Hybrid Meta-heuristic Frameworks : A Functional Approach

by

Richard James Senington

Submitted in accordance with the requirements for the degree of Doctor of Philosophy.

UNIVERSITY OF LEEDS

The University of Leeds
School of Computing

February 2013

The candidate confirms that the work submitted is his/her own, except where work which has formed part of jointly-authored publications has been included. The contribution of the candidate and the other authors to this work has been explicitly indicated below. The candidate confirms that appropriate credit has been given within the thesis where reference has been made to the work of others. Further details of the jointly-authored publications and the contributions of the candidate and the other authors to the work are included in the **Declarations** below.

Acknowledgements

I would like to thank my supervisor David Duke for his guidance, advice and tolerance over the three years of this project. For their many discussions with me regarding optimisation research in general and metaheuristics in particular I would like to thank Sarah Fores, Ray Kwan and Natasha Shakhlevich. For providing a wonderful example of metaheuristic optimisation on a real application written in Haskell and subsequently a useful case study I would like to thank Noah Daniels and Norman Ramsey of Tufts University. I would also like to thank Peter Wortmann for his advice on the inner workings of the Haskell compiler and the impact this would have upon my own work.

Finally I would like to thank Karolina for her forbearance as I have stared at computer screens for hours and my parents for listening to me talk when I needed to think out loud.

Abstract

Problems requiring combinatorial optimisation are routinely encountered in research and applied computing. Though polynomial-time algorithms are known for certain problems, for many practical problems, from mundane tasks in scheduling through to exotic tasks such as sequence alignment in bioinformatics, the only effective approach is to use heuristic methods. In contrast to complete strategies that locate globally optimal solutions through (in the worst case) the enumeration of all solutions to a problem, heuristics are based upon rules of thumb about specific problems, which guide the search down promising avenues.

Work in the field of Operations Research has gone further, developing generic *metaheuristics*, abstract templates which may be adapted to tackle many different problems. Metaheuristic researchers have created a variety of algorithms, each with their own strengths and weaknesses, and development of metaheuristics now often tries to combine concepts from a number of existing strategies to balance the advantages of the originals, known as *hybridisation*.

This interest in hybridisation has led to the creation of a number of frameworks in imperative languages to assist programmers in the rapid creation and experimentation upon the algorithms. However these existing frameworks have struggled to enable hybridisation of the two major classes of metaheuristic, point based and population based, while being large and complicated to use.

This Thesis investigates a functional approach to hybridisation. Despite superficial analogies between hybridisation and function composition, there are substantial challenges: unlike global search methods that can be explained elegantly in terms of graph traversal, prior work on local search has struggled to articulate a common model, let alone one that can accommodate more esoteric optimisation techniques such as ant colony optimisation. At the same time, these implementations cannot ignore the fact that the development of these techniques is driven by large-scale problems, and computational efficiency cannot be ignored. Given this background, this Thesis makes three substantial contributions. It decomposes metaheuristic search methods into a set of finer-grained concepts and tools that can be reassembled to describe both standard search strategies and more specialised approaches. It resolves problems found in implementing these abstractions in the widely used language Haskell, developing a novel approach based on dataflow networks. The value of functional abstraction in the practice of metaheuristic development and tuning is demonstrated through case studies, including a substantial problem in bioinformatics.

Declarations

Some parts of the work presented in this thesis have been published in the following articles:

Senington R. and Duke, D. (2012), ”*De*composing Metaheuristic Operations”, *Implementation and Application of Functional Languages*, (conditionally accepted)

Senington R. and Duke, D. (2012), ”Combinators for Meta-heuristic Search”, *Journal of Functional Programming*, (Under review)

All these articles are substantially the candidates own work, with guidance from his co-author and PhD supervisor David Duke. No other co-authors have contributed directly to these papers.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis re-examines how the design and implementation of metaheuristics and hybrid-metaheuristics for combinatorial optimisation problems is approached in the pure functional language Haskell. Tackling a problem using metaheuristics often involves a process of experimentation upon existing algorithms, or combining characteristics of existing algorithms, to provide a strategy shaped to the task. The process of rapid experimentation and hybridisation is ideally supported by a system which is very flexible and this thesis argues that a functional approach can deliver flexibility in implementation, hybridisation and adaptation while not sacrificing performance. This is achieved through expressing the component characteristics of a range of well known metaheuristics in terms of fine grained, higher order combinators.

1.1 Motivation

Combinatorial problems, such as factory scheduling problems, are discrete assignment problems which are ubiquitous in areas including the sciences, engineering, economics, business and logistics [37]. These problems are known to be NP-Hard, with the number of solutions to any given problem rising exponentially against the number of assignments to be made. The textbook examples of these problems are Boolean Satisfiability (SAT) and the Travelling Salesperson Problem (TSP), how-

ever other well known examples are Timetabling and Machine Shop Scheduling.

Combinatorial problems can be solved, i.e. the subset of *best* solutions sharing an equally high *quality* may be found, by various *complete* algorithms known to Operations Research, including Depth First and Breadth First Search, Branch & Bound, Dynamic Programming and Integer Linear Programming solvers. These complete algorithms examine every possible solution to a problem, either explicitly or implicitly. Algorithms such as Depth First Search explicitly examine every possible solution, where algorithms such as Branch & Bound or Branch & Cut (for Integer Linear Problems) avoid explicitly examining solutions through *pruning* the search process.

In general however the NP-Hard nature of the problems means that these *complete* algorithms are unable to finish in practical computational time once problem sizes have become too large. While finding provably optimal solutions to these problems is impractical, finding solutions which are superior in quality to the best solutions previously known is still of great value. Hence algorithms which can be shown to provide sufficiently good solutions, in practical computation times, can be seen as "solving the problem".

Heuristics are general "rules of thumb" for specific problems which may be used to guide search processes. The computational cost of the heuristic function must be low, and the solutions that it produces must be of good quality by comparison to results that can be found by brute force search such as a simple Depth First Search. For example, in the well known Travelling Salesperson Problem, a simple but effective heuristic is to add legal edges to a solution in a shortest first order. Using heuristics to guide search gives rise to algorithms such as best first and A* search methods, which can be seen in most standard AI textbooks [68].

Heuristics are problem specific[1] and can require significant work to design methods for new problems, or to create better heuristics for known problems. By comparison *metaheurstics* are template methods that operate in a heuristic like way, and can be *fitted* to many different problems through the provision of standard low level interaction functions. They are iterative methods which generate new solutions through the transformation of previously found solutions, often through stochastic, or semi-stochastic methods. Their iterative nature allows metaheuristics to be run as long as is practical for the problem, exploring increasing numbers of potential solutions.

[1] For example, the heuristic for the Travelling Salesperson Problem, of adding the shortest edge first, cannot be directly applied to other problems, which may not use graphs as the problem model.

While heuristics and metaheuristics cannot guarantee the quality of solutions that are generated, an analysis of the qualities of solutions that are found can give a level of confidence regarding how likely it is that a better solution exists. Metaheuristics have been found to be effective in finding high quality solutions to combinatorial problems in practical time limits[2].

All metaheuristics have many parameters which must be set before the algorithm will perform effectively. Some of these 'settings' are simple types, such as a population size in a Genetic Algorithm, however more often these settings are functional in nature, such as the functions which make changes to the solutions of a problem. Through the settings of parameters, both simple and functional, metaheuristic algorithms are *tuned* to improve performance on specific problems, by taking advantage of the characteristics of the problem. Further attempts to improve performance make use of *hybridisation*, where different metaheuristics, or aspects of them, are combined to overcome the weaknesses of the individual components.

However the *No Free Lunch Theorem* [86] says that there is no single metaheuristic which will perform best on all problems. For each algorithm, with a particular set of parameters, there will be problems that it works well upon, and problems where it is the worst possible strategy.

Evidence for the no free lunch theorem is seen frequently, for example it is rare for initial implementations of metaheuristic search for a problem, with an initial choice of parameters to be effective at generating high quality solutions. The process of exploring a problem, testing different metaheuristics, different combinations of parameter settings and different hybridisations is a time consuming task in terms of both human and computational time.

Researchers have worked to overcome these issues in the application of metaheuristics. One approach is to make use of machine learning techniques, and metaheristic methods themselves [3, 12], in the setting of the parameters to an algorithm for a problem, with the termination criterion being either the rate of convergence, or the quality of solutions found after a specific period. A second approach which has gained popularity over the last decade has been the field of *hyper-heuristics* [7], an approach where the concept of using machine learning and evolutionary computation is applied to the design of metaheuristic algorithms.

[2]Metaheuristics are primarily used to tackle large instances of discrete combinatorial problems. They may also be used when dealing with continuous optimisation problems, however the characteristics of these problems lend themselves to solution through linear programming or scientific computation algorithms. Hence this thesis restricts itself to examples of discrete problems.

The most common approach however has been the design of libraries, toolkits and Domain Specific Languages (DSLs) for the implementation and hybridisation of metaheuristic algorithms. A number of such libraries have been previously created for standard imperative and object oriented languages, such as HotFrame [21] for C++ and OPT4J [55] for Java. However it has been noted that these existing libraries have limitations [56]:

- often requiring expertise in the libraries not just the programming languages;
- being specialised to either population or individual solution based methods but;
- not supporting hybridisation between population and point based metaheuristics well.

1.2 Research goals and Hypotheses

The focus of this thesis is the design and implementation of toolkits and libraries for the construction of hybrid metaheuristics. The goal is to lay the foundations for future research into metaheuristics in Haskell, and provide a tool for other functional programmers to easily leverage metaheuristic methods for their own work. Functional languages have a number of powerful features that may be well suited for this task:

- Higher order functions, that can both take functions as parameters and return functions as results, giving rise to combinator libraries, libraries of orthogonal higher order functions that can be combined to express complex concepts concisely, enabling a greater degree of abstraction than is possible in traditional languages.
- Related to higher order functions is the compositional style of expression, to break down more complex expressions into the combination of smaller blocks, which encourages modularity in the design of libraries.
- Lazy evaluation, which aids declarative expression, such as infinite data structures, for example streams, which provides the *glue* for combining the higher order functions [42].

- Advanced compilers, that are more capable of bridging the gap between natural expression of concepts at the level of source code while maintaining computational traction in the underlying execution model.

There is evidence from other domains that these features can be exploited to provide new methods and insights into a range of problems [45, 75, 85].

It is the hypothesis of this thesis that the use of pure functional languages will provide a more expressive foundation for a toolkit for hybrid metaheuristics than traditional imperative languages. This constitutes the creation of a shallowly embedded Domain Specific Language for metaheuristics in Haskell[3]. The shallow embedding facilitates extension and rapid adaptation of the library through the use of higher order functions and *parachuting* new functionality into chains of composition.

1.3 Main Contributions

The main contributions of this thesis are:

- A revision of the approach to implementing metaheuristics in a functional context through...
- ...an adaptation of a data flow model of computation to the implementation of metaheuristics, through stream processing functions. These provide a basis for a high level transformative and compositional style for hybridisation of the algorithms. The stream model also successfully aligns the pure functional model of computation with the awkward issues in metaheuristics including stochastic functions and a modular method for encapsulating state.
- A combinator library for the expression and implementation of metaheurstics.
- Logical transformations, some of them encoded as compiler rules, to improve the marriage of the high level stream transformation functions with low level performance.

[3] Shallow embedding refers to a situation where the DSL for a problem is created using a subset of the existing language (in this case Haskell) and operating directly upon values. A deep embedding is where the DSL gives rise to a data structure representing a computation that can be evaluated to run the program being created, and often involves parsers. Deep embedding has the advantage of allowing reflection in languages which do not naturally support it [26].

- A demonstration of the application of the library to facilitate rapid experimentation on non-trivial problems, and the provision of effective specialised metaheuristics that practically solve the problems.

1.4 Overview of the Thesis

The Thesis is divided as follows:

- *Chapters 2 & 3* provide background on the two major subjects of the Thesis. Chapter 2 provides the background of metaheuristics, their implementation, tuning, hybridisation and usage, while chapter 3 provides background on functional programming, the key concepts and its previous usage for optimisation in general and metaheuristics in particular.

- *Chapter 4* discusses a variety of approaches that can be used to implement and hybridise metaheuristics in pure functional languages, showing how the direct approaches are limited and introducing the stream based model that this Thesis proposes.

- *Chapters 5, 6 & 7* detail the stream based approach that has been taken. Chapter 5 describes the combinators and shows how they may be used to implement the major metaheuristic algorithms. Chapter 6 shows how the combinators can be used to structure and manipulate functions that interact with problem specific data at lower levels. Chapter 7 shows how the combinators provide the major forms of hybridisation that have been identified by the wider research community.

- *Chapter 8* examines alternative approaches to constructing and manipulating the stream processors that form the core of the framework.

- *Chapters 9 & 10* engage in evaluations of the framework that has been proposed. Chapter 9 provides a comparison with two frameworks implemented in traditional object oriented languages, focusing upon the clarity of the implementations of example metaheuristics and a comparison of performance data. Chapter 10 presents the successful usage of the combinators to explore the design space of possible metaheuristics used in an algorithm for detection of homology in proteins.

- *Chapter 11* concludes the Thesis, summarising what has been presented in the previous chapters and proposing a number of possible directions for future work and research.

Chapter 2

Combinatorial Problems and Metaheuristics

This chapter will describe combinatorial problems in greater detail. It sets out the issues involved in finding high quality solutions to them, how these difficulties manifest themselves when using exhaustive search algorithms and how metaheuristics can avoid similar pitfalls. Five major families of metaheuristics are then described, with details of their traditional imperative expression.

2.1 Discrete Combinatorial Optimisation Problems

A discrete combinatorial problem consists of a set of *variables* and a set of *constraints*. Each variable has a finite domain, and a solution to a problem is a set of assignments of values to the variables. A constraint is a logical predicate which provides additional relationships between the variables of a problem and the values that these variables can take.

For example, consider assigning jobs to a machine in a factory. The machine has four timeslots, and four named jobs. The domain of each timeslot variable is the set of jobs. In any given solution each of these jobs is assigned to a timeslot, where each job can be used only once, and each timeslot can be used only once. This uniqueness characteristic is captured as the constraints of the problem. This

can be expressed more formally as:

$$
\begin{aligned}
&vars = \{timeSlot_0, timeSlot_1, timeSlot_2, timeSlot_3\} \\
&jobs = \{a, b, c, d\} \\
&S = \{\{(var, job)\} \mid var \in vars, job \in jobs\} \\
&\text{subject to} \\
&\{j \mid (\_, j) \in s, s \in S\} \equiv jobs \\
&\{v \mid (v, \_) \in s, s \in S\} \equiv vars
\end{aligned}
$$

A possible solution to this problem is:

$\{(timeSlot_0, a), (timeSlot_1, b), (timeSlot_2, c), (timeSlot_3, d)\}$

In the problem above the constraints described the unique usage of timeslots and jobs, which would be considered common sense in real life. The example will now be extended with a set of time constraints, drawn from the user requirements, where time is represented as the index of the time slot to which the job is assigned.

$timeOf(a) < timeOf(b), timeOf(d) < timeOf(a), timeOf(c) > timeOf(d)$

The presence of these new constraints to the example previously shown reduces the set of possible real, or valid, solutions to three;

$$
\begin{aligned}
&\{(timeSlot_0, d), (timeSlot_1, a), (timeSlot_2, b), (timeSlot_3, c)\} \\
&\{(timeSlot_0, d), (timeSlot_1, a), (timeSlot_2, c), (timeSlot_3, b)\} \\
&\{(timeSlot_0, d), (timeSlot_1, c), (timeSlot_2, a), (timeSlot_3, b)\}
\end{aligned}
$$

Where the problem consists of only variables and constraints, the problem is called a *decision problem* and the task is to find a legal assignment. Other problems introduce the concept of an objective function, a way to price or order assignments that obey all the constraints. In these *combinatorial optimisation* problems the task is to find the best possible assignment, with respect to the objective function. The previous example can be extended with an objective function of the following form;

$$
\begin{aligned}
&objective = minimise(lateness) \\
&\text{where} \\
&lateness = \Sigma\, max(0, n - target_{timeSlot_n}) \\
&target = [(a, 2), (b, 3), (c, 1), (d, 4)]
\end{aligned}
$$

This objective function can be used to calculate the values of the three possible solutions to the problem:

$$\{(timeSlot_0, d), (timeSlot_1, a), (timeSlot_2, b), (timeSlot_3, c)\}$$
$$\begin{aligned} lateness &= max(0, 0-4) + max(0, 1-2) + max(0, 2-3) + max(0, 3-1) \\ &= 0+0+0+2 \\ &= 2 \end{aligned}$$
$$\{(timeSlot_0, d), (timeSlot_1, a), (timeSlot_2, c), (timeSlot_3, b)\}$$
$$\begin{aligned} lateness &= max(0, 0-4) + max(0, 1-2) + max(0, 2-1) + max(0, 3-3) \\ &= 1 \end{aligned}$$
$$\{(timeSlot_0, d), (timeSlot_1, c), (timeSlot_2, a), (timeSlot_3, b)\}$$
$$\begin{aligned} lateness &= max(0, 0-4) + max(0, 1-1) + max(0, 2-2) + max(0, 3-3) \\ &= 0 \end{aligned}$$

Hence for this toy optimisation problem the optimal solution is

$$\{(timeSlot_0, d), (timeSlot_1, c), (timeSlot_2, a), (timeSlot_3, b)\}$$

2.1.1 Complexity

Problems exist which fall within the field of discrete combinatorial optimisation but can be solved with efficient polynomial time algorithms, by exploiting regularities in the problems. A well known example of this is the calculation of the minimum spanning tree of a weighted graph, which may be constructed in polynomial time in the number of vertices and edges of the graph.

The types of combinatorial optimisation problems that metaheuristics are called upon to deal with are NP-Hard problems. For these problems there are no efficient polynomial time algorithms which will find provably optimal solutions, and the number of solutions that must be considered to (provably) solve them rises exponentially with the sizes[1] of the problems. This rising number of solutions to consider causes the computational time required to increase exponentially with the size of the problems.

For example in the previous toy problem there were 4 time slots and 4 values. The number of basic solutions is the number of ways to arrange 4 values in a sequence, which is 4! (24). However if the number of slots and elements to arrange increases to 6, then the number of basic solutions becomes 6! (720), where as if only the values increases then the number of possible arrangements becomes $\binom{6}{4}!$ (10^{12}).

[1] The size of a problem will be measured in terms of the number of assignments that are required to produce a solution. For example a TSP with 10 cities is size 10, since there are 10 positions in the sequence that must be assigned values.

2.1.2 Exhaustive Search Methods

Exhaustive, or complete, search algorithms guarantee to find an optimal solution to a combinatorial problem, given enough time. The most naive approach to search is to use a generate and test methodology. In this algorithm every solution to the problem is constructed, and each in turn is tested for the constraints of the problem, evaluated for the objective function and finally the best selected.

Many well known algorithms such as depth first and breadth first search on trees are in the category of naive algorithms. However these tree based algorithms do allow for an optimisation of the construction process, the embedding of the constraint testing within the tree. Using this approach partial solutions (and every solution that might have been created from them) may be *pruned* from the tree when they fail to satisfy any constraint in the problem. These pruned trees are the basis for the *constraint programming* methodology.

Branch and bound algorithms operate in a similar way to the constraint-pruned trees. These algorithms make use of a *bounding heuristic*, which gives a limit on the quality of solutions which may be derived from a partial solution. This allows the tree to be pruned with respect to the best solution found at that point, without removing branches that may contain the optimal solution, and thus reducing the number of solutions that must be evaluated.

While the pruning of search trees is a powerful tool, they will usually only be able to remove a proportion of the possible solutions that must be considered. Given that the number of solutions rises exponentially as problems grow in size and complexity, these algorithms eventually all reach a point where they cannot complete within practical time limits. Most real world optimisation problems are significantly larger than the sizes of problem which can reasonably be dealt with by complete algorithms.

2.1.3 Example Problems

While optimisation tasks are highly varied in practice, the textbook example problems are Satisfiability (SAT) and the Travelling Salesperson Problem (TSP). In this section these two problems will be described in more detail, and the TSP will subsequently be used for the example problems in this Thesis.

2.1.3.1 Satisfiability

SAT is a the most abstract form of a decision problem, the task of finding an assignment of variables such that all constraints (known as clauses) are *satisfied*. Formally a SAT is defined as;

$$
\begin{aligned}
vars &= \{x_0, x_1, x_2 \ldots\} \\
& x \in \{True, False\} \forall x \in vars \\
terms &= \bigcup_{x \in vars} \{x, \not{x}\} \\
clauses &= \{\{a, b, c \ldots\} \mid a, b, c \ldots \in terms\} \\
& \text{subject to} \\
& \bigwedge_{c \in clauses} (\bigvee_{t \in c} t)
\end{aligned}
$$

While it is possible to tackle this problem without an objective function, as a pure decision problem, it is often approached as a minimisation of the number of currently unsatisfied clauses. This removes the constraints in the previous model, and introduces the following objective function;

$$
objective = minimize(\sum_{c \in clauses} \text{if } (\bigvee_{t \in c} t) \text{ then 1 else 0})
$$

Libraries of these problems can be found online for the purpose of experimentation, for example SATLIB [36].

2.1.3.2 TSP

The Travelling salesperson problem can be seen as the task of finding an optimal route within a transport network. However more abstractly it is the task of finding a shortest Hamiltonian cycle in any connected weighted graph, that is a cycle passing through every vertex exactly once, where the sum of the edge weights is minimised. Formally;

G is a graph (V,E) of vertices and edges

G is a complete graph

p is a pricing function from $E \to \mathbb{R}$

$S = \{\{e\} \mid e \in E\}$

subject to

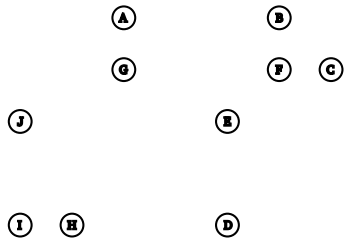
s is a hamiltonian cycle of $G\ \forall s \in S$

$objective = minimize(cycle\_cost \in S)$

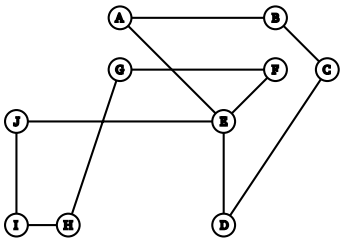
where

$$cycle\_cost(s) = \sum_{e \in s} p(e)$$

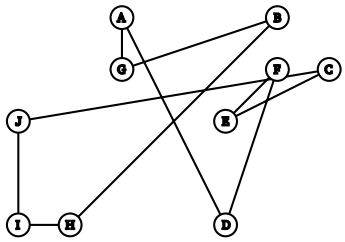
Figure 2.1 illustrates a simple Euclidean TSP, with some potential solutions to the problem. This example was generated stochastically, however TSP problems can be drawn from the locations of real cities, or found in other fields of science and technology such as X-Ray crystallography [5]. These problems can be symmetric (the length of every edge in the graph from A to B is the same as that from B to A) or asymmetric.



(a) Points in a plane, the basis of a Euclidean TSP.



(b) An set of edges that is not a legal solution.



(c) A non-optimal solution.

Figure 2.1: An example Euclidean TSP.

Some groups of travelling salesperson problem have been proved to have properties that give rise to bounding functions for branch & bound, for example a lower

bound in a Euclidean TSP is the minimum spanning tree of a set of points. Effective bounding functions for special classes of the TSP has allowed for the solution to very large instances in practical computational time. However not all problems admit such functions, for example asymmetric problems often do not have good functions for the lower bound of a tour. In all cases as the size of the instances grows (measured in terms of the number of vertices in the problem), the travelling salesperson becomes impractical to solve, whilst easy to describe.

Many examples of the TSP can be found online, for example the TSPLIB [66] provides a library of known problems of various forms and drawn from a variety sources.

2.1.3.3 Real world problems

In practice real world problems, while having the same underlying issues as SAT and TSP, will often exhibit many additional properties and characteristics which must be dealt with. For example when timetabling for a school or university there may be multiple, possibly contradictory, objectives to try to optimise against, such as the preferences of the teachers, the students and the administration.

While it is often possible to transform these problems into instances of the TSP or SAT, in practice it is often more practical to design specialised algorithms, exploiting specific properties of the problems, to more efficiently resolve conflicts in the constraints.

2.2 Metaheuristic Concepts

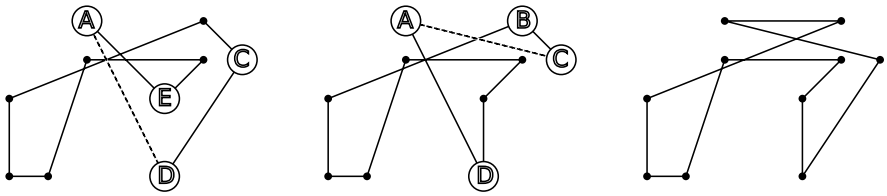
While large problem sizes preclude finding provably optimal solutions, this does not preclude making an attempt to solve a problem. One approach is to use an exhaustive method[2] while limiting the time allowed for search and taking the best solution found at the point of termination [64, 68]. However to make exhaustive methods effective requires discovering good bounding functions for each specific problem, and this can be a time consuming task. In practical terms heuristic and metaheuristic methods tend to outperform exhaustive search on new or rapidly changing problems.

[2]It is presumed that the choice of exhaustive algorithm would be the best possible which is known for the particular problem, using appropriate bounding functions and exploring the tree in a best first order.

Heuristic methods use problem specific *rules of thumb* to find solutions to problems, usually in a constructive manner. For example, a solution to a TSP may be constructed through iteratively adding edges to a solution, where the next edge to be added is both the minimum available and legal in the context of the previous edges.

Heuristic methods suffer from the drawback of requiring problem-specific decision algorithms to be created. To overcome these weaknesses researchers developed a group of iterative algorithms known as *meta*heuristics[3]. These are heuristic in that they make use of rules of thumb for investigating problems, but are more abstract and may be applied to many different problems. They can be described as *generically definable rules of thumb* for optimisation algorithm construction.

Metaheuristics tackle optimisation problems by moving between *complete* solutions to problems i.e. assignments which do not break any of the constraints of the problem, but which may not be of optimal quality. The moves that the algorithm makes are between solutions that are defined as *similar* to one another. The definition of similarity is problem specific, and a problem may admit more than one definition of similarity. For example, in the TSP one form of similarity is the concept of adjacent city exchange, where two solutions are similar if their sequences are the same, except two of the cities, which are adjacent in the sequence. The use of this form of similarity can be seen in Figure 2.2.



(a) CDAE ⇒ CADE (b) BCAD ⇒ BACD (c) Final State

Figure 2.2: A sequence of TSP solutions, where pairs of nodes are exchanged in each solution.

[3]This Thesis will restrict its definition of a metaheuristic to iterative algorithms, however, if the definition were taken as *a generally applicable rule for writing algorithms which generate solutions* then greedy constructive algorithms could also be seen as metaheuristics. Greedy constructive algorithms make use of a problem specific heuristic to define the quality of decisions in a constructive process and always take the *best* decision, and hence are based upon a more generally applicable approach.

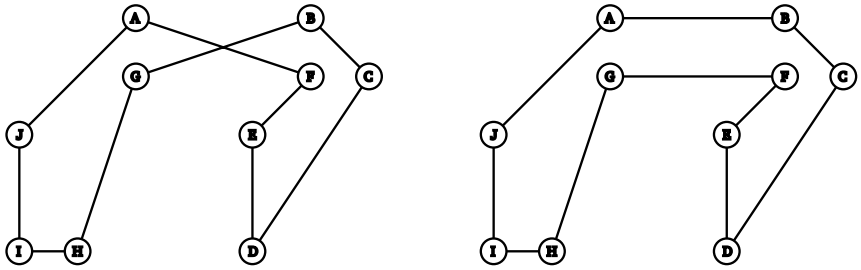


Figure 2.3: An example Euclidean TSP demonstrating similarity of solutions under limited edge modification.

2.2.1 Why should metaheuristics be effective?

The effectiveness of metaheuristics is based upon the assumption that solutions with a similar structure will also have similar values. For example, consider the TSP in Figure 2.3. This example gives two possible hamiltonian cycles through the problem, which differ in terms of edges $(A,F),(B,G)/(A,B),(G,F)$. The difference in value will only be affected by the differences in cost of these edges; the remaining edges and their contribution to cost remaining unchanged.

This intuition suggests that there are *regions* of higher quality solutions, connected by their similarity, and regions of low quality solutions. Metaheuristics seek to explore the set of solutions, following the patterns of the solution qualities towards groups of higher quality solutions.

This assumption of similarities of value in the same locality is not always well founded. For any strategy it is possible to design *trap* problems, where the *landscape* of the problem will lead the search process away from the optimal solution rather than towards it. However many problems, including most real world problems, do have the high locality property, and so metaheuristics may be used successfully in practice.

2.2.2 Similarity and Neighbourhood

The concept of similarity of solutions gives rise to the concept of the *neighbourhood* of a solution, the set of all solutions which are similar to a given seed solution[4]. A function that maps every solution in a problem to its complete neigh-

[4]A neighbourhood is usually defined constructively as those solutions which may be generated from a seed, through a function which generates similar solutions. The use of constructive methods gives rise to the alternative definition of similarity, solutions that are one *edit step* apart.

bourhood is called a *neighbourhood function*.

Perturbation functions are a related approach to neighbourhood functions, which generate only one arbitrary solution from a particular seed, typically through a stochastic decision making process. The complete set of solutions which a perturbation function can generate from any particular seed is equivalent to a neighbourhood relationship. In practice they may be implemented in terms of a choice from a neighbourhood function, but are more often a separate type of operation. In imperative languages the stochastic component is usually hidden as the implicit use of a random number generator or other internal state.

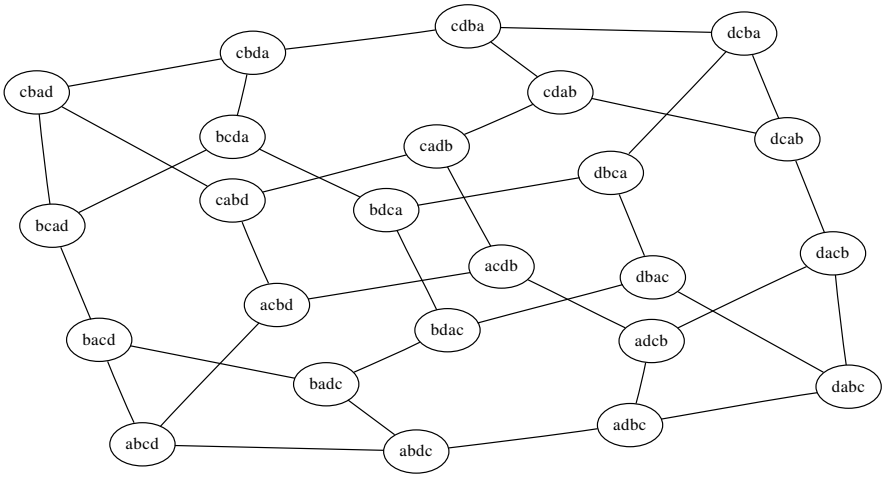


Figure 2.4: The graph of a solution space with adjacent city exchange neighbourhood relations for a small TSP.

Each definition of neighbourhood for a particular problem imposes a structure on the set of solutions to that problem, a graph structure where each solution is a vertex and edges are the neighbourhood relationships. The search algorithms can be thought of as *navigating* these graphs, exploring the landscape. An example of such a graph structure for a small TSP problem can be seen in Figure 2.4.

Different metaheuristics typically make use of either neighbourhood or perturbation functions to interact with the problem representation. For example Simulated Annealing tends to use perturbation functions while Iterative Improvers tend to use neighbourhood functions.

2.2.3 Recombination

Neighbourhood and mutation operations are the basis of so called *point based* algorithms, which operate upon one solution a time. Population based algorithms

operate differently, maintaining a collection of solutions at each stage, and proceeding through the *recombination* of either the whole, or subsets of the collection. For example *Genetic Algorithms*, from which the name *population* comes, operate upon a collection, selecting small numbers (typically two) and *breeding* solutions from these selections to generate a new population.

The primary method for this is the *recombination* function, which maps a collection of solutions onto a single new solution. Recombination functions are typically designed to preserve characteristics such that as the number of parent solutions possessing a trait rises, so does the likelihood of the new solution possessing this trait. Choices must often be made between mutually exclusive components of different parents and in these cases stochastic processes are often employed. Hence recombination functions will usually generate a single arbitrary solution from an input collection, where several are possible.

This Thesis further describes recombination functions as an analysis operation that yields information about characteristics and commonalities of these elements in the parent solutions. The choice of characteristics is then carried out by a separate *constructive* function, taking into account the constraints of the optimisation problem being considered. A final repair function is then used, as the selection of characteristics can often leave *holes* in the final solution, such as is seen in common edge detection and selection for the TSP [58]. This description is adopted because it gives stronger guidelines for how to create recombination operations for many problems which have the desired characteristic preservation properties.

2.2.4 Generically definable search processes

All metaheuristics seek to define a generically applicable process of decision making which can be used to tackle search problems. At an implementation level there is a tension between the need to produce executable code and the desire to abstract the general patterns of search that the metaheuristics seek to use. This tension is resolved through the definition of a set of operations that provide a structure for the generically defined metaheuristic to operate upon. For example

Iterative Improvement requires that solutions to the problem are orderable and that a neighbourhood function is provided. Generically defined selection methods for orderable collections are then employed to drive the search process.

Genetic Algorithms require that solutions are orderable and that a recombination

operation is provided. Generically defined stochastic selection processes are then employed to manage selection from populations for the recombination process, and hence drive the search process.

It is interesting to consider if this is the right level of abstraction for metaheuristics to be defined over, and this question will be returned to in later chapters.

2.3 Metaheuristic Algorithms

Metaheuristics can be grouped into a number of different *families*, where each family is based upon an underlying idea for how to conduct the search process. This section lists the families which will be the focus of the remainder of the thesis, with a description and some of their most common variations. These families have been chosen as the focus of this thesis because they are the most common examples in the literature and encompass a range of major features of the algorithms such as (1) population and point based methods (2) memory and learning (3) stochastic and deterministic operations.

2.3.1 Random Generation & Random Walk

Random generation of new solutions and strategies that move from solution to solution within a search space making all decisions randomly are degenerate search processes, in that there is no strategy. These algorithms typically operate through iterated perturbation operations, or stochastic selection from neighbourhood functions if it is desirable to preserve the concept of neighbourhood, otherwise random assignment can be used to create entirely new solutions.

Both Random Generation and Random Walk are often used as a component of other hybrid strategies, as methods to *escape from local minima*. This is the situation where another strategy has explored and found solutions, but has now ceased to find further improving solutions. At this point it is desirable to move the search on to a new part of the search space where better solutions may be found, either by restarting at a new solution or allows the process to wander for a while.

These degenerate approaches are also used to provide an initial baseline for assessing the performance of other metaheuristic methods.

2.3.2 Iterative Improvers

Iterative Improvers, also known as hill climbers, are the simplest form of guided metaheuristic whose defining feature is that the algorithm should not move away from a solution unless the new solution improves upon the previous solution. These algorithms have the useful property that they often rapidly improve upon initial solutions; their disadvantage is that they also rapidly become stuck in solutions with no local improving move.

These are usually implemented using a *neighbourhood function*[5]. In this formulation, a number of solutions to a problem are generated from the current solution, and one is chosen which improves upon the current solution. A number of variations can be created through the method used to choose from among these *improving* solutions;

First Found: accept the *first* improving solution seen in a neighbourhood.

Maximal: select the *best* solution from those which improve within a neighbourhood.

Minimal: select the *weakest*[6] solution from those which improve within a neighbourhood.

Stochastic: select a solution at *random* from those which improve within a neighbourhood. While a uniform distribution is the most obvious, it is also possible to use other distributions, and when combined with an ordering of the neighbourhood (e.g. best to worst) can give rise to many more variations within this family of algorithms.

This family of subtly different techniques suggests a description of Iterative Improvers as the pairing of a piece of selection logic with the generic concept of an improving neighbourhood.

The imperative template for this metaheuristic is given in Figure 2.5.

[5]It is also possible to implement Iterative Improvers using a perturbation method only. Solutions are generated by perturbation and only when an improving solution is found does the algorithm move to it. This can be seen as a variation upon first found iterative improvement.

[6]The success or failure of a slow ascent strategy (relative to the other options) is dependent upon the landscape of the problem. In practice this is more likely to be a useful variation in hybrid strategies where a range of tactics is employed.

1. *currentSol* $\leftarrow$ *a random solution*
2. *repeat*
   (a) $n \leftarrow$ *neighbourhood of currentSol*
   (b) $n \leftarrow \{ s \mid s \in n, s$ ***better than*** *currentSol* $\}$
   (c) terminate program ***if*** $n \equiv \{\}$
   (d) *currentSol* $\leftarrow$ *perform select logic upon n*

Figure 2.5: An imperative implementation of the template for Iterative Improvers

2.3.3 TABU search

TABU search [27, 28] is usually introduced as an adaptation of the first found Iterative Improver algorithm, which makes use of memory to allow the strategy to attempt to make further progress once the Iterative Improver would have ended. Recently seen solutions are stored in a TABU (FIFO) list (the memory) of fixed size and as the algorithm proceeds older solutions are *forgotten*. At each stage newly generated solutions are only considered if they are not present on the current TABU list and in this way the algorithm limits how likely it is to revisit solutions.

Like Iterative Improvers, TABU search is usually based upon a neighbourhood function. These neighbourhoods are then filtered to remove solutions that can be matched by elements present in the TABU list. The simplest version of TABU search, based upon pseudo code from [27] is presented in Figure 2.6. This version will always take the best solution in the TABU pruned neighbourhood, and so at first will act like a maximal iterative improver. Once a local minimum is reached, the TABU extension begins to have more impact, allowing the process to leave the local minimum, moving to the best solution in the local neighbourhood that is not in the TABU list. This precludes the process retracing the path that led to the minimum, instead forcing it to follow a new search path. In this way TABU search can be thought of as an escape strategy for an iterative improver.

