Fantasia: An Interactive Synthesiser of Polymorphic Recursive Functions

Interim Report

MSci Individual Research Project (COMP4027)

Written by:
Zac Garby (20176069)
psyzg5@nottingham.ac.uk

Supervised by:
Professor Graham Hutton
graham.hutton@nottingham.ac.uk

School of Computer Science
University of Nottingham

2022/2023

University of Nottingham
UK | CHINA | MALAYSIA

Contents

1 Background

Programming is usually thought of as a creative activity, but in a lot of cases this is not so true. 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.

Hopefully this gives some idea of the scope of application of program synthesis methods. To be more specific, when I refer to program synthesis I mean in particular the process of automatically deriving a program from some specification. Many things lie under this definition, the most obvious of which being a process which takes a description (perhaps some examples, perhaps some mathematical properties, perhaps a natural language description) and outputs a working function.

Interesting, a standard compiler also fits this definition, the "specification" being the source text and the "program" being the machine code, as do many other types of programs. So, to narrow down the topic further, I will be focusing in this project on program synthesis based on input-output examples.

To explain why I will be focusing on synthesis based on input-output examples, I will explain what I would consider a fundamental problem of program synthesis.

Consider what we want from a program. The main thing that comes to mind is that it is correct—that is, it performs the function that it is designed to do, always, and for every possible input. If a synthesiser is to guarantee this, it needs a complete, formal specification of the desired program in the first place. It turns out that this is as hard, or indeed harder, than writing the program in the first place.

The alternative would be to give a vague description and then manually verify the program, by inspection, after it has been synthesised. This, too, is often slower than writing the program from scratch, because to prove something correct which has been written by somebody (or some*thing*) else requires insight into *how* they solved the problem, i.e. the algorithm they have chosen, plus then proving that their method is correct. Unit tests or property-based testing could be used here, but then these will have to be written by hand.

While it may seem that this dilemma renders program synthesis useless, I can see at least one solution to the problem. The second case of the dilemma—giving a vague description and manually verifying—is deceptive. My hypothesis, at least, is that verifying a program by manual inspection gets harder at a rate greater than linear as the size of the program increases. In other words, a large program is disproportionately difficult to verify by hand compared to a smaller one. This is my conclusion based on a number of factors which I will come back to, but for now suffice to say that this suggests that small enough programs are verifiable practically instantly by a programmer, and so may be amenable to synthesis.

This research project explores the effectiveness of program synthesis as a tool for writing real programs, and methods by which we can make this more useful. Some of these

I have already discussed, but another point I am interested in is the interplay between interactivity and program synthesis.

Interactive editing is a way of writing programs which moves away from the traditional view of programming as an individual activity, checking in with the compiler once the program has been written, to an interaction (or conversation) between the programmer and the programming language. The later parts of my project will focus on the integration of program synthesis into an interactive editing system, working towards an implementation of the *Synthesis-Aided Live Programming* [5] editing paradigm.

2 Related work

There are a number of ways to categorise approaches to program synthesis. In this project, I'll be focusing specifically on methods which are applicable to the synthesis of functional programming languages, but there are still many options to consider.

2.1 Enumerative and Analytical Approaches

Approaches to synthesis for functional programming languages can be split into two categories ([8], [10]): *enumerative* and *analytical*. Both are based on input-output examples, meaning the user of the system provides some examples that their program must satisfy. For example,

$$
\begin{aligned}
&\mathit{stutter} \ : \ [a] \to [a] \\
&\qquad \{ \ [1,2,3] \Rightarrow [1,1,2,2,3,3] \\
&\qquad ; \ [] \qquad\quad \Rightarrow [] \ \} \\
&\mathit{stutter}\ \mathit{xs} \ = ?
\end{aligned}
$$

Other syntaxes for the same idea are used in the literature, but all essentially come down to the idea that a number of inputs are provided (in this case [1, 2, 3] and [], and their corresponding desired outputs are specified (here [1, 1, 2, 2, 3, 3] and [], respectively).

Enumerative approaches to synthesis would go about this problem by "blindly" listing off a series of potential function definitions, then testing these against the examples until they find one that fits. Typically, the definitions will be generated in such a way that smaller options are considered first, so that the smallest possible definition will be produced by the end. With these approaches, recursion is typically introduced via higher order recursion combinators, like *foldr* to encapsulate the idea of a catamorphism—or a *fold*. The generation of potential solutions is usually guided by type information, which makes it particularly suitable in a language with a strong type system.

Analytical approaches make less use of typing information, and instead analyse the examples' structure to procedurally build up the solution to the synthesis problem. For example, it may be the case that if all given examples result in a particular constructor instantiation (say, *cons* of two arguments) then the analytical synthesiser could guess an application of this constructor as the next "step" in the solution. The examples will be refined, either by being split apart into different cases (or *worlds* [16]) or otherwise dismantled. Backtracking will always be required in such techniques, since only a best guess can be made at each point. This fact, it turns out, makes *partial* synthesis—the process of

synthesising the next few "steps" in a function definition, rather than the whole thing—difficult.

Typically, enumerative methods will be better suited to synthesis of terms composed of standard library functions and other existing components, composed with each other in some way. For example, *Hoogle+*, described in [7], synthesises compositions of Haskell library functions to meet a given specification (analogous to how *Hoogle* [13] searches for single Haskell functions which on their own meet a specification—usually a type).

Similarly, *SnipPy* ([4], [5]) use an enumerative approach to synthesise programs; they refer to the technique as a "*generate-and-test synthesizer*". SnipPy is interesting with respect to my project in its own right as well, being essentially an interactive synthesis environment. SnipPy's goal is to augment the Python editing experience with live programming features (in this case the live viewing of intermediate values) as well as a synthesiser. They refer to such an editing paradigm as "Synthesis-Aided Live Programming".

2.2 Machine Learning

Many modern program synthesis techniques rely on machine learning. Typically, this would be training a language model to produce code based on either a natural language description or to continue existing human-supplied code. A popular example of this is GitHub's *Copilot*, an AI model based on GPT-3 [3] which is specialised to generating code snippets. It is trained on code from a huge number of open source projects from GitHub and is designed to act as a "pair programmer" in interactive editing sessions.

Another similar, more recent example is OpenAI's *ChatGPT* [1], which uses similar technology but applies it to a conversational context, so the programmer can interact in a more natural way. For example, the user might ask "Write me a program which calculates the Fibonacci sequence", and then later clarify "Oh, but do this in C". This is a powerful interaction paradigm, as any flaws which the programmer identifies can usually be rectified automatically. Additionally, *ChatGPT* understands more than just code, being able to provide natural-language feedback to the programmer to aid understanding.

