Why Functional Program Synthesis Matters (In the Realm of Genetic Programming)

Why Functional Program Synthesis Matters (In the Realm of Genetic Programming)

Fraser Garrow
Heriot-Watt University
Edinburgh, UK
fg28@hw.ac.uk

Michael A. Lones
Heriot-Watt University
Edinburgh, UK
m.lones@hw.ac.uk

Robert Stewart
Heriot-Watt University
Edinburgh, UK
r.stewart@hw.ac.uk

ABSTRACT

In Genetic Programming (GP) systems, particularly those that target general program synthesis problems, it is common to use imperative programming languages to represent evolving code. In this work, we consider the benefits of using a purely functional, rather than an imperative, approach. We then demonstrate some of these benefits via an experimental comparison of the pure functional language Haskell and the imperative language Python when solving program synthesis benchmarks within a grammar-guided GP system. Notably, we discover that the Haskell programs yield a higher success rate on unseen data, and that the evolved programs often have a higher degree of interpretability. We also discuss the broader issues of adapting a grammar-based GP system to functional languages, and highlight some of the challenges involved with carrying out comparisons using existing benchmark suites.

CCS CONCEPTS

• **Software and its engineering** → **Genetic programming**; **Functional languages**; **Automatic programming**.

KEYWORDS

genetic programming, program synthesis, functional programming

ACM Reference Format:
Fraser Garrow, Michael A. Lones, and Robert Stewart. 2022. Why Functional Program Synthesis Matters (In the Realm of Genetic Programming). In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3520304.3534045

1 INTRODUCTION

Automatic programming has the potential to transform the software development process by changing the role of developers and empowering non-expert users. Program synthesis is one aspect of automatic programming where the goal is to automatically generate programs based on some specification or user prompt. Historically, GP has had many successes in the automatic creation of programs. It has achieved this by evolving programs in a variety of languages, *e.g.* C++ [5, 24], Java [8, 28], most recently, Python [12, 13, 35], and many more. These are all examples of imperatively styled programming languages. On the other hand, the literature on programs evolved in a functional style is sparse [1, 22, 23, 40, 44]. Other languages targeted by GP include several domain specific languages and Push [25, 38].

Recently, the introduction of program synthesis benchmarks has increased interest in this domain within the GP community. Helmuth and Spector's General Program Synthesis Benchmark Suite [18] (from here on referred to as *the benchmark suite*) introduced 29 general programming tasks. It has so far been tackled by two GP systems in particular: the first using PushGP to evolve Push programs [18], and the second using Grammar Guided GP (GGGP) to evolve Python programs [12]. Both systems were able to solve several of these problems, but, despite this, low success rates over multiple runs are commonplace and there is often a lack of generalisation to unseen data. Further work has been carried out to address both issues [13, 17, 35], but this remains an open area.

In this work, we tackle a subset of the benchmark suite with programs evolved in a purely functional programming language. GGGP evolves programs using a context-free grammar that ensures programs are type safe. Context-free grammars can be created for any programming language. We use the program synthesis grammar design pattern to construct our grammar for the Haskell language. This design pattern, presented in [12], allows GP to tackle any arbitrary program synthesis problem by creating an independent grammar for all data types considered; these can then be selected according to the data types required for a given problem.

It is known that program representation plays an important role in the success of a GP system [15]. GGGP and stack based GP have both been shown to be effective at program synthesis. What is less well known, with respect to GGGP, is what role language choice plays in the effectiveness of the technique. There is good reason to believe that evolved functional programs will be able to generalise better to unseen data than imperative programs [44].

It is also reasonable to suggest that because we can compose functions and make use of higher order functions we will be able to achieve the same results (as evolved imperative programs) with smaller instruction sets, which will reduce the search space. It may also be feasible to evolve generic functional programs that can operate over multiple data types. Previous work in GP has considered similar ideas, particularly Yu [44], who created a GP system with functional programming paradigms to show the benefits of this style of GP over standard (at the time) GP methods.

In this study, we investigate the evolution of purely functional style programs for tackling a subset of the problems from the benchmark suite. We find that, for a number of these problems, the functional approach yields higher success rates and generalises well to

GECCO '22 Companion, July 9–13, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9268-6/22/07…$15.00
https://doi.org/10.1145/3520304.3534045

unseen data. Despite this success, the results also suggest that adjustments should be made to adapt the program synthesis grammar design pattern for use with functional programming languages.

This paper is organised as follows: Section 2 gives an overview of program synthesis in GP, Section 3 discusses different programming paradigms, Section 4 presents the approach we have taken for evolving functional programs, Sections 5 and 6 describe the experimental setup and results, the discussion follows in Section 7. Finally, Section 8 concludes the study and suggests future work.

2 PROGRAM SYNTHESIS

Program synthesis refers to the automatic generation of executable code based on some description of how the system should operate. The automatic synthesis of programs has the potential to revolutionise computing by making programming easier for developers and non-developers alike. As such, many different research communities are working on the problem independently, each achieving different successes: Programming Languages [27, 33, 37], Machine Learning [4, 9] and Evolutionary Computing [12, 13, 17–19].

For a full survey of GP in the program synthesis domain we direct the reader to [36]. In GP, program synthesis is mostly viewed as an inductive programming problem; that is, the program is generated based on input/output examples, although some work has considered more formal specifications [22, 23, 43] and partial programs [3]. It is also worth noting Genetic Improvement [31] approaches, which work via the manipulation of existing programs. The benchmark suite from Helmuth and Spector [18] (see Section 5.1) was a catalyst for increasing interest in general program synthesis within the GP community. Two GP approaches to the benchmark suite are discussed here: PushGP and GGGP.