In practice the comparison of solutions to detect membership of the TABU list is a very slow operation, and so it is more usual to store TABU *operations*. Under this system, the modification used to create a new solution is stored, and the reversal of the *modification* is TABU, thus still disallowing undoing recent actions. For example, in the TSP a pair of cities may be exchanged to create a new solution, the swapping of these cities subsequently would be TABU.

The use of TABU operations introduces a new issue to TABU search, that an

1. $currentSol \leftarrow$ *a random solution*
2. $tabuList \leftarrow \{\}$
3. *repeat*
   (a) $tabuList \leftarrow tabuList \cup \{\ currentSol\ \}$
   (b) $n \leftarrow$ *neighbourhood of currentSol*
   (c) $possibleCandidates \leftarrow \emptyset$
   (d) *for each* $s \in n$
   if $s \notin tabuList$ *then*
   $possibleCandidates \leftarrow possibleCandidates \cup \{s\}$
   (e) $currentSol \leftarrow$ *best solution* $\in possibleCandidates$
   (f) ***if*** *tabuList too large then*
   $tabuList \leftarrow tabuList - \{$*oldest element of tabuList*$\}$

Figure 2.6: An imperative implementation of the template for TABU Search, based upon a description in Glover [27].

operation may be on the TABU list, but its application might produce an unseen and improved solution to the problem. To combat this the concept of *aspirational acceptance* [27] was introduced, where the first instance of an improving solution is always accepted, and solutions that do not improve and are on the TABU list are removed from the neighbourhood. The imperative template for this aspirational TABU search can be found in Figure 2.7.

In a similar manner to Iterative Improvers, the formulation of TABU search allows for a number of variations based upon the selection methods in use. The standard method, as seen in the imperative template, is to use a first found selection with the improvement element until that fails, at which point it is normal to take the best of the elements not found on the TABU list, a maximal selection operation. Both of these selection operations can be varied, making use of the options that have already been described in the previous section 2.3.2.

The model of TABU search described in Figure 2.7 is also known as *short term memory* [27] TABU. More complex variations have been created with medium and long term memory, each of which is prioritised in a different way.

Introduction of stochastic elements into TABU search has also been used. These may take the form of stochastic selection routines, stochastic deletion from the neighbourhood set or stochastic modification of the TABU list, at any given

1. *currentSol* ← *a random solution*
2. *tabuList* ← {}
3. *repeat*
   (a) $n \leftarrow$ *neighbourhood of currentSol*
   (b) *nextCandidate* ← $\emptyset$
   (c) *nextCandidate* ← *for each* $s \in n$
      i. ***if*** s ***better than*** *currentSol then*
         return s and exit loop
      ii. ***if*** $s \in$ *tabuList then*
         $n \leftarrow n - \{s\}$
   (d) ***if*** *nextCandidate* $\equiv \emptyset$ *then*
      nextCandidate ← *perform select logic on n*
   (e) *tabuList* ← *tabuList* $\cup$ { *information about nextCandidate* }
   (f) ***if*** *tabuList too large then*
      tabuList ← *tabuList* $-$ {*oldest element of tabuList*}
   (g) *currentSol* ← *nextCandidate*

Figure 2.7: An imperative implementation of the template for aspirational TABU Search, based upon a description in Glover [27].

step of the process. The use of stochastic modification of the strategy can improve speed of search, by reducing the number of computations required at each step and often improves robustness of the search strategy through improved variation in the solutions explored. Stochastic modification of the TABU list is a key component of *Robust Taboo Search* [78].

2.3.4 Simulated Annealing

The Simulated Annealing (SA) metaheuristic was proposed by [50] and later independently by [9]. The algorithm draws inspiration from statistical thermodynamics, specifically the modelling of the annealing process. It uses the perturbation functions rather than neighbourhoods and will usually use stochastic perturbation rather than deterministic selection.

At each step in SA a current solution is perturbed to generate an alternative solution. The perturbation may be defined in terms of the random selection of a value from a small neighbourhood, as might be found in Iterative Improvement

1. *currentSol* ← *a random solution*
2. *t* ← *initial temperature*
3. *repeat*
   (a) *p* ← *perturb currentSol*
   (b) *acceptProbability* ← *movementProbability (currentSol,p,t)*
   (c) *r* ← *generate random value*
   (d) ***if*** *acceptProbability* > *r then*
   currentSol ← *p*
   (e) *t* ← *update temperature t*

Figure 2.8: An imperative implementation of the template for Simulated Annealing

or TABU search, however SA is usually more effective where the range of solutions that are possible is larger. This is computationally feasible because only one solution is considered at each step.

Once an alternative solution has been created it is then accepted or rejected by a decision process which is controlled by a real valued *temperature*, which constrains the quality of a solution that is likely to be accepted. Temperature is determined by a *temperature strategy*, which may be defined mutually recursively with other parts of the system. A template for the standard imperative implementation of Simulated Annealing can be found in Figure 2.8.

The standard computation to calculate the probability of acceptance or rejection of a solution at a given temperature is the same as that proposed in the original papers on Simulated Annealing and derived from statistical thermodynamics. The function is defined as

$$saAccept(c,p,t) = e^{\frac{energy(c)-energy(p)}{t}}$$

where c is the current solution, p is the perturbed solution, t is the current temperature of the system, and the *energy* function gives the quality of the solutions. The output of this acceptance function is treated as a probability, and movement occurs where a uniformly generated value between 0 and 1 is higher than the result of this function. The *saAccept* equation will always give a value greater than 1 for situations where the new solution p is better than (where the objective is minimisation)

the previous solution, and hence be certain of being accepted. Hence Simulated Annealing will act like an Iterative Improver where the opportunity presents itself.

The most common component of Simulated Annealing which is varied from problem to problem is the temperature strategy (also known as a cooling schedule, where the temperature can only reduce). Commonly only the constants of the cooling schedule equation are varied when tuning SA. The three major groups of cooling equation are used:

linear, a linear relationship exists between the temperatures, usually a subtraction of some constant, however this strategy is not usually effective, with the temperature dropping too quickly.

logarithmic, a logarithmic relationship between the temperatures, $\frac{c}{log(t+d)}$, where c is start temperature, t is the time that has elapsed (or the number of iterations) since the start of the process and d is a further parametrising constant.

geometric, a geometric relationship between the previous and the current temperature, usually multiplication by a constant between 0 and 1.

More complex temperature strategies are possible, involving reheating the system once a point of stability or convergence of the solution quality over a specified period has been reached, or staying at a specific heat level until convergence has been reached. An alternative approach that has been tried is to apply a heating strategy rather than cooling [54]. Where the temperature strategy depends upon the relative qualities of the solutions being found by the search it is known as an *adaptive* temperature strategy.

Geometric cooling strategies are the most commonly used, providing a good balance between the linear schedules which are too fast, and the logarithmic strategies which are too slow. It has however been proved that a logarithmic strategy, with a starting temperature which is high enough relative to the energy values of the solution space, and with a small enough value for d will converge to the optimal solution of the optimisation problem [31]. This result is not usually used in practice because the cooling rates and corresponding time requirements for the algorithm tend to be impractically long.

Simulated Annealing can be thought of as a hybrid of Iterative Improvement and Random Walk, where the balance between the two strategies is controlled by the current temperature of the system. At high temperatures the algorithm will tend to act more like a Random Walk, but as temperatures drop it acts more and more like an Iterative Improver.

2.3.5 Genetic Algorithms

Genetic Algorithms are an approach to optimisation inspired by the process of evolution found in nature, first formalised in their current form in 1975 [35]. In each iteration these algorithms operate over a *population* of solutions rather than from an individual *point* in the solution space. The algorithm mimics natural evolution through *selecting* and *breeding* solutions from the population such that *better* solutions are more likely to contribute to solutions in the future population. In order to encourage movement through the solution space away from known high quality solutions, genetic algorithms also make use of perturbation in the role of mutation in natural systems, making stochastic changes to some members of the population at each iteration.

The process of recombination usually operates upon only two solutions at a time, in a similar manner to the most common pairwise reproduction processes found in nature. Mutation rates can vary a great deal, but are usually set quite low, as high mutation rates lead to far too chaotic a search process. The template for imperative genetic algorithms can be seen in Figure 2.9.

1. *population* $\leftarrow$ *generate popSize solutions*
2. *repeat*
   (a) *nextPopulation* $\leftarrow \{\}$
   (b) *loop from 1 to popsize*
      i. $r \leftarrow \{\}$
      ii. *loop from 1 to how many to recombine*
         $r \leftarrow r \cup \{$ *select 1 from population* $\}$
      iii. *nextPopulation* $\leftarrow$ *nextPopulation* $\cup \{$ *recombine r* $\}$
   (c) *mutateSet* $\leftarrow$ *select for mutation from nextPopulation*
   (d) *mutateSet'* $\leftarrow \{$ *mutate m* $\mid m \in$ *mutateSet* $\}$
   (e) *population* $\leftarrow$ *(nextPopulation* $-$ *mutateSet)* $\cup$ *mutateSet'*

Figure 2.9: An imperative implementation of the template for Genetic Algorithms

Selection processes in biological systems are based upon the concept of *fitness* of the individual, with the concept that fitter individuals will breed with greater success. This is mimicked in genetic algorithms through basing selection likelihood on the value of the objective function of the optimisation problem.

Most variation and tuning of genetic algorithms is achieved through the size of

populations and the rates of mutation. More complex refinements are based upon how selection for recombination and mutation is performed. For example selection for recombination may be allowed to select each solution many times, only once, or several times with some limit. The advantage of allowing each solution to be selected many times is that good solutions are more likely to be selected for recombination, however this can also limit the portion of the population which is actively selected from, as it tends to pick from the best, ignoring the rest. Similarly, mutation can be limited to the worst, or be applied with equal likelihood to all solutions, or be based upon a fixed repeating pattern.

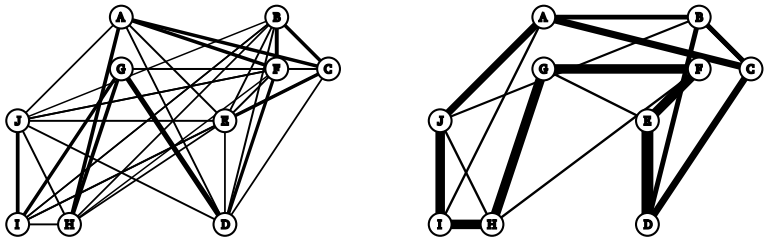
Other common patterns of effect in biological evolution can be used as inspiration for further variations upon the basic genetic algorithm pattern. For example the *island* evolution pattern [29], where a number of genetic algorithms are run in parallel and occasionally exchange solutions, known as *migration*.

The standard methods of recombination are based upon taking two solutions and creating a third, with elements of each. For example the *crossover* method [68], applicable where the solution to a problem can be represented as a string or list of numbers, creates a new solution by cutting these two strings and taking the first half of one and the second half of the other. Crossover is effective for SAT problems but quickly runs into problems where the optimisation problem has more constraints, such as TSP, because simple string slicing and concatenation often produces illegal solutions [6]. Due to this problem the more general term, recombination, for functions which produce solutions in some way based upon the *parents* is preferred in this Thesis.

2.3.6 Ant Colony Optimisation

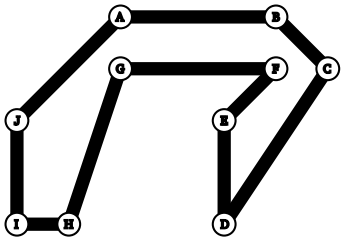
Ant Colony Optimisation (ACO) is also usually described as a population based method, although it can be implemented with a population size of one, making it a point based algorithm. The algorithms draw inspiration from the study of insect colony construction and food foraging methods, first developed as algorithms for optimisation in the 90s [16, 17].

ACO can be thought of as a reinforcement learning algorithm, which seeks to learn how to construct good solutions. This is managed through the weighting of possible decisions at each stage of the construction process, using the pheromone metaphor, where the pheromones are computed from the solution or solutions of the previous iteration.



(a) Random solutions to the TSP as Pheromone Trails

(b) Pheromone trails for the TSP after a number of iterations



(c) The pheromone trail for the TSP in the final state

Figure 2.10: An example of the iteration of pheromone trails for an ACO solution to a TSP.

Once a set of solutions is created and the value of their objective functions is found, the objective values are converted into pheromone weightings, and summed together to create an overall map of the problem. The future ants run over this map making stochastic decisions, but weighting each one by the thickness of the pheromones on the options ahead of them. Each group of ants add new pheromones over the old trails, while the older trails decay with time, usually managed by simply multiplying every value by some constant between 0 and 1. This concept is particularly well suited to the TSP, since both are related to navigation of a graph structure. The TSP is used to illustrate the concept in Figure 2.10. A template imperative implementation of the ACO can be found in Figure 2.11.

The original ACO methods operated over single solutions at a time, managing the pheromone build up using the fading concept. The alternative method which has been explored is to use the ACO as a population based method, with many solutions being generated at each iteration, and their pheromones collectively making the new map [30]. Under the population based model fading of previous trails can be eliminated, or not, as the programmer chooses. The ACO method of pheromone based construction for solutions has also been proposed as a more general way to perform recombination in genetic algorithms [6].

1. *pheromones* $\leftarrow \{$*initialise pheromones*[a]$\}$
2. *repeat*
   (a) *solutions* $\leftarrow \{\}$
   (b) *for i in* $\{1 \ldots$ *number of ants* $\}$
   solutions $\leftarrow$ *solutions* $\cup \{$ *create solution from pheromones*$\}$
   (c) *pheromones* $\leftarrow$ *degrade pheromones*
   (d) *for s in solutions*
   pheromones $\leftarrow$ *pheromones* $+$ *pheromone trail of s*

[a] The initialisation can be done in a variety of ways, including a random setting of values for each decision, but more usually all weights are initialised to 0.

Figure 2.11: An imperative implementation of the template for Ant Colony Optimisation

2.4 Models of metaheuristic operation

The description of metaheuristic algorithms as a series of instructions or steps shows some similarities between the algorithms, such as TABU and Iterative Improver sharing a first found improvement concept. A number of alternative models are presented here for the purposes of (i) showing how metaheuristics are taught and thought about, (ii) illustrating how they operate in a more conceptual way and (iii) finding a model that can form a basis for a practical functional library for metaheuristics.

2.4.1 Navigation of a Graph

A common metaphor for the operation of metaheuristics is that of graph navigation[7], and one such description can be seen in [37]. The graph that is to be navigated is formed of the candidate solutions, which give the vertices, and the neighbourhoods of solutions which give the edges, such that if one solution is a neighbour of another, then an edge will exist between them in the graph. At each point in the process a metaheuristic will be at one of these vertices or solutions, and will be able to examine the local neighbourhood, before making a decision to move to one of these. In this way the action of decision-making is placed at the heart of the description.

This metaphor provides a way to decompose the description of meta-heuristics into two activities; (i) the construction of the graph and (ii) the navigation of the graph. Given that a graph represents the solution space of a problem that is to be solved, the size of a graph will grow at the same rate (exponentially) as the number of solutions to a problem[8]. This exponential growth is relative to the number of assignments in the model of the problem, which is the measure of the size of the problem, as described in Section 2.1.1.

The use of a lazy language, such as Haskell, aids this division of the model description from the search logic, as has previously been observed [42]. It is a harder task in a traditional imperative language to achieve this, not least because a naive implementation will attempt to allocate and compute the entire search space

[7] Another metaphor for metaheuristics, similar to that of exploring graphs, which is often used is that of exploring a labyrinth.

[8] The graphs will also grow exponentially in terms of the edges as problem sizes increase, however the size of the neighbourhoods is usually constrained, to keep the search process at each iteration manageable. This leads to the vertex growth being the key issue, with the increase in edges being a function of the increase in vertices.

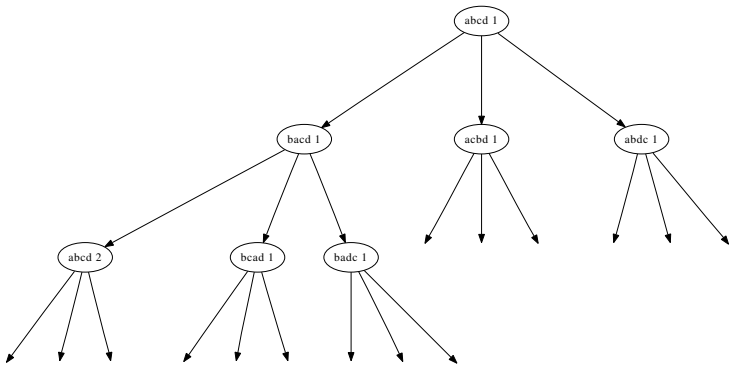


Figure 2.12: An illustration of part of a repeating tree, representing a search process.

for a problem, before beginning the search of the data structure. By comparison a lazy language provides a memoized, on demand data structure, that either returns values that are known, or augments the structure as it is explored.

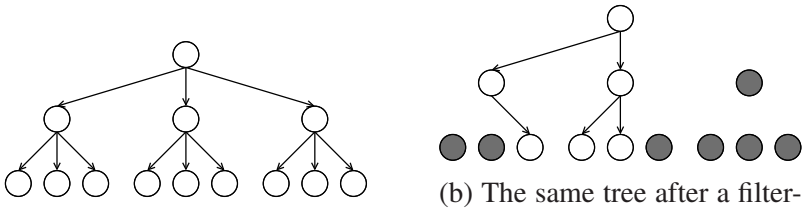
2.4.1.1 Tree representation

The process of navigating a graph of decisions can be implemented directly in declarative languages through the creation of a *repeating tree of decisions*, where every vertex of the tree carries a solution of the problem, and where the levels of the tree represents time, or number of iterations of the search process. The tree can then be *navigated* in an equivalent way to the process of exploring the graph of the solution space. Where two vertices share the same solution this approach will cause the recomputation of the solution (Figure 2.12), however this is also true of imperative implementations [9].

The use of an infinitely generating tree to represent the search space can be used to allow the expression of some aspects of metaheuristics in terms of transformations of the tree. For example a recursive modification of the tree to remove all branches where the local relationship between the parent and child is not improving, converts a tree into an improving tree, the basis for Iterative Improvement algorithms as seen in Figure 2.13.

The repeating structure of the tree is more usefully employed when dealing the the concept of a TABU filter. This would be a transformation which prunes

[9] In Haskell it is possible to memoize the function that computes each node of the tree [33]. If this is done then the resulting data structure is just a lazy representation of the graph of the problem, however the longer the program runs for the larger the memory footprint that will be built up. Ultimately this would result in exponential memory use and so may not be practical in all circumstances.



(a) An unfiltered tree

(b) The same tree after a filtering transformation

Figure 2.13: An illustration of how a Rose tree structure, capturing neighbourhood relationships may be filtered.

branches of the infinitely repeating tree based upon the recent series of solutions used to get to that point. Since one solution may be reached several times, at each point the pruning process can be different, giving a subtly different structure to the tree.

Each level of the tree can also have the order of the child elements rearranged, capturing other aspects of metaheuristics. For example, shuffling every level implements stochastic aspects seen in algorithms such as random walk and stochastic Iterative Improvement, whereas sorting the trees prepares them for algorithms such as maximal improvement.

The trees can then be explored, or flattened, to yield the solutions of the algorithm as a sequence. This approach of generation, transformation and flatten is also seen in other work on search strategies in pure functional languages [47, 59]. Using this approach it is now possible to use the tree model of search to express a number of point-based metaheursitics (such as Iterative Improvers, Random Walk and TABU) in a *compositional* style. A sketch can be seen in Figure 2.14.

1. *make tree*
2. *improvement prune*
3. *sort tree*
4. *left edge flatten*

Figure 2.14: An abstraction of the process of Iterative Improvement through the composition of a series of tree transformations.

2.4.1.2 Limitation

This metaphor for the representation of search spaces as a labyrinth of choices to be navigated by the algorithm is most effective when dealing with the *point*

based algorithms such as TABU, Iterative Improvement and Simulated Annealing. However the population based methods do not fit the model, operating over multiple solutions and jumping to new solutions through recombination, rather than navigating the labyrinth directly.

It is possible to adapt the tree model of search by replacing the solutions at each vertex of the tree by populations, and the search moves between the *populations* of the system, rather than through *individual* solutions. However finding a satisfactory equivalent to the successor relationship for population based method has been difficult. For the point based methods the successors of a node represented a finite and relatively small *neighbourhood* of the solution. For a population based method, the successors should represent the possible successor populations, which is a neighbourhood of every possible population that could be created through the recombination of the original population. This is at odds with the basis of a genetic algorithm which already incorporates a clever marriage of stochastic elements (the selection of each parent could be any in the population) with the improving component (the selection is biased towards better parents). This raises many questions such as (i) in what order are the populations generated by the construction process, and (ii) what do transformations of the tree, such as sorting, now mean? The size of this neighbourhood of populations also presents practical difficulties for many of the transformations previously considered such as sorting and selecting, which must now consider combinatorially large neighbourhood sizes.

2.4.2 Expansion and Contraction of Decisions

The graph model of metaheuristic operation is not practical for population based methods, but a model of the algorithms as a series of instructions does not clearly identify the underlying patterns of computation being used. The ”expansion and contraction” model is proposed in this Thesis as a more abstract model, one based around the decision-making processes of the algorithms.

Figure 2.15 shows an example of this model when applied to a first found iterative improver. The neighbourhood function increases or *expands* the options that are available within the metaheuristic. The later transformations then reduce or *contract* the number of options that are available until a final decision is made. The process of the algorithm becomes a cycle of enumerating options, reducing these options and finally making specific selections from these options.

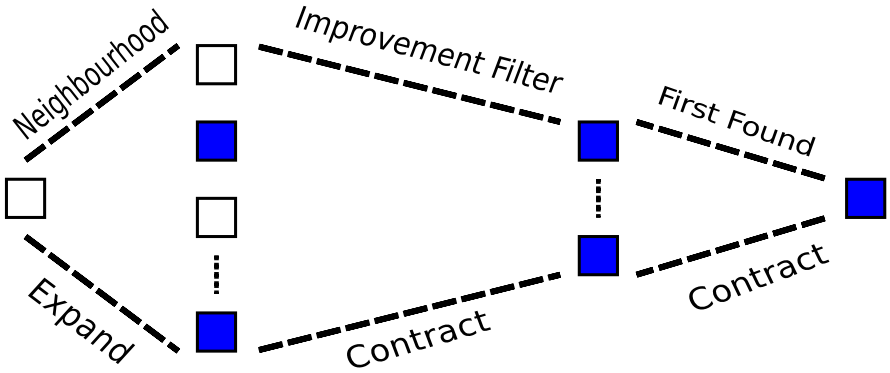


Figure 2.15: Model of first found iterative improvement as the expansion and contraction of decisions.

This model of metaheuristics can be applied to population based algorithms such as genetic algorithms, where the population provides the initial supply of options, and selections reduce this down until there are two parents. The recombination action is then applied to reduce these two into one new solution. The repeated application of this action to a collection gives rise to a new population of options.

This model can also be applied at a finer grain to the process of perturbation, where the perturbation of a solution is modelled as the creation of options and the filtering and selection between these options until a new complete solution is created. This concept will be revisited to in Chapter 6.

2.5 Tuning, Fitting, and Hybridisation

While used to tackle large instances of combinatorial problems, metaheuristic algorithms are not silver bullets. The No Free Lunch Theorem [86] says that there is no one metaheuristic, nor one set of settings for any given metaheuristic, which is ideal for all problems. Another way to describe this concept is that maximising for performance on one class of problem will be offset by worse performance on another class of problem.

Alleviating these issues is a major topic in metaheuristics and has led to a number of different approaches. One method of avoiding the danger of a single method for a problem is work on hybridisation of algorithms, combining two or more algorithms to diversify the search and enable each individual technique to build upon the results of the others [65]. One approach to hybridisation is to examine the actions and patterns of computation in existing metaheuristics and pro-

vide a general template structure for the metaheuristic designer [24,81]. The other method is to use computers to automate the processes of tuning settings [4] and the automation of the design of metaheuristics, a field known as *hyper-heuristics* [7].

A further time consuming task is the creation of the low level operators, such as neighbourhood and recombination, for an optimisation problem. These operators must be created for each problem, to bridge the gap between the high level metaheuristic logic and the problem specific data structures.

2.5.1 Design of Low Level Operators

The three key forms of low level operator have already been discussed and are; neighbourhood, perturbation and recombination. Each problem will usually admit more than one of each form of operator. Each operator imposes a different structure on the search landscape, which the metaheuristic will be required to navigate. How effective a final metaheuristic will be is dependent upon how well the search process is able to explore the landscape created by the operator. Changing the operator can improve the alignment between the search process and the solutions of the problem, enabling it to find better solutions more easily.

For example, in the TSP perturbation and neighbourhood operations can be based upon the swapping of cities in the sequence. However this tends to be less effective than operations based upon the concept of a set of edges forming a path in the graph, where change occurs through deleting and inserting edges [58]. This pattern extends to recombination, with the most effective recombination method being based upon constructing solutions where as many edges as possible are found in one or both of the parents [58].

A simple solution to this difficulty of operator design is to reduce the *dimensionality* of the problem. In the case of neighbourhood functions, this can be interpreted as increasing the size of the neighbourhoods, to allow the search process to explore more of the search space more quickly. This quickly leads to the theoretical best situation of every solution being in the neighbourhood of every other solution, and the search process simply picks the best. However in practical terms this *best* situation is clearly impractical, and illustrates the need for operators to balance a trade off between the structure of the landscape that they impose upon the problem and the computational cost of generating new solutions.

The choice and design of low level operators is as important as the tuning of basic parameters to metaheurstic success.

2.5.2 Hybridisation

The hybridisation of metaheuristic techniques is, perhaps, an artificial concept, which has arisen for historical reasons. Raidl puts it this way;

> At the beginning, however, such hybrids were not so popular since several relatively strongly separated and even competing communities of researchers existed who considered their favourite class of metaheuristics generally best and followed the specific philosophies in very dogmatic ways. [65, p. 2]

Hybridisation is the exploration of algorithms with the aim of finding more successful methods for optimisation, through the combination of various concepts from the available *toolbox*. The topic of hybridisation will be covered in more detail in Chapter 7.

2.5.3 Tuning

When first implementing a metaheuristic for a combinatorial problem the initial setting of parameters for the algorithms is often done with very little information about the problem. It is subsequently found that the initial metaheuristic created this way is fairly ineffective, and experimentation must be carried out in order to achieve reasonable and consistent solution quality. The setting of parameters to maximise performance for particular problems or classes of problem is known as *tuning*.

The tuning of the parameters of metaheuristics for particular problems is known to be a difficult task, where "trial and error" [82] is often used to discover the correct values. The problem of discovering how to set the parameters in a more scientific way has been investigated (for an example see [13]) however the range of possible values for the parameters can make this into a combinatorial problem itself. For this reason the approaches of machine learning [3] and Hyperheuristics have also been turned towards the task of automating the *tuning* of parameters for metaheuristic algorithms.

It has previously been pointed out that some parameters to metaheuristics, such as problem specific operators, are *functional* parameters, rather than static values. This Thesis will restrict itself to considering these functional parameters, and not consider the tuning of static values further.

2.5.4 Hyperheuristics

Hyperheuristics [7] refers to a form of metaheuristic research that focuses upon automatic development of metaheuristic search strategies. This can either be carried out as *offline*[10] development of a metaheuristic before using the algorithm in a practical context, or *online* development and execution on problems.

2.6 Summary

This chapter has dealt with the background material on metaheuristics, *generically definable rules of thumb for decision making in iterative search processes*, and the problems that they are typically called upon to deal with. The major algorithms that will be the focus of this Thesis have been introduced and several models of how they operate, both for metaphorical description and practical implementation, have been described.

The final model that was described, using a concept of expanding and contracting sets of decisions, provides additional insight into the description of metaheuristics as generic patterns of decision making. The model causes the design of metaheuristics to become the task of understanding where choices are coming from, before applying a number of standard computations (such as selection or filtering) to decide how to choose between them.

While metaheuristics have limitations, these have been described and the approaches that are taken by metaheuristic researchers and practitioners to overcome them (tuning and hybridising) have been described. In general the various parameters, operators, tuning and hybridisation are all treated as different issues and tasks in the literature. However in a functional programming context, where functions can be parameters and return types, all of these different components in metaheuristics can be treated as first class components of the language. For example operators are just parameters to metaheuristic algorithms and hybridisation is the combining of metaheuristic functions.

[10]The terms offline and online here are used in the same sense as in machine learning, where offline learning means development against training data and evaluation against test data, as separated from the execution of the result in a practical context. Online learning is the development of the metaheuristic simultaneous to its usage.

Chapter 3

Functional Programming

Functional languages are based upon the *lambda calculus* of Alonzo Church [10], and working implementations have been in existence since the 1950s. The first functional language was LISP and since it was created a number of other languages have been created including ML, Scheme, Single Assignment C (SAC) and Haskell.

All of these languages share the concept that functions should be considered "without special treatment" [2] in the language, being able to be passed as parameters, and can be returned as results of the evaluation of functions[1]. Functional languages then employ higher order functions to abstract common patterns of computation, to achieve separation of functionality, code reuse and modularity. Examples of this are *map* and function composition ($\circ$), where *map* abstracts the pattern of performing an operation on every element of a list and $\circ$ allows for creating a new function from the sequencing of the actions of two others. The types of these functions in Haskell are shown below.

$$map :: (a \to b) \to [a] \to [b]$$
$$(\circ) :: (b \to c) \to (a \to b) \to a \to c$$
$$foldl :: (a \to b \to a) \to a \to [b] \to a$$

These examples also include the two key functions that form the basis of *map*-

[1] A function which either takes other functions as parameters and/or returns a function as its result is called a *higher order function*.

reduce, here named *map* and *foldl*. This illustrates how functional programming is finding major uses in traditional branches of programming and industrial applications.

This Thesis exclusively uses the Haskell programming language for all program and function examples. A summary and introduction to the notation of Haskell is provided in Appendix C, and a more detailed guide and description of the background is readily available in various textbooks and papers [39,41,60].

3.1 Haskell

Haskell [32, 62] is a pure lazy functional language, characteristics which significantly impact on the issues and results presented in this Thesis.

Purity requires that there are no hidden input or outputs (usually called side effects) from any function. This requirement enables many useful features of functional languages such as;

- the clarity of the programs, because the meaning of a function should be apparent from its local definition; and
- powerful compilation techniques, e.g. stream fusion [11].

Lazy evaluation is the combination of normal order evaluation with memoization of shared expressions, and provides demand driven computation while avoiding unnecessary recomputation of values. Normal order evaluation enables the declaration and use of recursive and potentially infinite data structures, such as the simple list $[0..]$ which refers to the list of integers, with no limit (save the computers memory). This is possible because a lazy language can create a list defined as a value, followed by the computation which will yield the rest of the list, when forced to do so. Previously computed values in an infinite data structure are preserved and can be shared between multiple other computations, including the computation that yields the remainder of the data structure.

3.2 Haskell and Operational Research

There has has been previous work combining functional programming and operational research, however far less concerted effort has been applied to the particular subfield of metaheuristics in functional languages. In this section the major

branches that have been explored using the Haskell programming language will be considered.

3.2.1 Accessing External Solvers

Many software systems for optimisation have been written and refined to a significant extent in standard imperative languages. The most direct way to access this existing capability is to make use of the Haskell Foreign Function Interface (FFI). This has been done for several existing systems such as the GNU Linear Programming Toolkit, which is accessed using *glpk-hs*[2] and a package for non-linear optimisation based upon the *CG_DESCENT library* [3].

This approach has not been applied to the field of hybrid metaheuristics. The possible reasons for this inaction can only be speculated upon, however the very large number of possible libraries, the lack of interface standards, and the lack of complete agreement within the field itself are likely causes.

3.2.2 Systematic Search

Systematic search is a mainstay of combinatorial optimisation in OR and also of decision and game playing systems in the field of Artificial Intelligence. Systematic search is based upon the metaphor of the *search tree*, where each branch leads to decisions and the leaves represent possible solutions. The most basic method of systematic search is to exhaustively generate every leaf of a tree, and compare them all to find the best.

This tree based search can be represented in a very natural way in functional languages, taking advantage of the recursive nature of the functions to call search upon nodes of the tree until leaves are discovered. This is an example of an implementation of depth first search in Haskell, for computing the set of permutations of a list, using recursive calls:

$$
\begin{aligned}
&dfs1 :: Eq\ a \Rightarrow [a] \rightarrow [[a]] \\
&dfs1\ [\,] = [[\,]] \\
&dfs1\ xs = concat\ [map\ (x{:})\ \$\ dfs1\ (filter\ (\not\equiv x)\ xs) \mid x \leftarrow xs]
\end{aligned}
$$

[2]Haskell library available on Hackage, LouisWasserman, 2012, `http://hackage.haskell.org/package/glpk-hs`.

[3]Haskell library available on Hackage, Felipe Lessa, 2012, `http://hackage.haskell.org/package/nonlinear-optimization`.

Similar functions can be programmed for other tree search algorithms such as Branch&Bound. This method of creating a new function for each strategy is monolithic in nature and lacks reuse of code.

One approach to providing improved code reuse and generality is to separate the broad structure of the tree, in terms of the set of descendents from any given node, from the ordering of these nodes and the exploration of the tree [47,59]. The data type of the tree itself is the standard Rose Tree implementation for Haskell[4], making use of a recursive data structure (see Appendix C). An initial tree is created through *unfolding* the structure. This tree may then be transformed to give the structure particular desired characteristics, such as ordering of the nodes of the tree at each level. Finally the tree is flattened or explored in a particular way, resulting the a sequence of solutions to the problem.

The creation, structuring and exploration of a search tree for a problem may be achieved in a compositional manner. In the following example depth first search is reimplemented and shown alongside breadth first search. In these examples *unfoldTree, levels* and *flatten* are found in the Haskell tree library, and *succF* is a successor function.

$$dfs2\ succF = flatten \circ unfoldTree\ succF$$
$$bfs\ succF = concat \circ levels \circ unfoldTree\ succF$$

To create more sophisticated strategies, such as best first or branch&bound requires the insertion of the tree transformations. What follows are the types of the functions which may be composed to achieve these effects [5].

$$stochasticRearrange :: RandomGen\ g \Rightarrow g \rightarrow Tree\ a \rightarrow Tree\ a$$
$$prune :: (a \rightarrow a \rightarrow Bool) \rightarrow Tree\ a \rightarrow Tree\ a$$
$$sortTree :: (a \rightarrow a \rightarrow Ordering) \rightarrow Tree\ a \rightarrow Tree\ a$$

These methods all take advantage of lazy evaluation of the data structure to only construct as much of the output as is needed to provide the next solution that is requested by the calling process. As solutions are examined and discarded the Haskell environment will free up the resources for reuse, thus making it practical to try to examine a large problem, for which the entire search tree could not be practically computed.

[4] The Rose Tree implementation may be found in the Haskell *Data.Tree* library

[5] In the example the type *Ordering* is a Haskell type with three possible forms, less than (*LT*), greater than (*GT*) and equal to (*EQ*).

A monadic approach has also been proposed [76, 77] to structuring the pattern of systematic search. This approach takes advantage of the monadic interface to lists to allow the structuring and pruning of the search process through the provision of alternative data structures, however ultimately it cannot overcome the underlying difficulty of exhaustive search in combinatorial problems.

3.2.3 Constraint Programming

Constraints are an integral part of optimisation problems and real world problems often have more complex sets of constraints on the solutions than do their academic counterparts. Constraint programming techniques focus upon the constraints of the problem, exploiting them to prune the search trees as far as possible, before resorting to investigating the remaining branches.

Constraint solvers are based around three key components;

Problem Modelling, the description of the constraints upon the problem, which have been explored in Haskell by Schrijvers et al [70].

Constraint Propagation, is the simplification of the model and pruning of the search space through the application of algebraic laws to constraint equations, allowing the inference of variable values and elimination of unneeded terms. The state of the art at the time of writing in the implementation of constraint propagation is *Constraint Handling Rules* (CHR), which were first explored in Prolog [23]. An implementation of CHR, taking advantage of Haskell's concurrency has been investigated [52], however the libraries are not being maintained at the time of writing.

Search Once constraint propagation has proceeded as far as is possible, progress is made through standard search methods, assigning values to variables and then proceeding with constraint propagation from these assumptions. Combinators for these search methods have been described previously and have been explored further in the particular context of constraint programming by Schrijvers et al [71].

Constraint programming is subject to wide interest in computer science, with two particular systems deserving mention here. The first is Mozart/Oz [67], a multi-paradigm language with an emphasis on declarative programming. Oz has been used for constraint programming (for example [72]), where the problem modelling, constraint propagation and search process can be implemented in a

unified way. Although Oz does have functional properties, metaheuristic search has not been emphasised in research using it.

The second constraint system is COMET [83], a domain specific language created for solving constraint problems with a Java like interface and parallelism implemented through explicit language features. COMET has successfully made use of metaheuristics as the search component for constraint solvers, preferring this approach to more traditional exhaustive search methods.

3.2.4 Metaheuristics

Metaheuristics have not received as much attention within the field of functional programming as the systematic methods that have been seen so far. Some algorithms have been implemented in Haskell such as Genetic Algorithms[6] and Simulated Annealing[7], however in both these cases the implementations are monolithic in design and limit the customisation of the search processes to predefined parameters such as the initial temperature of a simulated annealing system.

There are also implementations of solvers for well known combinatorial problems, such as SAT and TSP[8]. These solvers are problem-specific and not easily adaptable to other problems, and do not necessarily provide library functionality for Haskell, but only standalone programs for tackling these problems. Other libraries have been written for the optimisation of continuously valued numerical functions, but these are not in the purview of this Thesis.