This does, of course, require that the programmer can notice any flaws in the AI's generated code, which is not trivial—especially for large programs. This is a problem inherent in almost all forms of program synthesis which rely on informal descriptions, but it is especially troublesome in machine learning based techniques, since we can never be sure exactly how these systems work.

2.3 Formal Methods & Proof Search

One of the most mature and developed fields of program synthesis is what I refer to as *formal methods*. These are the techniques where a formal, usually mathematical description of a function is given, and a formal, well-defined process is followed to synthesise a corresponding function definition.

One example of this is [12], which describes a "deductive approach" to program synthesis. Here, the programmer would specify their intentions in a very precise way. For

example, to specify the problem of synthesising an integer square-root function:

$$\begin{aligned} \textit{sqrt}(n) \quad \Leftarrow \quad & \textit{find } z \textit{ such that} \\ & \textit{integer}(z) \textit{ and } z^2 \leqslant n < (z+1)^2 \\ & \textit{where integer}(z) \textit{ and } 0 \leqslant n. \end{aligned}$$

This approach then uses *sequents*, essentially tables containing assertions and goals (corresponding essentially to implications—if all of the assertions in a sequent are true, then at least one of the goals is also). Each sequent may also hold an output, which does not come into the proof itself, but is used to build up the proof (i.e. the synthesised program).

This method, *if* it produces a solution, is guaranteed to have written a program which fits the specification given. And, since the specification is given as a general mathematical formula (though in a real implementation would likely be limited at least somewhat), very complicated properties can be described. This gets around the issue of underspecified outputs having to be manually checked for correctness, at the expensive of extensive, complex specifications being necessary.

For this reason, I choose not to follow this particular path of program synthesis, since for an interactive editing situation it would be jarring to make the mental jump from writing a program to thinking about a formal specification. Input-output examples will lead to a smaller necessary context switch in the programmer's mind. However, these "deductive proof search" methods are not without merit, and a lot of more modern techniques based on input-output examples are based on similar deductive rules.

Proof search in this form is still used in much more recent works, for example [6] is based, theoretically, on proof search over the sequent calculus in the same way. This approach doesn't require such a strict mathematical definition—instead it "encodes" input-output examples into the type system.

2.4 Interactivity & Live Programming

Interactive programming is another area which this project will be exploring. Live programming environments aim to "provide programmers ... with continuous feedback about a program's dynamic behaviour as it is being edited". For example, *projection boxes* [11] show, for each line in a program, the value of all relevant runtime values. Projection boxes are built on further in [4] and [5] with program synthesis, using the projection boxes as both a live programming tool and an input to the synthesis engine—a way to specify examples.

Another example of interactivity in programming environments is *Hazel* ([14], [2]), a functional programming language with a "structure editor calculus", allowing for direct structural editing so that every possible editor state is a valid program. This is important when live programming is desired. Invalid editor states (e.g. a syntax error) can make some forms of live programming impossible, since the system needs to be able to understand the program to give live contextual information.

Interactivity is a key component of my project, but since I will be focusing on it only in the second half of the year, I won't delve too deep into specifics in this report. Nevertheless, at this point in time it seems likely that my approach to interactivity will be similar in some way to projection boxes [11], since this lends itself nicely to example-based synthesis as shown by [4] and [5].

3 Aims of This Project

There are many powerful techniques in the existing literature for program synthesis, and many interesting approaches to interactive editing. These are what my project will build upon, with the aim of creating a synthesis environment for *Fugue*, the functional programming language I made last year.

This environment will be called *Fantasia*[1], and the rest of this report details the state of my current work on it. This section will provide more details on the long-term goals for the project which I hope to achieve by the end of the year.

There are essentially two parts to *Fantasia*: the synthesiser, and the interactive editing environment. I will be basing my synthesiser on the *analytical* (as opposed to *enumerative*) approaches taken in [6], [15] and [16]. In [6], Frankle et al use a technique similar to proof search, translating input-output examples into types and searching for an inhabitant of that type. The other two, [15] and [16], frame the problem as analytically building a *program* rather than a proof (although of course the two approaches are identical from a theoretical standpoint). These approaches, in general, suit my application well—analytical program synthesis demonstrably works for functional programming languages, and *Fugue* is not dissimilar from the languages with which these papers work.

I choose not to take the machine-learning approach to program synthesis for a number of reasons. The main reason is that it is unpredictable and difficult to verify. For program synthesis to be a useful tool, the programmer should have a reasonable level of trust in the system that it does what is expected. For a pure machine-learning system, this is almost never possible. Machine learning could be used alongside an enumerative synthesis technique, as the program generator portion of the system, but that is far out of the scope of this project.

My aim of this project, then, at least in terms of program synthesis, is to take these existing methods and develop them further. Specifically, I would like to amend some issues with the techniques described in [6], [15] and [16]:

1. These techniques do not allow for the synthesis of auxiliary or mutually recursive functions.

2. For synthesis of recursive functions, these techniques require provided examples to be "trace complete", meaning that *specific* smaller and smaller examples must be given to guide the synthesiser towards the recursion.

I go into more detail about these two problems, and my solutions to them, in the next section. The synthesis engine I build will be called *Fantasia*.

The other aim of this project is to explore the interaction between a human and such a program synthesis engine. The details of this are yet unclear, except that I will create a live programming system which allows users to write programs in an exploratory manner with the help of *Fantasia*.

[1] A reference to J. S. Bach's *Fantasia and Fugue*, plus the link between "fantasy" and program synthesis.

4 Technique

In their most general form, the synthesis problems that I will be tackling look like this:

$$
\begin{aligned}
&f \ : \ \tau_1 \to \tau_2 \to \cdots \to \tau_n \\
&\quad \{ \ x_{1,1},\ x_{1,2},\ \cdots,\ x_{1,n-1} \quad \Rightarrow y_1 \\
&\quad ;\ x_{2,1},\ x_{2,2},\ \cdots,\ x_{2,n-1} \quad \Rightarrow y_2 \\
&\quad \vdots \\
&\quad ;\ x_{m,1},\ x_{m,2},\ \cdots,\ x_{m,n-1} \Rightarrow y_m \ \} \\
&f\ a_1\ a_2\ \cdots\ a_{n-1} = ?
\end{aligned}
$$

Where $n, m \geqslant 1$.

That is, a function of any amount of arguments (potentially zero—then a constant) to a value of any type, supplied with at least one example of the desired functionality. The examples are written beneath the type signature between the { and }. Each example represents a property of the function which the synthesiser *must* preserve.

The aim of the synthesiser, then, is to fill in the hole, written as ?, with a `Fugue` expression meeting the following properties:

- The expression should be of the correct type, when given correctly typed arguments.
- When applied to any of the examples, the expression should yield the correct value.

