synthesis. If the type of an argument is polymorphic (i.e. universally quantified), the synthesiser knows that it cannot possibly use the *value* of this argument in the synthesis; it can only move the value around as a "black box".

This lets us tacitly ask the synthesiser to ignore the specific value of an argument, which is useful for example in the `length` function. If we use the example $[1] \Rightarrow 1$, the synthesiser might be tempted to give us a function analogous to `head`. However, if we specify the type $[a] \rightarrow \text{Int}$, it is forced to generalise more effectively. In FANTASIA, this example plus $[] \Rightarrow 0$ is enough to fully specify `length`.

Another helpful property of FANTASIA is its regimented way of implementing recursion. Recursion is only allowed after a **case**-split, and the recursive arguments we supply *must* come directly from the pattern-match following the **case** analysis. This guarantees that functions are not infinitely recursive, and as a result, guarantees that all synthesised programs will terminate.

This is an extremely powerful property which, while we do not prove here, the proof is trivial and follows the narrative above—essentially, if we can only recurse on parts of the constructor, then—as long as the value we are recursing on is not itself infinite—we are forced to call the function on strictly "smaller" values. We will eventually reach the base case then, by definition, as this is the smallest case.

We also note that, though this bureaucracy involved with introducing recursion may seem restrictive, in does in fact cover most cases where we would need recursion. The pattern of **case** analysis followed by a different recursive call for each branch is more powerful than, for example, a simple "folding" operation.

One notable drawback of our technique, however, is that it does not make use of pre-defined functions. For example, there would be no way for a synthesised program to make use of the built-in `(==) : a → a → Bool` function to check if two values are the same. Or, no way to add two integers together without synthesising an auxiliary function for addition (which, while possible, will increase the synthesis depth required, and thus take a long time).

We explicitly opt to not use these additional functions, or "components" as [17] refers to them, for primarily performance reasons. As demonstrated in [17], and as discussed by [8] and [4] (as a recurring problem with enumerative techniques), the possibility of using arbitrary pre-defined components increases the search space for synthesis greatly.

Naïvely, one might expect that allowing the use of one additional function will give one more possibility at each "node" in the search tree. However, polymorphic functions make the problem exponentially worse. As discussed in [8], for example, a definition of the function `const : a → b → a` effectively introduces an "unbounded" number of distinct functions, one for each pair of types.

Some solutions, such as [17], work around this somewhat by restricting the set of components the synthesiser is allowed to use. For example, to synthesise a function to count the nodes in a tree, they provide the set of component function `(+)`, along with the component literals `0` and `1`. A similar approach is taken in [5], where components are extracted from the examples directly (for example if an example contains the literal `0`, that will be present in the set of components). Notably, however, [5] does not need to consider function components such as `(+)`, since it is enumerative.

Also, [16], an analytical synthesiser similar to ours, uses component functions to great success. They use a "guessing" method to introduce calls to these components functions, a possibility which we discuss later in Section 5.1.

We can see then that there is a tradeoff between performance and ease of use, depending on whether we allow the full set of possible components (slow, but transparent to the user) or ask the user to provide a set of components (faster, but less convenient). Unfortunately, the design goals of FANTASIA were such that we want as much performance *and* ease of use as possible, and as a result we chose to not allow these extra components to be used in synthesis at all. This does limit the system somewhat in which programs it can generate, but we discuss potential solutions to this in Section 5.1.

5.1 Future Work

There are lots of exciting areas which can be explored relating to the techniques we have shown in this paper. In this section, we present a selection of these.

Enumerative Synthesis. It is likely that an additional synthesis rule implementing an enumerative synthesiser, perhaps named R-ENUMERATIVE, would provide a number of benefits to our (primarily analytical) synthesiser. Such a rule might look for any possible fully-formed expressions (though likely only expressions within the depth allowable by the current state of iterative deepening) and "test" these against the example-set.

A rule implementing enumerative synthesis as a search over all (or some subset of) possible expressions would likely be the end of the line; any function calls it produces would almost definitely not be able to use the analytical synthesis techniques to produce their arguments, as doing so would likely require an *inverse* of the function to produce the subsequent examples.

This is probably the improvement to the system which would have the biggest impact. Also, since no further synthesis sub-goals would be produced from such a rule, it should not slow down the synthesis significantly. Such an approach would be similar to the divide-and-conquer enumerative synthesis proposed in [2].

Higher-order Function Synthesis. Synthesis of higher-order functions is something which FANTASIA does not currently support, but would be useful. Straightforwardly, adding a rule which synthesises applications of functions found in arguments to some set of other arguments would be an important step towards this goal, and would make some higher order functions (e.g. `map`) synthesisable.