None of these implementations address metaheuristic algorithms in general but only the specific method of interest to the programmer at the time. Customisation of the processes is typically limited to the setting of some of the parameters, such as population sizes and mutation rates within a genetic algorithm. Hybridisation of metaheuristics is not directly addressed, however some hybridisation is possible through the setting of functional parameters, such as mutation or breeding methods in genetic algorithms.

Metaheuristic methods are being used in other projects written in Haskell.

[6] Haskell library available on Hackage, Kenneth Hoste, 2011,
`http://hackage.haskell.org/package/GA`.

[7] Haskell library available on Hackage, Louis Wasserman, 2010,
`http://hackage.haskell.org/package/concurrent-sa`.

[8] SAT Haskell library available on Hackage, Andrii Zvorygin, 2007,
`http://hackage.haskell.org/package/sat`.
Interface to Concorde solver for TSP, Haskell library available on Hackage, Keegan McAllister, 2011,
`http://hackage.haskell.org/package/concorde`.

Of particular interest to this Thesis will be the program *MRFy* [14], a project in computational biology which makes use of metaheuristic methods to identify homology between proteins. The approach taken in this project involved creating a more general framework for metaheuristics and then implementing a variety of general purpose algorithms to tackle their particular problem. MRFy is used as a case study in Chapter 10, where a comparison is made of their framework and equivalent metaheuristics written in the framework of this Thesis.

3.3 Summary

This chapter has presented a very brief introduction to functional programming and the features of Haskell that will be relied upon throughout the rest of the Thesis, with further background provided in Appendix C. The chapter has also shown where techniques from optimisation have already been implemented in Haskell or other functional languages including exhaustive search, constraint programming, external solvers and monolithic implementations of some metaheuristic algorithms.

This Thesis will go beyond the existing work on metaheuristics in Haskell, to consider how a generalised library of generic combinators may be created in a functional setting. It will be shown how the general operators of this library can also be used to abstract patterns of computation in low level operators such as problem specific perturbation and recombination methods. The flexibility of the compositional approach that is utilised will be shown to be highly effective for hybridising metaheuristics both at a high and low level. Finally the use of the library will be demonstrated upon a discrete combinatorial problem of practical importance drawn from computational biological research.

Chapter 4

Metaheuristics in a functional setting

The purpose of this Thesis is to identify a framework for the hybridisation of metaheuristics, both in terms of *whole* algorithms and the interaction or exchange of smaller components. For this to be successful the framework that is used must allow for separation of common elements of the algorithms, such as the termination conditions from the search strategy itself.

This chapter sets out 3 approaches, beginning with implementing metaheuristics as straight forward recursive functions threading state data through function parameters. Implementation issues will be highlighted and attempts at solving these problems will spur the subsequent approaches that are developed in this chapter.

4.1 Generalising Comparison of Solutions

Combinatorial problems vary in terms of their conceptual models (e.g. the use of graphs for TSP and set of boolean assignments for SAT), their data structures their objectives and their pricing functions. In addition to this, in operational research some problems are maximisations, such as maximising the profit from a business, while others are minimisations, such as minimising the travel distance

```
class Optimisable a where
  (=:=) :: a → a → Bool    -- equal value
  (>:) :: a → a → Bool     -- better than
  (<:) :: a → a → Bool     -- worse than

best, worst :: Optimisable s ⇒ [s] → s
best (x:xs) = foldl (λa b → if a >: b then a else b) x xs
worst (x:xs) = foldl (λa b → if a <: b then a else b) x xs

bestOf :: Optimisable s ⇒ s → s → s
bestOf a b = if a >: b then a else b

sortO :: Optimisable a ⇒ [a] → [a]
sortO = sortBy (λa b → if a >: b then LT else GT)
```

Figure 4.1: The optimisable class, to provide generic access to solution content.

in the TSP. A more general characterisation of each of these is the search for *better* solutions, where the definition of better is problem dependent. Libraries for the creation of search strategies must provide a mechanism which allows for the generic interaction with the underlying problems.

While it is possible to mathematically transform a maximisation into a minimisation and vice versa, this still requires specifying the type of optimisation that is required. Since the form of a problem does not usually change once it is defined, this Thesis will restrict itself to an explicit interface for defining the ordering of solutions.

The concept of solution comparison could be captured using the standard Haskell type classes *Eq* and *Ord*, however this would suggest that the problems should always be in the form of either a minimisation or maximisation, depending upon how the functions of the library are written. For this reason *Ord* will not be used, instead a new type class is provided *Optimisable*, which shares strong similarities with *Ord* but specifies that its comparisons are the problem specific optimisation concepts of *better than* and *worse than*.

A further distinction must be made between a solution's *quality* and a solution's *structure*. The type class *Eq* provides equality testing but in optimisation problems it is possible for two solutions to share the same quality but not represent the same solution to the problem. For this reason the type class *Optimsiable* will provide a new equality operator that is specifically for comparing the values of solutions, and *Eq* will be reserved for total equality of solutions. The type class

Optimisable and related functions can be seen in Figure 4.1.

4.2 Naive Implementations

A naive implementation of a metaheuristic is as a monolithic function that maps seed solutions (or seed populations) onto an output solution. Additional parameters that are static for a particular method, such as the cooling rate in simulated annealing or the maximum size of a TABU list can be passed as additional parameters to such a monolithic function. Where the metaheuristic makes use of internal state, such as TABU list or more commonly random number generators, then the data to support these must also be threaded through the recursion as a parameter to the functions.

4.2.1 Random walk

```
randomWalk :: Optimisable solution
              ⇒ (rng → solution → (solution, rng))
              → solution → rng
              → Int → solution
randomWalk perturbF seed = go seed seed
  where go currentBest currentSol rng 0
              = currentBest
        go currentBest currentSol rng n
              = let (s, rng') = perturbF rng currentSol
                    cb' = bestOf s currentBest
                in go cb' s rng' (n − 1)
```

Figure 4.2: An example implementation of the Random Walk metaheuristic, as a recursive function, threading the state of the system as parameters.

Random Walk, conceptually the simplest metaheuristic method, is used as an example in Figure 4.2. The simplest way to implement the iterative process is through the threading of data through parameters to recursive functions;

- the current best solution seen,
- the current solution of the system,
- a random number generator, and

- a counter to indicate when to terminate the process.

While the implementation is correct it fails to separate out the process of random walk, the process of termination, and the process of selecting the best solution seen within the run of the algorithm. Nor does it provide a convenient framework to parametrise over.

Hybridisation of Random Walk with other metaheuristics introduces new problems. The overall termination and selection of the best solution should apply to the overall hybrid, rather than to the individual components. The state of the system, in terms of random number generator and solutions must be correctly managed in the new hybrid. Using the implementation seen here this would involve seeding with a new random number generator each time this component was called.

This simple example highlights the key problems: *a lack of abstraction of the metaheuristic search logic, and a lack of encapsulation of the states of individual components*.

4.2.2 Termination Functionality & Output

The random walk implementation in Figure 4.2 also has the following properties, that it is terminated after a number of iterations, and the final output is the best solution that was found. Hughes notes that a significant advantage of functional programming when dealing with indefinite processes is related to the modularisation that can be achieved through separating operational concerns such as these:

> *"[Program] f can even be a non-terminating program, producing an infinite amount of output, since it will be terminated forcibly as soon as g is finished. This allows termination conditions to be separated from loop bodies - a powerful modularization." [42, p. 9]*

A number of methods for terminating a search strategy can be proposed. While these cover the majority of foreseen circumstances, the list cannot be guaranteed to be exhaustive.

- *Iteration Limit*, is the approach taken in the example seen in Figure 4.2. It is to take the best solution seen over a preset number of steps.

- *Time limited*, in practice it is often of more use to know the best solution found after a predefined period of real time has passed. While there is a direct correlation between the number of iterations possible and the time

taken, this can be difficult to precompute, and so operating in terms of time limits can be preferable.

- *Quality Limited* when a solution is found that is better than a predefined quality. This is the weakest, as it requires prior knowledge of the qualities that are likely to be discovered.

- *Convergence* of the search process, can also be thought of as the stagnation of the search process. Metaheursitics generate improving solutions which follow the same pattern as a *learning curve* in machine learning, and the rate of improvement slows the longer the process is performed for. Information about a series of recent solutions may be used to estimate the current rate of improvement, and the process may be terminated when the estimate of the rate has dropped below a specific value, or reached 0.

The final output of a metaheuristic search is the best solution that is seen during that run. However, while developing a new search algorithm a metaheuristic designer can expect to require a range of other outputs, based upon the solutions that are encountered. For this reason a number of other output transformations should be available, and the following list proposes some, however it is not exhaustive.

- Preserve the best solution seen over the metaheuristics execution;

- Calculate the average quality of solutions over the course of the metaheuristics execution;

- Calculate the rate of improvement in solution quality; and

- Calculate an approximation of the changing rate of improvement over the course of process.

In any given run of a metaheuristic any number of these different outputs might be desired, so it is important that the approach taken enables the clean separation and recombination of functionality. There is also an overlap between the functionality required to provide an approximation of the first differential of the rate of change, and the convergence criteria in the list of termination criteria, encouraging further code reuse.

Every option proposed here requires its own stateful information to be threaded through the program. However the naive implementation of random walk did not

permit this. A simple way to provide an increased element of flexibility to the implementation of termination conditions would be through providing a function of the following form:

type *Termination sol* = *sol* → *Bool*

This would allow filtering-like functionality to be parachuted into a metaheuristic, however this does not provide for more complex termination criteria, such as the runtime of the process, nor does it help in the implementation of the various output options. A more flexible option is to have a metaheuristic take both a termination, and output function as parameters, each threading their own state. These would have the following types:

type *Output state sol output* = *state* → *sol* → (*output*, *state*)
type *Termination state sol* = *Output state sol Bool*

While this approach successfully separates the implementation of termination conditions from the rest of the system, it does not address the problem of threading the state data types seen here. Each implementation of each metaheuristic algorithm is required to take the various initial state types as parameters, and correctly thread the state through the search process. The approach also offers no clear insight into how to manage other forms of state data in a search process (RNGs, memory, cooling strategies), besides threading each separately through the recursion, or combining them into a single state.

4.2.3 Functional properties of metaheurstics

One approach to defining a library for the implementation of hybrid metaheuristics is to enumerate a complete set of functional characteristics that are employed by various metaheuristics, how they are used, and the state information that is required to support them. This would then allow for a single state data type, representing the state of search for any given metaheuristic, or hybrid metaheuristic of unlimited complexity. Table 4.1 shows a reasonable selection of metaheuristics, their variants and the forms of state that they make use of. It is assumed that all of these examples maintain a *best so far* memory.

Table 4.1 indicates that many of the state types are used in only a few of the metaheuristic algorithms that are the focus of this Thesis. Given that there are few truly common elements it does not seem appropriate to use a single data type for

metaheuristic name	**a**	**b**	**c**	**d**	**e**
Random Walk	✓	×	✓	×	×
Iterative Improver	✓	×	×	×	×
Iterative Improver (S)	✓	×	✓	×	×
TABU	✓	×	×	✓	×
Stochastic TABU	✓	×	✓	✓	×
Simulated Annealing	✓	×	✓	×	✓
Simulated Annealing (A)	✓	×	✓	✓	✓
Genetic Algorithms	×	✓	✓	×	×

a: A single solution to a problem being worked upon, such as is used in point based metaheuristics.

b: A population of solutions to a problem being worked upon.

c: The use of random numbers in a strategy requires the threading of the current state of the RNG.

d: A memory of solutions or characteristics of them, may be short term or for the length of the program.

e: Temperature of the system, used in Simulated Annealing and derivatives.

Table 4.1: A table of some common types of state information and the metaheurstics they appear in.

all forms of metaheuristic. It does not seem sensible that every form of memory should be present in every form of metaheuristic, or even how many forms of memory should be provided for. A single monolithic data type for metaheuristics is not the answer, because when new metaheuristics are proposed (for example Ant Colony Optimisation), or new hybrids developed they require either new state information or a different mix of the existing options that may not be provided for.

Such changes to the nature of the search algorithms would require rewriting of the data type for search each time new discoveries were made, and subsequent modification of existing code to support the updated data type. Further more, all updates would be required within the library for metaheuristics itself, they would not be able to be applied as lightweight extensions of functionality by users, a particular problem when the user is proposing a new hybrid of existing components, which should not be a radical shake up of the underlying combinators.

Haskell provides a variety of tools which may be used to enable access to elements of a monolithic state, the construction of state, the composition of state data types and the abstraction of the threading of such information in an automated

way. These will be considered in the next section.

4.2.4 Extensible State

The precise mixture of data that will be required by a given metaheuristic is determined by the functionality that is required of it. Hence the *state* that will eventually be used should be able to be derived from the functions that are used in the construction of the algorithm.

The issue of managing stateful computations in pure functional languages is not new, and a variety of approaches have been developed to aid programmers. This section will examine four approaches in Haskell that can be used to manage state directly; Records, HList, Accessor Classes and Monad Transformers.

4.2.4.1 Records & HList

Haskell provides the concept of a *record* with named fields (seen in Appendix C), allowing both accessor methods and syntactic sugar for update of individual fields in a complex data structure. However records and accessors are not easily extensible, and cannot be derived from function definitions. A proposal has been put forward [48] for an alternative form of record, where accessors are tightly associated with their data types. This would allow a function to be defined over any data type that supported certain forms of named access, for example (from [48]):

$$average :: (Fractional\ a, r\,/\,x, r\,/\,y) \Rightarrow \{\, r \mid y :: a, x :: a \,\} \rightarrow a$$
$$average\ r = (r.x + r.y)\,/\,2$$

This would have provided a mechanism for creating a library of generically defined functions for the characteristics of metaheuristics, and the subsequent derivation of the type of the monolithic state required. However this proposal does not seem to have been widely accepted at the present time, and an implementation does not appear to be available.

An extensible record system does exist in Haskell, based upon the Heterogeneous List library [51]. Two weaknesses exist in this approach, the first is that the Heterogeneous List must provide an explicit wrapper for each data type that it can carry, limiting what can be carried within the records. The second is that access is potentially quite slow, having to search a linked list for each record, as it is requested. Given that during a particular run, the structure of the monolithic state will not change this use of extensible records is inefficient and unnecessary for this application.

4.2.4.2 Accessor Classes

Type Classes provide another alternative way to access elements of a monolithic state, and allows the derivation of how the state type should appear for the programmer to implement. The approach is to define a class for each form of state, and then create functions in terms of these generic accessors, rather than in terms of specific data types. For example, this interface to a TABU list:

$$
\begin{aligned}
&\textbf{class } \mathit{Tabu\ sol\ s} \textbf{ where}\\
&\quad \mathit{addToTabu} :: \mathit{Int} \to \mathit{sol} \to s \to s\\
&\quad \mathit{inTabu} :: s \to \mathit{sol} \to \mathit{Bool}\\
&\mathit{tabuPrune} :: \mathit{Tabu\ sol\ s} \Rightarrow s \to [\mathit{sol}] \to [\mathit{sol}]\\
&\mathit{tabuPrune\ st} = \mathit{filter}\ (\mathit{inTabu\ st})
\end{aligned}
$$

This approach has the disadvantage of requiring the programmer to provide appropriate interfaces to all forms of state for the data type, once the strategy is constructed from the component functions. This task is called *boilerplate* [53] code and while work has been done on automated construction and other generic definition techniques, these have been in terms of writing generic functions that operate over generic data types. The task in metaheuristics is closer to composing a number of existing state types, with associated interface information into a larger more complex data type, while deriving the previous accessor information and doing so automatically to provide a more declarative way to program metaheuristics.

4.2.4.3 Monads and Monad Transformers

Monads have been used successfully to simplify the structure of computations in functional programming and *hide the plumbing* of the data structures that are present. The State Monad generically defines this concept, for some given state we have computations that can yield values, and these computations can be chained together.

$$
\begin{aligned}
&\textbf{newtype } \mathit{State\ s\ a} = \mathit{State}\ \{\mathit{runState} :: s \to (a,s)\}\\
&\textbf{instance } \mathit{Monad}\ (\mathit{State\ s}) \textbf{ where}\\
&\quad \mathit{return\ x} \qquad = \mathit{State}\ (\lambda s \to (x,s))\\
&\quad (\mathit{State\ f}) \gg\!= g = \mathit{State}\ (\lambda s \to \textbf{let } (a,s') = f\ s \textbf{ in } \mathit{runState}\ (g\ a)\ s')
\end{aligned}
$$

The State Monad would appear to be a good choice for modeling the stateful operations that have been discussed here. However while it models state transitions it provides no guidance on the structure of the state data type itself, and the variation in requirements of the state between metaheuristics is an issue that must be overcome.

Monad Transformers [60] provide one way to overcome this issue, by providing a method for composing monads, so as to allow the merging of desired functionality. A monad transformer is defined in Haskell as a new data type which is both a Monad, and part of the *MonadT* type class, which provides the function *lift*. The *lift* function provides a way to access *deeper* parts of the *Transformer Stack*, by transforming functions that operate on that monad in isolation. In the following toy example a State Monad transformer lies between a *Maybe* and a *MaybeT*. An operation on state (*withStateT*) may be accessed by lifting the function over the standard return operation.

$$f :: Num\ s \Rightarrow a \rightarrow (MaybeT\ (StateT\ s\ Maybe)\ a)$$
$$f\ x = lift\ (withStateT\ (+1)\ (return\ x))$$

Monad transformers allow a more compositional approach to the construction of state, however the state is still a stack of components, which will be represented in the type of functions. The programmer using such a system is still required to implement versions of all functions they require for each component they will use, and must know the depth in the stack of each component's state data, so that they can *lift* functions the correct number of times. Where the stack is being changed, as the programmer experiments with a metaheuristic, bringing in new functionality and removing old, changing and modifying the accessor functions is expected to become a burden. Lenses [22] provide an alternative way to define the accessors more easily. However where the data structure is likely to change as the programmer experiments, they must still manage the code for the lenses.

It has been shown that this basic approach to monad transformers can be significantly improved upon [69], with the ideas of this paper being integrated into the Monatron library[1]). The approach taken is to define *masks* and *views* for layers of the monad stack, which may then be used to define operations that operate over components of those masks, and those only. For example, the following two could be masks for a simple implementation of TABU search, where *i* refers to

[1]Haskell library available on Hackage, Mauro Jaskelioff & Tom Schrijvers, 2010, http://hackage.haskell.org/package/Monatron.

an identity element, and a component that is not part of the current mask, and o refers to a layer that is of interest to the current mask.

$$currentSeedMask = o$$
$$tabuMask = (vlift\ i)\ `hcomp`\ (vlift\ o)$$

These two masks may be used to define operations for TABU search such as these;

$$\begin{aligned}
&inTabu\ x = \mathbf{do}\ (xs, i :: Int) \leftarrow getv\ tabuMask \\
&\qquad\qquad return\ \$\ notElem\ x\ xs \\
&updateTabu\ x = \mathbf{do}\ (xs, l :: Int) \leftarrow getv\ tabuMask \\
&\qquad\qquad\qquad putv\ tabuMask\ (take\ l\ \$\ x : xs, l) \\
&getCurrentSeed = getv\ currentSeedMask \\
&setNewSeed\ x = putv\ currentSeedMask\ x
\end{aligned}$$

Finally these operations are used to define the following function for a single iteration of TABU search.

$$\begin{aligned}
&tabuSearch\ nF = \mathbf{do}\ x \leftarrow getCurrentSeed \\
&\qquad\qquad\qquad xs \leftarrow nF\ x \gg\!= filterM\ inTabu \\
&\qquad\qquad\qquad updateTabu\ x \\
&\qquad\qquad\qquad setNewSeed\ (maximum\ xs)
\end{aligned}$$

While this example only offers a simpler way to define the lifting, in the form of named masks, the paper [69] extends the approach with the concept of named layers within the monad stack. The *monad zipper* then allows implicit definitions of the masks and the lifting. A library built in this way would first define a number of layer names, for example;

$$\mathbf{data}\ TabuLayer = TabuLayer$$

An operation defined in terms of this name makes use of the functions *use* and *expose*, defined in [69], to move operations to the correct layer.

$$\begin{aligned}
&updateTabu\ x = \mathbf{do}\ (xs, l :: Int) \leftarrow getv\ `use`\ TabuLayer \\
&\qquad\qquad\qquad putv\ (take\ l\ \$\ x : xs, l)\ `expose`\ TabuLayer
\end{aligned}$$

This creates an operation that can be run on any stack that provides enough layers with the correct names. A limitation in this example is that only one Tabu

layer may exist for any metaheuristic, however this can be overcome by providing functions that take layer names as parameters, with generic implementations.

While this work using masks and the monad zipper makes it much easier to create generic components it still fails to address the relationships between point based and population based algorithms, requiring layers to represent one, or both depending upon the form of hybridisation.

4.3 Functional Reactive Programming (FRP)

Functional Reactive Programming [39,40] was created to enable the implementation of programs whose behaviour varied with respect to time and unique events. It has been used to express computations in the fields of functional user interfaces, animation and simulation. FRP models systems as collections of continuous behaviours[2], which are implemented as functions from time to values, with the following type (*Time* is a constant type defined elsewhere in the program);

$$\textbf{type } Beh\ a = Time \to a$$

For example, the following is the behaviour of a point in two dimensions rotating about the origin. In this example "time" is used directly as the angle in radians, and the point is defined in terms of its distance from the origin.

$$rotatingPoint :: Double \to Beh\ (Double, Double)$$
$$rotatingPoint\ dist\ t = (dist * cos\ t, dist * sin\ t)$$

It is then possible to create a range of higher order functions to create new behaviours in terms of existing code. For example *mapB* operates like *map*, transforming every output of a behaviour by some basic function, and *changeTime* allows a behaviour to be sped up or slowed down by some transformation of time. A more detailed treatment of this subject can be found in Chapters 13 and 15 of [39].

$$changeTime :: (Time \to Time) \to Beh\ a \to Beh\ a$$
$$changeTime\ p\ b = b \circ p$$
$$mapB :: (a \to b) \to Beh\ a \to Beh\ b$$
$$mapB\ f\ b = f \circ b$$

[2] This Thesis will ignore *events*, which while important in FRP do not aid in the implementation of metaheuristics.

4.3.1 FRP for Metaheuristics

FRP is relevant because metaheuristics often make use of self-contained functional properties that vary over the lifetime of the program. A strong analogy can also be seen between simulations and metaheuristics, most obvious in Simulated Annealing which was developed from models of physical systems. For example, Simulated Annealing makes use of a temperature strategy, which can be implemented as a function which varies the temperature of the system with respect to the step of the process. This function is usually defined as a geometric progression, or transformation from one state to another, but can be redefined as a behaviour, shown here mathematically with two externally defined constants.

$$t(i) = tempSeed * geoDrop^{i}$$

Inductive behaviours are also possible, where each new value is based upon the preceding value, back to an initial seed. The use of an inductive behaviour is illustrated in the following reimplementation of the temperature function (where i is time or *iteration*). In general, most behaviours used for implementing metaheuristics can only be implemented in this inductive style, with seed data and chained data dependencies.

$$t(i) = \begin{cases} tempSeed & \text{if } i \leqslant 0 \\ t(i-1) * geoDrop & \text{otherwise} \end{cases}$$

FRP is designed for a continuous model of time, whereas in general metaheuristics will operate largely in an inductive form over discrete steps and so a simpler model of time (such as integers) can be used for this specific purpose. The choice of the type for time does not impact upon the rest of the implementation.

Behaviours can provide modularity through the clean separation of the various constituent parts which result from the decomposition of the monolithic metaheuristic. For example, consider this mathematical model of simulated annealing[3].

[3] In this example the condition *accepted* is a place holder for the functionality of the standard simulated annealing acceptance function, ignored here to reduce complexity. In general this would provide further scope for modularity and tuning.

$$
\begin{aligned}
r(i) &= \begin{cases} randomSeed & \text{if } i \leqslant 0 \\ next(r(i-1)) & \text{otherwise} \end{cases} \\
r'(i) &= \begin{cases} randomSeed & \text{if } i \leqslant 0 \\ next(r'(i-1)) & \text{otherwise} \end{cases} \\
p(i) &= permute(sa(i), r(i)) \\
sa(i) &= \begin{cases} seedSolution & \text{if } i \leqslant 0 \\ sa(i-1) & \text{if not } accepted(i) \\ p(i-1) & \text{otherwise} \end{cases} \\
accepted(i) &= e^{\frac{energy(sa(i-1)) - energy(p(i-1))}{t(i)}} > r'(i)
\end{aligned}
$$

To change the cooling strategy it is only necessary to modify function t; the other functions and the more general pattern of the simulated annealing algorithm remain unchanged. Similarly the perturbation behaviour p, which is a simple function defined over two behaviours, could be more complex, taking into account other details such as history of the process, and implementing this would not change the code for the *sa* function.

A naive implementation of these functions results in a recursive explosion of computation for higher values of time, an inefficiency due to the recomputation of intermediate values [19]. Memoization of these intermediate values can be used to fix this, however this results in space leaks as all previous values are preserved. Ideally we wish to allow sharing of only required previous results between behaviours, with intermediate values being cleaned up once they are no longer required. Research into these issues is ongoing [20] (see also the *Reactive* library[4]).

4.3.2 Threading Behaviour

The traditional approach to FRP provides a mechanism that allows for an efficient execution of iterative behaviour [19] under some circumstances, presented here in a simplified form. The approach taken is to change the definition of a behaviour to a function from a time to both a value and a new behaviour. This technique is possible if it is known that the new behaviour will only be sampled at times later than the previous sampling point, a promise that is satisfied in the case of iterative algorithms. The simplified type is presented below.

newtype *Beh2 a* = *Beh2* (*Time* → (*a*, *Beh2 a*))

[4]Haskell library available on Hackage, Conal Elliott, 2010, `http://hackage.haskell.org/package/reactive`.

The use of this form of continuation generation can allow the efficient sampling or unfolding of a behaviour into a list of values, using a function like the following.

$$\begin{aligned} &\mathit{unfoldB} :: \mathit{Beh2}\ a \to \mathit{Time} \to [a] \\ &\mathit{unfoldB}\ (\mathit{Beh2}\ f)\ \mathit{startTime} = \mathbf{let}\ (v,b) = f\ \mathit{startTime} \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ v : \mathit{unfoldB}\ b\ (\mathit{startTime} + 1) \end{aligned}$$

4.3.3 The problem of shared iterative behaviour

The use of continuations for behaviours resolves many issues of efficiency in inductive functions, where they are sampled sequentially. However where mutually evolving relationships exist between two or more behaviours then a new issue arises of how to share the results of continuations as each behaviour is sampled[5].

For example, where SA is adaptive the temperature behaviour is dependent upon older solutions which have been found, while the solutions are themselves dependent upon older temperatures. Implementing this in terms of evolving behaviours as seen in section 4.3.2 will still cause the space and time leaks that the evolving behaviours were introduced to avoid. This may be resolved by explicitly defining behaviours that operate over the related components, however this loses the clean modularity that made FRP attractive, for example:

$$\mathit{adaptiveSA} :: \mathit{Beh2}\ (\mathit{Solution}, \mathit{Temperature})$$

4.3.4 Impure short term memoization

An alternative, that would provide a cleaner interface, would be a form of function memoization, where the memoizing function is impure, maintaining only a limited number of steps using internal state. While impure this memoization function would still provide a pure function as its result. This would have a form like this:

$$\mathit{impureMemo} :: \mathit{Int} \to (a \to b) \to a \to b$$

This could then be used to memoize an iterative temperature function in the following way:

[5]The issue being described can be avoided where two behaviours depend exclusively upon one another's previous values, or when a single behaviour depends only on its own previous values. The issue appealatex rs when behaviours depend both upon their own histories and other behaviours histories, however this is a rather common condition in metaheuristic algorithms.

```
temp :: Time → Temperature
temp = impureMemo 3 t    -- the parameter 3 is used here as an example
                         -- of a limited number of steps to memoize over
```

This short-term memoization of the values of behaviours would require that the programmer obeys the conventions that behaviours are always sampled sequentially, and only be sampled within the specified time window. It would however allow efficient access to and sharing of values within the window.

This concept for short-term memoization would result in only a small code overhead, but is not currently supported. It is possible it would only be of use for this project at the present time, explaining the lack of support. This Thesis did not attempt to resolve this issue, because a more natural approach was conceived while examining FRP.

4.4 Dataflow & Stream Programming

FRP is a form of dataflow programming, specialised to operate over continuous time. The discrete and step-wise nature of metaheuristic algorithms makes the explicit use of time, either discrete or continuous, an unnecessary overhead. Instead, dataflow techniques may be used in pure functional languages, with automatic sharing of results, using lazy lists.

Dataflow programming dates back at least to the mid 1970's [1,49] and can be done in a variety of ways within the Haskell language, such as FRP. In a Dataflow program, data *"flows between instructions"* [46] of the program, forming a directed graph, with instructions being executed when the parameters become available. In functional languages the *instructions* can be represented as computations that construct streams of data, and these computations proceed by acting upon the elements of 0 or more other streams as data appears upon them.

This Thesis models a stream as a lazy list, and expresses the flow of data in terms of the interactions of these lists. A summary of the relationship between pure functional and data flow languages, along with a description of how to create a shallow embedding of the *Lucid* data flow language can be found in Appendix D.

4.4.1 Sharing and memoization

Lazy lists can be defined in a mutually recursive way, as with FRP behaviours, however lists defined this way do not suffer from the same runtime issues as were

seen in FRP. A simple example of a system of recursively defined streams can be seen in Figure 4.3. In this example, both *as* and *bs* are streams, each with their own initial seed value, followed by a stream which is dependent upon the previous value of the stream and the corresponding value of the other stream. As values are produced in each stream, they are implicitly shared with the computation generating the other stream.

$$\begin{aligned} \mathit{demo}\ \mathit{seedA}\ \mathit{seedB} = \textbf{let}\ & \mathit{as} = \mathit{seedA} : \mathit{zipWith}\ (+)\ \mathit{as}\ \mathit{bs} \\ & \mathit{bs} = \mathit{seedB} : \mathit{zipWith}\ (*)\ \mathit{bs}\ \mathit{as} \\ \textbf{in}\ & \mathit{as} \end{aligned}$$

Figure 4.3: A toy example of a function which implements a pair of mutually recursive streams.

Limited memoization is provided automatically in this system, with all values being preserved until they are no longer needed, and subsequently being subject to garbage collection. This inductive computation can cause memory leaks under certain circumstances, such as trying to directly compute a high value, which would result in a build up of computations on the stack until the base case was reached and all could be resolved. The solution to this is to *push* the computation, the default effect when printing a stream to the screen. When computations are pushed then automatic memoization will prevent time leaks, and garbage collection will prevent space leaks.

4.4.2 Simulated Annealing

Section 4.3 introduced a construction of simulated annealing based upon mutually recursive behaviours. Figure 4.4 shows how streams can be used to express a similar system of equations, with a very close correspondence to the FRP model. The use of streams however allows this system to be executed efficiently, through examining the stream of values bound to *s*.

4.4.3 Supporting streams using Haskell Prelude

For a shallow embedding of data flow functionality and style within a pure functional language, it is necessary to provide for the construction and manipulation

```
initTemp = ...
prop = ...
t = iterate (∗prop) initTemp

r, r′ :: [Float]
r = unsafePerformIO (newStdGen >>= return ∘ randoms)
r′ = unsafePerformIO (newStdGen >>= return ∘ randoms)

perturb :: Float → s → s
perturb = ...
p = zipWith perturb r s

seed = ...
saChoice :: Float → Float → s → s → s
saChoice = ...
s = seed : zipWith4 saChoice r′ t s p
```

Figure 4.4: An implementation of simulated annealing, using a collection of mutually recursive streams to model the intermeshed behaviours. This implementation uses *unsafePerformIO* to bypass the usual restrictions on interaction between pure and impure code. This is an acceptable break with the pure functional system, because it will be resolved once, on the initial examination of the related stream variable, and will subsequently be an unfolding list of values.

of groups of streams. The manipulation of streams is performed by applying functions over elements of the underlying streams, which can be seen as equivalent to lifting a function from acting upon elements to acting over streams. The Haskell *prelude* library provides a variety of useful tools, which can be seen as a mini-DSL for streams.

- *repeat*, takes a single value and creates a constant stream.
- *cycle*, takes a finite list and creates a stream which infinitely loops/generates the values of the list.
- *zip*, combine two streams into a single stream of tuples.
- *unfoldr*, a function that lifts a deterministic operation and a seed into a stream of values, threading internal state.
- *map*, a function that lifts a deterministic transformation of a data type to produce a function which transforms one stream into another.

- *zipWith*, describes the relationship between three streams, two inputs and one output. This may be used to create stream transformations like *map*, but hiding an internal state. For example, a stochastic perturbation operation hides the internal state of a random number generator, as seen in the simulated annealing example in Figure 4.4.

- $(\circ)$, function composition may be used to combine stream transformations, as it is usually used to compose functions. When composing stream transformations the result is often referred to as a *pipeline*[6].

4.4.4 Separation of termination conditions

The advantages of separating the termination conditions of a process from the process itself were previously considered, along with the various potential outputs that a metaheuristic can give other than a solution itself. The use of streams enables this separation by applying functions over the stream of solutions produced by a search process, predominantly using standard functions of Haskell.

- list index (!!), terminates a process and returns a single solution from a particular point. For example, this could be applied to the stream of solutions in the simulated annealing example as simply $s\,!!\,n$ to access the nth solution produced in the process.

- *take* will terminate a process, returning the first n values of the stream.

- *drop* removes the first n values of a stream (though they are still computed) and may be used in conjunction with *take* to provide a segment of a stream.

- *takeWhile* provides a way to terminate the search when some condition is met. These conditions are tests of the values in the underlying stream, making this effective for taking until a certain quality is met, but not suitable for convergence testing.

Extracting solutions until a convergence criterion is met requires a new function. This must take, not a test of the values of a stream, but a transformation of a

[6]It should be noted that using composition does not preserve the intermediate streams. If these values are of interest, for the purposes of debugging, or as part of a final output, then more complex functions are needed that can preserve and return multiple streams. Such transformations may be constructed by the programmer as an extension to the library, but have not tended to be required for implementing the standard metaheuristics. This issue is not considered further in this Thesis.

stream into a stream of test results. Results from the first stream are then produced while the convergence criterion is not met. An implementation of this adaptation of *takeWhile* is presented below as *takeS*

$$takeS :: ([s] \to [Bool]) \to [s] \to [s]$$
$$takeS\ t\ xs = map\ snd \circ takeWhile\ (\neg \circ fst) \circ zip\ (t\ xs)\ \$\ xs$$

The different forms of analysis of the results may be constructed through the parameterisation of *takeS*. For example, the following function to extract solutions from a stream until convergence is attained, where convergence is defined as two solutions of equal value separated by w steps.

$$converge\ w = takeS\ (\lambda xs \to zipWith\ (=:=)\ xs\ (drop\ w\ xs))$$

Many metaheuristics, for example random walk, do not produce a stream of solutions that guarantees to improve in value. In these cases the convergence function presented may not operate as desired. Because of this most metaheuristics will require processing by the following transformation:

$$bestSoFar :: Optimisable\ s \Rightarrow [s] \to [s]$$
$$bestSoFar\ (a : as) = scanl\ bestOf\ a\ as$$

4.4.5 Abstracting Recursive Construction

In Simulated Annealing, seen in Figure 4.4, the iterative nature of the algorithm is captured through inductive definition of the functions. This inductive process is common to all metaheuristics and is the way that the search processes are evaluated at the top level of the program. The control structure can be abstracted in the following looping functions.

$$loopS :: ([s] \to [s]) \to [s] \to [s]$$
$$loopS\ f\ seed = \textbf{let}\ as = seed +\!\!+ f\ as\ \textbf{in}\ as$$

$$loopP :: ([s] \to [s]) \to s \to [s]$$
$$loopP\ f\ seed = loopS\ f\ [seed]$$

Each of these loop functions takes a stream transformation and seed information, to produce a single stream which is now inductively defined so that the future values of the stream depend upon the previous values.

This allows the construction of inductively defined streams in terms of the composition of a number of stream transformation operators. This can be seen in this example of the Fibonacci sequence.

$$fib = loopS\ (map\ sum \circ map\ (take\ 2) \circ tails)\ [0,1]$$

4.4.6 The use of top level terms

The examples in this chapter have used Haskell to define a number of streams of data and the relationships between these streams. While this approach works, it is defining top-level processes for the computation of data, rather than functions that may be reused. In this Thesis *let-in* expressions are used in preference to defining streams of data at the top level of programs.

4.4.7 Managing stochastic components & *unsafePerformIO*

Many metaheuristics make use of stochastic components, either for perturbation, as seen in Simulated Annealing, or to introduce variation in decision making processes, such as is seen in Genetic Algorithms. However the use of random numbers, functions that give different outputs each time they are called on the same parameters, conflicts with the concept of *pure* functions. Streams present one solution to this issue, by creating stream transformations, pure functions which transform one stream into another. For example, the following function f increments every input value by a random value between 0 and 1;

$$f :: System.Random.RandomGen\ g \Rightarrow g \rightarrow [Double] \rightarrow [Double]$$
$$f\ g = zipWith\ (+)\ (randoms\ g)$$

The use of stream transformations can also be seen in Figure 4.4. In this example r is a stream of random values, and is used to define a stochastic perturbation transformation p, which will perturb each solution on its input stream, giving a stream of perturbed solutions. The stream p is created through a pure computation (subject to the source of the random values), which provides stochastic effects on the values in the input stream, but does not require the user to explicitly manage the threading of the RNG itself.