Furthermore, this should be done in the most "general" way possible. By this I mean that, while a solution could be trivially synthesised simply by enumerating the examples as a series of `if-else` chains and returning the correct value at each point, this would not be a good solution. Ideally, the minimal possible definition should be given, but proving this property would be extremely difficult. In practice, as long as the synthesiser doesn't "cheat" like this, it will be good enough—the aim is that extrapolation outside of the example set should be reasonable in most cases.

My approach is similar to the approach taken in [6], [15] and [16], where a data structure called a refinement tree is constructed and processed throughout the synthesis. These treat the problem as a proof search (as in [12]), in the constructive logic sense; then the refinement tree tracks which "proof goals" are active at any moment.

(It is worth noting that while the proof-search perspective on things helps to understand the process, it would be just as well to see the synthesis from a more syntactic perspective: searching for expressions of a given type. Indeed this is exactly what is happening. The two perspectives are identical from a Curry-Howard point of view [9].)

The rest of this section explains the specifics of my approach to the problem, and specifically how it improves upon existing methods.

4.1 Auxiliary Functions

While the refinement tree is shown in [6] and [16] to be an effective and elegant solution to this problem for some cases, it has its drawbacks. Most notably, it dictates that the entire synthesis must lie inside one function. This is acceptable for most applications,

however for some functions it is necessary to split off into an auxiliary function. For example, suppose we are presented with the following synthesis problem:

```
chew : String → String
     { "aaaabbbca" ⇒ "aaaa"
     ; "dddddd" ⇒ "dddddd"}
chew xs = ?
```

This requests a function which chews off a run of characters from the beginning of a string, as long as all of the characters are the same. There is a way to define this function without any auxiliary functions, but it would be useful to create a second function, after pattern-matching `xs`, to keep hold of the first character of the list and match against in the future:

```
chew : String → String
chew xs = case xs of
  [] → ""
  (x :: xs) → x :: go x xs
go : Char → String → String
go x xs = case xs of
  [] → ""
  (y :: ys) → if x ≡ y then x :: go x ys else ""
```

This is, essentially, the worker/wrapper pattern, and as well letting us store intermediate state in new arguments, it can help with efficiency in some cases [**?**] (the classic example being the transformation of a polynomial naïve *reverse* function into linear time).

My solution to this problem is at first glance a little heavy-handed. Essentially, at each decision point in the synthesis, I collect together all of the variables I have access to and hand them to a new function to synthesise that point in the definition. Thus, each synthesis step constructs an entire function body, synthesising entirely new functions for each "hole" in the body template.

While this does lead to a lot of small functions that don't need to be separate, it's not too difficult to unwind a pile of connected functions back into a minimal definition with as few as possible. I will come back to this later on.

4.2 Synthesis Rules

Framing the problem as program synthesis as constructing a number of auxiliary functions, we can see that all that needs to be done is to somehow decide on the structure of each of these functions. We have two pieces of information to guide us: the type signature of the function, and the provided examples.

Recall the structure of a synthesis problem from before:

$$
\begin{aligned}
f \;:\; & \tau_1 \to \tau_2 \to \cdots \to \tau_n \\
& \{\; x_{1,1},\; x_{1,2},\; \cdots,\; x_{1,n-1} \quad \Rightarrow y_1 \\
& ;\; x_{2,1},\; x_{2,2},\; \cdots,\; x_{2,n-1} \quad \Rightarrow y_2 \\
& \vdots \\
& ;\; x_{m,1},\; x_{m,2},\; \cdots,\; x_{m,n-1} \Rightarrow y_m \;\} \\
f\; & a_1\; a_2\; \cdots\; a_{n-1} = ?
\end{aligned}
$$

Where $n, m \geqslant 1$. Additionally, for the time being, I assume (without any loss of generality) that all of the example values ($x_{i,j}$ and y_k) are at least in weak-head normal form, so we can inspect their outermost constructors.

There are then a number of cases to consider, based primarily on the examples given inside the $\{\cdots\}$ section.

Case 1: Trivial (Solution already present)

Perhaps the simplest case is where the solution to the problem is already present as one of the arguments to the function. This occurs when there is some argument, a_i, such that for *all* given examples, the value of a_i agrees with the desired output value.

This usually won't be the initial problem which the user provided, but will often come up later in the synthesis, acting almost as the "leaves" of the synthesis tree. For example, an example taken from the next case we look at:

$$
\begin{aligned}
&g \ : \ \tau \to \mathit{List}\ \tau \to \tau \\
&\quad \{\ 1,\ [1,1] && \Rightarrow 1 \\
&\quad ;\ 2,\ [2,2,3,4] && \Rightarrow 2 \\
&\quad ;\ \mathit{True},\ [\mathit{True}, \mathit{False}] && \Rightarrow \mathit{True}\ \} \\
&g\ a_1\ a_2 = ?
\end{aligned}
$$

Here, trivially, $g\ a_1\ a_2 = a_1$.

More formally, this is described by the rule that if $1 \leqslant i < n$, and $\forall j . x_{j,i} = y_j$, then:

$$
\begin{aligned}
&f \ : \ \tau_1 \to \tau_2 \to \cdots \to \tau_n \\
&\quad \{\ x_{1,1},\ x_{1,2},\ \cdots,\ x_{1,n-1} && \Rightarrow y_1 \\
&\quad ;\ x_{2,1},\ x_{2,2},\ \cdots,\ x_{2,n-1} && \Rightarrow y_2 \\
&\quad \vdots \\
&\quad ;\ x_{m,1},\ x_{m,2},\ \cdots,\ x_{m,n-1} && \Rightarrow y_m\ \} \\
&f\ a_1\ a_2\ \cdots\ a_{n-1} = a_i
\end{aligned}
$$

It's worth noting that the "=" here means just that f *could* be defined like this—in other words, this is a valid definition. It's possible that some synthesis rules may overlap in certain cases, and in these cases, their bodies, while defined as "equal" in this syntax, would not necessarily be semantically the same. I come back to this when I discuss backtracking in the implementation later on.