PushGP [39] is a GP system that evolves programs in the stack-based programming language, Push [39]. PushGP has achieved impressive results across various domains [25, 38]; it has also been successful at tackling many of the problems from the benchmark suite. Despite potential benefits in terms of evolvability, an argument against PushGP is that Push is not a commercial language, it was designed specifically for GP, and therefore programs evolved in Push are less likely to be adopted for use in real-world applications. However, this is debatable, since one of the aims of GP must be to tackle program synthesis problems that humans cannot, which would make language choice less relevant; a solution in an unfamiliar language is better than not having a solution in a familiar language. Despite Push being an imperatively styled language, our focus is on language choice in the context of GGGP and as Push is synonymous with PushGP, we do not consider it further in this study.

GGGP uses a context-free grammar to construct genotypes that are type safe, syntactically sound individuals (so long as the grammar has been designed to ensure this property). In GGGP there are two approaches to representation: derivation trees and linear genomes. When evolving using a grammar with a large number of non-terminals (like those for arbitrary program synthesis problems), linear genomes can produce invalid solutions; this occurs during the mapping process when all the genes in the genome have been spent yet the program still contains non-terminal nodes. For this reason, in program synthesis, derivation tree GGGP has been preferred [12, 35]. In derivation tree GGGP the grammar acts as a guide to ensure the derived trees are valid, and the usual GP operators on trees can then be used (again ensuring valid programs).

In grammar design, one concern is that the grammar is as small as possible, so that evolution can efficiently traverse the search space, but still remain expressive enough to encode a solution. In program synthesis we want a grammar to be able to solve any program synthesis problem, and this may include many data types and functions that operate on these. To address these competing concerns, Forstenlechner et al. [12] presented the grammar design pattern for program synthesis problems. This suggests creating grammars for each individual data type and selecting these, component-wise, corresponding to the data types of the given problem. This is an effective way of reducing the size of the grammar for each problem, whilst preserving the ability to tackle any program synthesis problem.

3 ON PROGRAMMING PARADIGMS

There are often clear reasons for choosing one programming paradigm over another. However, possibly more often, the choice comes down to personal preference. Here we discuss some of the reasons, good and bad, for both the use of a purely functional style of programming (in Haskell) and imperative programming (in Python).

3.1 Imperative Programming

In imperative programming the programmer lays out, step by step, the actions that must be taken by the computer to achieve the program's goal. Each of these steps may change the program's internal state. The main focus of the programmer is how to perform the underlying task and to maintain the correct program state. As programs are designed to follow this step-wise structure, the order of execution is important and follows the program's structure. This flow may be controlled by loops and conditionals.

The imperative programming style is by far the most common in the developer ecosystem. Of the 10 most abundant languages on GitHub, all 10 are of imperative style [14]. The state of the existing programming ecosystem seems to be one of the main drivers of why these languages are so popular; for instance, it is no surprise that Javascript tops the list of languages on GitHub when it is the main language for web development. Once a language is as pervasive as the mainstream languages are (because of their historical use) then they tend to continue to grow. Then, because so many tools depend on them, they do not disappear; this has been called "The complete absence of death" [32].

Imperative style programming is relatively easy to learn, and it is usually the style first taught (another reason for its dominance). This is perhaps because the conceptual model of a problem translates directly to a step by step program; all the programmer needs to know is the syntax to implement it. The small leap required from conceptual model to code also means they are quick to prototype in. Often these grow into much larger projects; again, another reason for how pervasive imperative languages are.

To compare the style of an imperative program with the functional version presented in the next section, we present a reference

implementation in Python of the well known programming problem, FizzBuzz[1]. The snippet was taken from RosettaCode [6].

```
def fizzbuzz(n):
    for elem in range(n):
        if elem % 3 == 0 and elem % 5 == 0:
            print("FizzBuzz")
            continue
        elif elem % 3 == 0:
            print("Fizz")
            continue
        elif elem % 5 == 0:
            print("Buzz")
            continue
        print(elem)

fizzbuzz(100)
```

In imperative programming, state is mutable, and a key task for the programmer is maintaining it correctly. This is not always easy. As an example, consider this code snippet where the intention is to remove odd numbers from a list:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def remove_odds(list):
    for i in list:
        if i %2 != 0:
            del list[i]
    print(list)

remove_odds(numbers)
```

We might expect this snippet to return `[2, 4, 6, 8, 10]`. However, lists are mutable and here the list we are iterating over changes as we iterate, resulting in Python trying to retrieve a list index that no longer exists. This is a so-called side effect of mutable state changing programs.

3.2 Functional Programming

Functional programming, compared to imperative programming, cares less about how to do it and more about what to do. The basis of functional programming is applying functions to arguments. Programs in a functional language can be viewed as mathematical functions. The main focus of the programmer is how to reason about the underlying task so that the program achieves its goal correctly.

In this study we are interested in purely functional programming. Purely functional programs use pure functions; that is, for the same input they will always return the same output. From this, it follows that there are no side effects. Side effects are a major source of bugs in programs, like the one highlighted in the previous section.

Here we show a reference implementation for the same FizzBuzz programming task as before, but this time in a functional style using the Haskell language. The snippet was again taken from RosettaCode [7].

[1] **FizzBuzz:** Given an integer n, return: "Fizz" if n is divisible by 3; "Buzz" if n is divisible by 5; "FizzBuzz" if n is divisible by both 3 and 5; and finally n as a string if it is not divisible by either 3 or 5.