Pseudo RNGs in computer programs require IO to be created, using properties of the system such as the clock for an initial seed. In Haskell this can be done in

the following manner, where the function f is applied to two different RNGs, creating two different operations, and then composed together.

$$\begin{aligned} main = \textbf{do}\ & g1 \leftarrow newStdGen \\ & g2 \leftarrow newStdGen \\ & print\,\$\,f\ g1 \circ f\ g2\,\$\,repeat\ 0 \end{aligned}$$

The general pattern for the construction of a metaheuristic in this manner is expected to follow the following form;

$$\begin{aligned} main = \textbf{do}\ & g \leftarrow newStdGen \\ & \textbf{let}\ vs = loopS\ (\ldots \circ stochasticComponent\ g \circ \ldots \\ & \qquad\qquad)\ seedSolution \\ & print\ vs \end{aligned}$$

Typically each metaheuristic strategy will only contain a few stochastic components, so this approach is acceptable and does not incur a significant overhead. However ideally the library should be as simple as possible to use, and this approach introduces a small degree of book keeping. The overhead can be reduced through the introduction of impure computations through the function *unsafePerformIO*. There are two key issues that must be considered in using *unsafePerformIO*, the order of the evaluations and the number of evaluations of each expression.

Haskell does not give any guarantees of the order in which functions are evaluated, so care must be taken that the logic of the program is not undone by computations occurring in different orders, such as could happen to an algorithm which depends on the particular order of memory access and update. However in this case *unsafePerformIO* is being used for the construction of RNGs only, with the all other operations being pure functions. So there is generally no concern for the order in which RNGs are created, as long as they are suitably unpredictable.

The number of evaluations of each expression is more of a problem, for example;

$$\begin{aligned} & g = zipWith\ (+)\ (unsafePerformIO\,\$\,newStdGen \gg\!= return \circ randoms) \\ & k = g \circ g \end{aligned}$$

Should each usage of g be the same transformation, with the same threaded RNG, or two different transformations? In this Thesis we typically want them to be different. This can be achieved by forcing inlining using the following *Pragma*.

{-# INLINE g #-}

This instructs Haskell to replace every instance of g with the body of g, causing k to be rewritten as;

$$k = zipWith\ (+)\ (unsafePerformIO\ \$\ newStdGen \gg\!= return \circ randoms)$$
$$\circ\ zipWith\ (+)\ (unsafePerformIO\ \$\ newStdGen \gg\!= return \circ randoms)$$

There is still a danger that the Haskell compiler could search for and share common sub expressions, such as the repeated expression in k above. This issue should continue to be a consideration in future uses of this technique, however at the time of writing this approach has been tested on the current version of the Haskell compiler and it has been seen to work correctly.

4.5 Summary

This chapter has looked at several methods for implementing metaheuristics in Haskell, moving from a direct implementation of imperative concepts, towards the construction of the search strategies in terms of mutually recursive streams. The following chapters approach metaheuristics, not explicitly defining mutually recursive streams, but using the composition of stream transformations, and will be broken down as follows:

Chapter 5 details the basic library of combinators and how the combinators may be used to implement each of the major metaheuristic algorithms.

Chapter 6 extends the use of the stream combinators into the expression of low level operators commonly used to manipulate combinatorial problems.

Chapter 7 discusses some perspectives on hybridisation of metaheuristics and shows how the stream combinators can be used to enable implementation of hybrid algorithms.

Chapter 5

Metaheuristic Combinators

Chapter 4 proposed streams and data flow programming as a suitable approach for implementing metaheuristics, which provided flexibility to the programmer while fitting well with the functional foundations of Haskell. This chapter elaborates upon the stream based approach providing a library of combinators for manipulating the structure and imposing computations upon streams. It then shows how these combinators may be used to implement the five metaheuristic families that were named in the introduction and how the low level operators such as perturbation and recombination also include functionality provided by the library.

5.1 Types for stream transformations

Two type synonyms for the standard Haskell list are introduced to improve readability, while enabling code reuse of functions over streams from the standard libraries.

- **type** *Stream* $a = [a]$, where there is an unenforced promise for the list to not end; and
- **type** *List* $a = [a]$, where there is an unenforced promise for the list to be finite.

The *List* type will be used to capture components of metaheuristics such as population and neighbourhood and in this way it abstracts a commonality between

these two, which is not generally shown in metaheuristics. A more generic approach, such as an abstracted group type could also be used, but to enable reuse of existing Haskell functions it is easiest to use the standard list.

While implementing search strategies the composite types *Stream* (*List a*) and *List* (*Stream a*) are frequently encountered. It is much rarer to find operations resulting in the type *Stream* (*Stream a*). Transformations between between basic streams and streams of lists are particularly common, corresponding to the expansion and contraction of choices suggested in Chapter 2. These common patterns are captured as the following types to simplify later function definitions.

- **type** *ExpandT a b* = *Stream a* → *Stream* (*List b*)
- **type** *ContraT a b* = *Stream* (*List a*) → *Stream b*

5.2 Common operations on Streams

Frequently operations require a stream of a particular form, or change the structure of a stream themselves. To facilitate this a group of combinators are needed to manipulate the type or structure of streams. These come in six variations, representing the transformations between the different structures, see Figure 5.1.

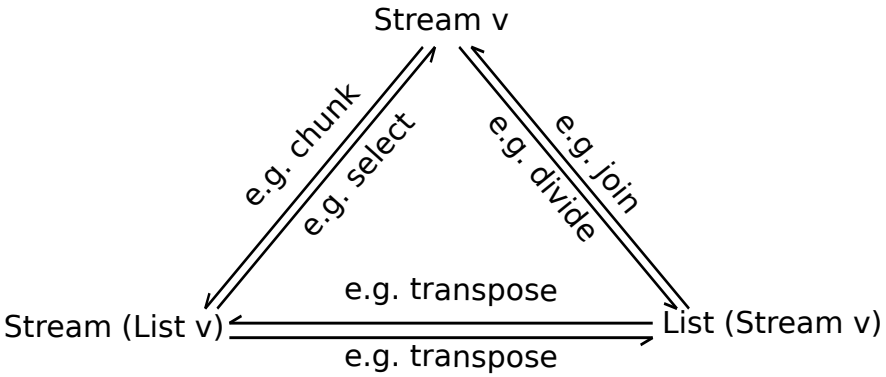


Figure 5.1: Transitions between common internal structures of streams.

Any number of functions can be created for these six transformation types, however what follows are those that have been found to be useful in the creation of a wide variety of other transformations. Implementations of these functions have been omitted for clarity, but may be found in Appendix B.

5.2.1 Stream⟺stream of lists

There are two main forms of transformation from a stream to a stream of lists *window* and *chunk*. Window is used to provide a stream of recent histories of a stream, by creating a sliding window on the values of the stream that it transforms. Chunk divides a stream of values into regularly sized blocks, however this relies upon there being enough values in the underlying stream at each point to construct a complete block before it *blocks*. Variations on both window and chunk exist which vary the sizes of the list produced at each step.

The inverse of window and chunk can be achieved using the standard *concat* function for streams of lists, however more commonly streams of lists are converted into lists through selection strategies. Some common selection strategies remain unnamed such as *map best* because their operation is clear from the verbose implementation. The more complex operation is called simply *select* and this takes a function that creates a probability distribution from the length of a list. The stream of lists is used to derive a stream of distributions from which can be sampled to select a single element of the original lists[1]. Simple selection operations can be seen in the variations of iterative improvement, while genetic algorithms make extensive use of distributive selection mechanisms.

5.2.2 Stream⟺list of streams

Two processes exist for converting a stream into a list of streams, the first is the standard functions *replicate*, which duplicates the stream multiple times to create a list of identical streams. This is of value when applying different transformations to the same source stream. The second action is *divide*, which takes a finite set of indices and a stream of instances of these indices. It then divides the input stream, based upon a pairing of each value with an index from the stream of indices, and creates one stream for each member of the index set. For example, in a genetic algorithm it is often useful to apply a mutation to a subset of elements in a population. Divide, using a boolean index type can be used to enable this, as will be seen in subsequent sections of this chapter.

join is the reversal of divide, taking the names and stream of name values. When selecting from the result of *replicate* to return to a single stream it will

[1]Due to the stochastic nature of the selection process *unsafePerformIO* is used to allow selection processes to be created *self seeded* with RNGs. This is the implementation found in the appendices and will be used subsequently in this Thesis.

usually make more sense to convert the list of streams into a stream of lists and consider the operations in the previous section.

5.2.3 List of streams $\Longleftrightarrow$ stream of lists

Conversion between streams of lists and lists of streams can be achieved using the standard *transpose* function. When applied to a list of streams *transpose* will create a single stream, where the first element is a list containing the first element of each input stream. When applied to a stream of lists *transpose* will create a list of streams, where each stream represents one *row* of the lists in the original stream. It is usually assumed that the lists in the stream will be of regular size.

5.2.4 Composite Operations

Many of the basic operations of the library introduce timing issues into the streams that they transform. For example, the use of chunk alone will typically suffer from starvation where the operation is *looped* over a finite source of data. To avoid issues like this the operations are used in pairs, with particular pairs appearing so often that they form their own operations. The code for these operations is also found in Appendix B.

The first function *stretch* is composed of the operation *map* $(replicate\ n)$ and concatenation. The map/replicate pair takes a stream as input and creates a stream of lists, where each list consists of multiple references to the underlying value of the original stream. It is rarely used alone, but is usually found as a component of *doMany*.

$$stretch :: Int \rightarrow Stream\ v \rightarrow Stream\ v$$

The *doMany* operation takes a stream transformation, but applies the operation to each value in the underlying stream multiple times and collects the results as a list. It is composed of *chunk* and *stretch* and is most useful when the stream transformation being applied uses stateful information internally threaded through the computation, such as a perturbation operation using a random number generator. This allows several perturbations to be seen from each solution in the underlying stream, before being selected between.

$$doMany :: Int \rightarrow (Stream\ a \rightarrow Stream\ b) \rightarrow Stream\ a \rightarrow Stream\ (List\ b)$$

The functions *divide* and *join* can be paired to divide a stream into multiple substreams, apply a different stream transformation to each substream and then recombine them. For the result to be stable it is important that *divide* and *join* share the set of indices and stream of index values, the sharing of which is provided by the function *nest*. This is then reformulated so that the type of the function is a list of index values and stream transformations paired together, a stream of index values and results in a stream transformation.

$$nest :: Eq\ n \Rightarrow List\ (n, Stream\ a \rightarrow Stream\ b) \rightarrow Stream\ n \rightarrow \\ Stream\ a \rightarrow Stream\ b$$

5.2.5 Restart Combinators

Metaheuristic search strategies *converge* with time, reaching a point where they cease to make significant improvements in the best known solution to the problem. At this point alternative strategies are used to *diversify* the search, allowing the strategy to continue. The simplest form of diversification is the random restart, where solutions are generated by a search strategy from one solution and when that strategy begins to struggle it continues from a newly generated solution.

The restart combinators were originally created to support this task, but were subsequently found to be of use more widely in hybridisation, perturbation and neighbourhood implementation and adaptive simulated annealing. Two versions are implemented *restart* and *restartExtract*, both sharing the same type:

$$\begin{aligned} restart, restartExtract :: & (Stream\ a \rightarrow Stream\ a) && \text{-- internal strategy} \\ & \rightarrow (Stream\ a \rightarrow Stream\ Bool) && \text{-- restart on} \\ & \rightarrow Stream\ a && \text{-- seed solutions} \\ & \rightarrow Stream\ a \end{aligned}$$

restart provides as output every solution encountered by the internal strategy until the restart condition is found. *restartExtract* acts as a stream transformer, giving only the last solution encountered by the internal strategy from each seed.

5.2.6 Combinators for eagerness

The use of streams to express metaheuristics can run the risk of memory leaks caused through the build up of unevaluated Thunks. This problem can be controlled by requiring that the solutions in the final stream are evaluated in the order

that they appear in the stream. This can be implemented using a function called *push* , which makes a stream *head strict*, that is the first value must be evaluated before any other values are[2].

$$push :: Stream\ a \rightarrow Stream\ a$$
$$push\ (x : xs) = x\ `seq`\ x : push\ xs$$

This function is applied to the output of a metaheuristic before applying further operations such as selecting the best of the solutions seen.

5.3 Hill Climbers

Iterative Improvers are more commonly known as hill climbers and gradient descent. They operate in a greedy manner, only moving to a new candidate solution if it improves upon the previous candidate. Usually they are based upon neighbourhood functions, where a number of candidates are generated from a previous candidate and only one is selected.

This means that the process can be divided into three parts; (i) the generation of the raw neighbourhood, (ii) the processing of this into a neighbourhood of improving solutions, where improving means *better than the parent solution*, and (iii) selecting from the improving neighbourhood. There are various ways in which the next candidate can be selected from the neighbourhood of improving solutions, with the most common being; first found, maximal, minimal and stochastic.

The creation of the stream of neighbourhoods from a stream of solutions is a problem specific operator, of the type *ExpandT s s*. The stream of improving neighbourhoods is a modification of the neighbourhood operator such that neighbourhoods produced are always improving. The function to perform this has been called *improvement*, a higher order function that transforms expansion operators.

$$improvement :: Optimisable\ s \Rightarrow ExpandT\ s\ s \rightarrow ExpandT\ s\ s$$
$$improvement\ nf\ sols$$
$$= safe\ (map\ (:[\,])\ sols)$$
$$\$\ zipWith\ (\lambda a\ b \rightarrow filter\ (>:a)\ b)\ sols\ (nf\ sols)$$
$$safe :: Stream\ (List\ v) \rightarrow Stream\ (List\ v) \rightarrow Stream\ (List\ v)$$
$$safe = zipWith\ (\lambda a\ b \rightarrow \textbf{if}\ null\ b\ \textbf{then}\ a\ \textbf{else}\ b)$$

[2]The *push* function is equivalent to the *evalList rseq* strategy found in the Haskell parallel libraries at http://hackage.haskell.org/package/parallel-3.2.0.3.

The improvement function works by internally creating the stream of neighbourhoods and then combining each neighbourhood with its seed in a filtering operation. When the hill climber reaches a local minimum, the filtered neighbourhood would be empty and this would cause problems for subsequent selection operators.

Iterative Improvers are often used as components of hybrid search strategies. To simplify composition of an Iterative Improver with other streams it is important that it will provide some output, regardless of the solution that is processed. To avoid the problem of local minima not being able to improve the helper function *safe* is introduced so that either the improving neighbourhood or the singleton seed is returned from an *improving* neighbourhood.

Iterative improvement becomes a combinator of the library, combining the concepts of the improvement transformation of a neighbourhood operation and the application of a selection operation to the result.

$$\begin{aligned} &\mathit{iterativeImprover} :: \mathit{Optimisable}\ s \Rightarrow \mathit{ExpandT}\ s\ s \rightarrow \mathit{ContraT}\ s\ s \rightarrow \\ &\qquad\qquad \mathit{Stream}\ s \rightarrow \mathit{Stream}\ s \\ &\mathit{iterativeImprover}\ \mathit{nf}\ \mathit{cf} = \mathit{cf} \circ \mathit{improvement}\ \mathit{nf} \end{aligned}$$

First found Iterative Improvement is implemented using this combinator by parametrising it with a contraction pattern which takes the first element of any neighbourhood it encounters. This is created through *map head*. Similarly maximal and minimal improvement are described in terms of *best* and *worst*;

$$\begin{aligned} &\mathit{firstFoundii}, \mathit{maximalii}, \mathit{minimalii} \\ &\quad :: \mathit{Optimisable}\ s \Rightarrow \mathit{ExpandT}\ s\ s \rightarrow \mathit{Stream}\ s \rightarrow \mathit{Stream}\ s \\ &\mathit{firstFoundii}\ \mathit{nf} = \mathit{iterativeImprover}\ \mathit{nf}\ (\mathit{map}\ \mathit{head}) \\ &\mathit{maximalii}\ \mathit{nf} = \mathit{iterativeImprover}\ \mathit{nf}\ (\mathit{map}\ \mathit{best}) \\ &\mathit{minimalii}\ \mathit{nf} = \mathit{iterativeImprover}\ \mathit{nf}\ (\mathit{map}\ \mathit{worst}) \end{aligned}$$

Stochastic Iterative Improvers are usually implemented as the selection of a random element, implicitly using a uniform distribution, from an improving neighbourhood. The library provides finer grained control of the degree of randomness in a search strategy through the use of alternative distributions and imposing additional structure on the neighbourhoods. In the following example three alternatives are shown, including a biased selection method sorting the neighbourhoods so that better solutions occur earlier, and then using a Poisson distribution to tend to select from the front of the list.

$\lambda nf \rightarrow iterativeImprover\ nf\ (select\ uniform)$
$\lambda nf \rightarrow iterativeImprover\ nf\ (select\ (poisson\ 1))$
$\lambda nf \rightarrow iterativeImprover\ nf\ (select\ (poisson\ 1) \circ map\ sortO)$

5.4 TABU

At its core TABU search is an exploration of neighbourhoods similar to an iterative improver, but the filtering process of the neighbourhoods involves the recent history of solutions to avoid returning to those previously seen. This concept can be implemented in a variety of ways, the most direct being the combination of two streams, one of neighbourhoods and one of TABU lists, filtering one against the other:

$zipWith\ (\lambda w\ n \rightarrow filter\ (elem\ w)\ n)\ tabuLists\ neighbourhoods$

This approach separates the source of TABU lists from the source of neighbourhoods, however commonly both of these are dependent upon the previous solutions. Due to this shared characteristic each of these computations will be passed not directly as streams, but as computations that transform streams. The following code also makes use of the *safe* function previously seen in Iterative Improvers, to ensure that the result of filtering the neighbourhoods is never completely empty.

```
tabuFilter :: Eq s => (Stream s -> Stream (List s)) ->    -- window
                      (ExpandT s s) ->                    -- neighbourhood
                      (ExpandT s s)
tabuFilter wF nF xs
  = safe (map (:[]) xs)
  $ zipWith (\ws -> filter (flip notElem ws)) (wF xs) (nF xs)
```

This allows the implementation of TABU search as the following function, which composes a choice function with a filtering transformation of a neighbourhood, following the pattern seen in iterative improvers.

$tabu\ cF\ wF\ nF = cF \circ tabuFilter\ wF\ nF$

However this approach does not follow the rules of TABU search laid out in Chapter 2. In the standard model of TABU search, the TABU filtered neighbourhood is used as an escape strategy in conjunction with an iterative improver. While neighbourhoods contain solutions which improve the result of the search strategy this is always taken, only being replaced with a choice from the TABU search when no further improvement is possible. In order to implement this *safe* is reused again, but taking the result of a TABU filter as the alternate result.

```
tabu :: (Eq s, Optimisable s) ⇒
        (ContraT s s) →                    -- choice
        (Stream s → Stream (List s)) →     -- window
        (ExpandT s s) →                    -- neighbourhood
        Stream s →
        Stream s
tabu cF wF nF xs = cF ∘ safe (tabuFilter wF nF xs) ∘ improvement nF $ xs
```

A TABU list may be computed in a variety of ways however the standard method, and that provided by the current functional library is to provide a snapshot of the recent history, through the *window* function. In this example a window size of 5 is used, for illustrative purposes.

```
> loopP (tabu (map head) (window 5) tsp_neighbourhood) seed_solution
```

As with iterative improvers any selection function may be used in place of *map head*, which is used here both for illustration and because operating in this mode is part of the standard implementation of TABU search.

5.4.1 Variable TABU List Size

The flexibility of these combinators to construct a range of variants can be demonstrated through implementing part of Taillard's Robust Taboo search [78]. In this work Taillard used a TABU list in which the size varied randomly between fixed bounds. This was found to produce better and more stable results than the basic TABU algorithm.

A function is provided to stochastically modify the elements of a stream of TABU lists. To simplify later usage this is automatically seeded with a random number generator.

```
varyWindow :: (Int, Int) → Stream (List s) → Stream (List s)
varyWindow range = unsafePerformIO k
```

$$\textbf{where } k = \textbf{do } g \leftarrow \mathit{newStdGen}$$
$$\mathit{return}\,\$\,\mathit{zipWith\ take}\ (\mathit{randomRs\ range\ g})$$

Construction of Taillard's TABU is now straightforward;

$$\begin{aligned}&\mathit{taillardTABU\ nf\ winSize\ range}\\ &= \mathit{loopP}\ (\mathit{tabu}\ (\mathit{map\ head})\\ &\qquad (\mathit{varyWindow\ range} \circ \mathit{window\ winSize})\\ &\qquad (\mathit{map\ nf}))\end{aligned}$$

5.4.2 Performance Considerations

The implementation of TABU shown so far provides a clean expression for where the functionality of the different elements interact, however it also will frequently duplicate the neighbourhood construction process. In order to avoid the cost of unnecessary function evaluation the result of the neighbourhood must be shared, which can be performed manually in the following way.

$$\begin{aligned}&\mathit{tabu\ cF\ wF\ nF\ xs}\\ &\quad = \textbf{let } \mathit{nF'} = \mathit{const}\ (\mathit{nF\ xs})\\ &\qquad \textbf{in } \mathit{cF} \circ \mathit{safe}\ (\mathit{tabuFilter\ wF\ nF'\ xs}) \circ \mathit{improvement\ nF'}\,\$\,\mathit{xs}\end{aligned}$$

This modification can be performed automatically, however the task of identifying all the cases where the common transformation could be extracted has not yet been performed. In general this does begin to highlight where functional programming can aid in the expression of metaheuristics at a high level while enabling efficient performance through rewriting of expressions during compilation.

5.5 Simulated Annealing

Simulated Annealing can be thought of as an adaptive filtering process similar to a basic TABU filter, where the elements that will be filtered change at each step of the iteration. The filtering process in Simulated Annealing is dependent upon the previous or seed solution, a temperature and a supply of random numbers.

Unlike iterative improvers and TABU search Simulated Annealing is not usually implemented as a neighbourhood based algorithm, but as a perturbation based algorithm where a seed solution is perturbed and a choice is made between the old

and the new. This is the approach, as a choice between values, rather than focusing upon the filtering characteristic, which has been taken by the library. The standard choice function[3] used by simulated annealing can be implemented in the following way, operating over values rather than streams.

```
saChoose :: (Floating v, Ord v) ⇒
  (s → v) →      -- solution to value
  v → v →        -- temperature and random numbers
  s → s → s      -- solutions
saChoose quality r t s s'
  | d ⩽ 0 ∨ e > r = s'
  | otherwise = s
  where
    e = exp (−(d / t))
    d = (quality s') − (quality s)
```

The *saChoice* function can be lifted to operate over streams using the standard *zipWith4* function. It is due to this direct implementation using a standard Haskell combinator that further work on abstracting the filtering characteristic was not pursued.

```
sa :: (Ord v, Floating v) ⇒
  (s → v)                    -- quality evaluation function
  → (Stream s → Stream s)    -- perturbation transformation
  → Stream v                 -- stream of stochastic values
  → Stream v                 -- temperature strategy stream
  → Stream s → Stream s
sa quality perturbF rs coolS sols
  = zipWith4 (saChoose quality) rs coolS sols (perturbF sols)
```

5.5.1 Standard Cooling Strategies

The streams of temperatures, usually called cooling strategies, are the most common way to adapt simulated annealing for particular problems. There are three well known strategies which are implemented below:

[3] It should be noted that unlike the other metaheuristics Simulated Annealing does not make use of the *Optimisable* type class. This is because the *saChoice* function selects between solutions based upon a comparison of the numerical representation of their qualities, rather than using a more generic ordering of solutions.

- Linear cooling

  $t_n = t_{n-1} + c$

$$\begin{aligned}&\mathit{linCooling} :: \mathit{Floating}\ b \Rightarrow b \to b \to [b]\\&\mathit{linCooling}\ c\ t0 = \mathit{iterate}\ (+c)\ t0\end{aligned}$$

- Geometric

  $t_n = t_{n-1} * c$

$$\begin{aligned}&\mathit{geoCooling} :: \mathit{Floating}\ b \Rightarrow b \to b \to [b]\\&\mathit{geoCooling}\ c\ t0 = \mathit{iterate}\ (*c)\ t0\end{aligned}$$

- Logarithmic

  $t_n = \frac{c}{log(n+d)}$

$$\begin{aligned}&\mathit{logCooling} :: (\mathit{Enum}\ b, \mathit{Floating}\ b) \Rightarrow b \to b \to [b]\\&\mathit{logCooling}\ c\ d = \mathit{map}\ (\lambda t \to c\ /\ (\mathit{log}\ (t+d)))\ [1..]\end{aligned}$$

As with the previous strategies, the stream transformer must be looped and provided with a seed solution, and like TABU search, simulated annealing does not always improve solution quality with time, so *bestSoFar* will be required on the final output.

$$\begin{aligned}&\mathit{exampleSA} :: (\mathit{Optimisable}\ s, \mathit{Floating}\ v, \mathit{Ord}\ v)\\&\quad \Rightarrow \quad (s \to v) \to \mathit{StreamT}\ s \to v\\&\quad \to \quad v \to [v] \to s \to \mathit{Stream}\ s\\&\mathit{exampleSA}\ \mathit{quality}\ \mathit{perturbF}\ \mathit{startT}\ \mathit{propT}\ \mathit{rs}\\&\quad = \mathit{bestSoFar} \circ \mathit{loopP}\ (\mathit{sa}\ \mathit{quality}\\&\qquad\qquad\qquad\qquad \mathit{perturbF}\\&\qquad\qquad\qquad\qquad \mathit{rs}\\&\qquad\qquad\qquad\qquad (\mathit{geoCooling}\ \mathit{propT}\ \mathit{startT}))\end{aligned}$$

5.5.2 Adaptive Simulated Annealing

At high temperatures simulated annealing will accept more candidates and so explore the solutions space more widely, while at lower temperatures the algorithm

will tend to move through solutions which improve the quality of the result. At very low temperatures the algorithm acts almost exactly like an Iterative Improver and so will become stuck in a local minimum and cease to change.

Adaptive simulated annealing attempts to select or manage the temperature of the system so as to improve the progress of the search system. This can take a variety of forms including (i) restarting the temperature strategy, (ii) reheating the system gradually and (iii) reducing the temperature in an irregular pattern. Typically each of these effects will be triggered by the detection of convergence in the stream of solutions being discovered. The following code sketches illustrate the use of restart for implementing types (i) and (iii), with the presumption of a function called *converge* which gives a stream of boolean values indicating convergence of the recent history at each step of the process.

$$\mathit{restart\ coolingStrategy}\ (\mathit{const}\ (\mathit{converge\ sols}))\ (\mathit{repeat\ initialTemp})$$
$$\mathit{restart\ id}\ (\mathit{const}\ (\mathit{converge\ sols}))\ (\mathit{coolingStrategy\ initialTemp})$$

The following more complete example shows the use of *window* to implement a simple convergence function, and the use of this to create an adaptive simulated annealing system with cooling schedule restarting.

$$\begin{aligned}
&\mathit{restartingSA}\ ::\ (\mathit{Eq}\ s, \mathit{Floating}\ v, \mathit{Ord}\ v, \mathit{Optimisable}\ s)\\
&\qquad\qquad \Rightarrow \mathit{Int} \to v \to v\\
&\qquad\qquad \to (s \to v) \to (\mathit{Stream}\ s \to \mathit{Stream}\ s)\\
&\qquad\qquad \to \mathit{Stream}\ v \to s \to \mathit{Stream}\ s\\
&\mathit{restartingSA\ wSize\ startT\ propT\ getVal\ perturbF\ rs1\ seed}\\
&\quad = \mathbf{let}\ \mathit{cs} = \mathit{map}\ (\lambda w \to \mathbf{if}\ \mathit{null}\ w\ \mathbf{then}\ \mathit{False}\ \mathbf{else}\ \mathit{head}\ w \equiv \mathit{last}\ w)\ \$\\
&\qquad\qquad \mathit{window\ wSize\ sols}\\
&\qquad\quad \mathit{ts} = \mathit{restart}\ (\mathit{map}\ (*\mathit{propT}))\ (\mathit{const\ cs})\ (\mathit{repeat\ startT})\\
&\qquad\quad \mathit{sols} = \mathit{loopP}\ (\mathit{sa\ getVal\ perturbF\ rs1\ ts})\ \mathit{seed}\\
&\quad\ \mathbf{in}\ \mathit{bestSoFar\ sols}
\end{aligned}$$

Alternating between heating and cooling of the system presents a different difficulty as it requires the switching between different temperature strategies (heating and cooling), in accordance to the stream of triggers. This is almost identical to the concept of event driven changes to behaviours found in functional reactive programming (FRP) [40] and so a similar function is constructed. The implementation can be found in Appendix B.

5.6 Genetic Algorithms

At first glance genetic algorithms (GA) call for a separate set of combinators due to their use of *populations* of candidate solutions rather than operating over individual solutions. However a stream of solutions can be converted into a stream of lists, which is isomorphic to a stream of populations, through the use of the *chunk* function, and a stream of populations may be converted into a stream of solutions through concatenation.

It is possible to describe a genetic algorithm as a stream transformation over populations, and has the following type;

$$popTrans :: Stream\ (List\ s) \rightarrow Stream\ (List\ s)$$

This gives rise to the following sketch of a skeleton for genetic algorithms;

$$concat \circ popTrans \circ chunk\ popSize$$

The core process of the population transformation is selection of parents and recombination of these parents to give rise to a new population. The future population needs to be the same size as the previous solution, so this task must occur *popSize* times and each solution is produced through the recombination of *rSize* solutions. Each of these tasks is of the pattern *doMany* and can be implemented through composite application of this function. In the following example *recombine* is presumed to be an externally defined recombination function, and a *Poisson* distribution has been used to give preference to the better solutions.

$$\begin{array}{l} doMany\ popSize \\ \quad (recombine \circ doMany\ rSize \\ \quad\quad (select\ (poisson\ 0) \circ map\ sortO)) \end{array}$$

Genetic algorithms also make use of *mutation*. Rather than implementing this within the population structure it can be applied separately using the *nest* function, to apply a perturbation to a substream of solutions. For example, in the following code fragment, a TSP perturbation function is applied to every other solution in the system.

$$nest\ [(False, tsp\_perturb), (True, id)]\ (cycle\ [False, True])$$

These components can be composed together to give the standard skeleton of a genetic algorithm. How this implementation operates on streams can also be seen graphically in Figure 5.2.

$$
\begin{array}{lll}
ga :: Int \rightarrow & & \text{-- population size} \\
\quad Float \rightarrow & & \text{-- mutation likelihood} \\
\quad (ContraT\ s\ s) \rightarrow & & \text{-- recombine, contraction} \\
 & & \text{-- of parents into child} \\
\quad (Stream\ (List\ s) \rightarrow Stream\ (List\ s)) \rightarrow & & \text{-- selection} \\
\quad (Stream\ s \rightarrow Stream\ s) \rightarrow & & \text{-- mutation} \\
\quad Stream\ s \rightarrow Stream\ s & &
\end{array}
$$

$$
\begin{array}{l}
ga\ popSize\ perturbProb\ recombine\ selection\ perturb \\
\quad = \quad nest\ [(True, perturb), (False, id)]\ mutGo \circ \\
\qquad concat \circ \\
\qquad doMany\ popSize\ (recombine \circ selection) \circ \\
\qquad chunk\ popSize \\
\quad \textbf{where} \\
\qquad mutGo = unsafePerformIO\ k \\
\qquad\quad \textbf{where} \\
\qquad\qquad k = newStdGen \gg\!= return \circ map\ (<perturbProb) \circ randoms
\end{array}
$$

In this implementation, the size of the selection, making use of *doMany* is left to the programmer, rather than passing an additional superfluous parameter to the function.

5.6.1 Performance Considerations

As with TABU search this approach to GAs is a high level description of the functionality, but gives rise to runtime performance problems. In the case of GAs the problem arises from the typical desire to select from the population so that better solutions are more likely to be picked.

In the implementations seen here the sorting operation is applied every time that the selection operation is used. This was chosen to provide clarity regarding what it does and why it is there, but also flexibility in varying how the list is sorted. However this will cause the population to be sorted many times, when it only needs to be sorted once.

Unlike TABU this can be handled by a Haskell rewrite rule.

```
"stretch/map"
  forall f n. map f . stretch n =
           stretch n . map f
```

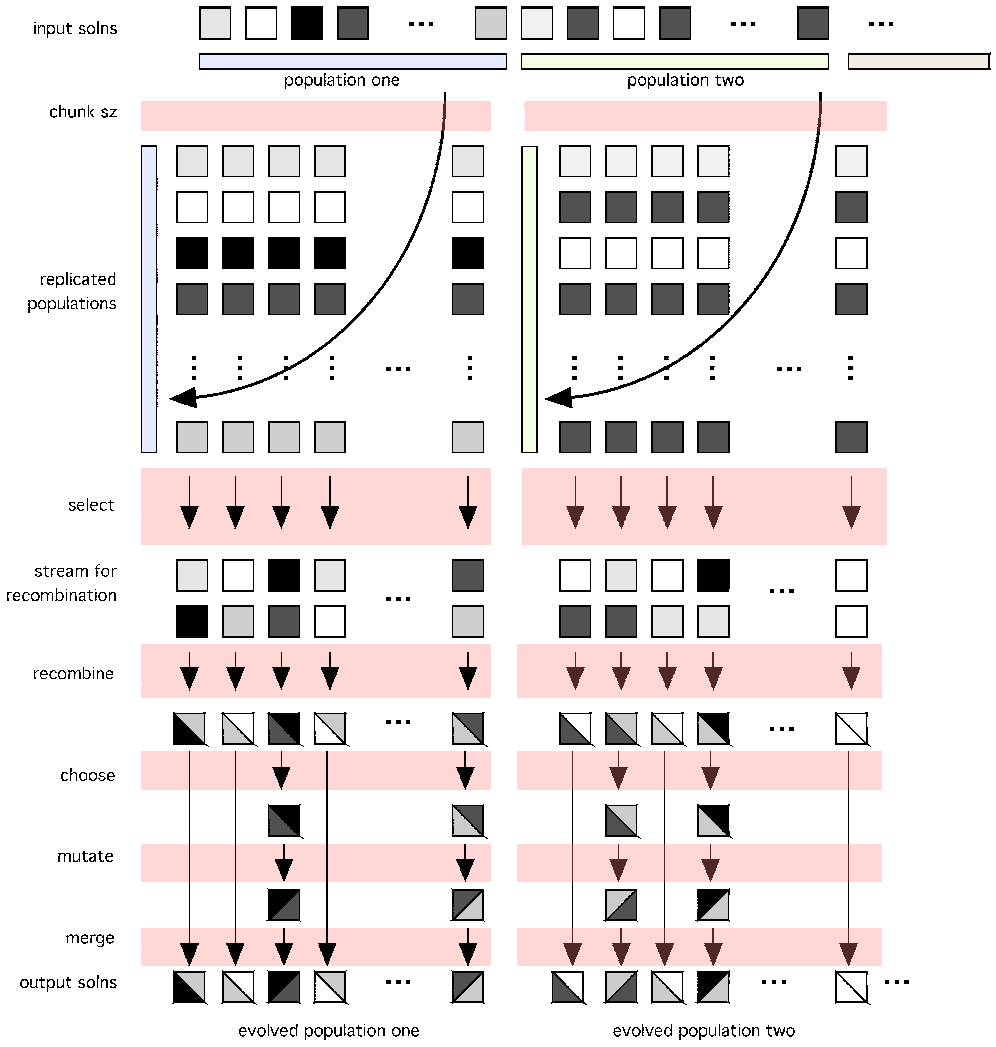


Figure 5.2: An illustration of how a genetic algorithm works, when implemented using stream transformations.

This rule takes advantage of the defintion of *doMany*;

$$doMany\ n\ f = chunk\ n \circ f \circ stretch\ n$$

Where f is actually $map\ f$, then the rule given can be used to rearrange the code automatically to improve performance. This rule can be proved in the following way. First the definition of stretch itself;

stretch n = concat . map (replicate n)

The proof, beginning from the left hand side of the rule.

map f . stretch n	The LHS
map f . concat . map (replicate n)	Definition of stretch
concat . map (map f) . map (replicate n)	standard property of map [39]
concat . map (map f . replicate n)	map composition
concat . map (replicate n . f)	can be proved by induction over finite lists
concat . map (replicate n) . map f	reverse map composition
stretch n . map f	reverse stretch definition

The result of those transformations is the right hand side of the rewrite rule, as desired.

The multiple selections that are performed in GAs also cause problems, computing the probability distribution, attaching it to solutions in the population and looping over the list many times more than is needed. This can be solved by detecting the use of selection in conjunction with stretch and computing an efficient data structure[4] once, and selecting from that many times. This can be implemented using the following compiler rule.

```
"multiselect" forall n g.
  select g . stretch n =
    zipWith (\r->snd . fromJust .
                 Data.Map.lookupGT r)
            (unsafePerformIO $
               newStdGen >>= return.randoms) .
    stretch n . map Data.Map.fromAscList
              . map (\x->zip (g.length $ x) x )
```

The argument for the practical validity of this transformation is to break down the selection function into its two constituent parts. The first takes an input list,

[4]The data structures and related accessor methods are all drawn from the standard *Data.Map* library.

and produces a data structure of pairs, $[(Probability, Value)]$, and this transformation is applied to every list in the input stream of lists, using map. This data structure is then processed by a stream transformation, which threads an internal random number generator, which will select one element from each list in the stream, based upon the probability distribution used. The selection mechanism is seeded using *unsafePerformIO*, and this is the same for both the standard selection mechanism, and the product of this rule.

Since *map* is used to apply the construction of the intermediate data structures within *select*, and the rule makes use of stretch, the previous proof can be used to prove that it is acceptable to move this across, giving (this is a code sketch only);

$$selectionProcess \circ stretch\ n \circ map\ dataStructureConstruction$$

The rule then improves performance by replacing the data structure, changing it from a flat linked list to a tree. The selection mechanism is similarly updated to match this change. Where stretch is used this will tend to give better access, by building the tree once, and querying it quickly, rather than searching the linked list (where it may not is if n is low enough that the size of the computation for creating the data structure is actually larger than just querying the linked list a few times).