Case 2: Common Constructors

Another case is where the output value of every example has the same outermost constructor. For example, if the problem at hand is the following polymorphic function:

$$
\begin{aligned}
&f \ : \ \tau \to \mathit{List}\ \tau \to \mathit{List}\ \tau \\
&\quad \{\ 1,\ [] && \Rightarrow [1,1] \\
&\quad ;\ 2,\ [3,4] && \Rightarrow [2,2,3,4] \\
&\quad ;\ \mathit{True},\ [\mathit{False}] && \Rightarrow [\mathit{True}, \mathit{True}, \mathit{False}]\ \} \\
&f\ a_1\ a_2 = ?
\end{aligned}
$$

This is quite a straightforward function—simply prepending an element twice to the beginning of a list—but illustrates the point nicely. At this point, the synthesiser can recognise that each of the examples' outputs are the *cons* of some value and another list (as is any non-empty list). This fact can be used to split the synthesis problem into two smaller auxiliary problems:

$$
\begin{aligned}
&g \;:\; \tau \to \mathit{List}\,\tau \to \tau \\
&\quad \{\; 1,\; [1,1] && \Rightarrow 1 \\
&\quad ;\; 2,\; [2,2,3,4] && \Rightarrow 2 \\
&\quad ;\; \mathit{True},\; [\mathit{True}, \mathit{False}] && \Rightarrow \mathit{True}\;\} \\
&g\; a_1\; a_2 = ?
\end{aligned}
$$

And,

$$
\begin{aligned}
&h \;:\; \tau \to \mathit{List}\,\tau \to \mathit{List}\,\tau \\
&\quad \{\; 1,\; [] && \Rightarrow [1] \\
&\quad ;\; 2,\; [3,4] && \Rightarrow [2,3,4] \\
&\quad ;\; \mathit{True},\; [\mathit{False}] && \Rightarrow [\mathit{True}, \mathit{False}]\;\} \\
&h\; a_1\; a_2 = ?
\end{aligned}
$$

Here, $g\; a_1\; a_2 = a_1$ (trivially from Case 1), and h will reduce further by a second application of Case 2, and a final application of Case 1.

To formalise this, we look back at the general structure of a synthesis problem:

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

Suppose there is some constructor, $C : \tau_1^c \to \tau_2^c \to \cdots \to \tau_k^c$, where $\tau_k^c \sqsubseteq \tau_n$ (meaning that the type which the constructor C constructs—τ_k^c—is more general than (or equal to) τ_n, the return type of the function to be synthesised.

Then if $\forall i \in [1,m].\; y_i = C\; c_{i,1}\; c_{i,2}\; \cdots\; c_{i,k}$ (meaning that the output value of each example is an instance of the constructor, C), we can synthesise f as follows:

$$
\begin{aligned}
f\; a_1\; a_2\; \cdots\; a_{n-1} = C[\tau_n]\; & (f'_1\; a_1\; a_2\; \cdots\; a_{n-1}) \\
& (f'_2\; a_1\; a_2\; \cdots\; a_{n-1}) \\
& \vdots \\
& (f'_k\; a_1\; a_2\; \cdots\; a_{n-1})
\end{aligned}
$$

Where f'_1, f'_2, etc are auxiliary functions synthesised separately (by the same process), specified by the following family of synthesis problems:

$$
\begin{aligned}
&f'_i \;:\; \tau_1 \to \tau_2 \to \cdots \to \tau_i^c \\
&\quad \{\; x_{1,1},\; x_{1,2},\; \cdots,\; x_{1,n-1} && \Rightarrow c_{1,i} \\
&\quad ;\; x_{2,1},\; x_{2,2},\; \cdots,\; x_{2,n-1} && \Rightarrow c_{2,i} \\
&\quad \vdots \\
&\quad ;\; x_{m,1},\; x_{m,2},\; \cdots,\; x_{m,n-1} && \Rightarrow c_{m,i}\;\}
\end{aligned}
$$

Each sub-problem (f'_1, f'_2, etc.) is supplied the full set of arguments available, in case it needs to make use of them later. After synthesis, any unneeded arguments can simply be removed.

The final piece of the puzzle here is $C[\tau_n]$. This represents the constructor C, specialised to the correct type as specified by the desired type signature of f. In general, this constructor specialisation is defined as:

$$
\begin{array}{l}
\textit{For } C : \tau_1 \to \tau_2 \to \cdots \to \tau_n, \\
\textit{and } \tau_n \sqsubseteq \tau, \\
\quad C[\tau] : \tau'_1 \to \tau'_2 \to \cdots \to \tau \\
\quad \textit{with } \tau'_1, \tau'_2, \cdots, \tau'_{n-1} \textit{ chosen such that } C \sqsubseteq C[\tau]
\end{array}
$$

In practice, this means finding a substitution σ which maps τ_n to τ, and then defining $\tau'_i = \sigma(\tau_i)$.

Case 3: Case-split

In a sense the opposite of Case 2, where a constructor is common to all output examples, is the case where one of the arguments is an algebraic data type—in this case, we can introduce a *case* expression to deconstruct the argument. This divides the set of examples up into a number of disjoint subsets, narrowing down the problem further.

Suppose we have a data-type, T, defined by $|T|$ constructors and z type parameters parameterised as:

$$C_t : \forall \alpha_1\ \alpha_2\ \cdots\ \alpha_{|T|}\ .\ \tau^T_{t,1} \to \tau^T_{t,2} \to \cdots \to \tau^T_{t,k_t} \to T\ \alpha_1\ \alpha_2\ \cdots\ \alpha_z$$

Here, the t^{th} constructor, C_t, takes k_t arguments and returns an element of the T (with potentially some polymorphic type parameters).

Imagine we have a problem of the form:

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

If we know that the type of the i^{th} argument is an instance of the algebraic type T (in other words, $T\ \alpha_1\ \alpha_2\ \cdots\ \alpha_z \sqsubseteq \tau_i$), we can synthesise f in this way (so long as the subproblems can also be synthesised):

$$
\begin{array}{l}
f\ a_1\ a_2\ \cdots\ a_{n-1} = \textbf{case}\ a_i\ \textbf{of} \\
\quad C_1\ b_1\ b_2\ \cdots\ b_{k_1} \quad \to g_1\ a_1\ a_2\ \cdots\ a_{n-1}\ b_1\ b_2\ \cdots\ b_{k_1}, \\
\quad C_2\ b_1\ b_2\ \cdots\ b_{k_2} \quad \to g_2\ a_1\ a_2\ \cdots\ a_{n-1}\ b_1\ b_2\ \cdots\ b_{k_2}, \\
\quad \vdots \qquad\qquad\qquad\quad \vdots \\
\quad C_{|T|}\ b_1\ b_2\ \cdots\ b_{k_{|T|}} \to g_{|T|}\ a_1\ a_2\ \cdots\ a_{n-1}\ b_1\ b_2\ \cdots\ b_{k_{|T|}}
\end{array}
$$

Here, the subproblems g_j are defined by extracting, for each case j, the relevant examples (i.e. the examples where the i^{th} argument is an instance of the case's constructor C_j). First, we need to find a substitution σ which unifies the argument type $T\ \alpha_1\ \alpha_2\ \cdots\ \alpha_z$ with the type of argument under scrutiny, namely τ_i. Such a substitution always exists, because we have already stipulated the fact that $T\ \alpha_1\ \alpha_2\ \cdots\ \alpha_z \sqsubseteq \tau_i$.