```
fizzbuzz :: Int -> String
fizzbuzz x
  | f 15 = "FizzBuzz"
  | f 3 = "Fizz"
  | f 5 = "Buzz"
  | otherwise = show x
  where
    f = (0 ==) . rem x

main :: IO ()
main = mapM_ (putStrLn . fizzbuzz) [0 .. 99]
```

The first thing to notice is that, unlike Python, Haskell is statically typed. However, it should be noted that many imperative languages are typed. Both implementations use a series of conditionals. The Python function does this in a for loop with `if` statements, step by step, with everything contained in the one function. On the other hand, the Haskell version abstracts the logic into a separate function, which is then passed as an argument to a mapping function to print the correct output. This is an example of modular programming (discussed below). The Haskell implementation also uses an anonymous function `f` for the conditional check, this makes the program less verbose.

Modular programming is supported in functional languages through higher order functions and is one of their most powerful features [21]. This is true for pure and non-pure functional languages. A higher order function is a function that either takes one (or more) functions as arguments or returns another function as an output. Well known examples are `map, reduce, filter`, and `fold`. The power of higher order functions is that we do not need to write every function we need, since we can compose functions with other functions. When combined with polymorphism it means that we can operate on data structures of any type using a small set of functions. The power of higher order functions combined with polymorphism is best shown by example:

```
filter even [1, 2, 3, 4]
filter (/='*') "GECCO '22, Boston*"
filter (/=[]) [[1,3], [], [34, 67]]
```

As shown, the `filter` function can operate over multiple data types and we can pass different functions as arguments to create different composed functions as required.

We have already seen that the most popular languages are of imperative design. One reason for functional languages not being as widely adopted corresponds to one of the reasons for imperative languages being popular — there is less of it currently in the ecosystem, so less perceived need for it. Functional languages are also regarded as being difficult to learn, with a steeper learning curve [42]. However, popularity of functional programming languages is increasing [20, 29]. It should also be noted that functional language motifs are now commonplace in many mainstream languages that can be classed as imperative. Many languages are now hybridised with support for higher order functions, generics, and other functional idioms.

Despite many desirable qualities, the literature on GP that includes some aspect of functional programming is limited when compared to GP that considers imperatively styled programs. It is over 20 years since Yu wrote a PhD thesis on the benefits of a