There are situations where it may give different answers and these depend upon the cumulative distribution function given by the programmer.

- Where the distributions are not monotonically increasing this approach will not work, however it would be incorrect in both the original and the new version of the code after the rule's application.

- Where the distribution has multiple identical probabilities. The left hand side of the rule will select the first, where as the right hand side will select the last. In practice this situation would not be desirable, unless it was the intention to make a large number of solutions inaccessible for this selection method, in which case a more explicit function would be preferred.

Apart from these considerations the selection process and data structure do match between the left and right hand sides of this transformation.

5.7 Demonstrations of flexibility

This section will demonstrate the flexibility of the library in three ways. The first will be to create new patterns for the combination of transformation building blocks, without having to modify the underlying library. The second will reuse existing components, built for the Simulated Annealing metaheuristic in the context of a genetic algorithm, to modify how breeding pairs are chosen. Finally it will be demonstrated how different search strategies can be combined using the library to create more complex strategies.

5.7.1 Creating a new operators

Consider a situation where the programmer wishes to apply several operators in sequence many times over, such as a number of different perturbation operations[5]. The operations of the library do not currently support this particular operation. To clarify how a sequence of operations can interact, the following is a toy example on numbers, starting from the number 1 and using the operations; $+1$, $*2$, *cos* and $/3$.

Seed	$+1$	$*2$	*cos*	$/3$	
1	2.0	4.0	-0.654	-0.218	
	0.782	1.564	$6.559e^{-3}$	$2.186e^{-3}$	
	1.002	2.004	-0.420	-0.140	
	0.860	1.720	-0.149	$-4.952e^{-2}$	...

One approach to combining these operations is simply composition of the functions, which can be done as follows, making use of the library operation *loopP*.

$$loopP\ (map\ ((/3) \circ cost \circ (*2) \circ (+1)))\ 1$$

However the output of this looped composition would be the following sequence. $1.0, -0.218, 2.186e^{-3}, -0.140...$
Composing the operators only outputs the result of the whole composition at each step, and many *potentially* useful values are hidden from the user. If the objective were to minimise the value seen in the sequence (as is often the case in an optimisation algorithm) then the value 0.654 can be seen to have been hidden by the composition.

[5]This can be the pattern of operation used by the product of simple hyperheuristics.

The desire is to have an operator that will apply the transformations in sequence, but will make visible all the values that are seen. There are at least two ways to implement this, presented here as two versions of *composeWithOutput*.

$$
\begin{aligned}
&\mathit{composeWithOutput} :: \mathit{List}\ (\mathit{Stream}\ a \to \mathit{Stream}\ a) \\
&\qquad\qquad \to \mathit{Stream}\ a \to \mathit{Stream}\ a \\
&\mathit{composeWithOutput}\ \mathit{fs} = \mathit{concat} \circ \mathit{transpose} \circ \mathit{zipWith}\ (\$)\ \mathit{fs} \\
&\qquad\qquad \circ \mathit{transpose} \circ \mathit{chunk}\ (\mathit{length}\ \mathit{fs}) \\
&\mathit{composeWithOutput}'\ \mathit{fs} = \mathit{nest}\ (\mathit{zip}\ \mathit{nms}\ \mathit{fs})\ (\mathit{cycle}\ \mathit{nms}) \\
&\quad \textbf{where}\ \mathit{nms} = [1\,..\,\mathit{length}\ \mathit{fs}]
\end{aligned}
$$

These may be used to implement the desired pattern of computation in the following way.

$$\mathit{loopP}\ (\mathit{composeWithOutput}\ \$\ \mathit{map}\ \mathit{map}\ [(+1),(*2),\mathit{cos},(/3)])\ 1$$

The result of using *composeWithOutput* is a stream transformation itself, and can hence be passed as a parameter to other components, for example using it as a perturbation operation, or composed into other more complex operations. For example;

$$
\begin{aligned}
&\mathit{loopP}\ (\mathit{composeWithOutput}\ \mathit{fs} \\
&\qquad \circ \mathit{nest}\ (\mathit{zip}\ [\mathit{False},\mathit{True}]\ [\mathit{id},\mathit{map}\ (+10)] \\
&\qquad\qquad (\mathit{cycle}\ \mathit{nms}) \\
&\qquad)1 \\
&\quad \textbf{where}\ \mathit{nms} = [\mathit{False},\mathit{True},\mathit{False},\mathit{False}] \\
&\qquad\quad \mathit{fs} = \mathit{map}\ \mathit{map}\ [(+1),(*2),\mathit{cos},(/3)]
\end{aligned}
$$

A similar operation is to combine two strategies, but in an unequal way, for example the first strategy for 10 steps, the second for 20, while outputting every intermediate solution seen. Like the second implementation of *composeWithOutput* this is a specialisation of the existing *nest* function.

$$
\begin{aligned}
&\mathit{combineWithTimeLengths} :: \mathit{Int} \to (\mathit{Stream}\ a \to \mathit{Stream}\ a) \\
&\qquad\qquad\qquad \to \mathit{Int} \to (\mathit{Stream}\ a \to \mathit{Stream}\ a) \\
&\qquad\qquad\qquad \to \mathit{Stream}\ a \to \mathit{Stream}\ a \\
&\mathit{combineWithTimeLengths}\ \mathit{l1}\ \mathit{t1}\ \mathit{l2}\ \mathit{t2} \\
&\quad = \mathit{nest}\ [(\mathit{True},\mathit{t1}),(\mathit{False},\mathit{t2})]\ \mathit{nms} \\
&\quad \textbf{where}\ \mathit{nms} = \mathit{cycle}\ (\mathit{replicate}\ \mathit{l1}\ \mathit{True} +\!\!+ \mathit{replicate}\ \mathit{l2}\ \mathit{False})
\end{aligned}
$$

Each of these new operators can interact with the existing library components, and be built with a minimum of understanding of how the existing functions act internally. This demonstrates the flexibility of the approach in enabling light-weight user augmentation of the existing system.

5.7.2 Reuse of existing components for new purpose

The genetic algorithm pattern previously presented selected parents making use of a probability distribution to guide the stochastic selection process. This basic concept can be varied, and indeed is experimented with by researchers in the field. For example, in the real world it is sometimes easy for almost any parent to reproduce, where as at other times only the best can breed, due to poor environmental conditions such as lack of food. Ideally implementing this concept should involve changing the selection mechanic alone, and should not involve modification of the basic pattern of the genetic algorithm.

One way to approach this is to vary the population that can be selected from by the existing selection operation. Given that the populations are already sorted this becomes the task of taking some group, from the best to some limit, which can be implemented using the following.

$$zipWith\ take\ ts :: Stream\ (List\ a) \rightarrow Stream\ (List\ a)$$

The symbol *ts* can be implemented in the above through an elaboration of one of the cooling strategies seen in Simulated Annealing. In the following example it is a form of geometric cooling that is chosen. The constants are arbitrarily chosen for this example, but the basic concept is to construct a segment of a cooling schedule, and then use that to construct a cycling pattern to limit the population selection pool.

$$cyclicalTemp\ popSize = map\ floor\ (cycle\ \$\ g +\!\!+ (reverse\ g))$$
$$\textbf{where}\ g = take\ 40\ \$\ geoCooling\ 0.97\ popSize$$

The previous mechanism for selection in a genetic algorithm might have been;

$$doMany\ 2\ (select\ uniform)$$

Which would select two parents using a uniform distribution. Typically a non-uniform distribution was expected, however, with the new mechanism for limiting the pool of choices, uniform selection becomes more useful, because any of the

pool should be equally eligible at the point of decision, or else the limiting of the selection pool is happening twice. This gives rise to the following example selection mechanic, making use of the pool limiting function, and uniform selection.

$$doMany\ 2\ (select\ uniform) \circ$$
$$zipWith\ take\ (stretch\ popSize\ \$\ cyclicalTemp\ popSize)$$

This composition may now be passed as the selection mechanic to a genetic algorithm, and provides a quite different breeding pattern to those previously seen. This is achieved without changing the underlying code, and making use of components originally not intended for use in a genetic algorithm.

5.7.3 Combining search components

A common approach to trying to improve the performance of a metaheuristic is to combine it with other search strategies, as seen in Hybridisation, returned to in Chapter 7. For example, Simulated Annealing has been described as operating using a stochastic perturbation of solutions, however this is not the only option. If p is a stochastic perturbation operation, with the type $Stream\ sol \rightarrow Stream\ sol$, then it can be varied and made more complex. For instance, the following could be used as a perturbation for simulated annealing, where a number of possibilities are created at once, and then a Poisson selection method will tend to present the best to the outer Annealing choice.

$$select\ (poisson\ 0) \circ map\ sortO \circ doMany\ nSize\ p$$

Another simple alternative is to always select the best from a generated group, making the Annealing choices operate over a series of fairly good choices (though they may still be worse than the seed at any given step), rather than just over stochastically generated options.

$$map\ best \circ doMany\ nSize\ p$$

An alternative is to make use of iterative improver as the perturbation mechanic, which has been previously investigated by other researchers. This makes the Annealing process a choice mechanism over a series of local minima rather than over a series of random solutions. It can be implemented as follows, making use of a new *convergeTest* function and the *restartExtract* function. It still uses a stochastic perturbation operation p to move away from the current solution, but then applies the iterative improver over the neighbourhood function nF.

$convergeTest = (False:) \circ map\ (\lambda[a,b] \rightarrow a \equiv b) \circ tail \circ window\ 2$
$newPerturb\ p\ nF = restartExtract\ (iterativeImprover\ nF)\ convergeTest \circ p$

Another approach to combining strategies is to combine them in succession. This may be achieved with function composition, however it can be more useful to use the recently defined *combineWithTimeLengths* function. A common approach in metaheuristics is to combine an Iterative Improver with a random walk, to allow the algorithm to escape from the local minimum that the Iterative Improver became trapped in. This can be done as follows, where the value 10 for the number of steps of the random walk is arbitrary and only intended for example;

$combineWithTimeLengths$
$1\ (restartExtract\ (iterativeImprover\ nF)\ convergeTest)$
$10\ randomWalk$

5.8 Ant Colony Optimisation

Ant Colony Optimisation can be seen as a learning algorithm which attempts to learn how to guide the construction process for new solutions. This is achieved through converting solutions into sets of *pheromone trails*, which for choices in the construction process, indicate the likely quality of solution that will be derived from making that particular choice. ACOs can be implemented in one of two ways, either as a point or population based metaheuristic. However it is now commonly treated as a population based method, and this will be the primary approach taken in this Thesis.

A population based approach has strong similarities to the structure of a GA, reusing the *chunk* and *concat* pattern to manage the population structure. A population is used to derive a single *pheromone map* of the problem and this is then used several times to create a new population of solutions. This separation of functionality is used to give rise to the following template.

$aco :: Int \rightarrow$ -- population size
$\quad (Stream\ (List\ s) \rightarrow Stream\ a) \rightarrow$ -- Pheromone analysis
$\quad (Stream\ a \rightarrow Stream\ s) \rightarrow$ -- solution generation
$\quad Stream\ s \rightarrow Stream\ s$
$aco\ popSize\ aP\ gN = concat \circ doMany\ popSize\ gN \circ aP \circ chunk\ popSize$

The process of pheromone analysis can be further decomposed into the conversion of individual solutions into pheromone information and the merging of this information into a single pheromone map. For example, pheromones based upon numbers can be implemented using a combination of map and foldl (left reduction) and makes use of an empty pheromone list (*p0*).

$$map\ (foldl\ (zipWith\ (+))\ p0\ \circ\ map\ convert)$$

Typically ACOs make use of a longer term memory about the evolution of the pheromone map, implemented by adding a proportion of the previous weights to the current map before generating new solutions. A stream transformation providing this functionality can be composed with the creation of the pheromone map previously seen and parachuted into the algorithm as desired by the programmer. Many implementations are possible but only one will be presented here, using *window* to emphasise the use of memory in this operation. A simple version can be implemented using a helper function to apply degrade correctly over the pairs of values in the histories produced by *window*, including the initial special case where there is only one value in the history.

$$\begin{aligned}
&acoMem :: Num\ a \Rightarrow (a \to a) \to Stream\ a \to Stream\ a \\
&acoMem\ degrade = map\ g \circ window\ 2 \\
&\quad \textbf{where}\ g\ [x] = x \\
&\qquad\quad g\ [x,y] = degrade\ x + y
\end{aligned}$$

In the above examples the linking and degrading processes have been implemented using simple functions and addition. It is more flexible to provide these more general stream transformers, passed as parameters to the function. This gives rise to the following implementation.

$$\begin{aligned}
&acoMem :: (Stream\ n \to Stream\ p \to Stream\ p) \to && \text{-- link} \\
&\qquad\qquad (Stream\ p \to Stream\ n) \to && \text{-- degrade} \\
&\qquad\qquad Stream\ p \to Stream\ p \\
&acoMem\ link\ degrade = g \circ window\ 2 \\
&\quad \textbf{where}\ g\ ([x] : xs) = x : link\ (degrade\ (map\ head\ xs))\ (map\ last\ xs)
\end{aligned}$$

A point based implementation of ACO can also be implemented either by using a population size of one, or by building a new version using only the composition of pheromone analysis, solution construction and degrade.

5.8.1 Mutation

ACOs are known to suffer from a similar issue to GAs, which are effective at exploring the space of solutions widely, but unable to make the final push towards local minima. To improve upon ACO a common hybridisation is to use an iterative improver to modify the solutions and hence allow the ACO to learn from a series of local minima, rather than a series of non-optimal solutions.

Other metaheuristics may be composed with ACO directly to provide additional functionality over all solutions considered. Alternatively only some solutions of the ACO can be modified in a similar way to mutation in a GA, making use of the *nest* function (see Section 5.6).

5.8.2 ACO for recombination

It is noted in [6], that recombination operators are particularly difficult to design and implement, and that since ACO seems effective on combinatorial problems it may be particularly suited in this role. Their work supported this hypothesis, finding that ACO was indeed a quite general and effective recombination operator.

This can be facilitated through importing the operators for pheromone map construction and ant colony solution rebuilt into the GA structure.

$$
\begin{aligned}
&ga\ popSize\ mutateLikelihood \\
&\quad (acoRebuild \circ map\ (foldl\ link\ p0 \circ map\ convert)) \\
&\quad (doMany\ (popSize * 9\ `div`\ 10)\ (select\ (poisson\ 2))) \\
&\quad mutation
\end{aligned}
$$

5.9 Summary

This chapter has described the major functions of the library and shown how each of the metaheuristic families can be implemented using them. For each metaheuristic one or more major variations upon them has also been implemented through small variations of the combinators used to express their key functional parameters. This completes one of the primary goals of the Thesis, a framework and library of combinators that can express the major metaheuristics using a unified functional approach.

Chapter 6

Low level operators

The low level operators of recombination, perturbation and neighbourhood have, so far, been treated as the operations that the programmer must create in order to enable metaheursitics to operate over a problem. For any particular problem the metaheuristic designer has a range of choices related to how to implement the low level operators that will be used. How the low level operators are implemented can have a significant impact upon the overall performance of the search. While any operation can be constructed monolithically, and it may sometimes be necessary to do so, this chapter considers strategies for implementing low level operators and proposes how they may be decomposed to illuminate the design space.

6.1 Using lifted functions

A monolithically defined perturbation operation has the type $Stream\ s \rightarrow Stream\ s$. These are most easily defined in terms of the lifting of a basic function. For example, the TSP can be perturbed through exchanging cities in a sequence. This can be given using a function of the following type, where TSP is a problem specific data type for the TSP problem.

$$tsp\_city\_swap :: Int \rightarrow Int \rightarrow TSP \rightarrow TSP$$

This can then be lifted to operate over streams in a variety of ways, for example:

- *map* can be used, where the problem operator is partially specialised to yield a deterministic function, for example

$$map\ (tsp\_city\_swap\ 0\ 1)$$

  While this example would be of limited value, in general the lifting of such a function through map may be of use, such as in the case of neighbourhood functions.

- *zipWith*, and its variants, can be used to embed an external state into the stream, for example

$$\begin{aligned}
&getRandomInts\ r = unsafePerformIO\ (newStdGen \\
&\qquad\qquad\qquad\qquad\qquad\qquad \gg\!= return \circ randomRs\ r) \\
&stochasticCityExchange \quad \text{-- purely stochastic city exchange} \\
&\quad = zipWith3\ tsp_city_swap\ (getRandomInts\ range) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (getRandomInts\ range) \\
&hybridStoCyc \quad \text{-- a demonstration of a deterministic cycle} \\
&\quad = zipWith3\ tsp_city_swap\ (cycle\ [0\,..\,numCities]) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (getRandomInts\ range)
\end{aligned}$$

Neighbourhood functions and recombination functions can be similarly defined and lifted to operate over streams, in combination with parameterising streams of values.

6.2 Perturbation and Neighbourhood

Perturbation functions and neighbourhood functions can be transformed into one another, using standard functions of the library. A neighbourhood function lifted to act upon streams is a transformation from a stream of solutions into a stream of collections of solutions. A perturbation function may be derived from a neighbourhood function by defining a way to select solutions from each collection in a stream, for example, using a uniform stochastic selection process.

$$select\ uniform \circ neighbourhood$$

A neighbourhood function, which operates over streams, can be created from a perturbation function via a process of applying the perturbation function to each

value in the input stream several times, and gathering these results as a collection. This can be achieved by reusing the *doMany* function, for example;

$$doMany\ n\ stochastic\_perturb$$

When the perturbation function is not stochastic, but makes use of a known pattern of values, the *doMany* operation can be used to allow the construction of deterministic neighbourhoods, rather than requiring a separate monolithic definition. For example, the construction of an adjacent city swap neighbourhood;

$$doMany\ nCities\ (tsp\_city\_swap\ (cycle\ [0\,..\,nCities-1]\ [1\,..\,nCities]))$$

The result of such a wrapping can then be combined with further selection functions, giving back more elaborate perturbation functions. For example, using a stochastic perturbation function, but applying it several times, and only keeping the best.

$$map\ best \circ doMany\ n\ perturb$$

6.3 Decomposing Perturbation

The construction of perturbation functions can often be broken down into two or more phases. The simplest is the damage-repair pattern. In this formulation the first operation *breaks* the input solution in some way, and the second operation then *repairs* it. These two phases can be directly composed as; $repair \circ damage$.

Both damage and repair can be constructed in a number of ways with the following being some common options;

- Uniformly random choice of modification
- Greedy choice of modification, e.g. always take the choice that will yield the best short term results, most obviously in repair, but can also be in damage with an effective measure of how much change will occur from each operation
- Probabilistic distribution that favours greedy choices
- An exhaustive search operation, only applicable to repair[1].

[1]The use of exhaustive repair with some form of damage closely mirrors the approach known as *Large Neighbourhood Search* in constraint programming which works through an iteration of relaxations and re-optimisations of subsets of constraints [73].

For example, when performing perturbation upon a TSP problem using edge modification a number of edges must be selected to be removed. Typically this is done randomly, i.e. with uniform likelihood of any particular edge being chosen, however selecting to favour longer edges tends to improve the performance of a metaheuristic. How greedy the selection process is, and the probability distribution that is used, are parameters to the process. Similarly when inserting new edges into a damaged solution the same range of approaches are available.

These formulations have a strong connection to the selection operators that have already been created for the library. The abstraction of this functionality is best implemented through the provision of a new type class of functions that allow this new form of generic interaction with a problem.

```
class DR s dopt ropt | s → dopt, s → ropt where
  damageOptions :: s → [dopt]     -- how a solution may be damaged
  applyDamage :: dopt → s → s     -- apply damage
  repairOptions :: s → [ropt]     -- how a solution may be repaired
  applyRepair :: ropt → s → s     -- apply repair
  complete :: s → Bool            -- can a solution be repaired?
```

For example, a TSP usage of *DR* is often found when dealing not in cities, but in particular edges. In this conception the solution to a TSP is a set of selected edges and a solution is *complete* when the set of edges forms a hamiltonian cycle. The options for damage at any point are the set of edges currently selected for the solution, implementing damage is the removal of an edge from this set. The options for repair at a given point are the set of edges that could be added to the set legally, and implementing a repair is adding an edge to the set.

6.3.1 Higher level damage repair operations

The *DR* type class can be used to give rise to more complex tools for damaging and repairing a solution to a problem. Multiple rounds of damage can be applied before repair is begun. Repair may be partial (performing some number of steps that may not result in a new complete solution) or complete (continues with one repair strategy until completion is achieved). Partial repair functions can be composed together to give rise to hybrid repair methods using multiple strategies.

A full enumeration of all possibilities is neither possible nor useful, however what follows are some examples to show how some of these possibilities can be implemented.

```
uniformDamage :: DR s dopt ropt ⇒ Int → Stream s → Stream s
uniformDamage n
  = restartExtract
    (λsols → let ds = map damageOptions sols
                 in zipWith applyDamage (select uniform ds) sols)
    (const $ cycle (replicate n False ++ [True]))
fullRepair :: DR s b c ⇒ (Stream s → Stream s) → Stream s → Stream s
fullRepair f = restartExtract f (map complete)
greedyRepair :: (Optimisable c, DR s b c) ⇒ Stream s → Stream s
greedyRepair = fullRepair
  (λsols → let rs = map best ∘ map repairOptions $ sols
               in zipWith applyRepair rs sols)
```

The usage of this approach reduces the reliance of metaheuristics on monolithically defined problem specific operators. Where it is known that the low level operators can be varied to tune performance, this functional decomposition explicitly extends the design space for metaheuristics into the construction of these operators.

6.3.2 Neighbourhoods from damage-repair

Damage-repair allows for two variations upon a neighbourhood to be proposed. The first has previously been given, where a perturbation method is applied several times to a specific solution and the results gathered together. This could now be written as;

```
doMany n (repair ∘ damage)
```

The alternative is to move the damage operation outside of the *doMany*;

```
doMany n repair ∘ damage
```

In this formulation the solution will be damaged once and then rebuilt several times. For this to be of interest the repair method is required to be stochastic, or otherwise have some varying internal state.

6.4 Recombination

A monolithic implementation of a recombination method forms a contraction transformation, taking a stream of parents and yielding a stream of new solutions. Recombination methods admit one obvious decomposition, before being entirely problem specific. This decomposition is a process which identifies common blocks within the parent solutions. These blocks can then be recombined using a repair strategy, similar to those seen for the decomposition of the perturbation methods.

It is harder to cleanly capture this as new operations, however a proposal can be made for extending the original *DR* class to *DRA*. In this extension a new function *analysis* is added, which takes a list of solutions and gives rise to an analysis, stored as *Either solution analysis*. The analysis is then implemented as part of the class, giving rise to further damage and repair options. Damage and repair are themselves changes to make use of either the solution or analysis system, with damage creating analyses, and repair removing them. The previous generic functions would have to be changed to incorporate these modifications.

class *DRA s s′ dOpt rOpt* **where**
  $analysis :: [s'] \to Either\ s\ s'$
  $damageOptions :: Either\ s\ s' \to [dOpt]$
  $applyDamage :: dOpt \to Either\ s\ s' \to Either\ s\ s'$
  $repairOptions :: Either\ s\ s' \to [rOpt]$
  $applyRepair\ \ :: rOpt \to Either\ s\ s' \to Either\ s\ s'$

This complex system is proposed to allow the unification of recombination with the action of ACO. In this system an analysis could be a set of common components in the parents, which subsequently give rise to the need to *finish* the solution with repair. Alternatively a set of solutions could give rise to a data structure, stored as an analysis, which represents a pheromone map. As this is queried for repair options, it will impose an ordering upon the options, dependent upon their analysis, and hence allow the creation of new solutions.

6.5 Summary

This chapter has shown that while low level operators can be defined monolithically they can often be seen as being constructed through the composition of the

damage, repair and analysis building blocks. It has been shown how to use standard functions of the library, such as *doMany*, *restart* and *select* to combine the lower level building blocks into a wide variety of operators. While there is still a separation between problem specific operators and the logic of metaheuristics, the use of these simpler building blocks changes the focus of the discussion and provides clear guidance for the design of perturbation, recombination and neighbourhood functions, how they may be varied and customised.

Chapter 7

Hybrid Metaheuristics

Research into methods for hybridsiation have taken two complementary directions;

- a theoretical study, classification and analysis of the hybrids that have been appearing; and
- the implementation of APIs and toolkits to enable the programming and hybridisation of metaheuristics (returned to in Chapter 9).

Various authors have created taxonomies and classification schemes for hybrid metaheuristics, such as Talbi [79] and Raidl [65]. These have classified hybrid algorithms by a number of characteristics which may be present. However the hybridisations do not form a simple hierarchy due to the way the characteristics may overlap in any given algorithm.

This chapter examines the theoretical breakdown of hybrid metaheuristics based upon these two authors. It shows how each form of hybridisation is supported by the functional combinators that have been proposed, and that this often provides a very clear description of the interactions of the algorithms being hybridised.

7.1 High and low level hybridisation

This is the most general distinction between hybrids, relating to the "level of coupling" between the components being hybridised. Talbi's description of the distinction between high and low level hybridisation is to specify that low level hybridisation is where

> *a given function of a metaheuristic is replaced by another metaheuristic. [79, p. 543]*

By comparison, Talbi describes high level hybridisation as the situation where each metaheuristic retains the complete individuality of both components. For example; simulated annealing using an iterative improvement method (beginning from a more stochastic jump) as its perturbation method would be a low level hybridisation and; running an iterative improvement algorithm on the best solution that an ACO finds would be a high level hybridisation.

```
sa qualityExtraction
   (iterativeImprover ∘ stochasticPerturbation)
   streamOfRandomDoubles
   coolingStrategy
```

Figure 7.1: A sketch of a hybrid metaheuristic using a stochastic iterative improver as the perturbation function of a simulated annealing system. In this example a number of *placeholder* functions and parameters (qualityExtraction, stochasticPerturbation, iterativeImprover, streamOfRandomDoubles,coolingStrategy) are used and expected to be provided by problem specific code or more complex expressions for practical usage.

Low level hybridisation is provided by the library in the form of functional parameters, which allow for the *parachuting* of functionality to lower levels of the program. An example of this can be seen in Figure 7.1, which demonstrates how the perturbation function in simulated annealing may be easily varied. High level hybridisation can be provided through the use of function composition. For example the high level hybridisation of ACO and iterative improvement is simply the composition of these two strategies.

High level hybridisation can also be implemented by combining fully evaluated strategies. For example, an alternative hybridisation of ACO and iterative

improvement is to run ACO to convergence and termination, and use the best solution found as the seed for an iterative improver. This may be seen in Figure 7.2.

$$last \circ convergeCheck' \circ loopP\ iterativeImprover \circ$$
$$last \circ convergeCheck \circ bestSoFar \circ loopS\ aco\ \$\ initialSols$$

Figure 7.2: An example of very high level hybridisation of algorithms, where both aco and iterative improver are run separately, with appropriate convergence checking and selection of best solutions found. Both aco and iterativeImprover are placeholders for more complex expressions, as are convergeCheck and convergeCheck', where the names serve to illustrate their purpose.

Raidl's conception of high level hybridisation is broadly the same as Talbi's however he uses a different description of low level hybridisation. For Raidl a low level hybridisation is the use of a characteristic or function from one metaheuristic within another. This makes the distinction between high and low level hybridisation clearer, where in Talbi's work each algorithm remains self contained, and the hybridisation is achieved by how they communicate.

Raidl extends the concept of low level hybridisation to include the situation where characteristics from the respective algorithms are combined, rather than using one complete metaheuristic as a component of another. This form of hybridisation is also supported through the same basic operations such as function composition and especially use of function parametrisation. Figure 7.3 gives an example of the hybridisation of the concept of avoiding revisiting previous solutions, drawn from TABU search, used in support of Simulated Annealing. It is created by the parametrisation of simulated annealing with a perturbation operation that shares streams of information with the outer strategy. Changes to simulated annealing at the top level are limited. Hybridisation of low level characteristics is more akin to the design of a new algorithm, which can be aided by the use of functional parameters, parachuting and function composition, but defies the implementation in terms of a single combinator.

7.2 Execution order

Talbi uses the distinction of *relay* and *teamwork* as a way to describe hybrids. In a relay the output of one strategy is fed into, or used by another. The simulated annealing / iterative improver example (seen in Figure 7.1) is an example of a

```
λxs → sa quality
          (p (window winSize xs))
          streamOfRandomDoubles
          temperatureStrategy
          xs
where
  p ws = restartExtract
     perturb
     (λvs → let f bs (w:wss) = w:f (tail bs) wss'
                    where wss' = if head bs then wss
                                            else (w:wss)
                 zs = zipWith (flip elem) (f zs ws) vs
             in zs)
```

Figure 7.3: An example of low level hybridisation through characteristic exchange, restricting the results of a perturbation for Simulated Annealing to values not recently seen, as used in TABU. Shows the reuse of self contained functions such as window for the memory and lack of changes to the Simulated Annealing strategy. Implementation of the new perturbation operation requires a restart combinator and an alignment of the windows with the variable rate of acceptance in the restart process. Contains placeholders for; window size, perturbation, quality extraction, temperature strategy and supply of random doubles for the simulated annealing choice operation.

relay hybrid. A simpler example is the direct composition of different strategies, using the standard composition operator. This can also be achieved using the *nest* combinator, as seen in Figure 7.4.

In a teamwork hybrid several processes produce solutions, aiming to find the best possible solution between them. A genetic algorithm can be thought of as a form of teamwork hybrid where a number of identical recombination processes act upon the population to generate new solutions to the problem. Algorithms where several strategies proceed independently, but periodically exchange information, are also teamwork strategies. Figure 7.5 shows how nest may be used to provide periodic communication of information between independent streams of computation. This example suggests the use of a recombination function as might be found in a genetic algorithm, but there is no requirement for this particular operation to be selected for the role, nor for there to be only one communication process.

Raidl describes two forms of serial operation, in addition to a more detailed

```
loopP (nest [(True, strategyA), (False, strategyB)]
            (cycle [True, False])
      ) initialSolution
```

Figure 7.4: A sketch of a relay hybrid metaheuristic, using two strategies, A and B in relay over a series of solutions. This approach allows the output of each strategy to appear on the final stream of solutions, where function composition would feed the output of A directly into B, and only provide the stream of solutions outputed by B.

```
solExch :: (Stream (List v) → Stream v) → Stream v → Stream v
solExch recombine
  = concat
  ∘ nest [(False, id),
          (True, map (λr → [r, r]) ∘ recombine)]
         (cycle [False, False, False, True])
  ∘ chunk 2
loopS (solExch ∘ nest (zip names solvers) (cycle names)
      ) (take (length solvers) sols)
```

Figure 7.5: This sketched example of usage shows how a set of solvers can proceed independently, and have all their solutions visible as a single output of the solver. Their independence is interrupted periodically by the function *solExch*, which uses an arbitrary cycling pattern to select solutions and recombine them using a problem specific recombination method. The new solution is reintroduced to each of the strategies that provided one of the seeds. This sketch involves a number of placeholder terms; names, solvers and recombine.

breakdown of parallel hybrids. The two serial forms are;

- batch, where each metaheuristic is run once in series (see the last section and Figure 7.2); and
- interleaved, where the metaheuristics run one after another, which can be seen as similar to the *relay* system of Talbi[1].

[1]Using the functional combinators, particularly *nest*, as seen in Figure 7.4, it is possible for the library to implement interleaved algorithms over any number of metaheuristics, using very complex, or even stochastic patterns of interleaving. Such modifications retain the separation of the search logic from the definition of the patterns and are quickly devised and implemented in this framework.

Raidl's break down of parallel algorithms describes a large number of physical characateristics of the machines such algorithms would run on, such as shared and distributed memory. This is counter to the conception of very high level declarative programming, where the programs can be run as easily on a parallel machine as on a serial machine. This Thesis chooses to focus upon the declarative programming aspects of hybridisation, and does not concern itself with these low level issues.

7.3 Heterogeneity

Both Talbi and Raidl describe hybrids in terms of the heterogeneity or homogeneity of their components. A homogeneous algorithm uses several instances of the same search strategy, and hence makes most sense in terms of population and parallel algorithms. For example, an *island* based genetic algorithm is several instances of the same algorithm, each on its own population. For this to be a hybrid strategy the islands must exchange solutions periodically, thus making this more than just parallel search. Heterogeneous algorithms are those where different search strategies are used, either in parallel or serial.

Heterogeneous algorithms require no further examples than those seen in the previous sections. Hybridisation of different algorithm types is simply the composition, or use of nesting, with heterogeneous strategies. Homogeneous algorithms require a combination of nesting of the different strategies, effectively dividing the underlying stream of solutions into several parallel tracks of computation, combined with a periodic communication strategy seen in the previous section.

Talbi proposes a further division, hybrids of general algorithms and hybrids of specialist algorithms. A general hybrid combines a number of algorithms that are optimising the same problem, for example an iterative improver and a genetic algorithm, both acting upon an instance of the TSP. In the functional context this classification does not change the composition of operators.

Specialist hybrids combine algorithms that solve different problems, though the combination is beneficial in some way. In the context of functional languages this may be seen as the combination of algorithms that operate over different data types. For example, an algorithm may be created for the TSP, where one part works on the TSP directly while another works on a transformation of the TSP into SAT. For this to work conversion algorithms will be needed, but the implementation in terms of the library will rely upon the composition operator alone.

Raidl also proposes another form of hybridisation based upon *solution space decomposition*. The classification here is whether the algorithms are all considering every solution to the problem, or if each one is considering a subset of the solutions. This is a form of divide and conquer. Implementation of solution space decomposition has not been attempted in this Thesis, however it can be hypothesised to require problem specific decomposition and reconstitution operators, and result in a pattern of computation not dissimilar to *nest*. An example of how this could be achieved can be seen in Figure 7.6.

$$
\begin{aligned}
&\mathit{decomposeProblem} :: \mathit{Stream}\ s \rightarrow \mathit{List}\ (\mathit{Stream}\ \mathit{decompS}) \\
&\mathit{reconstitue} :: \mathit{List}\ (\mathit{Stream}\ \mathit{decompS}) \rightarrow \mathit{Stream}\ s \\
&\mathit{reconstitute} \circ \mathit{zipWith}\ (\$)\ [\mathit{solverA}, \mathit{solverB}] \circ \mathit{decomposeProblem}
\end{aligned}
$$

Figure 7.6: A sketch of the operation of a heterogeneous hybrid, where the heterogeneous search strategies operate upon sub-subproblems, created through the decomposition of the solution space.

7.4 Source of algorithms

Raidl proposes that a final description of the form of hybridisation is a question of what is being hybridised, pointing out that the algorithms being hybridised are not limited to metaheuristics themselves. He proposes the use of problem specific solution methods, fuzzy AI and complete methods such as branch and bound.

The use of exhaustive methods, for example as one option when repairing a solution during a perturbation, has been seen in Chapter 6. These methods can be made available by designing stream transformations that encapsulate them, or lifting them using *map* and *zipWith* if this is an option. Hybridisation involving these alternative algorithms then proceeds using the combinators that have been discussed.

7.5 Conclusion

This chapter has addressed one of the key hypotheses of this Thesis, that functional languages and combinators can enable the hybridisation of metaheuristic algorithms. It has shown the ways in which other researchers have classified the

methods of hybridisation for metaheuristics, and how these methods can be supported concisely using the framework of this Thesis. The majority of hybridisation techniques seen here revolve around the operators $(\circ)$ and *nest*, requiring little or no modification of individual metaheuristic strategies being hybridised. While more sophisticated low level hybridisation of characteristics still poses problems, these can often be assisted by the approach taken, exploiting the parametrisation of strategies to combine characteristics deeply within the algorithms with minimal changes at the top level.

Chapter 8

Monads and Arrows

The previous chapters have presented a library for metaheuristic implementation based upon combinators for Streams and Dataflow processing of information. Monads have been avoided since Chapter 4, where it was shown that the direct usage of monads for sequencing operations of metaheuristics lacked the flexibility desired in the system. None the less monads and arrows represent powerful tools for structuring computations, and so this chapter examines using them in this fashion, to manage the construction of the streams previously seen.

8.1 Monads

Throughout this Thesis streams have been presented as lazy infinite lists, in this section this must change, as conflicting definitions exist for the Monad of Lists and the Monad of Streams.

8.1.1 The List Monad

The definition and proof of the List Monad was provided in the earliest papers on monads for functional programming [84], and can be described in Haskell as follows;

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ [a]\ \textbf{where}\\
&\quad \mathit{return}\ x = [x]\\
&\quad (\gg\!=) = \mathit{concatMap}\\
&\quad (\gg)\ a = \mathit{concat} \circ \mathit{replicate}\ (\mathit{length}\ a)
\end{aligned}
$$

It is unclear how these functions will aid the expression of data flow programming, and especially the implementation of metaheuristic algorithms. The *concatMap* function is rarely used in the combinators that have been seen, although it does present one possibility for an alternative definition of stretch as;

$$stretch\ i\ x = x \gg\!= replicate\ i$$

However this is an exception rather than a common case and in general the List Monad does not seem to aid the task of implementing metaheuristics.

8.1.2 The Stream Monad

A definition of Streams as a monad, using lists as the underlying data type, is provided by Wu and Gibbons[1] in an exchange on their blogs. This approach can be implemented in a number of ways, for example;

$$\begin{aligned}
&\textbf{type}\ Stream\ a = [a] \\
&\textbf{instance}\ Monad\ Stream\ \textbf{where} \\
&\quad return = repeat \\
&\quad (\gg\!=)\ as\ f = [s\ !!\ i \mid (s,i) \leftarrow zip\ (map\ f\ as)\ [0..]]
\end{aligned}$$

In this example bind ($\gg\!=$) applies the bound computation to each value resulting from the original computation for a stream. This yields a *stream of streams*. This *matrix* of values, is then reduced into a single stream, by selecting one value from each stream. In order to obey the Monad laws the values selected for the *leading edge* of the matrix of streams.