Then, we can specify g_j as follows:

$$\begin{aligned} g_j \ : \ & \tau_1 \to \tau_2 \to \cdots \to \tau_{n-1} \to \sigma(\tau^T_{j,1}) \to \sigma(\tau^T_{j,2}) \to \cdots \to \sigma(\tau^T_{j,k_j}) \to \tau_n \\ & \{ \ x_{l,1}, \ x_{l,2}, \ \cdots \ , \ x_{l,n-1}, \ c_1, \ c_2, \ \cdots \ , \ c_{k_j} \Rightarrow y_l \\ & \mid \ x_{l,i} = C_j \ c_1, \ c_2, \ \cdots \ , \ c_{k_j}, \quad 1 \leqslant l \leqslant m \ \} \end{aligned}$$

The examples here are defined using list-comprehension syntax over the set of examples from before, to select only the examples relevant at this point in the synthesis.

Again, as well as the new arguments introduced through the case-analysis ($c_1, \ c_2, \ \cdots \ , \ c_{k_j}$), we always include the arguments that were present before. If the synthesiser needs them, it can use them—otherwise, they will be removed at a later cleanup stage.

Other Cases

While Cases 1, 2, and 3 are sufficient in themselves for synthesis of a number of functions, there are obviously some things missing here.

Firstly, we would like some way to "look up" and use existing functions in the environment. Being an analytical synthesiser, this is not quite as important as it would be in an enumerative one, however it would still be desirable to use existing functionality rather than being forced to reimplement everything.

The lookup of existing functions (and compositions of these functions) to fill a type is a project in and of itself, and since it isn't the main focus of *Fantasia*, I will likely opt for a fairly simplistic approach. It seems at first that it would be easy to just try all of the possible functions in the environment whose return type fits the desired type. The problem with naïvely doing this lies in the synthesis of the function call's arguments—what examples should we provide? To deduce the desired outputs of the examples, we would need an "inverse" of the function at hand; and to deduce the inputs is tricky too.

Other options for usage of pre-defined functions are explored extensively in literature. The *TYGAR* algorithm described in [7], used for the implementation of *Hoogle+*, synthesises programs consisting entirely of composition of existing functions. This, or a similar algorithm, could be used in a second "mode of operator" of *Fantasia* to synthesise such programs when necessary. In any case, this is something which I will explore more in the second phase of the project.

The other notable missing rule from my list is a way to introduce recursive calls. Clearly, this is extremely important for synthesis of a huge number of *Fugue* programs. I explain the issues with, and my approach to, the problem in the next section which I dedicate entirely to recursion.

Fortunately, [16] shows that case-analysis, plus "guessing" (which corresponds to this idea of using existing functions and constructs to build up a solution) and recursion is sufficient for the synthesis of a vast array of programs. With my addition of auxiliary functions into this, plus what I hope to be a more effective approach to recursion, *Fantasia* should be a powerful system when it is complete.

A final case to consider, which I will refer to as Case 0, is the case when there are *no* examples. This can happen most notably after a case-analysis, if no examples were given for a particular constructor. In this case, it would be reasonable to fail, but since *Fantasia* is ideally geared towards interactive programming (and since *Fugue* has built-in support for holes: incomplete programs), it is better to simply return a new hole in such cases.

4.3 Recursion

Recursion is unsurprisingly a difficult problem in example-directed program synthesis. To see why, consider the problem of synthesising a function *stutter*, which duplicates each element in a list.

$$
\begin{aligned}
&\mathit{stutter} \;:\; \mathit{List}\;\tau \to \mathit{List}\;\tau \\
&\qquad \{\; [1,2,3] \Rightarrow [1,1,2,2,3,3] \\
&\qquad ;\; [] \qquad\;\; \Rightarrow [] \;\} \\
&\mathit{stutter}\;\mathit{xs} \;=\; ?
\end{aligned}
$$

After some synthesis steps, we end up with the following state:

$$
\begin{aligned}
&\mathit{stutter} \;:\; \mathit{List}\;\tau \to \mathit{List}\;\tau \\
&\qquad \{\; [1,2,3] \Rightarrow [1,1,2,2,3,3] \\
&\qquad ;\; [] \qquad\;\; \Rightarrow [] \;\} \\
&\mathit{stutter}\;\mathit{xs} \;=\; \textbf{case}\;\mathit{xs}\;\textbf{of} \\
&\quad \mathit{[]} \to \mathit{f}\;\mathit{xs} \\
&\quad \mathit{y} :: \mathit{ys} \to \mathit{g}\;\mathit{xs}\;\mathit{y}\;\mathit{ys}
\end{aligned}
$$

At this point, a case-analysis has been performed on *stutter*'s argument, *xs*. This results in two cases, one for the empty list, and one for *cons* of its head and its tail.

$$
\begin{aligned}
&\mathit{f} \;:\; \mathit{List}\;\tau \to \mathit{List}\;\tau \\
&\quad \{\; [] \Rightarrow [] \;\} \\
&\mathit{f}\;\mathit{xs} \;=\; []
\end{aligned}
$$

The synthesis for f proceeds trivially, returning the empty list. g applies the *cons* operator, constructing a new list:

$$
\begin{aligned}
&\mathit{g} \;:\; \mathit{List}\;\tau \to \tau \to \mathit{List}\;\tau \to \mathit{List}\;\tau \\
&\quad \{\; [1,2,3], 1, [2,3] \Rightarrow [1,1,2,2,3,3] \;\} \\
&\mathit{g}\;\mathit{xs}\;\mathit{y}\;\mathit{ys} \;=\; \mathit{h}\;\mathit{xs}\;\mathit{y}\;\mathit{ys} :: \mathit{i}\;\mathit{xs}\;\mathit{y}\;\mathit{ys}
\end{aligned}
$$

Introducing a constructor here creates two new functions, h and i, one for each argument to the constructor. h, again, follows trivially, as does the next step in i. As this is just an example, I will skip to the part where recursion is required, which is a few steps down the line inside i, after a second use of the *cons* operator:

$$
\begin{aligned}
&\vdots \\
&\mathit{k} \;:\; \mathit{List}\;\tau \to \tau \to \mathit{List}\;\tau \to \mathit{List}\;\tau \\
&\quad \{\; [1,2,3], 1, [2,3] \Rightarrow [2,2,3,3] \;\} \\
&\mathit{k}\;\mathit{xs}\;\mathit{y}\;\mathit{ys} \;=\; ?
\end{aligned}
$$

The Problem With Recursion

Intuitively, we can see that the solution here is $k\;xs\;y\;ys = stutter\;ys$—a recursive call on *ys*, part of the deconstruction of *xs*. The difficulty encountered here in existing solutions ([6], [15], [16]) is that the possibility of this call is determined based on the examples we

supplied to *stutter* in its definition. In other words, they require that *stutter* be given a further example,

$$[2, 3] \Rightarrow [2, 2, 3, 3],$$

in order to facilitate this recursive call. While this does work, it takes away from the goal of *automatic* synthesis. For a user to decide to provide this particular example, they need to recognise that recursion will be necessary, and they will also need to mentally dismantle the arguments to figure out *which* recursive example to provide.