```
<code> ::= <int_list>
<int_list_var> ::= 'xs'|'ys'
<number> ::= <num><num><num>
<num> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
<int_list_const> ::= '[]'
<arith_ops> ::= '+'|'-'|'*'
<int_arith_ops_protected> ::= 'safeIntDiv'|'safeMod'
<prefix> ::= ''|'-'
<comp_op> ::= '=='|'/='|'<'|'>'|'>='|'<='
<fold> ::= 'foldr'|'foldl'
<int> ::= <int_value>|<int_expression>
<int_value> ::= <number>
<int_expression> ::= '('<prefix>' '<int>')'|'(abs '<int>')'|'(max '<int>' '<int>')'|'(min '<int>' '<int>')'
                   |'('<arith_ops>' '<int>' '<int>')'|'('<int_arith_ops_protected>' '<int>' '<int>')'
                   |'(length '<int_list>')'|'(safeMinimumInts '<int_list>')'|'(safeMaximumInts '<int_list>')'
                   |'(safeHeadInts '<int_list>')'|'(safeLastInts '<int_list>')'|'(sum '<int_list>')'
                   |'('<fold>' ('<arith_ops>') '<int>' '<int_list>')'
                   |'('<fold>' ('<int_arith_ops_protected>') '<int>' '<int_list>')'
<int_list> ::= <int_list_value>|<int_list_expression>
<int_list_value> ::= <int_list_const>|<int_list_var>
<int_list_expression> ::= '(filter ('<comp_op>' '<int>') '<int_list>')'|'(safeReplicate '<int>' '<int>')'
                   |'(map abs '<int_list>')'
                   |'(zipWith ('<arith_ops>') '<int_list>' '<int_list>')'
                   |'(zipWith '<int_arith_ops_protected>' '<int_list>' '<int_list>')'
                   |'('<int>':'<int_list>')'|'('<int_list>' ++ '<int_list>')'
                   |'(safeTail '<int_list>')'|'(safeInit '<int_list>')'
                   |'(take '<int>' '<int_list>')'|'(drop '<int>' '<int_list>')'
```

Figure 1: BNF Haskell grammar for the Vectors Summed problem of the General Program Synthesis Benchmark Suite

functional programming approach in GP [44]. Despite promising results, little work followed directly from this.

Integration of tools available to pure functional languages that exploit the mathematical properties of programs are perhaps the biggest motivation for evolving code in functional languages. Tools like Speculate [2] and QuickSpec [34] for example can discover properties about functions. This could be used to find semantically similar individuals to remove from populations, discover unreachable branches for pruning, or rewrite individuals into simpler forms. Other tools can be employed that reason about the correctness of programs based on some specification. Recently, some researchers have begun investigating how such tools can be incorporated with GP including using counter examples to drive the search [23] and program-sketching [3], both ideas originating from the programming languages community. Other work has considered type refinements [11, 41] to constrain the search space.

4 SYSTEM DESCRIPTION

We select Haskell as the purely functional language for evolution as it is mature, popular (in the context of purely functional languages) and has a rich ecosystem. Motivated by [12] we use the grammar design pattern for arbitrary program synthesis problems to construct our grammar for evolving Haskell programs (Section 4.1). Evolution is carried out using the derivation tree representation in PonyGE2 [10]. Adjustments were made to PonyGE2, where required, to allow for the evolution of Haskell programs and the experiments carried out. The code, grammars and data are available online[2]. For the imperative approach we use PonyGE2 to evolve Python programs using the grammars from [12] also available online[3]. The Python grammars make use of imperatively styled code such as function scoped mutable variables and looped control flow.

4.1 Grammar Design Pattern for Haskell

Historically for GGGP a bespoke grammar has been crafted for each problem — this is not a trivial task when the benchmark is 29 problems of varying type. The grammar design pattern for program synthesis problems reduces this burden by creating a grammar for each data type for use in the system and a grammar that handles code structure (control flow etc.). For a single problem the data type to be considered can then be specified, allowing the corresponding grammars to be selected individually and combined; this stops the search space being unnecessarily large.

For this work we have followed the design pattern and created a Backus-Naur Form (BNF) grammar that includes all the following Haskell data types we consider for the benchmark suite: `Int`, `Double`, `Bool`, `Char`, `String`, `[Int]`, `[Double]` and `[String]`. The grammars for each problem were constructed manually by pruning the parent grammar based on the types selected for each problem. An example of one of the resulting Haskell grammars is shown in Figure 1.

[2]A persistent version of the implementation is located at 10.5281/zenodo.6499027
[3]https://github.com/t-h-e/HeuristicLab.CFGGP

Table 1: Details of the subset of the General Program Synthesis Benchmark Suite [18] used for evaluation.

Problem	Training/ Test Cases	Input Type(s)	Output Type	Additional Available Type(s)	Terminals
Number IO	25/1000	Int	Double	-	Int ERC, Double ERC
Small Or Large	100/1000	Int	String	Bool	"small", "large", Int ERC
Compare String Lengths	100/1000	3 x String	Bool	Int	Bool ERC
String Lengths Backwards	100/1000	[String]	[Int]	Bool	Int ERC
Last Index of Zero	150/1000	[Int]	Int	Bool	0
Vector Average	100/1000	[Int]	Double	-	-
Super Anagrams	200/2000	2 x String	Bool	Bool, Int, Char	Bool ERC, Int ERC, Char ERC
Vectors Summed	150/1500	2 x [Int]	[Int]	-	[], Int ERC
Negative To Zero	200/2000	[Int]	[Int]	Bool	0, []
Median	100/1000	3 x Int	Int	Bool	Int ERC
Smallest	100/1000	4 x Int	Int	Bool	Int ERC

The benchmark suite advises that any system should use a similar set of instructions and to not cherry pick instructions. One of the motivating factors of evolving in a functional language is that by higher order functions and composition it is possible to achieve similar results with less instructions. The Haskell grammars are smaller and make use of different functions, but where possible, similar functions to those used in [12] are selected for comparison. As a rule we only selected functions that are available in the Haskell Prelude, which is imported by default into all Haskell programs. Some of the Prelude instructions were modified to ensure evaluation safety (see Section 4.2).

An additional factor that affects grammar size is the use of code structuring: in the Python grammars there is the possibility to construct `for` and `while` loops, but these constructs are not included in the Haskell grammar due to the nature of functional languages. We do include `if-then-else` statements in the Haskell grammar; however, these are included in the individual data types as they are simply functions that return a type depending on a conditional. For each type the grammar first splits the type by values and expressions (functions). In problems where a type has no values, i.e. neither a variable or ephemeral random constant (ERC) is defined for the problem, the grammar is reduced to only expression productions. The Python grammars include three variable placeholders for each datatype used, together with instructions to assign or change these values, whereas mutable variables are not legal in Haskell and have therefore been omitted. This means that for the Haskell grammar there was no design choice to be made in regard to how many variables need to be specified in advance.