However this interpretation of streams has similar computational issues to the naive approaches to FRP previously seen. This can be seen more clearly if the bijection of streams to functions from an index to a value is used.

$$\begin{aligned}
&\textbf{type}\ Stream\ a = Int \rightarrow a \\
&\textbf{instance}\ Monad\ Stream\ \textbf{where} \\
&\quad return\ x = \lambda_ \rightarrow x \\
&\quad (\gg\!=)\ f\ g = \lambda time \rightarrow g\ (f\ time)\ time
\end{aligned}$$

In this context the bind function can be seen to operate somewhat like a composition operation, and be efficient where the functions f and g are not inductively defined. Where these functions are inductive this has equivalent execution problems to those seen in FRP.

[1] The final version that this code is drawn from can be found at http://patternsinfp.wordpress.com/2010/12/31/stream-monad/.

8.2 Co-Monads for Streams

Co-monads have been shown to provide a strong theoretic foundation for describing the semantics of data flow languages. This had previously not been done; and as Uustalu and Vene remark,

> *meaningfulness or right meaning of higher-order dataflow has been seen as controversial. [80, p. 136]*

Comonads are the *dual* of monads, defined in Haskell by the type class;

> **class** *Comonad w* **where**
> $extract :: w\ a \to a$
> $cobind :: (w\ a \to b) \to w\ a \to w\ b$

Where monads represent a computation yielding a value, a comonad represents a computation for a value in a context. For example, the following is one possible implementation of the comonad for streams;

> **instance** *CoMonad* $[a]$ **where**
> $extract = head$
> $cobind\ f = map\ f \circ tails$

In this example the *cobind* function applies its parameter f to *every* possible context (in this case current and future values) of a stream, giving rise to a new stream. However this approach fails to adequately describe the function of *zipWith*, which is used regularly in this Thesis for computations that rely upon more than one stream. It is possible to build functions with the type $w\ a \to w\ b \to w\ c$, however this does not directly make use of the comonadic machinery. Functions such as *window* cause greater issues, with the need for limited history of the stream. One way to overcome these issues is to use the bijection of streams to functions from an index to a value, so window could access previous values of a stream:

> **type** $StreamF = Int \to a$

However this is a behaviour, as seen in FRP, over a discrete time parameter, and while it would provide the desired flexibility would suffer from similar time and space leak problems to FRP.

The approach taken in [80] is to represent streams as data structures which both know their current position, the history of all values that the stream has produced, and the computation for the future values of the stream. There are still

issues with this approach, relating to time and space leaks in running computations.

While comonads present a possible future avenue for data-flow techniques in functional languages they do not seem to be mature enough at the current time to aid in the implementation of metaheuristics, nor do they provide a simplification of the expression of metaheuristic algorithms.

8.3 Arrows

A Monad captures a pattern of computation. The computation can, but is not required to, be run on an initial set of parameters, such as the String in a parser, or a representation of the initial state of a system. Arrows are a more general model of computation proposed by Hughes [43, 61]. An Arrow is more flexible because it does not represent the computation of a value, but the *transformation* of a value, which is more suited for the representation of streams and data flow computations. Hughes describes it in the following way;

> *whereas monadic computations are parameterised over the type of their output, but not their input, arrow computations are parameterised over both. The way monadic programs take input cannot be varied by varying the monad, but arrow programs, in contrast, can take their input in many different ways depending on the particular arrow used. The stream function example above illustrates an arrow which takes its input in a different way, as a stream of values rather than a single value, so this is an example of a kind of computation which cannot be represented as a monad. [44, p. 77]*

The definition of Arrows for Streams that will be used here will not be the same as the model used by Hughes for Stream Processors, but will instead remain closer to the model of data flow and stream transformation seen in Chapter 5. Technically, all the stream transformations that have been seen are simply functions, and as such are already Arrows, however this can be made more explicit, by encapsulating them in a new data type *StreamT* and making this data type an instance of the Category[2] and Arrow classes.

$$\textbf{newtype}\ StreamT\ a\ b = StreamT\ \{runStream :: Stream\ a \rightarrow Stream\ b\}$$

[2] In Haskell all *Arrows* must also be part of the type class *Category* and *Category* provides the definition of composition, and hence the $\ggg$ operator.

```
instance Category StreamT where
  id = StreamT Prelude.id
  (∘) (StreamT a) (StreamT b) = StreamT (a ∘ b)
instance Arrow StreamT where
  arr = StreamT ∘ map
  first (StreamT f)
    = StreamT (λxs → let (as, bs) = unzip xs in zip (f as) bs)
```

This definition permits the reimplementation of the basic functions of the library, but does not provide a significant improvement in expressiveness, though the arrow composition function captures the notion of a pipeline more clearly than standard function composition.

```
chunk :: Int → StreamT a [a]
chunk s = StreamT (iterate (drop s)) ⋙ arr (take s)
stretch :: Int → StreamT a a
stretch s = arr (replicate s) ⋙ StreamT concat
doMany :: Int → (StreamT a b) → StreamT a [b]
doMany s f = stretch s ⋙ f ⋙ chunk s
```

However some of the more specialised functions such as *improvement* can be given quite different implementations using *arrow notation* [61] in Haskell. These alternative implementations can hide some elements of lifting to operations over streams, in this case filter, thus giving a potentially more intuitive description of how the system is operating.

```
improvement :: Ord a ⇒ StreamT a [a] → StreamT a [a]
improvement nF = proc xs → do ns ← nF ≺ xs
                              returnA ≺ filter (<xs) ns
  -- represented using functions directly
improvementBasic :: Ord a ⇒ StreamT a [a] → StreamT a [a]
improvementBasic nF = arr (<) &&& nF ⋙ arr (uncurry filter)
```

This can be used to give an implementation of first found iterative improvement which is almost equivalent to the version that has been presented in the preceding chapters.

```
firstFound :: StreamT [a] a
firstFound = arr head
```

$$ffii :: Ord\ a \Rightarrow StreamT\ a\ [a] \rightarrow StreamT\ a\ a$$
$$ffii\ nF = improvement\ nF \ggg firstFound$$

It can be seen from these examples that stream transformation operations can be implemented easily using arrows, and that in doing so the programmer can gain access to features such as the specialised arrow notation. However, these examples do not exploit features or generalisations that are specialised to Arrows, with most of the data types being in terms of the *StreamT* type, and not the more general arrow type.

These examples have made use of arrow operators such as &&& and $\ggg$, and access to these functions for structuring streams of computations is very valuable. However since stream transformations are functions, and functions are arrows, these useful arrow operators are available already to any programmer that wishes to use them. In general is does not seem that the explicit use of arrows are inherently more powerful or expressive than not doing so.

8.4 Summary

Monads, CoMonads and Arrows can be used to represent stream computations. Arrows provide the most directly useful combinators for implementing stream based computations, and it has been seen how they can be used to implement elements of the library. However none of these systems significantly improve upon the expressiveness of the standard combinators for lists found in Haskell itself.

Chapter 9

Comparison with object oriented frameworks

There are many existing toolkits and frameworks to aid in the implementation of and hybridisation of metaheuristics. Some are based upon procedural languages, most are object oriented and all are imperative. For a more complete examination of these frameworks please see [56].

This chapter will provide examples of code written in two of the frameworks that are compared in [56], and contrast it with code written for the same task using the Haskell framework. The examples will illustrate the improved clarity of the algorithms when written using the Haskell framework, where all of the logic of the search algorithm is present at the top level of the program, rather than in a complex object hierarchy. To illustrate the performance of the Haskell library, timing data will be presented for each implementation, using a TSP instance as the example problem.

The choice of the frameworks for this comparison have been made based upon the following requirements:

- an example of a solver for the TSP should be available, and it is preferable if the code is provided by an expert in the framework being compared to, rather than an inexpert implementation.
- the frameworks being compared should be active projects, rather than old

projects that have been discarded by both users and the original creators.

- the two frameworks chosen should be written in different programming languages, so as to allow potential differences to be seen.

The first framework that will be used is *ParadiseEO*, a generic framework for optimisation that is implemented in C++. ParadiseEO provides tools for both population based and point based algorithms, but here will be used to provide an implementation of Simulated Annealing, to contrast with the second framework. The second framework is *OPT4J*, implemented in Java and only providing tools for population based metaheuristics.

It is preferable not to compare the loading routines and data models used for TSP, however in the implementations of both Opt4J and ParadiseEO, the inheritance mechanism used means that these components cannot be easily separated from the search logic. Similarly it is not the aim to compare the memory management models of the different languages, Java, C++ and Haskell, however it is not possible to separate these concerns from the implementations. The approach taken will be to focus upon the high level implementation of the algorithms, that the frameworks should assist with, highlighting these issues only when they become problematic.

9.1 ParadiseEO

ParadiseEO [8] is a *"white-box"* suite of systems in one framework for optimisation problems. The architecture focuses upon evolutionary algorithms and support for parallelism, but also has modules for point based algorithms and multi-objective optimisation problems.

9.1.1 Architecture of ParadiseEO

ParadiseEO employs a class based architecture with three key components;

Runners encapsulate a solver and perform one "run" of it, from seed solution to result solution, according to internal logic. An example of this is a Simulated Annealing *runner*.

Solvers provide both generic top level control of runners (such as restart functionality and preservation of the current best solution) and provide hybridisation through providing a particular order to execute runners in.

Helpers are utilities which are used by other parts of the system. The helpers in the system include; evaluation of solutions, crossover of solutions, neighbourhood exploration, cooling schedules for simulated annealing and stopping criteria for the algorithms.

Default implementations of TABU search, Simulated Annealing and a variety of hill climbers are provided, with the various helper classes needed to provide their functionality.

Extension of the functionality is provided through the object system, replacing components that share certain superclasses or interfaces. Hybridisation can be implemented in a top level harness, by running one algorithm and feeding the output of that search into later phases, or through the exchange of parameterising components. This leads to the complex class hierarchy that is seen.

It is interesting to note that ParadiseEO makes sporadic use of generators[1] to provide on demand computation, for example in the generation of neighbourhoods, where the neighbourhood is a generator for new solutions. However the use of generators is sporadic within ParadiseEO, without an abstraction of the general pattern. Instead each component that intends to act as a generator implements the pattern individually.

9.1.2 A Simulated Annealing Implementation

Figure 9.1 shows an implementation of Simulated Annealing for the TSP written using ParadiseEO. The associated classes for modelling and loading the TSP can be found in Appendix E.1.1.

This top level harness does provide an indication of the search logic being used, based upon the classes and parameters to constructors used. For example, the cooling strategy is clearly defined, with a recognisable starting temperature and cooling rate, though only examining the documentation or the library source code will indicate that it is a *geometric* cooling schedule. More unusual parameters are the final temperature and the delay of the cooling strategy for a number of steps at each temperature.

The perturbation operation is less clear, being provided by a neighbourhood generator. The name of the generator suggests that it will randomly swap cities in

[1] Generators provide a method for creating streams and on demand computation in imperative languages.

```
// Neighbours defined as swaped elements of a list.
typedef moIndexedSwapNeighbor<Route> Neighbor;

int main (int __argc, char * __argv []) {
    eo::rng.reseed(time(NULL));

    Graph :: load (__argv [1]) ; // Problem instance

    Route solution;     // solution data structure
    RouteInit init;     // solution creator
    RouteEval fullEval; // solution evaluator

    init(solution);
    fullEval(solution);

    // defines a random neighbourhood
    moRndWithoutReplNeighborhood<Neighbor>
               neighborhood(416);
    // evaluate through modification not copy
    moFullEvalByModif<Neighbor>
               neighborEval(fullEval);
    moSimpleCoolingSchedule<Route>   // geometric cooling
               coolingSchedule(1000, // start temperature
                               0.99, // cooling rate
                               5,    // stay at each temp. for
                               0.01);// terminate at this temp.

    moSA<Neighbor> hc(neighborhood, fullEval,
                      neighborEval, coolingSchedule);

    std::cout << "initial solution: "
              << solution << std::endl ;
    hc(solution);
    std::cout << "final solution: "
              << solution << std::endl ;
  return 0 ;
}
```

Figure 9.1: The main file for a Simulated Annealing based solver, implemented using the ParadiseEO-MO library in C++. This is modified from source code originally downloaded from the ParadiseEO website, written by Sébastien Cahon and Thomas Legrand.

the sequence, however an examination of the code in the class *moRndWithoutReplNeighborhood* indicates the following logic:

- When first initialised set a maximum index counter to the size of the neighbourhood.

- When generating neighbours, pick a random index up to the current maxi-

mum, and swap with the current maximum.

- Reduce the current maximum by one.

The functions used for this code are reproduced in Appendix E.1.2. Due to the way that neighbours are generated, through selecting a random number up to an index counter, the ParadiseEO code must specify the size of the neighbourhood to be the number of cities in the TSP, despite Simulated Annealing only generating a single solution from each neighbourhood. This is hard coded for convenience on line 13 of Figure 9.1.

It is not clear whether the search process will remain in a specific instance of a neighbourhood until a new solution is accepted, or create new neighbourhoods at each step of the process. An examination of the files *moSAExplorer*, for exploring a neighbourhood, and *moLocalSearch* the super class of *moSA* reveals the full process. At each step the algorithm will create a new neighbourhood, and draw the first randomly generated solution from it and then decide to move to it, or not. The full code from the original classes can be found in Appendix E.1.2.

The search logic can be described as follows:

- The perturbation of a solution is to swap a random city in the sequence with a specific index, nominally the last.

- The cooling strategy is geometric, starting at 1000, reducing by $*0.99$ each time it is cooled, and remaining at each temperature for n (in this example 5) steps.

- The process terminates once $n * 1146$ steps have been taken. The constant 1146 can be computed based upon the number of steps for the geometric progression to reach 0.01 with no delay.

- The movement choice is the standard function, as seen in the SA neighbourhood explorer class.

The Haskell code is shown in Figure 9.2. While the Haskell implementation is almost as long as the C++ version, it provides a more detailed description of the search logic at the top level; including the separation of the termination logic, the selection of the best solution encountered, the basic geometric cooling strategy, the characteristic of holding at each temperature for a number of steps and the perturbation routine being used. The functions *saChoose* and *geoCooling* are

```
main :: IO ()
main = do
    -- loading the problem
  problem ← do g ← newStdGen
               p ← loadTSPFile TriangularMatrix "fl417.tsp"
    -- and creating an initial solution
               return $ randomiseRoute g p
    -- creating the perturbation operation
  perturbOp ← do g ← donewStdGen
                 let n = numCities problem − 1
                 let rs = randomRs (0,n) g
                 return $ zipWith (swapCitiesOnIndex n) rs
    -- creating the strategy
  g ← newStdGen
  let strat xs
      = zipWith4 (saChoose solutionValue)
                 (stretch 5 $ geoCooling (0.99 :: Double) 1000)
                 (randoms g)
                 xs
                 (perturbOp xs)
    -- running and terminating the strategy, getting the best
    -- solution, and printing it to screen
  print $ (solutionValue :: TSPProblem → Double) ∘
          best ∘
          take (5 ∗ 1146) ∘
          loopP strat $ problem
```

Figure 9.2: The main file for a Haskell implementation of the Simulated Annealing TSP solver, following the same logic as the ParadiseEO version. In this implementation the cooling function has the same parameters as the ParadiseEO version, and to achieve the characteristic of holding at each temperature for 5 steps the basic pattern is *stretched*.

drawn in from the metaheuristic library, while the data model of the TSP, the loading routines and file parsers for TSPLIB and basic interactions are provided by the combinatorial problems library[2].

	Average Score			Average Time		
Cooling Delay	10	20	40	10	20	40
ParadiseEO	458000	445000	448000	1.12s	2.10s	4.05s
Haskell	296000	257000	250000	0.96s	1.86s	3.68s

Table 9.1: A runtime comparison of Simulated Annealing for TSP fl417, in ParadiseEO and Haskell. Each has been computed with varying cooling delays, causing a linear increase in the number of required computations. Each test was run 10 times, and the average, to 3sf is presented here. Times are presented to the nearest 100^{th} of a second.

9.1.3 Performance Comparison

Table 9.1 provides results from testing both the Haskell and ParadiseEO versions of the Simulated Annealing algorithm. The results are surprising, in that it is apparent from the poor correlation between the quality of the solutions found that the search logic being used is not the same, though at the time of writing it is not understood why.

The timing results are similarly interesting. Each shows the same linear increase as the cooling strategy is modified, and the termination process updated to run the search processes for longer. However it appears that the Haskell implementation is faster than the C++ version. This difference may be caused by tasks such as memory management, being performed by the programmer in C++, but using specialised garbage collection in Haskell, however it would be premature to draw detailed conclusions, given the difficulties in matching the search logic.

9.2 Opt4J

Opt4J [55] is a framework for using genetic algorithms for optimisation written in Java. It consists of 326 classes for describing variations upon genetic algorithms (version 2.7), and also possesses a sophisticated graphical interface to allow for lay usage of the algorithms and not just programmer usage. It has built in support for a number of standard ways to perform mutation and perturbation upon solutions and standard setting for population size, mutation rates and length of time to run the program for.

[2]The combinatorial problems library can be found on hackage and is called *combinatorial-problems-0.0.4*.

9.2.1 Architecture of Opt4J

Opt4J implements the single method of evolutionary algorithms, with a focus upon the concept of multi-objective optimisation and models for problems composed of a number of submodels.

The library defines *individuals*, comprised of a number of *genotypes*, where each genotype is a way to model a part of the problem. Where an individual has only one genotype, this represents a problem which has not, or cannot, be decomposed into subproblems. Opt4J provides a number of built in ways to represent genotypes, including lists, bits and numbers. The provision of built in representations allows for the provision of a selection of built in mutation, recombination and crossover methods, to allow a programmer to quickly model a problem and get an evolutionary mechanism set up.

The library encourages the programmer to break the problem down into two representations, the genotype and phenotype. The genotype can be thought of as a generic problem independent structure with generic modification operators, provided by Opt4J. The phenotype is the problem specific model of a solution, constructed from the genotype. This philosophical division is derived from nature, with the division of genetic material from the final life form.

At the top level, when implementing a new problem, the programmer is required to provide Opt4J with three classes;

Creator provides initial solutions to the evolution process.

Completer is required to ensure the validity of potentially complex problem models. Even the TSP can easily result in invalid solutions when performing stochastic recombination of two or more solutions.

Decoder encapsulates the genotype to phenotype conversion, and the subsequent pricing of solutions to allow Opt4J to perform natural selection and preserve the best seen solutions.

While designed around the use of a population and recombination, it is possible to create point based algorithms using Opt4J by setting the population to one, disabling recombination and providing an appropriate mutation operation, encapsulating a local search method.

The extension of functionality is provided through the creation of new classes, or inheritance of existing functionality. This can be seen in the library as it exists at the time of writing, for example in the creation of generic cross-over operators

for genetic algorithms, where the most specialised class *CrossOverIntegerDefault* extends the more generic *CrossOverIntegerRate* and finally this extends the concept of list cross over, defined in *CrossoverListRate*.

```
public class Harness{
  public static void main(String[] args)
  {
    EvolutionaryAlgorithmModule ea;
    ea = new EvolutionaryAlgorithmModule();
    ea.setGenerations(1000); // generation limit
    ea.setAlpha(100);        // population size
    ea.setMu(25);            // children per generation

    SalesmanModule tspModule = new SalesmanModule();
    Collection<Module> modules = new ArrayList<Module>();
    modules.add(tspModule);
    modules.add(ea);

    Opt4JTask task = new Opt4JTask(false);
    task.init(modules);
    try {
        task.execute();
        Archive archive = task.getInstance(Archive.class);

        for (Individual i : archive) {
          System.out.println(i.getObjectives());
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        task.close();
    }
  }
}
```

Figure 9.3: The main file for a Genetic Algorithm based solver, implemented using the Opt4J library in Java.

9.2.2 A Genetic Algorithm Implementation

The TSP implementation for Opt4J is drawn from the standard documentation and tutorials which can be found at:
opt4j.sourceforge.net/documentation/2.7/book.xhtml
The top level *harness* for the solver can be seen in Figure 9.3, while the classes used to model, load and evaluate the TSP itself can be found in Appendix E.2.1.

The implementation shows a GA where the population size is 100, and it will run for 1000 generations. A third setting is called *mu*, and is set to 26; this is

the number of children per generation. There is no indication of how parents are selected from the population, if mutation is happening, what the recombination method is, nor how new solutions are used to create the new population.

The search for the settings of these important characteristics proved unsuccessful and so likely looking classes have been presumed to be the defaults.

- Crossover for lists has been selected from *CrossoverListXPoint*. It operates by splitting each parent at a randomly selected index. Each child is made of the first part of one parent, with the second part having cities in the order that they appear within the second parent. The crossover operation returns two solutions.

  A specialised function called *crossover* was created for the Haskell version to implement this and can be found in the Appendix E.2.3. While it includes random numbers, the function itself achieves this through the use of *unsafePerformIO*, thus reducing the code overhead in the top level implementation. The function has the following type;

  $$crossover :: Stream\ (List\ TSPProblem) \rightarrow Stream\ (List\ TSPProblem))$$

- The effectiveness of a genetic algorithm is significantly influenced by the selection method for the parents of additional offspring. However at the time of writing it has not been possible to find, in the source code or documentation, details about how the selection process is implemented or what the default settings are. Nor does the library offer runtime interfaces to query the current settings of the selection process.

- As with the selection operators, the source code and documentation do not provide information regarding the default setting for *mutation* operators. The mutate operation for lists was assumed to be *MutatePermutationSwap*, however there were a number of other choices. This mutation strategy operates by swapping cities in specific solutions. The likelihood of any particular city being swapped with a randomly selected city is defined by the *mutation rate* variable. In this way it indicates a mutation rate for the genetic material of the entire population, rather than the likelihood of any individual being mutated.

  This mutation operation is assumed to be applied to every new solution generated, due to the code found in the file *MatingCrossoverMutate*, in the EA *optimiser* folder.

The implementation of this operation required a specialised Haskell function which has been called *mutate*, see Appendix E.2.2. It provides random numbers through the use of *unsafePerformIO*, and hard codes a number of parameters to the specific test problem that is being examined. This code has the following type;

$$mutate :: Double \rightarrow Stream\ TSPProblem \rightarrow Stream\ TSPProblem$$

- No mutation rate has been found at the time of writing, nor has any way to query the Opt4J implementation for the setting of this value been found.
- Population management appears to operate by adding all new solutions to the old population and then removing the worst solutions so that the population returns to the population size limit. This is based upon code that appears in the *EvolutionaryAlgorithm* class.

  In order to implement this alternative form of population management a new operator was created and this can be seen in Figure 9.4. This takes 2 streams of populations, assumed to be the stream of *old* populations and a stream of *new* solutions. These populations are merged, sorted and then the best solutions are preserved.

```
populationMerge :: Optimisable s
                => Int
                -> Stream (List s)
                -> Stream (List s)
                -> Stream s
populationMerge psize xs
  = concat o                          -- merge populatons into stream
    map (take psize) o                -- keep the best
    map sortO o                       -- reorder, best to worst
    zipWith (++) (chunk psize xs)     -- merge populations
```

Figure 9.4: A new population management strategy, to more closely mimic the operation of the Opt4J counterpart.

The implementation of the proposed genetic algorithm in Haskell can be seen in Figure 9.5. In order to ensure performance, the transformations of the GA selection method from populations, proposed as a compiler rule in Section 5.6.1,

have been implemented manually and are contained in the functions *makeDistPop* and *selectFromMap*. Full implementations of these functions can be found in the appendices.

```
randomStarts :: RandomGen g ⇒ TSPProblem → g → [TSPProblem]
randomStarts x g = let gs = unfoldr (Just ∘ split) g
  in map (flip randomiseRoute x) gs
main = do x ← loadTSPFile TriangularMatrix "fl417.tsp"
  starts ← newStdGen >>= return ∘ take 100 ∘ randomStarts x
  print ∘ minimum ∘ take (100000) ∘ push ∘
    loopS (λxs → populationMerge 100 xs
                 (chunk 26 ∘    -- mu
                 mutate ∘ concat ∘
                 doMany 13    -- half mu, because crossover
                              -- yields 2 new solutions
                 (crossover ∘ doMany 2 select) ∘
                 map (makeDistPop dist) ∘
                 chunk popSize $ xs)
      ) $ starts
dist = take 99 (iterate (∗1.0165) 0.2) ++ [1]
```

Figure 9.5: An implementation of the Genetic Algorithm search strategy in Haskell, following as closely as possible the search logic used by the Opt4J implementation.

The Haskell version is complicated, requiring the usage of *concat*, *chunk* and the use of specific numeric parameters, linked to the parameter *mu* in the Opt4J version, to manage the streams correctly. The number of solutions generated is high, and this results in space leaks unless the *push* combinator is in use. However it does specify the selection methods that are being used, where they are being used, the probability distributions that are controlling the selection operations, and the location of crossover and mutation at the top level of the program, which is not the case in the Opt4J version. It should also be noted that much of the complexity introduced has been done to mimic the assumptions made about the Opt4J algorithm being used, and does not obviously aid in the solving of the TSP problem itself.

9.2.3 Performance Comparison

Each of these implementations was built, with varying values for population size, and number of children, these two parameters being the most easily set in Opt4J. Each implementation was timed using the standard Linux *time* program and the results can be seen in Table 9.2.

	Average Score			Average Time		
Parameters	a	b	c	a	b	c
Opt4J	210000	261000	262000	21s	43s	43s
Haskell	255000	254000	223000	188s	190s	379s

- a : $p = 100$ and $\mu = 26$
- b : $p = 200$ and $\mu = 26$
- c : $p = 200$ and $\mu = 52$

Table 9.2: A runtime comparison of a Genetic Algorithm for TSP fl417, in Opt4J and Haskell. Each has been computed with varying population sizes (p) and number of parents per generation (μ). Each test was run 10 times, and the average, to 3sf is presented here. Times are presented to the nearest second.

As seen in the results for ParadiseEO, the quality of the solutions produced by the Haskell and the Opt4J versions are not the same, indicating that these are not obeying the same search logic. The number of solutions being examined is presumed to be the same in each version, however the precise number in Opt4J is hidden at the top level.

The results are not fully understood, in that the increase in population size and number of children does not improve the quality of the solutions being found in the Opt4J version. The Haskell version offers more understandable results, with a significant increase in quality with the number of children, population size and hence number of solutions examined during the run of the program.

The timing data suggests that the Haskell version is 10x slower than the Java version. This is believed to be caused by the approach to mutation that is being used, which requires the threading of three Streams of data in an asynchronous pattern, and will hence be difficult for the Haskell compiler to optimise.

It should be noted that if, rather than trying to mimic the operations of Opt4J, the standard template for genetic algorithms in Chapter 5 is used for the TSP then the quality of results found is superior to those in Table 9.2. The runtime for this

program is not 10x slower than Java, but only 1.5x slower.

9.3 The meaning of the results

While the logic of the search strategies used in this implementation of Simulated Annealing and Genetic Algorithms has obviously not been replicated (as seen in the poor correlation of the results), some conclusions can be drawn. The times of the algorithms in each case study can be compared because it is believed that the same number of solutions are being evaluated, and that correcting the logic would not change this number of evaluations. In the case of the Simulated Annealing implementation, one solution was generated each iteration, and the number of iterations was controlled by the cooling strategy, as previously stated. The number of solutions that were expected to be evaluated was calculated and used in the Haskell implementation. In the case of the Genetic algorithm the number of solutions was expected to be two children per pair of parents, with μ giving the number of parents in a test case. Hence the number of solutions to be evaluated was $\mu * \texttt{generations} + \texttt{initial population}$, In both cases the number of solutions that are evaluated by ParadiseEO and Opt4J are not explicitly available and must be inferred from examination of the source code of each library system.

Two weeks was spent trying to understand the algorithms being used, in order to replicate their search logic. However it is not apparent from the implementations of TSP, nor the top level harnesses what the algorithms are doing. To understand this an investigation of the libraries, their documentation and their source code was carried out, to try to understand the search logic within the class hierarchies. The results from the tests indicate that this was not successful, and this supports the criticisms of the current approaches made in [56].

The results for ParadiseEO suggest that this library is outperforming C++. This is not thought to be correct, in that if the specific algorithm being used was implemented in C/C++, with no thought to code reuse or generic components, then the C version would make use of inplace modification of data structures and thus provide quicker iteration times. However such an approach would lose the advantages of the general library, which allows for more rapid experimentation with other algorithms, from a generic TSP model. So these results are comparing the general libraries in the two systems, and seem to support the use of Haskell for experimentation and investigation of possible algorithms for solving these hard

combinatorial problems.

9.4 Summary

This chapter has discussed two existing frameworks for metaheuristic algorithms, written in Java and C++ respectively. The top level implementation of a solver for TSP has been shown in each, with an equivalent written using the Haskell framework. In each case the Haskell framework is seen to be approximately equivalent in length, but providing a clearer description and greater access to the details of the search logic.

Each version has been tested with three sets of parameters, to demonstrate both runtimes and the quality of solutions found. In each case it has shown that the search logic of the C++ and Java versions are not fully understood. This throws the timing data into doubt, while lending support to the use of the Haskell library, which makes the logic more available at the top level of the program.

Chapter 10

Case Study : Homology analysis in computational biology

Biologists studying proteins are interested in knowing what these chemicals do within cells. However to fully characterise the functions of a particular protein is a complex task in the lab, and hence biologists are eager to know where to focus their efforts.

Computational biologists seek to provide such guidance through a process known as homology detection [18], the analysis of characteristics in organic systems that share a common ancestor. When applied to proteins if it can be shown that one protein shares common characteristics, with one or a family of well understood proteins, then this suggests that its function should be related to the functions of that group.

Homology detection can be done through matching of sequence patterns within DNA, or sequences of amino acids. When dealing with a family of proteins, they can be used to train a Hidden Markov Model (HMM). This may then be used to give a probability that, given the next amino acid in a particular query sequence, with respect to the ones that have been given before, it is likely to be in the same family as those used to construct the model.

MRFy [14], a homology detection program written in Haskell, augments this model to take into account *beta-strands*. Beta-strands are interactions in a folded protein, based upon hydrogen interactions between specific amino acids. These

interactions are stronger than relationships between unbonded amino-acids, and can aid in the detection of homology.

MRFy implements a new algorithm, which breaks up the HMM for a particular family around the known beta-strands and provides a number of different smaller HMMs. A new protein is represented as a query string of the sequence of amino-acids that make it up. This query string is then broken up around the beta-strands, with each sub-sequence fed to a respective HMM, which gives the likelihood of a match. However the exact position of the beta-strands is not known and, within some bounds, they can lie anywhere within the query string. Once they are positioned, the likelihood of their positioning can be computed, and coupled with the output of the HMMs to give a final score (representing the likely homology) for the protein being tested.

Because the position of the beta-strands is not known and they can be moved, completing the detection of homology requires a search of their possible positions. The model is described as *beads on a wire*, where each bead is one beta-strand, which can be shifted left and right. Bead positions are all integer values, as these represent interactions with discrete amino-acids. The bead positions must be within a range from 0 to some upper limit provided by the problem data. In addition each bead must be a specific distance from the beads to its left and right on the wire, with this information provided by the problem data.

The pricing of a particular assignment, through the evaluation of the different HMM's, can only be performed once every bead has been positioned, because the position of the last bead can impact how the query sequence is broken up. This precludes the evaluation of partial solutions and hence precludes the use of exhaustive methods such as branch & bound or dynamic programming for this task.

Instead metaheuristics were chosen to provide the search component. A number of methods were tried including Iterative Improvers and Simulated Annealing, however genetic algorithms were found to be particularly effective [15].

This case study will:

1. examine MRFy's in-house approach to constructing the metaheuristics;

2. describe their conversion to the stream based library;

3. demonstrate that the converted code admits a greater number of metaheuristic methods while improving readability and shortening the code length;

4. and design a metaheuristic which will produce better results than the existing system in less computational time.

10.1 Problem instances and test environment

The MRFy development team provided three problem instances for experimentation and testing of the various metaheuristics in this case study. These were selected to be reasonably representative of the range and scale of the problems that MRFy would encounter.

barwin the smallest problem in terms of the number of beads, and correspondingly a rapid evaluation time for each solution. However each bead can be positioned in a wider range of locations increasing the number of potential solutions to the problem. This problem has 7 beads to position over a range of between 0 and 345.

sandwich was the intermediate problem instance. This problem has 17 beads to position over a range of between 0 and 235.

"8" the largest of the problems, characterised by a large number of quite small HMMs during the evaluation phase. This problem has 40 beads to position over a range of between 0 and 592.

Unless otherwise stated, all results in this Chapter and associated timing data have been computed on an HP netbook with 1GB of RAM and an Intel Atom N550 dual core processor with 4 logical processors each clocked at 1.5GHz, however only a single core is used for the tests.

10.2 Implementation of Metaheuristics in MRFy

The original approach taken to implementing metaheuristics in MRFy was to focus upon the stochastic nature that most metaheuristics exhibit. This was then formalised using a monadic approach, where the random number generator was a piece of state data threaded through the computation.

10.2.1 Elements of a search strategy

Using the original MRFy code the design of a new metaheuristic involves providing the following functions.

- A monadic computation for generating stochastic solutions to the problem, to be used as the initial solutions for the metaheuristic search.

- A step function which is parameterised over either populations or single solutions. This function is of the form

  $s \rightarrow Rand\ s$

  and can hence be chained together using the binding function of the monadic instance of random number threading.

- A *utility* function. This can either be identical to the solution's score, or transform the result to incorporate other information such as time. In this way it allows the use of time dependent pricing functions. It is of the form:

  $Move\ s \rightarrow Rand\ (Utility\ s)$.

- The stopping criterion for the strategy, which is encoded as a transformation from a list of solutions to a list of solutions.

- The final output function which allows for the transition from populations to single solutions. It is presumed that this will yield the *best* solution in any given population.

Each search strategy is implemented through a function that creates an instance of the data structure described, from information about the problem and a set of *search parameters*. The search parameters are provided in a data structure of named fields including every parameter used by every metaheuristic that has been implemented, for example population size for GAs and cooling rates for Simulated Annealing.

Final execution of the search strategy is provided through the conversion of the step function into a monadic computation from the random seed into a stream of solutions. This computation is then composed with termination criteria and extraction of the final result from the computation.

10.2.2 Low level operators

Two operators were used for all the metaheuristic algorithms that were provided with MRFy, one for perturbation and one for crossover. In each case the operators acted over a *Placement* data type, that is a type synonym for $[Int]$, and as such must be *priced* separately.

The stochastic perturbation operation works by iteratively moving each bead in the solution to a new value within its legal range, given the placement of the other beads in the solution. The following is the data type of the operation to ease understanding of how the function is used, but the body of the code will not be presented.

$$
\begin{aligned}
&\mathit{randomizePlacement} \\
&\quad :: \mathit{RandomGen\ r} \\
&\quad \Rightarrow [\mathit{BetaStrand}] \\
&\quad \rightarrow \mathit{Placement} \\
&\quad \rightarrow \mathit{Int} \\
&\quad \rightarrow \mathit{Rand\ r\ Placement} \\
&\mathit{randomizePlacement\ betas\ oldp\ maxRight} \ldots
\end{aligned}
$$

Usage requires the problem specific information relating to the beta-strands and the maximum right hand value that could be taken by the last bead.

Recombination of solution in the original implementation of GAs was provided by a function called *crossover* with the following type. Again the body of this function has been omitted.

$$\mathit{crossover} :: \mathit{Placement} \rightarrow \mathit{Placement} \rightarrow \mathit{Placement}$$

The logical process used was to pair the values from each solution and then in each cycle compare the first and last pair currently available. Of the first pair the lowest would be kept, of the final pair the highest. The process was then recursed over the remainder of the pairs. Once this pairing and selection was complete the resulting list was sorted to yield a new and valid placement of the beads.

10.2.3 Implemented Metaheuristics

Using this monadic framework, 3 metaheuristic algorithms were implemented, and are described here. Each algorithm has been tested with settings suggested by Tufts, and the results shown in Table 10.1.

Stochastic Hill Climber was implemented as a series of stochastic perturbations of a solution, where the perturbation was only accepted if it improved upon the seed.

Simulated Annealing was implemented by threading an additional *cost* value attached to each solution, which was used in the place of time in the cooling strategy. The cooling strategy was the standard geometric method.

Genetic Algorithms were implemented by using the stepwise update, to modify a population rather than one solution at a time. The algorithm implemented took the population, sorted it into improving order, then paired the 1^{st} value with the 2^{nd}, the 3^{rd} with the 4^{th} and so on. Each pair was used to generate a single new solution through crossover. Every solution so generated was then perturbed using the random mutation operation. Finally the new solutions were merged with the old population, and the entire list was resorted into improving order, and the new population was the same size as the old population, consisting of the best solutions from the unified group.

Using these algorithms Tufts have given the best known results as being;

Barwin	Sandwich	8
1017	771	2079

	SA		GA		SHC	
	value	time	value	time	value	time
barwin	1058	12m	1071	81m	1069	19m
sandwich	841	9m	891	79m	897	43m
8	2142	11m	2121	44m	2134	25m

Table 10.1: The value and time results of the original version of MRFy. Each algorithm was sampled 10 times with a time limit of 2 hours on the total runtime. Where the time limit was exceeded the result was discarded, but the time was kept as 2 hours. Barwin and Sandwich are the averages of 6 successful samples, while 8 is based upon 9. All results are rounded to the nearest whole number.

10.3 Stream Model

In converting the MRFy algorithms to the stream model, the aim was to implement the search logic as accurately as possible, but to demonstrate that the code is shorter and simpler using the stream based framework. Some modification of the solution representations was needed, as was the adaptation of the perturbation and recombination functions to fit with the new approach.