This is put into words in [6]: "To generate a recursive function, we demand that refinements be trace complete. That is, when $[2, 1]$ is an argument refinement, the synthesiser also needs refinements involving the arguments $[1]$ and $[]$ to clarify the results of structurally recursive calls." For the end-user, this is an annoyance, as many more examples are required.

My Solution

For Fantasia, I am exploring a solution to this problem based on partially evaluating recursive calls to functions in an attempt to estimate their desired behaviour, at which point an extra example can be produced mechanically, and the synthesis solved as normal.

I will first explain my approach using the *stutter* function, again. *stutter* is a function which takes a list and duplicates each element in the list, resulting in a list of twice the length. I can specify this with two examples:

$$\begin{aligned} &\mathit{stutter} \ : \ \mathit{List}\ \tau \to \mathit{List}\ \tau \\ &\quad \{ \ [] \quad\ \Rightarrow [] \\ &\quad ; \ [1, 2] \Rightarrow [1, 1, 2, 2] \ \} \end{aligned}$$

A key component of recursion—in my approach, at least—is a synthesis rule to introduce simultaneously a case-analysis and a recursive call. Grouping these operations covers the vast majority of recursive definitions, and lets us ensure that the recursive call always acts upon a "smaller" input. This helps to avoid the generation of non-terminating programs. (I have not proved this property. A proof would be an interesting direction to explore in the next phase of this project.)

In this case, the application of such a rule would lead to this definition of *stutter*:

$$\begin{aligned} &\mathit{stutter}\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\ &\quad [] \quad\ \to [], \\ &\quad \mathit{y} :: \mathit{ys} \to \mathbf{let}\ \mathit{t} = \mathit{stutter}\ \mathit{ys} \\ &\qquad\qquad\ \ \mathbf{in}\ \mathit{f}\ \mathit{xs}\ \mathit{y}\ \mathit{ys}\ \mathit{t} \end{aligned}$$

Note, here I've "synthesised" and unwound the case of the empty list because it is entirely trivial, but in a real implementation this would also be split out into an auxiliary function. I will continue to fill in these trivial cases for the sake of this example, but keep in mind that the real algorithm is slightly more verbose than what I'm showing here.

The next step is now to synthesise *f*, and the natural question of "what input-output examples do we need to pass to the synthesis of *f*?" arises. This is the question which the literature ([6], [15], [16]) has struggled with. The reason for this is that the input to

f—specifically of the final parameter, t in this case—contains a call to *stutter*, but at this point in time, *stutter* is not entirely defined: we have yet to finish its auxiliary function f.

The solution I propose to this problem is to just run with it. Call the input example *stutter ys* and simply leave it unevaluated. A thunk, in other words. Each time we make another step in the synthesis process, we can evaluate the thunk a little further, until we reach a point where the example is "obvious". In this case, we can evaluate *stutter ys* one step, yielding *stutter ys* $= f\ [2]\ 2\ []\ []$.

This leads to the following problem statement for f:

$$\begin{aligned} &f\ :\ \mathit{List}\ \tau \to \tau \to \mathit{List}\ \tau \to \mathit{List}\ \tau \to \mathit{List}\ \tau \\ &\quad \{\ [1,2],\ 1,\ [2],\ \langle f\ [2]\ 2\ []\ []\rangle \Rightarrow [1,1,2,2]\ \} \end{aligned}$$

Here I use the $\langle \cdots \rangle$ syntax to denote a partially evaluated argument to an input example.

Synthesis proceeds by Case 2, the rule dealing with common constructors in the output examples, leading to:

$$f\ xs\ y\ ys\ t = y :: g\ xs\ y\ ys\ t$$

(Again, the head-part of the cons operator introduced here follows trivially, so I'm omitting its derivation.)

Another function has arisen now to synthesise: g. This proceeds much like the previous one, giving us:

$$\begin{aligned} &g\ :\ \mathit{List}\ \tau \to \tau \to \mathit{List}\ \tau \to \mathit{List}\ \tau \to \mathit{List}\ \tau \\ &\quad \{\ [1,2],\ 1,\ [2],\ \langle 2 :: g\ [2]\ 2\ []\ []\rangle \Rightarrow [1,2,2]\ \} \end{aligned}$$

Evaluation of the thunk argument in the example has again proceeded by one step, producing the partially evaluated expression $\langle 2 :: g\ [2]\ 2\ []\ []\rangle$. Synthesis can now continue, again using Case 2 since "all" of the example outputs share the same constructor.

$$g\ xs\ y\ ys\ t = y :: h\ xs\ y\ ys\ t$$

Synthesis of h follows naturally, and we can evaluate the thunk one step further still:

$$\begin{aligned} &h\ :\ \mathit{List}\ \tau \to \tau \to \mathit{List}\ \tau \to \mathit{List}\ \tau \to \mathit{List}\ \tau \\ &\quad \{\ [1,2],\ 1,\ [2],\ \langle 2 :: (2 :: h\ [2]\ 2\ []\ []) \rangle \Rightarrow [2,2]\ \} \end{aligned}$$

Something interesting has happened here. The partially-evaluated argument $\langle 2 :: (2 :: h\ [2]\ 2\ []\ []) \rangle$ is looking very similar to the desired output, $[2,2]$! In fact, if we were to let $h\ [2]\ 2\ []\ []$ be simply the empty list, they would be identical. This, then, would be a *unifying example* for h.

In other words, we can append an additional example to h, like so:

$$\begin{aligned} &h\ :\ \mathit{List}\ \tau \to \tau \to \mathit{List}\ \tau \to \mathit{List}\ \tau \to \mathit{List}\ \tau \\ &\quad \{\ [1,2],\ 1,\ [2],\ 2 :: (2 :: []) \Rightarrow [2,2] \\ &\quad ;\ [2],\ 2,\ [],\ [] \qquad\qquad\ \Rightarrow [] \ \} \end{aligned}$$

When we do this, note how the $\langle \cdots \rangle$ brackets have disappeared from the first example, indicating that it is now fully evaluated. Its final value, $2 :: (2 :: [])$, is the same thing as $[2,2]$, but I choose to write it in this way here to show how it was derived.

Now, using Case 1—the synthesis rule which deals with cases where a solution already exists trivially as one of the arguments—we can finish the synthesis:

$$h\ xs\ y\ ys\ t = t$$

Finally, we can unwind this using the technique described in the following section, to produce exactly the function we would expect:

$$\begin{aligned} &stutter\ xs = \textbf{case}\ xs\ \textbf{of} \\ &\quad [] \quad\ \ \rightarrow [], \\ &\quad y :: ys \rightarrow y :: (y :: stutter\ ys) \end{aligned}$$

Notice that all of the auxiliary functions are folded into one function, and the let-binding which introduced the recursive call at the start has been pushed down into the function to the location where it is actually used.