The Haskell grammars allow for the use of higher order functions such as `zipWith, foldl, filter` etc. However, as this is a preliminary investigation, their application is currently limited to arithmetic operations, i.e. there is currently no functionality for passing any function (of the correct type) to a higher order function. Neither do the grammars include the scope for creating local variables within functions, Lambda expressions, or recursion (of the evolved function). Other code structuring patterns common in Haskell such as guards and pattern matching have not been allowed for; these constructs are syntactic sugar for the base language and it is not clear if they would make the code more or less evolvable.

As in [12] we use a skeleton of a Haskell module so that individuals can be evaluated. The evolved function from each individual is injected into the skeleton. The skeleton is then executed and returns a fitness to the evolution framework.

4.2 Haskell Program Details

To run an evolved Haskell program, a separate subprocess was created and the program was compiled and executed using the `runghc` Glasgow Haskell Compiler [26], version 8.10.7. The subprocesses were assigned a timeout, which if exceeded kills the subprocess, resulting in the individual being assigned the worst possible fitness.

Each Haskell skeleton imports a Haskell module `Functions`, which includes protected functions and functions required for fitness evaluation e.g. the Levenshtein distance metric.

The grammar has been designed such that all evolved programs will be type safe, and therefore no compilation errors can occur. Any other exceptions that occur are penalised by the individual being assigned the worst possible fitness.

5 EXPERIMENTAL SETUP

We investigate the evolution of the purely functional language, Haskell, and the imperative language, Python, using a GGGP system. We do this via a performance comparison using problems from the benchmark suite (Section 5.1) and inspection of the evolved artefacts. We first discuss the benchmark suite and then the parameter settings for the experiment.

5.1 Benchmark Suite

For this work we consider a subset of the benchmark suite [18]. The full benchmark suite consists of 29 general programming problems. Details of the subset of 11 problems are presented in Table 1. By covering all possible types in the benchmark suite, we regard the subset as somewhat representative of the full benchmark. The subset selected uses four of the seven problems selected as a subset by Sobania and Rothlauf [35].

In this study we use the data as provided the PonyGE2 framework, primarily because it was formatted correctly for the Python implementation. Minimal reformatting of the data was required to enable it to be used with Haskell. All of the data used was checked

Table 2: Experimental parameters

Parameter	Value
Runs	100
Generations	100
Population size	1000
Selection	Tournament
Tournament size	7
Crossover probability	0.9
Mutation probability	0.05
Elite size	1
Node limit	250

with a correct Haskell implementation to determine its veracity and that perfect fitness was possible with the data. Some of the correct Haskell implementations of the functions were taken from the Conjure library [27], and others were hand coded. The fitness functions used have been translated from the Python fitness functions used in [12].

5.2 Parameters

The parameters used in the experiment are presented in Table 2. The same parameters have been used for the Python and Haskell approaches. Compared to the original Python study [12] we reduce the number of generations; all other parameters remain the same. Forstenlechner et al. [12] previously showed that not all generations were required to solve many of the problems, Sobania and Rothlauf also used fewer generations (100); however, they significantly increased the population size (3000) [35]. The evolutionary search was stopped when a correct solution was found on the training data.

6 EXPERIMENTAL RESULTS

To evaluate the experiment we simply check for how many of the 100 runs an evolved program was able to produce all of the correct outputs given all of the training examples. If a program was found that solves all of these, it was then given all of the test examples; if it produced all of the correct outputs for these, it was said to have generalised to unseen data. This is shown in Table 3 for both the evolved Haskell and Python programs. Both approaches found solutions to 8 of the 11 problems on the training set. Haskell found solutions that generalised to unseen data (the test set) for all 8 problems it solved, Python managed 7. For 6 of the 11 problems considered the Haskell approach finds more correct solutions with respect to the test set, Python found more solutions for 4. It should be noted that the original implementation of the Python approach, which used a higher number of generations, achieved better results for 5 of the problems evaluated, two were unchanged, and 4 problems realised better results than the original implementation. For the problems where Haskell performs better it does so against this implementation and the original Python implementation [12].

How quickly correct solutions were found was also investigated across the runs. In Table 4 we present minimum, maximum and mean number of generations for problems that were solved across the 100 runs for both approaches.

Table 3: Number of times a solution passed all of the training and test examples for each problem of the benchmark suite, presented for the Haskell and Python methods.

	Haskell		Python	
Name	Train	Test	Train	Test
Number IO	100	99	100	100
Small Or Large	30	24	0	0
Compare String Lengths	94	85	12	0
String Lengths Backwards	0	0	35	34
Last Index of Zero	0	0	2	2
Vector Average	67	64	0	0
Super Anagrams	30	25	51	38
Vectors Summed	100	68	0	0
Negative To Zero	0	0	68	66
Median	100	96	39	21
Smallest	100	100	99	89

Additionally, we present some examples of evolved artefacts for both approaches. We present evolved code for two problems that both approaches found solutions to and one that only Haskell found a solution to. These were selected primarily as they open up points for discussion; further examples for each problem are available on the GitHub page[4].

Number IO[5]

For the Haskell approach most solutions take some form of:

```
evolvedFunction :: Int -> Double -> Double
evolvedFunction a b = ((+) b (fromIntegral (a)))
```

note that only the expression after the = symbol is evolved.

However, from further inspection of the results, we found that many evolved needless instructions, e.g.

```
 = ((+) (min b b) (fromIntegral (a)))
```

where `min b b` is not in the canonical form `b`. Others took advantage of the input data range of $[-100, 100]$, leading to solutions that would not generalise under a different input range, e.g.

```
 = ((+) (min (fromIntegral (661)) (fromIntegral (a))) b)
```

For the Python approach, some evolved examples have a similar degree of interpretability, but these are far less common. For example, a typical Python solution:

```
def evolve(in0, in1):
  # evolved code
  i0 = i1 = i2 = int()
  f0 = f1 = f2 = float()
  res = float()
  f0 = in1
  res = (in0 + div(in1,int(round((res +(int(math.floor(
  max(int(res), ( f0 * f2 ))))- i1 ) )))) )
  # end of evolved code
  return res
```

[4] http://frsrgrrw.github.io/why-functional-program-synthesis-matters

[5] Given an integer and a float, return their sum.

Note that the function has been annotated both to show which section is evolved, and for aesthetic purposes. Also, a portion of the code is predefined, i.e. the first grammar production initialises the variables. Perhaps the function is less interpretable, but this is still a correct solution; notice that the second term in the sum that increments `res` computes to 0 as it uses many non assigned variables, this is a commonality in Python solutions. Despite the Python solutions often being less readable than Haskell solutions, Python in fact generalised to every test input for this problem, Haskell evolved solutions that solved all but one.

Median[6]

For the Median problem, typical Haskell solutions are more complex than those shown above, and require careful inspection to understand how they work, e.g.

```
evolvedFunction :: Int -> Int -> Int -> Int
evolvedFunction x y z = (min (max x z) (max (if (if (even
    (abs 37)) then (z <= y) else (odd 18)) then (max y z)
     else (min (min z x) (max z 80))) y))
```

Note that we have formatted the output for aesthetic purposes; in reality, it is one long expression with no line breaks or indentations.

Again, and similar to the Haskell effort at Median, the Python evolved solution requires some inspection before understanding how the result is produced.

```
def evolve(in0, in1, in2):
  #  evolved function
  b0 = b1 = b2 = bool()
  i0 = i1 = i2 = int()
  res = int()
  if int(4.0) is max(i2, +int(27.0))
    b0 = True
  else
    i1 = -i2
  res += min(max(in2, min(in1, in0)), +( max(+min(
  min(abs(i0), in2), max(abs(in1), mod(i0,i1))),
  abs(-min(-i1, -res))) + max(( int(01.0) * in0 ),
  (in1 + res))))
  # end of evolved code
  return res
```

Vectors Summed[7]

No Python runs returned a correct solution for this problem. We show two Haskell solutions as they prompt an interesting discussion point. Many of the evolved Haskell solutions are similar and realise a function that looks like a human-coded solution, e.g.

```
evolvedFunction :: [Int] -> [Int]
evolvedFunction xs = (zipWith (+) (ys ++ []) xs)
```

The addition of the empty list is surplus to requirements in this case, but the evolved code still computes the correct output and it will in fact do this for any sound input. However, similar to the Number IO problem, we can evolve code that will not achieve these results, yet it will pass all test cases due to the data ranges considered, e.g.