10.3.1 Representing the problem

To represent a solution the following data type was introduced, encapsulating;

- the placement of the beads as a list of integers,
- the scoring algorithm itself,
- and the score of the current placement.

This is represented using the following data type;

$$\textbf{data}\ \mathit{PricedSol} = \mathit{PricedSol}\ ([\mathit{Int}] \rightarrow \mathit{Double})\ [\mathit{Int}]\ \mathit{Double}$$

This data type is made part of *Eq*, comparing the list of integers (the placement of beads), and part of *Optimisable* comparing the *Double* valued price.

The perturbation and recombination functions are rewritten to operate over *PricesSol* and correctly maintain the reference to the pricing function of the system.

10.3.2 Perturbation and Recombination

The operations that applied perturbation to solutions were originally implemented as self contained monolithic functions. Due to the restriction that only a completed placement can be priced, the concept of damage and repair would both be restricted to purely stochastic methods, and hence is not particularly useful. However to enable experimentation upon the operators of the problem the following underlying computation was identified, which provides a way to modify a solution through the movement of one bead a specific distance left or right. Solutions are guaranteed to be valid.

```
detPerturb :: QuerySequence →
              [BetaStrand] →
              Int →            -- bead to move
              Int →            -- change position by
              PricedSol →      -- the solution to change
              PricedSol
detPerturb qs bs p v s
  = let (as, c : cs) = splitAt p $ solution s
        as' = concat [as, [c + v], cs]
```

```
    in if checkPlacement qs bs as'
       then s {solution = as', underlyingScore = pricer s as'}
       else s
```

The *detPerturb* function makes use of an additional function *checkPlacement* which is provided in Appendix F.3. This function validates the new placement against the *QuerySequence* and *BetaStrand* information of the problem.

When lifted to operate over streams *detMutate* does not predefine the sources of the information related to which bead to move, nor which way to move it nor how far. This allowed it to be the basis for all subsequent perturbation operations that will be seen in this Chapter. The original method of randomly selecting one bead and moving it a random distance was implemented as follows:

```
stochasticPerturb :: RandomGen g =>
                     QuerySequence ->
                     [BetaStrand] ->
                     g ->
                     Stream PricedSol ->
                     Stream PricedSol
stochasticPerturb q b g
  = uniformChoice g1
  . zipWith (\r -> checkAllOptions (detPerturb q b r))
  . randomRs (0, length b - 1) $ g2
  where (g1, g2) = split g
```

The *stochasticMutate* function makes use of a further function which is provided in Appendix F.3, named *checkAllOptions* which yields all possible solutions where this bead is moved, or if no legal options exist, the original solution as a singleton list.

The recombination method was preserved in its original form, though updated to operate over the *PricedSol* data type.

```
recombine :: PricedSol -> PricedSol -> PricedSol
recombine ...
```

10.3.3 Constructing Random Solutions

The approach taken to constructing random placements was to generate a set of integers within the range $(0, maxRight)$, where *maxRight* is a problem specific

variable. Once sorted most lists generated in this fashion are valid placements for the problem instance. The removal of invalid sequences can be achieved with an auxiliary function *invalidPlacement*[1], against which the stream of potential placements is filtered.

$$
\begin{aligned}
&\mathit{generateSolutions} :: \mathit{RandomGen}\ g \\
&\qquad \Rightarrow \mathit{QuerySequence} \rightarrow [\mathit{BetaStrand}] \rightarrow \mathit{Int} \\
&\qquad \rightarrow g \\
&\qquad \rightarrow \mathit{Stream}\ [\mathit{Int}] \\
&\mathit{generateSolutions\ query\ betas\ maxRight} \\
&\quad = \mathit{filter}\ (\neg \circ \mathit{invalidPlacement\ query\ betas\ maxRight}) \\
&\quad \circ\ (\mathit{map\ sort}) \\
&\quad \circ\ \mathit{chunk}\ (\mathit{length\ betas}) \\
&\quad \circ\ \mathit{randomRs}\ (0, \mathit{maxRight})
\end{aligned}
$$

The pricing of these placements and conversion into a stream of priced solutions can be performed separately.

10.3.4 Termination Criteria

MRFy makes use of convergence checking on the qualities of the solutions that are encountered, terminating the process when solution quality has not improved for a given number of iterations. To implement this the following functions was created, for application to a stream of always improving solutions.

$$
\begin{aligned}
&\mathit{convergenceCheck} :: \mathit{Eq}\ a \Rightarrow \mathit{Int} \rightarrow \mathit{Stream}\ a \rightarrow \mathit{Stream}\ a \\
&\mathit{convergenceCheck\ width\ as} \\
&\quad = \mathit{map\ snd} \circ \mathit{takeWhile}\ (\neg \circ \mathit{fst}) \\
&\quad \circ\ \ \mathit{map}\ (\lambda(a,b) \rightarrow (\mathit{length}\ a > 2 \wedge (\mathit{last}\ a \equiv \mathit{head}\ a), b)) \\
&\quad \$\, \mathit{zip}\ (\mathit{window\ width\ as})\ \mathit{as}
\end{aligned}
$$

10.3.5 Comparing algorithm implementation

The simplest search strategy, stochastic iterative improver, is used to illustrate the transition from the original MRFy implementation to the stream based implementation. Further code for implementing Genetic Algorithms and Simulated

[1] The source code for this function is complex and problem specific, and will not be provided.

Annealing, both the original version from MRFy and the Stream implementations can be found in Appendix F.

The construction of the data structure that represents a stochastic iterative improver in the original MRFy code is shown below.

```
nss :: NewSS
nss hmm searchP query betas scorer = fullSearchStrategy
  (fmap scorer $ initialize hmm searchP query betas)
  (mutate searchP query betas scorer)
  scoreUtility
  (takeByCCostGap (acceptableCCostGap searchP))
  id
```

What follows is a version of a stochastic iterative improver, mimicking the search logic used by the MRFy version, but using the stream based framework.

```
stochasticII :: (Stream (PricedSol [Int]) → Stream (PricedSol [Int]))
             → Int
             → PricedSol [Int]
             → Stream (PricedSol [Int])
stochasticII perturb convergeLimit
  = convergenceCheck convergeLimit ∘ keepBest
  ∘ loopP (map head ∘ improvement (doMany 1 perturb))
```

The stream method is not a direct translation of the MRFy version of the code. For example while MRFy includes references to the termination criteria in this code, it must be processed by a complex harness before final evaluation. By comparison, while the stream implementation requires a perturbation method to be specified, and a seed solution provided, it is otherwise complete. In general the stream based method contains simpler expressions providing a clearer understanding of the search logic which is being used, specifically that this is improvement over a singleton neighbourhood.

Both these examples need harnesses to run correctly, providing the problem specific data, random number generators and gathering the results. The Stream implementation can be placed into a larger program using the following line, creating a mutator from the parameterisation of *stochasticPerturb*. In this example convergence width is set to 200, a value used in the MRFy testing of the last section for the iterative improver and simulated annealing strategies.

```
stochasticII (stochasticPerturb query betas g)
             200 initialSol
```

10.4 New Strategies

10.4.1 Iterative Improvers

Modification of the initial stream based iterative improver to make use of other possible characteristics of the algorithm family was straight forward. Neighbourhood size can be changed directly in the code, or extracted as a parameter set by the user at runtime.

To allow greater variation in the search process it is also possible to remove the taking of the first improving solution, seen in the original version, and replace with simply taking the best at each step, even if it does not improve.

$$loopP\ (map\ best \circ doMany\ neighbourhoodSize\ stochasticPerturb)$$

The other alternative that was tried was to use a different neighbourhood with more predictable properties than the stochastic neighbourhoods so far considered. The approach taken was to create a cyclical perturbation following a regular pattern, and then exhaustively generating all possible solutions from each *seed*.

$$\begin{aligned}
&cyclicalPerturb\ q\ b = zipWith3\ (detPerturb\ q\ b)\\
&\quad (cycle\ (stretch\ 4\ [0\,..\,length\ b-1]))\\
&\quad (cycle\ [1,-1,2,-2])\\
&cyclicNeighbourhood\ q\ b = doMany\ (4*length\ b)\ (cyclicalPerturb\ q\ b)
\end{aligned}$$

10.4.2 TABU search

TABU search was implemented through direct application of the library function using a basic TABU memory and a stochastic neighbourhood function as seen in iterative improvers. The size of the memory and the size of the neighbourhood were selected to be one apart so that at maximum usage the TABU list would always leave one option open for movement.

Several memory sizes were tried out, however none proved significantly effective. This is believed to be caused by the very large number of possible solutions

that can appear from perturbing any given solution, rendering TABU lists of neighbourhoods generated in this way useless. The TABU lists cannot capture enough of the possible neighbourhoods to ever prune the neighbourhoods in a meaningful way, unless very large TABU lists are used, which are themselves impractical.

10.4.3 ACO

ACO had also not previously been tried in MRFy, so it was a useful candidate as a new strategy. The objective of this algorithm here is to *learn*, through pheromone build up, how to *build* good solutions to the problem. To try to achieve this, the approach taken was to associate a probability distribution with each bead in a solution and generate bead positions against these distributions.

The distribution for each bead was created through the *blending* of a number of distributions. The base distribution was a uniform distribution. Once a solution was created (generated by an *ant*), it was inversely weighted against known solution qualities, and a number of *normal* distributions were created, where the mean was the position of the bead in the generating solution. The standard deviation was based upon the *weighting* of the solution, and the entire distribution was then scaled by the weight so that a better solution would provide more dominance to the generation of new solutions. The code for this ACO was stream based, but reasonably complex and will not be presented here.

In order to improve the quality of solutions used for the ACO recombination more quickly, the strategy was hybridised with an iterative improver, using the structured neighbourhood, and restart functionality seen in previous chapters.

10.4.4 Summary of experimentation

What has been seen is that the implementation of these new variations and strategies is simple with the stream library, including entirely new methods such as TABU search. The implementation of the ACO was harder, with problem specific methods for pheromone recombination and evaluation being needed, however subsequent hybridisation and experimentation with ACO was similarly simple.

In this section there has been no discussion of how effective, in terms of time and solution quality, these algorithms were. What was found was that all the strategies tended to stagnate quickly, and not be significantly stronger than the existing MRFy strategies. The ACO and iterative improvement hybrid was the

strongest, acting more consistently than the existing algorithms in terms of the predictability of the qualities of the solutions that would be found.

Rather than allowing them to run for long periods as was done with the MRFy strategies previously, they were given a 10 minute limit, as a pragmatic trade off between reasonable evaluation of the strategies and practical experimental time constraints. This was provided by the following new termination operation.

```
takeFor :: (NFData a, Integral target) ⇒ target → [a] → IO [a]
takeFor targetTime (v : vs)
  = do endTime ← getCurrentTime >>= return ∘ floor
                                  ∘ toRational ∘ utctDayTime
    if endTime ⩾ targetTime
      then return [ ]
      else (return ∘ (v:))          =<< ((v `using` rdeepseq)
                                    `seq` takeFor targetTime vs)
```

10.5 Examining the effects of the operators

The neighbourhood based Iterative Improvers and the construction based upon the learning ACO were reasonably successful, in some cases generating results equivalent, or slightly better, than the best that had been previously reported for these problems. A cursory examination of these superior results yielded a useful insight into the problem, that high quality solutions would often share *blocks* of beads, but offset differently in different solutions. Figure 10.1 illustrates a number of solutions with widely varying solution qualities.

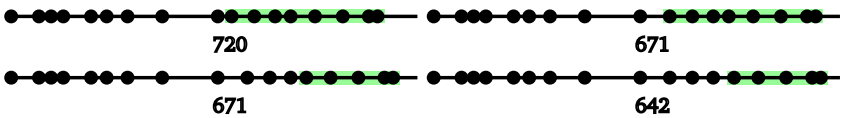


Figure 10.1: Block similarity, with quite different solution qualities. *sandwich* problem file.

The existing perturbation methods for the metaheuristics operated by modifying only one bead at a time. However if the beads needed to move in blocks, it would be expected that moving only one bead would be more likely to break the internal pattern of a block and hence lead to worse solutions, as seen in Figure 10.2.

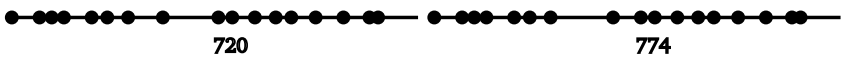


Figure 10.2: Example of movement of a single bead within a solution. *sandwich* problem file.

Consideration of this characteristic by experts at Tufts revealed that the way a placement pattern was broken down into inputs for the HMMs would cause this concept of blocks that can be internally optimised and then moved. This adds support for the concept that arbitrary movement of single beads would be unlikely to help.

10.6 Building a new operator

To exploit the idea that it was often useful to move not just single beads but blocks of beads simultaneously a block perturbation operation was created through an alternative lifting of the *detPerturb* function. An auxiliary function *blockOptions* was created, that would take a block defined by a pair of numbers and returned the possible solutions that would result from moving the block. The function *blockOptionsS* then lifts the *blockOptions* function to operate over streams, using a random number generator to provide a stream of blocks.

```
blockOptions :: Eq a ⇒
                (Int → Int → PricesSol → PricedSol) →
                (Int, Int) →
                PricedSol →
                List PricedSol
blockOptions f (lower, upper)
  = checkAllOptions g    -- generate every possible movement of a block
  where                  -- define how to move this block (g)
    g offset sol = foldl (λs i → f i offset s) sol blockIndices
      where blockIndices = if offset < 0 then [lower..upper]
                                         else reverse [lower..upper]

blockOptionsS :: RandomGen g ⇒ QuerySequence →
                                [BetaStrand] →
                                g →
                                Stream (PricedSol) →
                                Stream (List PricedSol)
```

$$
\begin{aligned}
&\mathit{blockOptionsS}\ \mathit{qs}\ \mathit{betas}\ g = \mathit{zipWith}\ \mathit{perturb}\ \mathit{pairs}\\
&\quad \textbf{where}\ \mathit{pairs} = \mathit{map}\ (\lambda[a,b] \to (a,b))\\
&\qquad\qquad \circ\ \mathit{map}\ \mathit{sort}\\
&\qquad\qquad \circ\ \mathit{filter}\ (\lambda[a,b] \to a \not\equiv b)\\
&\qquad\qquad \circ\ \mathit{chunk}\ 2\\
&\qquad\qquad \circ\ \mathit{randomRs}\ (0, \mathit{length}\ \mathit{betas} - 1)\ \$\ g\\
&\qquad \mathit{perturb} = \mathit{blockOptions}\ (\mathit{detMutate}\ \mathit{qs}\ \mathit{betas})
\end{aligned}
$$

The *blockOptions* function is then composed with selection methods to create a perturbation function using block motion as the underlying operation. Typically the number of possible options from any particular selected block will be too high to evaluate all solutions, so to allow for speed a uniform selection method was chosen and is seen in *blockPerturb*.

$$
\begin{aligned}
&\mathit{blockPerturb}\ \mathit{qs}\ \mathit{bs}\ g = \mathit{uniformChoice}\ \mathit{g1} \circ\\
&\qquad\qquad\qquad\qquad \mathit{blockOptionsS}\ \mathit{qs}\ \mathit{bs}\ \mathit{g2}\\
&\quad \textbf{where}\ (\mathit{g1}, \mathit{g2}) = \mathit{split}\ g
\end{aligned}
$$

10.7 Developing the strategy

The *blockPerturb* operation may be used alone, or as a new operation composed into a larger metaheuristic. This was done, composing it with the existing iterated maximal improvement method, from which better solutions were successfully generated. It was found that trying a small number of block perturbations from each solution and selecting the best provided further improvements. This final lifting is seen in *blockPerturb′*, which generates a neighbourhood using *doMany* in the standard way. The size of the neighbourhood (10) was selected after a number of trials and is a reasonable balance of range of solutions generated and practical execution speed.

$$
\begin{aligned}
&\mathit{blockPerturb'}\ \mathit{qs}\ \mathit{bs}\ g = \mathit{map}\ \mathit{minimum} \circ\\
&\qquad\qquad\qquad\qquad \mathit{doMany}\ 10\ (\mathit{blockPerturb}\ \mathit{qs}\ \mathit{bs}\ g)
\end{aligned}
$$

10.7.1 Memory and Backtracking

It was noted in the previous experiments that the search strategies would often move away from promising solutions too eagerly. To avoid this a new *back tracking* transformation was introduced, that could be composed into a search strategy

as needed. This function builds up a collection of solutions to a predefined limit and then replaces the current solution with the best solution seen within the preceding collection. It is implemented in terms of chunking and mapping as a simple stream transformation.

$$
\begin{aligned}
&\mathit{backTrack} :: \mathit{Ord}\ a \Rightarrow \mathit{Int} \rightarrow \mathit{Stream}\ a \rightarrow \mathit{Stream}\ a \\
&\mathit{backTrack}\ n = \mathit{concatMap}\ (\lambda a \rightarrow \mathit{take}\ (n-1)\ a +\!\!+ [\mathit{minimum}\ a]) \\
&\qquad\qquad \circ\ \mathit{chunk}\ n
\end{aligned}
$$

10.7.2 The Final Version

The final version of the metaheuristic makes use of the most successful features seen in this study, combining:

- the iterated improvement method using a local neighbourhood search;
- the block perturbation method, used to created a manageable neighbourhood and subsequently explored for the best solution within it;
- and the back tracking feature to allow the system to focus upon useful avenues of exploration.

To avoid recomputation of the quality of specific placements, memoization of the pricing function was introduced. This used Memo-Trie.

$$
\begin{aligned}
&\mathit{twolevelIterativeImprovement}\ \mathit{query}\ \mathit{header} \\
&\quad = \mathbf{do}\ \mathit{rng} \leftarrow \mathit{newStdGen} \\
&\qquad \mathit{intialSols} \leftarrow \mathit{newStdGen} \gg\!= \mathit{return} \\
&\qquad\qquad\qquad\qquad \circ\ \mathit{map}\ (\mathit{mkPricing}\ \mathit{scorer}) \\
&\qquad\qquad\qquad\qquad \circ\ (\lambda g \rightarrow \mathit{basicGuesser}\ g \\
&\qquad\qquad\qquad\qquad\qquad \mathit{query}\ (\mathit{betas}\ \mathit{header})) \\
&\qquad \mathit{return} \circ \mathit{loopP}\ \$ \\
&\qquad\quad (\ \mathit{backTrack}\ 15 \\
&\qquad\quad \circ \mathit{localsearch} \\
&\qquad\quad \circ \mathit{blockPerturb}'\ \mathit{query}\ (\mathit{betas}\ \mathit{header})\ \mathit{rng} \\
&\qquad\quad)\ (\mathit{head}\ \mathit{initialSols})
\end{aligned}
$$

The *backTrack* parameter had to be chosen so as to allow reasonable exploration from any given solution, but would allow backtracking quickly enough so as to

avoid unproductive avenues. Given the slow speed of the pricing computation it was known that only hundreds of solutions could be examined in practical time limits, and 15 was chosen as a balance between these factors on a somewhat trial and error basis.

This new metaheuristic was then tested, continuing with the 10 minute time limit on the search, on a single core. For each of the three problem files the program was run 15 times and the results of these tests can be seen in Table 10.2. The high degree of reproducibility in the algorithm and the high quality of the results relative to those seen previously, in comparatively limited time, are all significant evidence that this metaheuristic is superior to the originals. Problem *"8"* has the highest variance, caused by the slow rate of evaluation for the largest problem, indicating that for problems in this class more computational time will probably be needed for the system to achieve similar levels of stability as for the smaller problems.

	min	max	average	Tufts Original Best
barwin	978	983	979	1017
sandwich	554	554	554	771
8	1905	1981	1946	2079

Table 10.2: Quality results of the final metaheuristic, created for MRFy. It should be noted that the original version of the code provided by Tufts is effectively running in unlimited time, but with a convergence check, while the new metaheuristic is run with a 10 minute time limit.

10.8 Library enhancement: parallel processing

The original versions of the MRFy program had been implemented taking advantage of Haskell's libraries for parallel processing, and it was natural to seek to allow the new metaheuristic to do the same. While parallel processing of search strategies has not been the focus of this Thesis, work on this metaheuristic for MRFy has provided a useful opportunity to explore extending the basic library.

The metaheuristic that has been presented makes use of the *doMany* function in 2 places to provide neighbourhoods of solutions. The evaluation of the qualities of the solutions in these neighbourhoods is known to be required due to the selection of the *best* element from them, which can only be done when all qualities are known. The process of evaluation of a placement to a quality value is highly

costly, but effectively a map operation and so inherently parallel.

This combination of inherent parallel processing and high cost of individual operations makes the generation of priced neighbourhoods an ideal location for a parallel operation. This is provided using an augmentation of the basic *doMany* operation called *parDoMany*, causing the solutions contained in each neighbourhood to be evaluated in parallel with one another, making use of the standard *Control.Parallel* libraries.

$$
\begin{aligned}
&\mathit{parDoMany} :: \mathit{NFData}\ b \Rightarrow \\
&\qquad \mathit{Int} \rightarrow \\
&\qquad (\mathit{Stream}\ a \rightarrow \mathit{Stream}\ b) \rightarrow \\
&\qquad \mathit{Stream}\ a \rightarrow \mathit{Stream}\ (\mathit{List}\ b) \\
&\mathit{parDoMany}\ n\ f = \mathit{map}\ \mathit{parChunks} \circ \mathit{doMany}\ n\ f \\
&\quad \textbf{where}\ \mathit{parChunks}\ x = x\ \text{`}\mathit{using}\text{`}\ \mathit{parList}\ \mathit{rdeepseq}
\end{aligned}
$$

The quantity of the code that required replacement was only 2 functions, specifically the use of *doMany* in the construction of neighbourhoods. This allowed the parallelisation of a significant number of operations, for example the neighbourhood size for the iterative improver, using the deterministic neighbourhood was $4n$ (e.g. 160 elements in problem "8"). These neighbourhoods could now be evaluated in parallel, although the use of memoization would impact upon the productivity of this parallelism, where a rapid look up might even cost more than in a serial implementation. However the productivity of the program did increase, examining twice as many solutions in an equivalent time when run on 3 processors rather than 1.

This demonstrates both the ease of extending the library with new operations and the ease with which this stream based approach can make use of parallel evaluation in reasonably non-specialised ways. By comparison, extending a sealed system like Opt4J is difficult and would require modifying the underlying library or rewriting significant parts of the object hierarchy. These advantages are well known generally in Haskell, and it is gratifying to be able to provide them in this library for metaheuristics.

10.9 Conclusion

This case study demonstrates the practical use of the library of stream based metaheuristic combinators presented in this Thesis, for exploring the design space of

search strategies and gaining greater insight into the problem. It has shown the process of creating a simple self contained operator for a problem (the *detPerturb* function) and then lifting and transforming it in a variety of ways, to yield a number of perturbation and neighbourhood functions; as well as how this enables rapid experimentation with these operators.

While the library itself does not *solve* the optimisation problem, the provision of a flexible framework and the ability to easily experiment with new algorithms enables a rapid exploration of the design space. Shorter and clearer code allows the underlying operation of the existing algorithms to be understood and manipulated. The rapid experimentation and the lack of progress with the existing operators and strategies emphasised the need for alternative approaches to the problem. This also serves to again illustrate the need for flexibility in experimenting with metaheuristic algorithms for combinatorial optimisation problems.

An examination of the solutions being produced suggested a new operator, which was implemented through the elaboration of the simplest underlying operator that was already available. The use of the new operator, in concert with existing elements that had been found to have some beneficial effect, then provided a new algorithm that surpassed the original search methods used for MRFy, using less time and providing higher quality results with a high degree of repeatability.

Further work can be done on the metaheuristic for this problem, examining further the block motion characteristic of the solutions and trying to identify for a particular problem instance:

- the number and size of blocks;
- the indices of blocks;
- and better ways to explore the neighbourhoods created through block motion.

Chapter 11

Conclusion

This Thesis has considered the use of metaheuristics in pure functional languages, specifically Haskell, and how to create a library of general purpose combinators to allow their implementation and hybridisation. The use of functional languages has highlighted a new approach to implementing metaheuristics as a general class of algorithms as opposed to a loose collection of monolithically defined algorithms.

There has previously been some research on implementing the algorithms of operations research and tackling discrete combinatorial problems in pure functional languages, however there has not been an effort to create a single general purpose library for metaheuristics. It is hard to be sure why this is the case, however it can be hypothesised that it is related to:

- the natural construction of tree based exhaustive search methods in Haskell (such as DFS), as opposed to the problems that are encountered with complex state management and iteration required by meta heuristics;
- as opposed to the apparent lack of a significant improvement in functional theory to be gained from investigating metaheuristics; and
- a lack of applications being created in Haskell that wish to make use of metaheuristics methods, and so no incentive to work on this approach.

Despite is lack of structured effort on the topic of metaheuristics several implementations of algorithms such as Simulated Annealing and Genetic Algorithms

exist and metaheuristics are in practical use in projects in Haskell such as the work on Homology analysis in MRFy.

The hybridisation of metaheuristics has been a topic of some interest in the field of optimisation, resulting in templates for how to approach the task, analyses of existing approaches and the construction of a number of frameworks to aid the programmer in these tasks. However other researchers have pointed out that these frameworks tend to require advanced knowledge of both the language and the frameworks to make use of them, while not enabling a full range of hybridisation between all algorithms.

This Thesis has selected five major types of metaheuristic algorithm and created a single framework and library of combinators to allow the implementation and hybridisation of these algorithms. The algorithms were selected to be among the most common strategies in use and not be naturally subject to hybridisation in existing frameworks [56].

Several approaches to the implementation have been tried, each with difficulties which have been illustrated. The method that this Thesis has settled upon was the use of a stream-transformation approach to structure metaheursitics in a data-flow style. This has successfully separated the termination conditions, the analysis of the results and the common structures of the internal workings of the algorithms into finer grained components.

The library has been presented and the difficulty of implementing new algorithms and extensions to it have been compared with two existing frameworks. Finally the library was utilised to examine the optimisation problem encountered in MRFy, and allowed for rapid experimentation on various metaheurisitic options. This experimentation resulted in a new search strategy, utilising parallelism and superior perturbation methods to yield consistently superior results to the problem, while simultaneously having faster runtime than the existing search strategies.

11.1 Review

This thesis has successfully created a library of combinators for the pure functional language Haskell, which enable the implementation and hybridisation of all the metaheuristics that were selected at the outset. The power of Haskell in terms of rewrite-rules and parallelism has been examined to a more limited extent but been found to be of use in aiding in both performance and high level expression of metaheuristics.

The library has been demonstrated to be an effective tool for experimentation and algorithm design, enabling rapid testing of concepts and ideas to provide metaheuristic solutions to combinatorial problems.

The framework has been used in the design of a metaheuristic for a non-trivial real world problem in computational biological research, resulting in an effective algorithm for this problem. This has demonstrated that the stream based combinators are effective tools for rapid experimentation in the course of designing algorithms to solve combinatorial problems.

It has been seen that the use of this framework in Haskell can be competitive, in terms of runtime, with metaheuristics implemented in frameworks written in C++ or Java. There are still issues relating to performance, most noticable in the direct use of the program logic seen in Opt4J, which resulted in a significantly slower program written in Haskell. In general it is believed that there does remain an issue of *memory churn* in the Haskell framework, however this is partially related to Haskell itself, a high level language, that incurs costs for use of single assignment to names and garbage collection. Unfortunately the approach of the library further encourages this memory churn, with the flow of data through the streams, and the pure nature of Haskell means that old data structures must be deleted and new ones created, rather than memory being reused.

In some cases Haskell has been shown to get close to C through the use of sophisticated compilation techniques such as stream fusion [11]. Further developing and exploiting these techniques is a significant part of the proposed future work of this project, and it is hoped that this will further reduce the performance gap between Haskell implementations of metaheuristics and those built in other languages.

11.2 Domain Specific Languages for Metaheuristics

One approach which could have been taken to providing a suitable abstraction for implementing metaheuristic algorithms is through a Domain Specific Language (DSL). The creation of a DSL can involve the creation of an entire new programming language, tailored for the specific domain, with a related interpreter or compiler (this is the approach taken by the Comet [83] language). The advantages of building a DSL include the ability to closely match the symbols of the domain to the new language, and improved performance, due to being able to take into account domain specific rules [57]. However it is also pointed out that this solution

involves building a complete programming language, known to be a difficult task.

The alternative is an Embedded DSL (EDSL), where a general purpose language is used as the *host*, and in a sense any Appliation Programming Interface (API) is this already. The advantages of this approach are that they are often more extensible, through adding new components in the host language, however it can be harder to match the domain symbols and the expressions of the domain as closely as in a DSL. Performance can also be an issue, due to using a general purpose compiler, which does not have specific domain knowledge.

DSLs actually fall on more of a continuum, with these two as the extremes. One alternative that falls between them is an EDSL where the API constructs a data structure in the host language for compilation and execution later. An example of this approach is the Co-Pilot language in Haskell, which is an EDSL for stream processing, for creating monitoring software for system properties [63]).

This Thesis has put forward the approach of using streams and stream transformations for implementing metaheuristics, does it make sense to build a DSL for this task? It has been pointed out that Haskell is a good host language for EDSLs [38], allowing for easy extensibility by users, often allowing for a close match to the semantics of the domain, providing a powerful compiler and giving some access to this compiler through the use of rewrite rules.

The toolkit that this Thesis has presented is a DSL for the stream transformations needed to express a wide variety of metaheuristic algorithms, lightly embedded in Haskell. It allows a programmer to build new stream transformations in Haskell as the need arises, and then compose these with existing transformations, or parachute them deeper into the code through parameters to higher order functions, while not sacrificing performance. This extensibility is particularly important in the domain of metaheuristics which is still evolving as a field.

The modelling of combinatorial problems using Haskell has not been focused upon in this Thesis. While this does mean that the library does not restrict the types of problems that can be tackled, it also means that it does not provide a modelling language for problems, such as is found in Comet. This lack of a language for problems suggests that either a new EDSL (or several) for modelling problems is needed to go with this library, or it is an indication that a separate DSL is needed for this domain.

11.3 Future Work

11.3.1 Compiler Optimisations

The current implementation makes limited use of rewrite-rules in Haskell[1] to provide the capability to construct algorithms focusing upon the logic of the process rather than the runtime performance. The use of the standard Haskell *list* type for the representation of Streams does give access to built in fusion rules such as *map composition fusion*, however only the standard[2] operations are currently applied.

Given the data flow and stream based approach that the final library has exploited, a technique known as *stream-fusion* [11], could be adapted to improve performance by reducing the number of intermediate data structures required. The manual fusion that was performed to achieve results in Chapter 9 would be the expected starting place to attempt to improve the generic use of rewrite rules for fusion in this system. Other forms of fusion, and a more generic approach to fusion, can be seen here [34].

A specialised form of rewrite for selection methods when combined with *doMany* was also used in Chapter 9, and could be generalised. This option appeared because the repeated selection from a collection, based upon a single probability distribution could be improved through the creation of a tree based data structure with the probability distribution forming the keys. However, where only a single selection is taking place, this imposes an additional cost, rather than providing a performance boost. A study examining the use of rewrite rules to automatically choose between these options depending upon the situation (matching against the specific pattern of *stretch* and *chunk* or just against *doMany*) would be a valuable exploration of using the power of functional programming and compiler reasoning to aid in metaheuristic programming.

The use of a specialised data structure, selected through rewrite rules, to improve performance suggests a final direction for compiler optimisation work, that of automated optimisation of data structures. For example, the TSP will often be very fast in C if the metaheuristics use a city swapping method based upon arrays, rather than trees as used in this Thesis. Automated mappings to efficient low level data structures where appropriate could be an effective way to bring performance

[1] For more detail on rewrite-rules see Appendix C.6.

[2] The standard fusion rules in Haskell are based upon *foldr/build* fusion proposed in [25]. At present a cursory examination of Haskell core code after compilation rules have been applied suggests that few if any of these optimisations are currently being applied to metaheuristics written using this framework.

of this library closer to that of a pure C approach, while maintaining the flexibility of expression that Haskell provides.

11.3.2 Improved Support For Parallelism

Parallelism has been a key tool for tackling large instances of combinatorial problems using both exhaustive and metaheuristic search methods. The purity of Haskell has been leveraged to support parallelism in a variety of flavours, however the research is still ongoing. Improving the exploitation of parallelism by the library should be a key subject for future research.

The basic library of this Thesis has not focused upon exploiting parallelism, although it has been seen in Chapter 10 that extending the library for this purpose can be done easily using Haskell's parallel combinators. This extension of the metaheuristic library and its use on a real world problem was performed manually at this time, though it was simple to accomplish and had successful results.

A further simple extension to the system could take advantage of properties of operations such as *doMany* to automatically enable limited parallelism, with automated settings of chunking sizes for nested lists computed from the number of cores provided at runtime. This extension can be achieved using existing code libraries at runtime, or at compile time through meta-programming provided by Template Haskell [74].

11.3.3 Evolutionary Programming

The combinators and stream transformation operations that have been proposed here are subject to logical constraints on their application and use, such as the types of input and output, and the ratio of input elements to output elements. It is hypothesised that these constraints should enable reasoning about these programs and the combination of subcomponents automatically, though guaranteeing the validity of the final combination of stream transformations.

This proposed technique is firmly in the realm of Genetic Programming, and the use of Genetic Programming for hyper-heuristics. Hyper-heuristics were previously mentioned in Chapter 2, as a generalisation of metaheuristic methods, which attempt to design, build, tune, hybridise or choose between existing metaheuristics to enable a more automated tool for complex combinatorial problems.

To enable a Genetic Programming approach this Thesis proposed a further meta-language, implemented in Haskell, for the types of selection operator, and

ranges of other parameters such as population size for the combinators in this library. This metalanguage would then be run to give rise to a specific collection of stream transformers and a grammar for their combination that would be subjected to evolution by an existing standard genetic programming tool.

11.3.4 Generators for Imperative Languages

In chapter 9 it was seen that imperative languages can make use of the *generator pattern* to facilitate on demand computations. ParadiseEO especially made use of objects of this type to manage neighbourhood construction and exploration, however the use of generators was not formalised nor made explicit, but rather used in an ad-hoc way throughout the library structure.

Generators have not been used widely for the implementation of metaheuristics, in either functional or imperative languages at the present time. It remains to be seen how well the support for generators in other languages can be leveraged to extend the use of these forms of combinators beyond Haskell, however it is expected that it should be both possible and effective.

Bibliography

[1] William B. Ackerman. Data flow languages. *IEEE Computer*, 15:15–25, 1982.

[2] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[3] Mauro Birattari. *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective*. PhD thesis, Université Libre de Bruxelles, 2004.

[4] Mauro Birattari. *Tuning Metaheuristics: A Machine Learning Perspective*. Springer, first edition, 2005.

[5] Robert G Bland and David F Shallcross. Large travelling salesman problems arising from experiments in x-ray crystallography: A preliminary report on computation. *Oper. Res. Lett.*, 8(3):125–128, June 1989.

[6] Jürgen Branke, Christiane Barz, and Ivesa Behrens. Ant-based crossover for permutation problems. In *Proceedings of the 2003 international conference on Genetic and evolutionary computation: PartI*, GECCO'03, pages 754–765, Berlin, Heidelberg, 2003. Springer-Verlag.

[7] Edmund Burke, Emma Hart, Graham Kendall, Jim Newall, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In F. Glover and G. Kochenberger, editors, *Handbook of Meta-Heuristics*, chapter 16, pages 457–474. Kluwer Academic Publishers, 2003.

[8] S. Cahon, N. Melab, and E.-G. Talbi. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380, May 2004.

[9] V. Černý. A Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, 1985.

[10] Alonzo Church. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics*, 2(33):346–366, 1932.

[11] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion. from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 315–326, New York, 2007. ACM.

[12] Steven P. Coy, Bruce L. Golden, George C. Runger, and Edward A. Wasil. Using experimental design to find effective parameter settings for heuristics. *Journal of Heuristics*, 7:77–97, January 2001.

[13] Steven P. Coy, Bruce L. Golden, George C. Runger, and Edward A. Wasil. Using experimental design to find effective parameter settings for heuristics. *Journal of Heuristics*, 7(1):77–97, January 2001.

[14] Noah M. Daniels, Andrew Gallant, and Norman Ramsey. Experience report: Haskell in computational biology. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 227–234, New York, NY, USA, 2012. ACM.

[15] Noah M. Daniels, Raghavendra Hosur, Bonnie Berger, and Lenore J. Cowen. SMURFLite: combining simplified Markov random fields with simulated evolution improves remote homology detection for beta-structural proteins into the twilight zone. *Bioinformatics*, 28(9):1216–1222, May 2012.

[16] M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[17] Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *Trans. Evol. Comp*, 1(1):53–66, April 1997.

[18] Sean R. Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–763, 1998.

[19] Conal Elliott and Paul Hudak. Functional reactive animation. *SIGPLAN Not.*, 32(8):263–273, August 1997.

[20] Conal M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 25–36. ACM, 2009.

[21] Andreas Fink and Stefan Voß. Hotframe: A heuristic optimization framework. In Stefan Vo, David L. Woodruff, Ramesh Sharda, and Stefan Vo, editors, *Optimization Software Class Libraries*, volume 18 of *Operations Research/Computer Science Interfaces Series*, pages 81–154. Springer US, 2003.

[22] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 383–396. ACM, 2008.

[23] Thom Fruewirth and Slim Abdennadher. *Essentials of Constraint Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[24] Michel Gendreau and Jean-Yves Potvin. Metaheuristics in Combinatorial Optimization. *Annals of Operations Research*, 140(1):189–213, 2005.

[25] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, 1993. ACM.