To conclude: this method of keeping partially evaluated terms as example inputs until they reach a unifiable state is able to produce the same class of recursive functions as other techniques on which *Fantasia* is based, but removes the need for the user to supply extra examples to indicate how the recursion should work.

4.4 Unwinding Auxiliary Functions

The final step of synthesis, once we have produced a root function along with a collection of auxiliary functions, is to piece them back together. This is not strictly necessary, since the unravelled form which synthesis leaves us with is perfectly valid. It would be inconvenient though, of course, since we are often left with dozens of small helper functions.

We start from the root function and unwind it, recursively working our way outwards along the call graph. To unwind one function, we first recursively unwind all functions which it refers to. Memoizing is necessary here, since two mutually recursive functions would otherwise result in an infinite loop at this point. Once all child functions have themselves been unwound, we attempt to *inline* each function call from their definitions.

We can inline a function if and only if it is non-recursive. As a result, unwinding a synthesised program in this way will produce a minimal unwinding: if a function call exists, it is always either a recursive call, or the application of another recursive function.

5 Implementation

Fantasia is implemented inside *Fugue*[2], so the whole system could be referred to as *Fantasia and Fugue*. The synthesiser is not fully implemented yet, the most notable gap at this point in time being recursive functions (though this will come soon).

The synthesiser lives inside a new module, *Synthesis*, though I also made some significant changes to the parser, type inference engine, and compiler, to extend *Fugue* with algebraic data types which are an essential component of the *Fantasia* synthesiser.

The first steps were to define some types to use. Firstly, *Context* represents a "synthesis context", in other words a set of examples plus the current *Fugue* environment (for access to pre-defined functions, etc.):

[2] https://github.com/zac-garby/diss/tree/master/fugue

```
data Context = Context
  {env      :: Environment
  ,examples :: [Example]}
data Example = Eg [Term] Term
  deriving Show
```

Additionally, I need a type to represent functions during synthesis, which are in effect regular *Fugue* functions augmented with a set of examples which they satisfy. To make the code easier to write, I also explicitly keep track of the functions' arguments (their types and also their names) and their return types.

```
data Function = Function
  {args :: [(Ident, Type)]
  ,ret  :: Type
  ,body :: Expr
  ,egs  :: [Example]}
  deriving Show
type Functions = [(Ident, Function)]
```

With this, I can begin on the synthesis itself. Synthesis as I've described it earlier is actually a monad, unsurprisingly. We keep track of a context (most importantly, the current examples to synthesise against) and accumulate over the course of execution a set of auxiliary functions.

These two pieces of functionality sound a lot like the *Reader* and *Writer* monads, though *Writer* isn't quite sufficient for the accumulation of auxiliary functions since we would like to be able to refer back to them later on in the synthesis to reuse components. Another part of synthesis is producing fresh variable names for function definitions and their arguments, so we group this together with the set of functions into a tuple and store the two in the *State* monad.

The final piece is a way to perform backtracking. Synthesis may proceed a number of steps and then get stuck. At this point, there is nothing we can do but fail and go back to an earlier synthesis state (which itself may also be wrong). This is not a problem, it just means we made a wrong turn somewhere, which is absolutely expected for an analytical synthesiser such as *Fantasia*. However, we need some way of encoding this into the synthesiser's implementation, and for this I use the *Maybe* monad. A synthesis attempt which reaches a dead-end with no recourse will *fail*, letting execution fall back to an earlier state.

With this, I can define the monad stack that I will be working with:

```
type Synth = MaybeT (StateT ([Ident], Functions) (Reader Context))
```

A synthesiser then would look something like this:

```
synth :: [Type] → Type → Synth Ident
synth argTypes ret = ...
```

Given the types of the arguments, and the desired return type, we synthesise an *identifier* (just a *String*) referring to the auxiliary function which this synthesis step has produced. The identifier refers to a function inside the *State* monad, and the examples which the synthesiser adheres to are found in the *Reader*.

My definition of *synth*, implementing synthesis Cases 0, 1, 2, and 3, looks like this:

```
synth :: [Type] → Type → Synth Ident
synth argTypes ret = do
  egs  ← asks examples
  args ← forM argTypes $ λt → do
    name ← fresh
    return (name, t)
  try [synthNoEgs args ret
    , synthTrivial args ret
    , synthCommonConstr args ret
    , synthSplit args ret]
```

First we fetch the examples from the *Reader* and fabricate some new variable names for each of the new function's arguments. Then, we try each of the cases in turn, making use of the *try* function defined like so:

```
try :: Monad m ⇒ [MaybeT m a] → MaybeT m a
try [] = fail "exhausted all possibilities"
try (x : xs) = let x' = runMaybeT x in MaybeT $ do
  m ← x'
  case m of
    Nothing → runMaybeT (try xs)
    Just r → return (Just r)
```

This attempts each *Maybe* action in turn, returning the first success or resulting in failure if none of the cases succeed.

The functions *synthNoEgs*, *synthTrivial*, *synthCommonConstr*, and *synthSplit*, implement the bulk of the synthesis functionality. I won't show the definitions of all of them here since they essentially mirror the formal definitions given earlier on, but as an example here's *synthTrivial*.

```
synthTrivial :: [(Ident, Type)] → Type → Synth Ident
synthTrivial args retType = do
  egs ← asks examples
  go egs 0
  where go egs n
    | n ⩾ length args = exit
    | all (λ(Eg egArgs egRes) → egArgs !! n ≡ egRes) egs =
      defineFunction $ Function {args = args
                                 , ret = retType
                                 , body = Var (fst $ args !! n)
                                 , egs = egs}
    | otherwise = go egs (n + 1)
```

This function takes two arguments again: the arguments, and the return type of the function to synthesise. The arguments now are named, using the names *synth* fabricated.

The first step again is to fetch the examples to synthesise against. We then step into an auxiliary function, *go*, which acts as a loop over each of the arguments in the definition. This loop checks each argument in turn, and if any of them is consistent with all example

outputs, it is placed into a new auxiliary function, which is then placed into the *State* using *defineFunction*.

To give the rest of *Fugue* a way to interface with the synthesiser, I define a helper function to unwrap all of the monads in the *Synth* monad transformer stack:

```
synthesise :: Environment → Type → [Example] → Maybe (Ident, Functions)
synthesise env goal egs =
  let r = runReader (runStateT (runMaybeT (uncurry synth (unfoldFnTy goal)))
                               (allVars, [])) ctx
  in case r of
    (Nothing, _)      → Nothing
    (Just i, (_, fns)) → Just (i, fns)
  where ctx = Context {env = env
                      , examples = egs}
```

The final step is unwinding the auxiliary functions into a minimal amount of functions, for readability and pragmatism purposes. This makes use of the *State* monad again:

```
type Unwind = State Functions
```

This is due to the fact that we need to update the functions in an out-of-order way, so we should be able to update any function at any point in time. This requires state.