```
= (zipWith (+) (take 183 xs) (take 966 ys))
```

This fails for vectors of length > 569. This is a manifestation of the brittleness of ERCs. Undoubtedly, similar problems exist in other evolved functions for other problems, but they are less interpretable so the possible error is unknown.

Of the problems that the Haskell approach could not solve, if we consider the reference implementations[2], all use higher order functions to find a solution. It is perhaps possible to encode solutions in other ways but they would be highly irregular. Higher order functions are included in the grammar but the implementation is not generic enough to allow any function (that is a sound type) to be used.

7 DISCUSSION

As shown in the results, both approaches find solutions to 8 of the 11 problems. The Haskell approach finds some solutions that generalise to unseen data for all the problems it solves, whereas the Python approach finds some solutions that generalise to 7. Moreover, the Haskell programs do this with a mostly higher success rate over the 100 runs. The nature of Haskell code means that the grammar does not need to include as many rules for control flow, structuring and variable assignments. This reduces the size of the grammar and therefore reduces the search space. It is reasonable to suggest this was one of the reasons for the success of the Haskell approach. It is also notable that many solutions are found in the first generation. This indicates that many solutions may have been found during random initialisation; this requires further investigation.

We suspect that to tackle some of the more complex problems in the benchmark, the Haskell grammar will require extension to enable it to make full use of higher order functions and functional motifs like Lambda expressions. Extending the grammar will obviously increase the number of rules and therefore it is likely that there may be some performance hit on the results for the solved problems presented here.

With differing grammar sizes a question that arises is how we conduct a fair comparison between different approaches. The fact that we expect a functional language approach to produce a smaller grammar is a motivating factor, but this will inevitably lead to the approach being critiqued for using a smaller grammar than others. Similarly, what functions to include or not include is a difficult question. Although the benchmark suite makes recommendations on this, when the programming paradigm changes and functions become intrinsically different, following these guidelines requires more thought.

Closely related, is the matter of which additional (to the input and output) types to use for a problem. Particular types are recommended in the benchmark. However, if the goal of automatic programming is to discover some unknown function or program, then should we be specifying explicit domain knowledge in the choice of which types can be used? One argument is to use whichever input and output types we are given, but many programs require internal types which differ from those expressed in the input and output data. An interesting challenge would be a benchmarking competition where each problem is given without its name such that no domain knowledge, other than input and output data, is known. This would possibly provoke GP, or any other program synthesis system, to evolve truly arbitrary programs.

[6]Given three integers, return their median.
[7]Given two equal-sized lists of integers, return a list of integers that contains the sum of the inputs at each index.

Table 4: Generation when a correct solution was found, if a correct solution was found. Minimum, maximum and mean across the runs is presented. Where no correct solution was found within the evaluation budget we mark with ✗.

	Haskell			Python		
Name	**Min.**	**Max.**	**Mean**	**Min.**	**Max.**	**Mean**
Number IO	1	6	1.59	1	14	2.90
Small Or Large	19	99	42.09	✗	✗	✗
Compare String Lengths	1	66	5.73	30	99	61.58
String Lengths Backwards	✗	✗	✗	5	90	33.46
Last Index of Zero	✗	✗	✗	5	75	40.00
Vector Average	1	67	7.03	✗	✗	✗
Super Anagrams	3	98	40.33	9	98	46.35
Vectors Summed	1	9	1.83	✗	✗	✗
Negative To Zero	✗	✗	✗	3	100	25.53
Median	2	12	5.51	11	100	57.23
Smallest	1	4	2.17	6	44	15.22

For evaluation we tried to select a subset of problems that were representative of the full benchmark. This is difficult to do, especially when comparing different approaches. Different approaches will perform better on different problems; inspection of Table 3 shows that the two approaches are mostly solving different problems well. This is the nature of optimisation problems: there are no free lunches here! But with respect to the Haskell approach, it is maybe unsurprising that it achieved good results for the two vector problems as it was able use functions that operated over entire data types. Again, how we compare approaches needs careful consideration.

We have shown that the functional approach may have some benefits; however, this is not to say that it is the best method, rather that it is a useful tool. In fact, as most mainstream languages support functional idioms, a hybrid language grammar may be of some utility. However, this would not result in pure programs and unwanted side-effects would still be possible.

8 CONCLUSIONS AND FUTURE WORK

In this preliminary work, we investigated the use of GGGP with a purely functional language as the target language for evolution to tackle program synthesis problems. The grammar design pattern for program synthesis was used to develop a BNF grammar for the Haskell language. Functional languages can express more programs with a smaller function set through the use of modular programming by higher order functions, and as such, realise a smaller search space. Furthermore, programs written in a purely functional programming language like Haskell have desirable properties that can be exploited after evolution that might make them preferable as code artefacts for general use, e.g. automatic refactoring to equivalent functions, simpler to test, lack of side-effects, concurrency for free (in certain cases).

The Haskell GGGP program synthesis system was evaluated on a subset of problems from the General Program Synthesis Benchmark Suite. It was compared against the same GGGP system with imperatively styled Python programs as the evolution target. For 6 of the 11 problems investigated, the evolved Haskell programs were able to find more correct solutions that generalised to unseen data. For 3 of the problems selected, we were not able to evolve a correct solution in Haskell. The Haskell grammars were smaller in size than the Python grammars, as they require fewer code constructions such as loops or variable assignments, and the smaller grammar is likely a factor in the better performance. It was found that the grammar design pattern requires further extension to enable the full use of the Haskell Prelude made available for evolution. The Haskell grammar lacked the expressiveness to solve some of the problems as it was not able to use all higher order functions as they would be used in hand crafted Haskell programs. A Haskell grammar perhaps requires, as well as data type grammars, function signature grammars, so that functions can be passed as arguments in the grammar productions.

Inevitably, this early work raised more questions than answers. Future work should address grammar design for the optimal evolution of functional programming languages. How can we create a grammar where we pass functions (possibly with corresponding arguments) to other functions via grammar rules and productions? Swan et al. [40] have previously investigated synthesis of functions as inputs for higher order functions which may be effective, another option is the use of attribute grammars [30]. Further considerations include: the introduction of local variables, Lambda expressions and the inclusion of Haskell's syntactical sugar, like pattern matching and list comprehensions.

Future work should also consider the updated program synthesis benchmark [16], Lexicase selection, and run for a greater number of generations for a complete comparison.

Finally, some of the proposed benefits of evolving in a functional language should be included in the evolutionary search, e.g. using the mathematical properties of Haskell functions to ensure that individuals in the population are unique, term rewriting (to the canonical form) for more interpretable evolved code and to reduce the search space, and type refinements, or dependencies, to constrain the search space.

ACKNOWLEDGMENTS

This work was supported by the EPSRC CDT in Robotics and Autonomous Systems (EP/S023208/1).

REFERENCES

[1] Franck Binard and Amy Felty. 2008. Genetic programming with polymorphic types and higher-order functions. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation (GECCO '08)*. Association for Computing Machinery, New York, NY, USA, 1187–1194. https://doi.org/10.1145/1389095.1389330

[2] Rudy Braquehais and Colin Runciman. 2017. Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. Association for Computing Machinery, New York, NY, USA, 40–51. https://doi.org/10.1145/3122955.3122961