We may also like to provide examples in general terms of "any function argument", such as $\{f,\ [1,2] \Rightarrow \langle [f\ 1, f\ 2] \rangle\}$. This would require thunks to be allowable example *outputs* as well as just inputs.

Specialised Rules. Some programming idioms are so ubiquitous that a huge number of functions can be implemented just using them. In many cases, these are higher-order functions, such as `map` and `fold`, which abstract away some specific form of recursion. It may be worthwhile to implement rules into FANTASIA for some of these, as a `map` can be significantly more efficient—in terms of program size—than re-implementing a similar procedure each time it is required.

Synthesising Infinite Data. We noted earlier that, due to the structured way we introduce recursion into synthesised programs, infinite recursion is not possible. We suggested that this is a good thing, as it is in most cases—why would we want a program which doesn't terminate? Sometimes,

however, due to lazy evaluation of programs, this might be desirable. It would be interesting to look into techniques and extensions of FANTASIA which would allow for the synthesis of, say, an infinite list of Fibonacci numbers.

5.2 Conclusion

In this paper, we have introduced a novel synthesis technique that is powerful, elegant, simple, and extensible. We have solved a standing problem present in similar techniques ([16], [10]) of the synthesis of recursive functions with minimal examples, by using thunks to guess at recursive behaviour during synthesis.

We have shown that our theory works in practice in our implementation, FANTASIA, which successfully synthesises non-trivial FUGUE programs. In our implementation, we show how the real-time synthesis speed can lead to an exploratory, conversational approach to synthesis.

While FANTASIA is a prototype at this stage, it shows promising signs that the techniques are useful, and we have discussed a number of ways in which it can be improved and extended in the future.

6. REFERENCES

[1] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 934–950. Springer. doi:10.1007/978-3-642-39799-8_67.

[2] R. Alur, A. Radhakrishna, and A. Udupa. Scaling enumerative program synthesis via divide and conquer. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 319–336. Springer. doi:10.1007/978-3-662-54577-5_18.

[3] M. V. Eekelen. Systematic search for lambda expressions. In *Trends in Functional Programming*, volume 6, pages 111–126. Intellect Books. Google-Books-ID: R7urDwAAQBAJ.

[4] K. Ferdowsifard, S. Barke, H. Peleg, S. Lerner, and N. Polikarpova. LooPy: interactive program synthesis with control structures. *Proc. ACM Program. Lang.*, 5:153:1–153:29. doi:10.1145/3485530.

[5] K. Ferdowsifard, A. Ordookhanians, H. Peleg, S. Lerner, and N. Polikarpova. Small-step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 614–626. ACM. URL: https://dl.acm.org/doi/10.1145/3379337.3415869, doi:10.1145/3379337.3415869.

[6] A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251. Publisher: Cambridge University Press.

[7] GitHub. GitHub copilot. URL: https://github.com/features/copilot.

[8] Z. Guo, M. James, D. Justo, J. Zhou, Z. Wang, R. Jhala, and N. Polikarpova. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.*, 4:1–28. URL: https://dl.acm.org/doi/10.1145/3371080, doi:10.1145/3371080.

[9] M. Hofmann. Schema-guided inductive functional programming through automatic detection of type morphisms. Accepted: 2019-09-19T15:35:55Z Journal Abbreviation: Schemagesteuerte Induktive Funktionale Programmsynthese durch Automatische Erkennung von Typmorphismen. URL: https://fis.uni-bamberg.de/handle/uniba/264.

[10] S. Katayama. An analytical inductive functional programming system that avoids unintended programs. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation - PEPM '12*, page 43. ACM Press. URL: https://dl.acm.org/doi/10.1145/2103746.2103758, doi:10.1145/2103746.2103758.

[11] S. Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In *Lecture Notes in Computer Science*, volume 5351, pages 199–210. Springer.

[12] S. Katayama. MagicHaskeller on the web: Automated programming as a service. In *Haskell Symposium 2013*.

[13] N. Mitchell. Hoogle overview. *The Monad. Reader*, 12:27–35.

[14] OpenAI. ChatGPT: Optimizing language models for dialogue. URL: https://openai.com/blog/chatgpt/.

[15] OpenAI. GPT-4 technical report. URL: http://arxiv.org/abs/2303.08774, arXiv:2303.08774[cs], doi:10.48550/arXiv.2303.08774.

[16] P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 619–630. Association for Computing Machinery. doi:10.1145/2737924.2738007.

[17] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 522–538. ACM. URL: https://dl.acm.org/doi/10.1145/2908080.2908093, doi:10.1145/2908080.2908093.