Unwinding an expression, then, is attempting to inline each variable and functional application inside it. *canInline* checks that a function can be inlined (i.e. that it is not recursive), and *inline* performs the actual substitution.

```
unwind :: Expr → Unwind Expr
unwind (Var x) = do
  fn ← lookupU x
  case fn of
    Just (Function {body = body, args = []}) → unwind body
    _ → return $ Var x
unwind app@(App e1 e2) = case unfoldApp app of
    (Var x, args) → do
      fn ← lookupU x
      case fn of
        Just fn → if canInline x fn then inline x args else app'
        Nothing → app'
    _ → app'
  where app' = App <$> unwind e1 <*> unwind e2
unwind (Abs x e) = Abs x <$> unwind e
unwind (Let x e1 e2) = Let x <$> unwind e1 <*> unwind e2
unwind (LetRec x e1 e2) = LetRec x <$> unwind e1 <*> unwind e2
...   -- etc.
```

I've omitted some uninteresting cases, since the rest of the code is simply applying *unwind* recursive inside the expressions using the fact that *State* is an *Applicative* functor.

Most of the functionality here is in the *App e1 e2* case—we first unfold the application to get its "function" and its full set of arguments. If the function being called is a variable-name referring to a function defined by this synthesis, the *inline* function takes care of the actual inlining of it back up into the expression.

At present, there is no interactive way for a user to ask *Fantasia* to synthesise a program: all interaction is currently done manually through Haskell for testing. I have, however, tested this synthesiser against a number of (non-recursive) function specifications for user-defined data-types like lists and trees. The next step, as I will discuss in more depth in the next section, is to finish the synthesiser so I can move on to the interactivity portion of the project.

Also in the course of this implementation, I have implemented user-defined algebraic data types and pattern matching into *Fugue*. Now, for example, I can declare new data types with the following syntax:

```
data List a = Nil | Cons a (List a);

data Maybe a = Nothing | Just a;

data Bool = True | False;
```

Some examples of functions defined over these types are below:

```
null : List a -> Bool;
null xs = case xs of
  Nil -> True,
  Cons x xs -> False;

(++) : List a -> List a -> List a;
(++) xs ys = case xs of
  Nil -> ys,
  Cons x xs -> x :: (xs ++ ys);

(!) : List a -> Int -> Maybe a;
(!) xs i = case xs of
  Nil -> Nothing,
  Cons x xs -> if i == 0 then
                 Just x
               else
                 xs ! (i - 1);
```

The relevant Haskell types used to represent these new data types are found in the *Types* module:

```
data DataType = DataType [Ident] [DataConstructor]
  deriving Show
data DataConstructor = DataConstructor Ident [Type]
  deriving Show
type DataTypes = [(Ident, DataType)]
```

The rest of the changes to implement this were fairly major in terms of lines changed, but not too interesting or surprising (mostly the addition of new cases to functions which operated on expressions). As it's not the focus of this project, I won't include too much of it here, but I think the type inference for case analyses is worth mentioning:

```
infer (Case t cases) = do
  tt ← infer t
  tb ← fresh
  env ← ask
  let (con1, _, _) = head cases
  dt ← dataTypeWithCon con1
  case dt of
    Just (DataType dtParams dtCons) → case missingCases dtCons cases of
      [] → do
        forM_ cases $ λ(constr, args, body) → do
          tArgs ← mapM (const fresh) args
          let schArgs = zipWith (λarg ty → (arg, Forall [] ty)) args tArgs
          (tb', tt') ← withMany schArgs $ do
            tb' ← infer body
            tt' ← infer $ foldl App (Var constr) (map Var args)
            return (tb', tt')
          tb ~~ tb'
          tt ~~ tt'
        return tb
      missing → throwError $ MissingCases missing
    Nothing → throwError $ UnknownConstructor con1
```

A case analysis, as specified on the second-last line here, requires that *all* possible cases—all constructors of the scrutinee's type—are considered. This requirement comes from the fact that *Fugue* doesn't have a $\bot$ value, and all expressions must lead to some value.

This concludes the explanation of the implementation of *Fantasia*, as well as some changes to the core language *Fugue*. Next I will discuss the state of this project from a management point-of-view, and what the future of it will look like.

6 Progress

This project has progressed well, despite going slightly slower than I would have necessarily liked.

In my project proposal, I said that I would like to finish the implementation of user-defined datatypes into *Fugue* by 31 October, which I did. Then, I proposed to finish the synthesis engine in its entirety by 12 December.

This ended up being an unrealistic goal, since a lot of literature review was required, and program synthesis is a difficult problem. I also had a lot of time taken away from me this term by another module—Music & Mixed Reality—which had a public showcase around the same time as the deadline for this interim report. As a result, I had to unfortunately focus most of my attention on the project which would be displayed to the public.

Despite this setback, the synthesis engine is nearly finished, and I planned for this contingency in my project proposal. I wrote that the implementation of the synthesiser may extend into the Christmas holidays if necessary, which is what will happen now.

Either way, I'm thoroughly enjoying this project so far. Program synthesis is a huge

topic with a huge amount of methods and approaches, and I'm pleased that I have been able to find two novel contributions to make in this research project.

As for the future progress of *Fantasia*: I will be working on finishing the synthesiser over the Christmas holidays. Next term I look forward to exploring in more depth the world of interactive programming to produce a synthesiser which programmers will find useful.

References

[1] ChatGPT: Optimizing language models for dialogue. 2.2

[2] Cyrus Omar et al. Hazelnut: A bidirectionally typed structure editor calculus. 2.4

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. 2.2

[4] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. LooPy: interactive program synthesis with control structures. 5:153:1–153:29. 2.1, 2.4

[5] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia 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. 1, 2.1, 2.4

[6] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. 51(1):802–815. 2.3, 3, 4, 4.1, 4.3, 4.3

[7] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. 4:1–28. 2.1, 4.2

[8] Martin 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. 2.1

[9] William A Howard. The formulae-as-types notion of construction. 44:479–490. 4

[10] Susumu 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. 2.1

[11] Sorin Lerner. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, pages 1–7. Association for Computing Machinery. 2.4

[12] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. 2(1):90–121. 2.3, 4

[13] Neil Mitchell. Hoogle overview. 12:27–35. 2.1

[14] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. 3:14:1–14:32. 2.4

[15] Peter-Michael Osera. Constraint-based type-directed program synthesis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development*, pages 64–76. ACM. 3, 4, 4.3, 4.3

[16] Peter-Michael Osera and Steve 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. 2.1, 3, 4, 4.1, 4.2, 4.3, 4.3