[3] Iwo Błądek and Krzysztof Krawiec. 2017. Evolutionary Program Sketching. In *Genetic Programming (Lecture Notes in Computer Science)*, James McDermott, Mauro Castelli, Lukas Sekanina, Evert Haasdijk, and Pablo García-Sánchez (Eds.). Springer International Publishing, Cham, 3–18. https://doi.org/10.1007/978-3-319-55696-3_1

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/ARXIV.2107.03374

[5] Gopinath Chennupati, R. Muhammd Atif Azad, and Conor Ryan. 2015. Synthesis of Parallel Iterative Sorts with Multi-Core Grammatical Evolution. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO Companion '15)*. Association for Computing Machinery, New York, NY, USA, 1059–1066. https://doi.org/10.1145/2739482.2768458

[6] Rosetta Code. 2022. *FizzBuzz*. Retrieved April 11, 2022 from https://www.rosettacode.org/wiki/FizzBuzz#Python3:_Simple

[7] Rosetta Code. 2022. *FizzBuzz*. Retrieved April 11, 2022 from https://www.rosettacode.org/wiki/FizzBuzz#Haskell

[8] Alexandre Correia, Juliano Iyoda, and Alexandre Mota. 2020. Combining model finder and genetic programming into a general purpose automatic program synthesizer. *Inf. Process. Lett.* 154 (2020). https://doi.org/10.1016/j.ipl.2019.105866

[9] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 835–850. https://doi.org/10.1145/3453483.3454080

[10] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Erik Hemberg, and Michael O'Neill. 2017. PonyGE2: Grammatical Evolution in Python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Berlin, Germany) *(GECCO '17)*. Association for Computing Machinery, New York, NY, USA, 1194–1201. https://doi.org/10.1145/3067695.3082469

[11] Alcides Fonseca, Paulo Santos, and Sara Silva. 2020. The Usability Argument for Refinement Typed Genetic Programming. In *Parallel Problem Solving from Nature – PPSN XVI (Lecture Notes in Computer Science)*, Thomas Bäck, Mike Preuss, André Deutz, Hao Wang, Carola Doerr, Michael Emmerich, and Heike Trautmann (Eds.). Springer International Publishing, Cham, 18–32. https://doi.org/10.1007/978-3-030-58115-2_2

[12] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2017. A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming. In *Genetic Programming (Lecture Notes in Computer Science)*, James McDermott, Mauro Castelli, Lukas Sekanina, Evert Haasdijk, and Pablo García-Sánchez (Eds.). Springer International Publishing, Cham, 262–277. https://doi.org/10.1007/978-3-319-55696-3_17

[13] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2018. Extending Program Synthesis Grammars for Grammar-Guided Genetic Programming. In *Parallel Problem Solving from Nature – PPSN XV (Lecture Notes in Computer Science)*, Anne Auger, Carlos M. Fonseca, Nuno Lourenço, Penousal Machado, Luís Paquete, and Darrell Whitley (Eds.). Springer International Publishing, Cham, 197–208. https://doi.org/10.1007/978-3-319-99253-2_16

[14] GitHub. 2021. *2021, The State of the Octoverse*. Retrieved April 11, 2022 from https://octoverse.github.com/

[15] Sean Harris, Travis Bueter, and Daniel R. Tauritz. 2015. A Comparison of Genetic Programming Variants for Hyper-Heuristics. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO Companion '15)*. Association for Computing Machinery, New York, NY, USA, 1043–1050. https://doi.org/10.1145/2739482.2768456

[16] Thomas Helmuth and Peter Kelly. 2021. PSB2: the second program synthesis benchmark suite. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/3449639.3459285

[17] Thomas Helmuth, Nicholas Freitag McPhee, Edward Pantridge, and Lee Spector. 2017. Improving Generalization of Evolved Programs through Automatic Simplification. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Berlin, Germany) *(GECCO '17)*. Association for Computing Machinery, New York, NY, USA, 937–944. https://doi.org/10.1145/3071178.3071330

[18] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15)*. Association for Computing Machinery, New York, NY, USA, 1039–1046. https://doi.org/10.1145/2739480.2754769

[19] Erik Hemberg, Jonathan Kelly, and Una-May O'Reilly. 2019. On domain knowledge and novelty to improve program synthesis performance with grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '19)*. Association for Computing Machinery, New York, NY, USA, 1039–1046. https://doi.org/10.1145/3321707.3321865

[20] Zhenjiang Hu, John Hughes, and Meng Wang. 2015. How functional programming mattered. *National Science Review* 2, 3 (07 2015), 349–370. https://doi.org/10.1093/nsr/nwv042 arXiv:https://academic.oup.com/nsr/article-pdf/2/3/349/31566307/nwv042.pdf

[21] J. Hughes. 1989. Why Functional Programming Matters. *Computer Journal* 32, 2 (1989), 98–107. http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf

[22] Zoltan A. Kocsis and Jerry Swan. 2018. Genetic Programming + Proof Search = Automatic Improvement. *Journal of Automated Reasoning* 60, 2 (Feb. 2018), 157–176. https://doi.org/10.1007/s10817-017-9409-5

[23] Krzysztof Krawiec, Iwo Błądek, and Jerry Swan. 2017. Counterexample-driven genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17)*. Association for Computing Machinery, New York, NY, USA, 953–960. https://doi.org/10.1145/3071178.3071224

[24] W. B. Langdon and Wolfgang Banzhaf. 2008. A SIMD Interpreter for Genetic Programming on GPU Graphics Cards. In *Genetic Programming*, Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcázar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–85.

[25] Michael A. Lones. 2021. Evolving continuous optimisers from scratch. *Genetic Programming and Evolvable Machines* (Oct. 2021). https://doi.org/10.1007/s10710-021-09414-8

[26] Simon Marlow et al. 2010. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)* (2010).

[27] Rudy Matela. 2022. *code-conjure: synthesize Haskell functions out of partial definitions*. Retrieved April 11, 2022 from https://hackage.haskell.org/package/code-conjure

[28] Yandu Oppacher, Franz Oppacher, and Dwight Deugo. 2009. Evolving Java Objects Using a Grammar-Based Approach. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation* (Montreal, Québec, Canada) *(GECCO '09)*. Association for Computing Machinery, New York, NY, USA, 1891–1892. https://doi.org/10.1145/1569901.1570220

[29] Stack Overflow. 2021. *Developer Survey 2021*. Retrieved April 11, 2022 from https://insights.stackoverflow.com/survey/2021

[30] James Vincent Patten and Conor Ryan. 2015. Attributed Grammatical Evolution using Shared Memory Spaces and Dynamically Typed Semantic Function Specification. In *18th European Conference on Genetic Programming (LNCS, Vol. 9025)*, Penousal Machado, Malcolm I. Heywood, James McDermott, Mauro Castelli, Pablo Garcia-Sanchez, Paolo Burelli, Sebastian Risi, and Kevin Sim (Eds.). Springer, Copenhagen, 105–112. https://doi.org/doi:10.1007/978-3-319-16501-1_9

[31] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2018), 415–432. https://doi.org/10.1109/TEVC.2017.2693219

[32] Simon Peyton Jones. 2003. Wearing the hair shirt: a retrospective on Haskell (2003). (Jan. 2003). https://www.microsoft.com/en-us/research/publication/wearing-hair-shirt-retrospective-haskell-2003/ Edition: invited talk at POPL 2003.

[33] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. *SIGPLAN Not.* 51, 6 (jun 2016), 522–538. https://doi.org/10.1145/2980983.2908093

[34] Nick Smallbone. 2022. *quickspec: Equational laws for free!* Retrieved April 11, 2022 from https://hackage.haskell.org/package/quickspec

[35] Dominik Sobania and Franz Rothlauf. 2021. A generalizability measure for program synthesis with genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '21)*. Association for Computing Machinery, New York, NY, USA, 822–829. https://doi.org/10.1145/3449639.3459305

[36] Dominik Sobania, Dirk Schweim, and Franz Rothlauf. 2022. A Comprehensive Survey on Program Synthesis with Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation* (2022), 1–1. https://doi.org/10.1109/TEVC.2022.

3162324

[37] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. phd. University of California at Berkeley, USA. AAI3353225 ISBN-13: 9781109097450.

[38] Lee Spector and Jon Klein. 2008. Machine invention of quantum computing circuits by means of genetic programming. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 22, 3 (2008), 275–283. https://doi.org/10.1017/S0890060408000188

[39] Lee Spector and Alan Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines* 3, 1 (mar 2002), 7–40. https://doi.org/10.1023/A:1014538503543

[40] Jerry Swan, Krzysztof Krawiec, and Zoltan A. Kocsis. 2019. Stochastic synthesis of recursive functions made easy with bananas, lenses, envelopes and barbed wire. *Genetic Programming and Evolvable Machines* 20, 3 (Sept. 2019), 327–350. https://doi.org/10.1007/s10710-019-09347-3

[41] Sabrina Tseng, Erik Hemberg, and Una-May O'Reilly. 2022. Synthesizing Programs from Program Pieces Using Genetic Programming and Refinement Type Checking. In *Genetic Programming*, Eric Medvet, Gisele Pappa, and Bing Xue (Eds.). Springer International Publishing, Cham, 197–211.

[42] Philip Wadler. 1998. Why No One Uses Functional Languages. *SIGPLAN Not.* 33, 8 (aug 1998), 23–27. https://doi.org/10.1145/286385.286387

[43] Thomas Welsch and Vitaliy Kurlin. 2020. Synthesis through unification genetic programming. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 1029–1036. https://doi.org/10.1145/3377930.3390208

[44] Gwoing Tina Yu. 1999. *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming*. Ph.D. University of London. https://discovery.ucl.ac.uk/id/eprint/10104592/ Accepted: 1999.