[26] Andy Gill, Tristan Bull, Andrew Farmer, Garrin Kimmell, and Ed Komp. Types and type families for hardware simulation and synthesis: the internals and externals of kansas lava. In *Proceedings of the 11th international conference on Trends in functional programming*, TFP'10, pages 118–133, Berlin, Heidelberg, 2011. Springer-Verlag.

[27] F. Glover. Tabu Search, Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[28] F. Glover. Tabu Search, Part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[29] Martina Gorges-Schleuter. Explicit parallelism of genetic algorithms through population structures. In *Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, PPSN I, pages 150–159, London, UK, UK, 1991. Springer-Verlag.

[30] Michael Guntsch and Martin Middendorf. A population based approach for aco. In *Proceedings of the Applications of Evolutionary Computing on EvoWorkshops 2002: EvoCOP, EvoIASP, EvoSTIM/EvoPLAN*, pages 72–81, London, UK, UK, 2002. Springer-Verlag.

[31] Bruce Hajek. Cooling schedules for optimal annealing. *Math. Oper. Res.*, 13:311–329, May 1988.

[32] Haskell.org, the haskell programming language, February 2013.

[33] Ralf Hinze. Memo functions, polytypically! In *Proceedings of the 2nd Workshop on Generic Programming, Ponte de*, pages 17–32, 2000.

[34] Ralf Hinze, Thomas Harper, and Daniel W. H. James. Theory and practice of fusion. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 19–37, Berlin, Heidelberg, 2011. Springer-Verlag.

[35] J. H. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[36] Holger Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. In T.Walsh I.P.Gent, H.v.Maaren, editor, *SAT 2000*, pages 283–292. IOS Press, 2000. SATLIB is available online at www.satlib.org.

[37] Holger Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., 2005.

[38] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

[39] Paul Hudak. *The Haskell school of expression - learning functional programming through multimedia*. Cambridge University Press, New York, 2000.

[40] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional*

Programming, 4th International School, volume 2638 of LNCS, pages 159–187. Springer-Verlag, 2003.

[41] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *In Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL-III*, pages 1–55. ACM Press, 2007.

[42] John Hughes. Why functional programming matters. *The Computer Journal*, 32:98–107, 1989.

[43] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 2000.

[44] John Hughes. Programming with arrows. *5th International Summer School on Advanced Functional Programming*, pages 73–129, 2005.

[45] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, 1998.

[46] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[47] A. Tolmach Jones, S.P. and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in ghc. In *R. Hinze, editor, 2001 Haskell Workshop*, 2001.

[48] Mark P. Jones and Simon Peyton Jones. Lightweight Extensible Records for Haskell. In *In 3rd Haskell Workshop*. Utrecht University, 1999. Technical Report.

[49] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[50] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[51] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell 2004: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[52] Edmund S. L. Lam and Martin Sulzmann. A concurrent constraint handling rules implementation in haskell with software transactional memory. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 19–24, New York, 2007. ACM.

[53] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 26–37, New York, 2003. ACM.

[54] Yi Liu, Shengwu Xiong, and Hongbing Liu. Hybrid simulated annealing algorithm based on adaptive cooling schedule for tsp. In *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, GEC '09, pages 895–898, New York, NY, USA, 2009. ACM.

[55] Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Jürgen Teich. Opt4J - A Modular Framework for Meta-heuristic Optimization. In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)*, pages 1723–1730, Dublin, Ireland, 2011.

[56] S. Masrom, Siti Z.Z. Abidin, P. N. Hashimah, and A. S. Abd. Rahman. Towards Rapid Development of User Defined Metaheuristics Hybridisation. *International Journal of Software Engineering and Its Applicatons*, 5, 2011.

[57] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[58] Peter Merz and Bernd Freisleben. Memetic algorithms for the traveling salesman problem. *Complex Systems*, 13(4):297–345, 2001.

[59] Thomas Nordin and Andrew Tolmach. Modular lazy search for constraint satisfaction problems. *J. Funct. Program.*, 11(5):557–587, 2001.

[60] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[61] Ross Paterson. A new notation for arrows. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, volume 36 of *ICFP '01*, pages 229–240, New York, October 2001. ACM.

[62] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *J. Funct. Program.*, 13(1):0–255, Jan 2003.

[63] Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Experience report: a do-it-yourself high-assurance compiler. In *Proceedings of the Intl. Conference on Functional Programming (ICFP)*. ACM, September 2012. Preprint available at http://www.cs.indiana.edu/~lepike/pub_pages/icfp2012.html.

[64] I. Pohl. Practical and theoretical considerations in heuristic search algorithms. *Mach. Intell.*, 8:55–72, 1977.

[65] Günther R. Raidl. A Unified View on Hybrid Metaheuristics. In Francisco Almeida, María J. Blesa Aguilera, Christian Blum, J. Marcos Moreno-Vega, Melquíades Pérez Pérez, Andrea Roli, and Michael Sampels, editors, *Hybrid Metaheuristics, Third International Workshop, HM 2006, Gran Canaria, Spain, October 13-15, 2006, Proceedings*, volume 4030 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 2006.

[66] Gerhard Reinelt. TSPLIB - A Traveling Salesman Problem Library. *INFORMS Journal on Computing*, 3:376–384, 1991.

[67] Peter Van Roy. *Multiparadigm Programming in Mozart/Oz: Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected Papers (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[68] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Prentice Hall, 2nd international edition edition, 2003.

[69] Tom Schrijvers and Bruno C. d. S. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 32–44, 2011.

[70] Tom Schrijvers, Peter Stuckey, and Philip Wadler. Monadic constraint programming. *J. Funct. Program.*, 19:663–697, November 2009.

[71] Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter Stuckey. Search Combinators. In *Principles and Practice of Constraint*

Programming, 17th International conference, Proceedings, pages 774–788. Springer, 2011.

[72] Christian Schulte. *Programming constraint services: high-level programming of standard and new constraint services*. Springer-Verlag, Berlin, Heidelberg, 2002.

[73] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, CP '98, pages 417–431, London, UK, UK, 1998. Springer-Verlag.

[74] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *In Proceedings of the ACM SIGPLAN Workshop on Haskell*, New York, 2002. ACM.

[75] Mary Sheeran. Functional and dynamic programming in the design of parallel prefix networks. *Journal of Functional Programming*, 21:59–114, 2011.

[76] Michael Spivey. Combinators for breadth-first search. *J. Funct. Program.*, 10:397–408, July 2000.

[77] Michael Spivey. Algebras for combinatorial search. *J. Funct. Program.*, 19:469–487, July 2009.

[78] E. Taillard. Robust Taboo Search for the Quadratic Assignment Problem. *Parallel Computing*, 17:443–455, 1991.

[79] E.G. Talbi. A Taxonomy of Hybrid Metaheuristics. *Journal of Heuristics*, 8:541–564, 2002.

[80] Tarmo Uustalu and Varmo Vene. The Essence of Dataflow Programming. In *Lecture Notes in Computer Science*, pages 2–18. Springer-Verlag, 2005.

[81] R. J. M. Vaessens, E.H.L. Aarts, and J.K. Lenstra. A local search template. *Computers and Operations Research*, 25:969–979, November 1998.

[82] Alex Van Breedam. Improvement heuristics for the vehicle routing problem based on simulated annealing. *European Journal of Operational Research*, 86(3):480–490, November 1995.

[83] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[84] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, pages 461–493, 1992. Special issue of selected papers from 6'th Conference on Lisp and Functional Programming.

[85] David Wakeling. Functional proxy programming or tinker tailor soldier spy. In *Implementation of Functional Languages*. Springer, 2012.

[86] David H. Wolpert and William G. Macready. No free lunch theorems for optimisation. *IEEE Transations on Evolutionary Computation*, 1(1):67–82, 1997.

Appendix A

Glossary

A description of a range of terms used in this Thesis for quick reference.

exhaustive search where the search process will, either explicitly or implicitly examine every solution to a particular problem. These will typically be tree based algorithms such as Depth or Breadth first search. Where branches of a tree are pruned (thus avoiding explicit examination of some solutions) it must be possible to *prove* that no better solution exists in that branch than some remaining branch. These algorithms can find provably optimal solutions to problems, but the runtime becomes prohibitive as problem sizes increase.

heuristic a rule of thumb for constructing solutions, or performing search that tends to give good results.

impure function a function which may read or modify properties of the environment in addition to its explicit parameters and returned results. See *pure function*.

lazy evaluation a form of normal order reduction where indirection is used to enable sharing of computations and hence avoid repeating work. The underlying concept is that a place holder to the computation is passed into each function as it is evaluated in normal order. When a result is required (demanded) the computation is run and the placeholder is replaced by the

value. This value is then present throughout the program, where the indirection leads to the placeholder.

low level operator describes functions that are problem specific and provide a generic interface for a metaheuristic to operate upon. While the interface is generic, the method of operation and data structure for the problem are not prescribed.

normal order evaluation the outermost expression is evaluated first, with the expressions for the arguments either substituted into the new expression or a note made as to how to reach the argument for later use. Where an expression is not needed it will not be evaluated, but where it is used many times it may be evaluated many times.

metaheuristic a general template for iterative heuristic methods. When combined with low level operators for a particular problem, and other related parameters, they give rise to a specific instance of a heuristic.

point based algorithm a metaheuristic that acts upon one seed solution at a time. For example an Iterative Improver has one *current* solution. Typically these algorithms do work by generating multiple options at each iteration and selecting one to be the new seed solution for the next iteration.

population based algorithm is a metaheuristic that acts upon a group of solutions, rather than only one. Internally they may then select one solution at a time to change, but this will return to the group, before the process can continue.

pure function a function that is exclusively a mapping from its input parameters to its output value and nothing else. The *sqrt* function is typically pure in that it maps from a value to a value, where as traditionally an *RNG* is not pure, because it takes no parameters and gives back a value that varies, based upon a hidden state.

strict evaluation where the expressions for the parameters of a function are evaluated down to their result values before the function body. Technically the value can also be a memory pointer or reference rather than a value but this distinction will not be of concern here. Also called *eager* and *greedy* evaluation. The name *Strict evaluation* does not define the order in which

arguments for functions are evaluated, only that all are evaluated before the evaluation of the function body. **The arguments are always (strictly) evaluated.**

Appendix B

Glossary of the Library

This appendix provides a summary of the major functions in the library of combinators for hybrid metaheuristics.

bestSoFar (p. 64), transforms a stream so that the next value that will appear upon it is always the best seen until that point.

$$bestSoFar :: Optimisable\ s \Rightarrow Stream\ s \rightarrow Stream\ s$$
$$bestSoFar\ (a : as) = scanl\ bestOf\ a\ as$$

chunk (p. 70), takes a constant value and a stream of values. It proceeds to divide the stream into regularly sized blocks of size n, and results in a stream of these blocks as lists. This does not typically preserve the *speed* of the underlying stream, consuming more values than it produces lists.

$$chunk :: Int \rightarrow Stream\ v \rightarrow Stream\ (List\ v)$$
$$chunk\ n = unfoldr\ (Just \circ splitAt\ n)$$

The *varChunk* command extends this concept by allowing each block to have its own size. The size of each block is provided by a stream of sizes, rather than a constant size as in *chunk*. While *chunk* may be implemented in terms of *varChunk* this is seen as unnecessary.

$$varChunk :: Stream\ Int \rightarrow Stream\ v \rightarrow Stream\ (List\ v)$$
$$varChunk\ ns = zipWith\ take\ ns \circ loopP\ (zipWith\ drop\ ns)$$

doMany (p. 71), is a composition of stretch and chunk. It enables a strategy to be applied multiple times to each value in a stream, and the results collected as a list.

```
doMany :: Int → (Stream a → Stream b) → ExpandT a b
doMany n f = chunk n ∘ f ∘ stretch n
```

divide (p. 70), takes a list of names and a stream of name values. Subsequently the input stream of the transformation is divided into a collection of streams, where each value appears in only one of the new streams. The streams are in the order of the names in the list, and the values of the main stream are linked with names in the stream of names to indicate where they should end up.

```
divide :: Eq n ⇒ List n → Stream n → Stream v → List (Stream v)
divide names ns xs = [substream (map (≡ n) ns) xs | n ← names]
  where substream bs = map snd ∘ filter fst ∘ zip bs
```

improvement (p. 73), provides a transformation from an expansion operation (a transformation from a stream of values into a stream of lists of values), into a new expansion where the output list contains only values that are *better* than the original seed values.

```
improvement :: Optimisable s ⇒ ExpandT s s → ExpandT s s
improvement nf sols
  = safe (map (:[]) sols)
  $ zipWith (λa b → filter (>:a) b) sols (nf sols)
```

join (p. 70), provides the inverse of divide. It takes the same additional parameters, a list of names and a stream of name values, but then takes a list of streams, and reverses the division process.

```
join :: Eq n ⇒ List (n, Stream v) → Stream n → Stream v
join streams ns = unfoldr f (ns, streams)
  where
    f (k : ks, vs) =
      let (as, (n, p : ps) : bs) = break ((≡ k) ∘ fst) vs
      in Just (p, (ks, as ++ (n, ps) : bs))
```

loopP (p. 64), provides a function for *tying the knot* on a stream described process. This links the outputs of the stream process to the inputs, with an initial value, and provides a single stream of values to the user.

$$loopP\ f\ seed = loopS\ f\ [seed]$$

loopS (p. 64), provides a function for *tying the knot* on a stream described process. This links the outputs of the stream process to the inputs, with a supply of initial values, and provides a single stream of values to the user.

$$\begin{aligned} &loopS :: (Stream\ s \to Stream\ s) \to [s] \to Stream\ s \\ &loopS\ f\ seed = \textbf{let}\ as = seed +\!\!+ f\ as\ \textbf{in}\ as \end{aligned}$$

nest (p. 72), is a composition of divide and join, which enables the insertion of the results of different transformations into a stream.

$$\begin{aligned} &nest :: Eq\ n \Rightarrow List\ (n, Stream\ a \to Stream\ b) \to Stream\ n \to \\ &\qquad Stream\ a \to Stream\ b \\ &nest\ fs\ ns = flip\ join\ ns \circ zipWith\ (\lambda(n,f)\ s \to (n,f\ s))\ fs \circ \\ &\qquad divide\ (map\ fst\ fs)\ ns \end{aligned}$$

A variation upon nest is *preNest* where the supply of names is provided by a stream transformation.

$$\begin{aligned} &preNest :: Eq\ n \Rightarrow (Stream\ a \to Stream\ n) \to \\ &\qquad List\ (n, Stream\ a \to Stream\ b) \to \\ &\qquad Stream\ a \to Stream\ b \\ &preNest\ d\ fs\ xs = nest\ fs\ (d\ xs)\ xs \end{aligned}$$

poisson (p. 74), a function that generates a Poisson distribution, of a particular number of elements, subject to a parameter for the most likely value in the distribution.

$$\begin{aligned} &poisson :: Double \to Int \to Distribution \\ &poisson\ mean\ sz = map\ f\ [0\,..\,(fromIntegral\ sz)] \\ &\quad \textbf{where}\ f\ k = exp\ (-mean) \\ &\qquad\qquad * \ sum\ [(mean ** i)\ /\ fi \mid (i,fi) \leftarrow zip\ [0\,..\,k]\ factorials] \\ &factorials = scanl\ (*)\ 1\ [1\,..] \end{aligned}$$

push (p. 73), converts a stream from inductive computation, to a consecutive form, so that earlier solutions in the stream are evaluated before the later solutions. It is typically applied to a stream of solutions generated by a metaheuristic, before other processing such as selecting the best of the solutions generated.

$$
\begin{aligned}
&push :: Stream\ a \to Stream\ a \\
&push\ (x : xs) = x\ `seq`\ x : push\ xs
\end{aligned}
$$

restart (p. 72), combinators are provided as a variation upon nesting. In these implementations they take a stream of solutions, the solutions to be restarted from, and a strategy or transformation. This transformation is applied to the first restart position, until a termination condition is detected. At this point the second restart value is inserted into the stream, and the transformation continued. *restartExtract* only provides the final value of each sequence of transformations, making the result a stable stream transformation in its own right.

$$
\begin{aligned}
&restart, restartExtract :: (Stream\ a \to Stream\ a) \\
&\qquad\qquad \to (Stream\ a \to Stream\ Bool) \\
&\qquad\qquad \to Stream\ a \\
&\qquad\qquad \to Stream\ a \\
&restart\ f\ r\ (a : as) \\
&\quad = \mathbf{let}\ ms = loopP\ (nest\ [(False, id), (True, const\ as)]\ (r\ ms) \circ f)\ a \\
&\quad\ \ \mathbf{in}\ ms \\
&restartExtract\ f\ r\ (a : as) \\
&\quad = \mathbf{let}\ ms = loopP\ (nest\ [(False, id), (True, const\ as)]\ rs \circ f)\ a \\
&\qquad\ \ rs = r\ ms \\
&\quad\ \ \mathbf{in}\ [l \mid (p, l) \leftarrow zip\ (drop\ 2\ \$\ window\ 2\ rs)\ ms, p \equiv [True, False]]
\end{aligned}
$$

safe (p. 74), provides functionality to combine two streams of lists. At each stage it will pick between them based upon which is *not empty*. It is intended that it be used where one stream may provide empty lists (e.g. improving neighbourhoods) and there for might need a default, or alternative.

$$
\begin{aligned}
&safe :: Stream\ (List\ v) \to Stream\ (List\ v) \to Stream\ (List\ v) \\
&safe = zipWith\ (\lambda a\ b \to \mathbf{if}\ null\ b\ \mathbf{then}\ a\ \mathbf{else}\ b)
\end{aligned}
$$

select (p. 70), provides selections from a stream of collections with a likelihood determined by a probability distribution. The probability distribution is provided as a function from an integer size to a list of values, where the list is expected to be in ascending order. The type of values of the distribution is expected to be a floating point number, but this is not required. Selections from each list are made with respect to a stream of values of the same type as the distribution.

```
{-# INLINE select #-}
select f = zipWith g (unsafePerformIO $
                        newStdGen >>= return ∘ randoms)
      ∘ map (λx → zip (f ∘ length $ x) x)
  where g r = snd ∘ head ∘ dropWhile ((<r) ∘ fst)
```

stretch (p. 71), extends a stream by duplicating values in place for a finite number of steps.

```
stretch :: Int → Stream v → Stream v
stretch n = concat ∘ map (replicate n)
```

tabuFilter (p. 75), a transformation from a neighbourhood expansion operation, replacing it with an expansion operation where the neighbourhoods are each filtered for recently seen solutions.

```
tabuFilter :: Eq s ⇒ (Stream s → Stream (List s)) →    -- window
                     (ExpandT s s) →    -- neighbourhood
                     (ExpandT s s)
tabuFilter wF nF xs
   = safe (map (:[]) xs)
   $ zipWith (λws → filter (flip notElem ws)) (wF xs) (nF xs)
```

uniform (p. 75), a function that generates a uniform distribution of a particular number of elements.

```
uniform :: Int → Distribution
uniform sz = [x / sz' | x ← [1..sz']]
  where sz' = fromIntegral sz
```

until (p. 80), combinators evaluate a stream of values until a change over point is reached (indicated by a stream of *triggers*). At this point, one of a number of *futures* replaces the original stream on the output.

$$
\begin{aligned}
&\mathit{until}\_ :: \mathit{Stream}\ a && \text{-- current stream of values}\\
&\qquad \to \mathit{Stream}\ \mathit{Bool} && \text{-- stream of triggers for switch over}\\
&\qquad \to \mathit{Stream}\ (\mathit{Stream}\ a) && \text{-- stream of potential futures}\\
&\qquad \to \mathit{Stream}\ a\\
&\mathit{until}\_\ (a{:}\ \_)\ \ (\mathit{True}{:}\ \_)\ \ (\_ : cs{:}\ \_) = a : cs\\
&\mathit{until}\_\ (a : as)\ (\mathit{False} : bs)\ (\_ : cs) \ = a : \mathit{until}\_\ as\ bs\ cs
\end{aligned}
$$

window (p. 70), provides a stream of lists, where each list is a recent history of the underlying stream. Several implementations are possible, for example using a queue data structure, however this has been found to be no quicker than the following implementation.

$$
\begin{aligned}
&\mathit{window} :: \mathit{Int} \to \mathit{Stream}\ v \to \mathit{Stream}\ (\mathit{List}\ v)\\
&\mathit{window}\ sz = \mathit{map}\ \mathit{reverse} \circ \mathit{tail} \circ \mathit{scanl}\ (\lambda xs\ x \to \mathit{take}\ sz\ (x : xs))\ [\,]
\end{aligned}
$$

Appendix C

The Haskell Language

In this section a brief summary and example of the syntax of the Haskell language will be given, to facilitate understanding of code segments present in this Thesis. More detailed information about Haskell may be found in many text books, such as [39, 60].

C.1 lhs2TeX

All the code presented in this Thesis has been processed by a Haskell pretty printing library called *lhs2TeX*, giving it a more mathematical appearance. This translation is one for one with the underlying Haskell code with the following symbol table giving the translations of the notation.

λ	$+\!\!+$	$\circ$	$\rightarrow$	$\leftarrow$	$\prec$	$\Rightarrow$	$\not\equiv$	$\equiv$	$\geqslant$	$\leqslant$
\	++	.	->	<-	-<	=>	/=	==	>=	<=

C.2 Functions

A function is defined in Haskell as a name, followed by a series of parameter names, the equals symbol and then the function body. All names are (mostly) a lower case first letter followed by other letters and numbers as in many other programming languages. For example the following are two very simple functions:

$theNumber5 = 5$ -- a function with no parameters
$id\ x = x$ -- the identity function

Functions are typically *applied* to parameters on their right until the function has *bound* all its variable names, at which point it may be evaluated. Due to functions themselves being first class members of the language, and hence able to be passed as parameters, some care must be taken over the order of symbols being passed. For example this usage of the square root function will fail, because the first *sqrt* will be applied not to the square root of 5, but to the function *sqrt*.

$f = sqrt\ sqrt\ 5$

Function application order may be controlled using brackets as in other languages, or using the *$* function provided by the basic Haskell libraries for right-association. For example, the *$* function may be used to correct the application order for the composed square root function in the following way.

$f' = sqrt\ \$\ sqrt\ 5$

The basic mathematical functions are provided and may be used infix within expressions. For example:

$add5\ x = x + 5$

The standard mathematical functions also predominantly follow the normal application precedence rules for mathematical functions, for example division binds more tightly than addition. More complicated mathematical functions are provided as named functions in standard libraries, such as the preceding example involving *sqrt*.

C.2.1 Lambda expressions

Anonymous functions may be created through the use of *lambda* expressions. These are similar to normal functions, but replace the name of the function with λ. A *normal* function is equivalent to a named lambda expression. For example:

$f = (\lambda x\ y \rightarrow (x + y)\ /\ y)$ -- a lambda expression, bound to the name f

C.2.2 Function types

Each function has a type, though the compiler can often infer the types, as in the cases above. Where the programmer wishes to be more precise, or overrule the compilers inferences, types can be provided. For example, this following type definition would instruct the compiler that the function f is a function which takes two parameters, both of type *Int* and the computation results in a boolean value.

$$f :: Int \rightarrow Int \rightarrow Bool$$

To specify a function as a parameter, the data type of the parameter function is enclosed in brackets, such as in the following example, where the first parameter will be a function from *Int*s to *Int*s.

$$f :: (Int \rightarrow Int) \rightarrow Double \rightarrow Bool$$

In the above examples the *types* at each stage are fixed to specific forms of data with specific internal structures. Haskell supports a more flexible type definition using *type variables*, which always begin with lower case letters. These generically defined functions can be used wherever the types of the parameters match. For example:

$$f :: (a \rightarrow b) \rightarrow c \rightarrow a$$

It should be noted that the standard mathematics operators are all binary functions internally, over the appropriate data types.

C.2.3 Sectioning functions

Functions in Haskell are also used in their higher order form, where a function of this type $a \rightarrow b \rightarrow c$ can be written as $a \rightarrow (b \rightarrow (c))$, that is a function from a parameter of type a to a new function. Using this perspective on functions allows for partial application of functions, or *sectioning*. For example it is possible to write $(+1)$, a function which adds one to any number given as a parameter. This approach may be used to share computations and partial results throughout Haskell programs.

C.2.4 Structuring code

To aid the construction of code two patterns are used in Haskell, *where* and *let-in*. Each of these defines a new level of the code, where all terms are equal and

have access to one another, as is seen in this Thesis in a number of the data-flow approaches. The following two example illustrate the same function expressed using each of these.

$$
\begin{aligned}
f\ x = \ & \textbf{let}\ y = 5 + x \\
& \quad\ \ z = y + y + x \\
& \textbf{in}\ \ (z, y) \\
g\ x = \ & (z, y) \\
& \textbf{where}\ y = 5 + x \\
& \quad\ \ z = y + y + x
\end{aligned}
$$

C.2.5 Making choices

Haskell offers a variety of ways to implement program logic forking. The common control structure of *if-then-else* is provided, but tends to be used only for simple choices because it only allows for a binary decision and so becomes verbose quickly. For example:

$$
\begin{aligned}
f\ x = \ & \textbf{if}\ x < 5\ \textbf{then}\ 0 \\
& \quad\ \ \textbf{else if}\ x < 7\ \textbf{then}\ 1 \\
& \quad\ \ \quad\ \ \textbf{else}\ 2
\end{aligned}
$$

Haskell provides three alternatives to basic *if-then-else* notation.

guards which allows for a less verbose chaining of if-then-else structures.

$$
\begin{aligned}
g\ x \mid & x < 5 \quad\ \ = 0 \\
\mid & \mathit{otherwise} = 2
\end{aligned}
$$

pattern matching provides a clean syntax for expressing different cases of a function body based upon how variables match in the parameters. This is more useful with more complex data structures such as lists and trees.

$$
\begin{aligned}
& g\ 5\ \_ = 0 \quad \text{-- under score is wild card} \\
& g\ 6\ 6 = 7 \\
& g\ \_\ \_ = 1 \quad \text{-- default}
\end{aligned}
$$

case is an alternative syntax for pattern matching, which may be used on arbitrary expressions.

$$
\begin{aligned}
&g\ x = \textbf{case}\ x\ \textbf{of} \\
&\qquad 5 \qquad\quad \rightarrow 0 \\
&\qquad \mathit{otherwise} \rightarrow 2
\end{aligned}
$$

C.3 Data types

A variety of basic types are present in Haskell. Bools, Ints and Doubles are simple data types, but the basic types of Haskell also include tuples, lists and strings. A user may create new type names through the use of the *type* keyword, for example;

$$
\begin{aligned}
&\textbf{type}\ \mathit{MyString} = [\mathit{Char}] \quad \text{-- the same definition as the basic library} \\
&\textbf{type}\ F\ a\ b = (a, [b]) \qquad \text{-- a tuple of a and the list of b} \\
&\textbf{type}\ G\ a = (\mathit{Int}, \mathit{Int}, \mathit{Double}, a)
\end{aligned}
$$

Other methods for user defined data are

newtype allows for the creation of a data type with a single constructor. For example, this data type from the *State Monad* definition.

$$\textbf{newtype}\ \mathit{State}\ s\ a = \mathit{State}\ \{\mathit{runState} :: s \rightarrow (s, a)\}$$

data is a more general purpose way to define data types, with multiple constructor functions. See the rose tree example below.

Each of these can be used to create recursively defined data structures, through the recursive application of their constructor functions. For example this definition of a rose tree, which has been chosen to illustrate this point, but is not how rose trees are defined in the standard *Data.Tree* library.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ a \\
&\quad = \mathit{Forest}\ [\mathit{Tree}\ a] \\
&\qquad | \mathit{Leaf}\ \{\mathit{payload} :: a\} \quad \text{-- payload is an accessor function} \\
&\qquad\qquad\qquad\qquad\quad \text{-- for a single field of a Tree}
\end{aligned}
$$

C.4 Lists in Haskell

Lists in Haskell are provided as a standard type, but their definition is as a recursive type similar to the trees above. As such they can be defined to be generated by computations lazily and be of indefinite or infinite length. They are used in this form throughout this Thesis as *streams*, rather than defining a new data type.

Lists can be constructed recursively or through the use of list comprehension notation. In the following example, each of the functions f, g and h will give the same output.

$$
\begin{aligned}
f &= [x*x \mid x \leftarrow [0..10]] \\
g &= map\ (\lambda x \rightarrow x*x)\ [0..10] \\
h &= \textbf{let}\ h'\ 10 = [10*10] \\
&\qquad\quad h'\ x = (x*x) : h'\ (x+1) \\
&\quad \textbf{in}\ h'\ 0
\end{aligned}
$$

Pattern matching may also be used with lists, as in this example of the standard *head* function.

$$head\ (x{:}\ \_) = x$$

C.5 Type Classes in Haskell

The Haskell type class system provides a further mechanism for generalising over data types. A class provides a number of function definitions which types within that class promise to support. A data type may then be defined as an *instance* of a class. A function defined over a type class can then be assured that it may use the operations of that class on any data it is applied to.

The following example contains a reproduction of part of the standard *Eq* class, which deals with equality of data types. The example also includes a generic instantiation of tuples for equality testing.

$$
\begin{aligned}
&\textbf{class}\ Eq\ a\ \textbf{where} \\
&\quad (\equiv) :: a \rightarrow a \rightarrow Bool \\
&\textbf{instance}\ (Eq\ a, Eq\ b) \Rightarrow Eq\ (a,b)\ \textbf{where} \\
&\quad (a,b) \equiv (c,d) = a \equiv c \wedge b \equiv d
\end{aligned}
$$

The not equal function ($\not\equiv$) can now be expressed generically for all types that are declared as instances of *Eq*.

$(\not\equiv) :: Eq\ a \Rightarrow a \to a \to Bool$ -- where a is in Eq, then the following holds
$a \not\equiv b = \neg\ (a \equiv b)$

C.6 Rewrite Rules

A key advantage of functional programming systems is the ability of the compilers to rearrange and reason about the structure of the program. In Haskell this is facilitated through *rewrite* rules[1], which allow the programmer to give the compiler additional information about how certain situations may be changed, typically for performance.

In the following example, a rewrite rule from the standard libraries, two *map* operations in sequence are rearranged to remove the intermediate data structure.

```
{-# RULES
 "map/map"    forall f g xs.  map f (map g xs)
                 = map (f.g) xs
#-}
```

[1]For more information about rewrite rules see the following page of the online Haskell users guide.

```
http://www.haskell.org/ghc/docs/5.04.3/html/
users_guide/rewrite-rules.html
```

Appendix D

Lucid Style Code in Haskell

The Lucid programming language is a *Data Flow* programming language, in which variables are represented as *varying* values within the lifetime of the program. The value taken by a variable is defined by an equation that may rely upon previous values of that same variable, or values from other variables. In this way Lucid provides a method to allow the creation of programs as graphs of data dependencies, expressed through the equations linking the evolution of the different variables of the problem. For example the natural numbers in Lucid can be expressed as;

```
n = 1 fby (n+1)
```

This describes a variable n, which takes the values 1 followed by $1+1$ followed by $2+1$ and so on.

D.1 Similarities with Haskell

Ackerman [1] proposed six key properties that are required of a data flow language. Three of these requirements are characteristics that are shared by modern lazy functional languages:

- *Freedom from side effects*, Ackerman notes that pure functional languages such as Pure Lisp share this property;

- *equivalence of instruction scheduling*, that subject to data dependencies, the precise order of instructions is irrelevant (a similar advantage is noted by [42], pointing out that it enables compiler technology through rearrangement and reduces the effort required by the programmer); and

- *a single assignment convention*, variables may appear only once in the parameter list and once set may not be changed.

The other three requirements that Ackerman proposed deserve a short note;

- *An odd notation for iteration*, is assumed to refer to the use of recursive definitions rather than loop constructs, which may have been less accepted at the time of writing;

- *a lack of history sensitivity in procedures*, no procedure has internal state variables, this appears to be a repetition of the freedom from side effects requirement; and

- *locality of effect*, it is unclear what this refers too, it may be (i) related to the need to compile into tight loops or (ii) for the purposes of caching in memory relating to performance. This confusion may be related to changing priorities in the field of computer science.

The high degree of overlap between these requirements and characteristics that have been argued as important in the field of modern functional languages, suggest that Haskell should provide an ideal location for a shallow embedding of data flow concepts.

D.2 Recursive lists in Haskell

In Haskell the implementation of a varying value may be done using an infinite list or stream, exploiting lazy evaluation to provide both values on demand and memoization of values in each stream once computed. For example, the natural numbers from the previous example can be implemented as follows;

$$n = 1 : map\ (+1)\ n$$

This describes a list n, which takes the value 1 followed by a computation defining the remainder of the list, in terms of the increment of previous values in the list by 1. Due to lazy evaluation and sharing this list will yield the value 1, then the

first result of the computation $1+1$, then the application of $+1$ to the result of the second computation and so on.

D.3 Shallow Lucid-like programming in Haskell

While there are clear similarities to Lucid in this implementation, it is possible to make Haskell resemble Lucid more closely. The Lucid operation *fby* (followed by) performs the same function as *cons* in this example, and so a direct synonym can improve the resemblance. In order to capture the operation of numbers in Lucid streams it is necessary to tell Haskell how to treat a list as if it were a number, so as to overload the meaning of addition, and the automatic lifting of constants to streams. This can all be achieved by making lists of numbers instances of the type class *Num*.

$$\begin{aligned}&\textbf{instance } \mathit{Num}\ a \Rightarrow \mathit{Num}\ [a] \textbf{ where}\\ &\quad (+) = \mathit{zipWith}\ (+)\\ &\quad (*) = \mathit{zipWith}\ (*)\\ &\quad \mathit{fromInteger} = \mathit{repeat} \circ \mathit{fromInteger}\end{aligned}$$

It is now possible to rewrite the code for the generation of natural numbers in a manner much closer to its appearance in Lucid.

$$n = 1\ `\mathit{fby}`\ (n+1)$$

Fibbonaci numbers provide a more compelling example of the use of data flow programming. What follows is a standard example written in Lucid, drawn from [80].

```
fibo = 0 fby (fibo + (1 fby fibo))
```

Fibbonacci numbers are also used as a standard example of functional programming, and the approach taken by Lucid can be described as short term memoization over lists.

$$\mathit{fib} = 0 : 1 : \mathit{zipWith}\ (+)\ \mathit{fib}\ (\mathit{tail}\ \mathit{fib})$$

The close correspondence is once again apparent, with some renaming of operations and the explicit using of $\mathit{zipWith}\ (+)$. This example has eschewed the use of the previous class instantiation, though it would have provided a direct translation of the Lucid code into Haskell. This Thesis has used the standard Haskell functions, finding them perfectly adequate in themselves.

Appendix E

Examples of usage of Object Oriented Frameworks

This appendix provides more detailed code for the implementation of solvers using the ParadiseEO and Opt4J frameworks.

E.1 ParadiseEO

E.1.1 TSP in ParadiseEO

This source code is based upon a genetic algorithm solver for the TSP found upon the ParadiseEO website, and written by Sébastien Cahon and Thomas Legrand. It was then modified using tutorials and information from the standard ParadiseEO documentation found at;

http://paradiseo.gforge.inria.fr

route

```
typedef eoVector <float, unsigned> Route;
```

route_eval

```
void RouteEval :: operator () (Route & __route) {
  float len = 0 ;
```

```
  for (unsigned i = 0 ; i < Graph :: size () ; i ++)
    len -= Graph :: distance
         (__route [i],__route [(i + 1) % Graph :: size ()]);
  __route.fitness (len) ;
}
```

route_init

```
void RouteInit :: operator () (Route & __route) {
  srand ( time(NULL) );

  // Init.
  __route.clear () ;
  for (unsigned i = 0 ; i < Graph :: size () ; i ++)
    __route.push_back (i) ;

  // Swap. cities

  for (unsigned i = 0 ; i < Graph :: size () ; i ++) {
    unsigned j = (unsigned) (Graph :: size () *
                             (rand () / (RAND_MAX + 1.0)));
    unsigned city = __route [i] ;
    __route [i] = __route [j] ;
    __route [j] = city ;
  }
}
```

E.1.2 Selected code from ParadiseEO

From moRndWithoutReplNeighborhood.

```
virtual void init(EOT & _solution, Neighbor & _neighbor) {
  unsigned int i, tmp;
  maxIndex = neighborhoodSize ;
  i = rng.random(maxIndex);
  _neighbor.index(_solution, indexVector[i]);
  tmp=indexVector[i];
  indexVector[i]=indexVector[maxIndex-1];
  indexVector[maxIndex-1]=tmp;
  maxIndex--;
}

virtual void next(EOT & _solution, Neighbor & _neighbor) {
  unsigned int i, tmp;
```

```
  i = rng.random(maxIndex);
  _neighbor.index(_solution, indexVector[i]);
  tmp=indexVector[i];
  indexVector[i]=indexVector[maxIndex-1];
  indexVector[maxIndex-1]=tmp;
  maxIndex--;
}
```

From moLocalSearch, the superclass of point based algorithms such as moSA.

```
virtual bool operator()(EOT & _solution) {
  if (_solution.invalid())
    fullEval(_solution);

    // initialization of the parameter of the search
    // (for example fill empty the tabu list)
    searchExplorer.initParam(_solution);

    // initialization of the external continuator
    // (for example the time, or the number of generations)
    cont->init(_solution);

    bool b;
    do {
      // explore the neighborhood of the solution
      searchExplorer(_solution);
      // if a solution in the neighborhood can be accepted
      if (searchExplorer.accept(_solution)) {
        searchExplorer.move(_solution);
        searchExplorer.moveApplied(true);
      } else
        searchExplorer.moveApplied(false);

      // update the parameter of the search
      // (for ex. Temperature of the SA)
      searchExplorer.updateParam(_solution);

      b = (*cont)(_solution);
    } while (b && searchExplorer.isContinue(_solution));

    searchExplorer.terminate(_solution);

    cont->lastCall(_solution);

    return true;
```

```
}
```

From moSAexplorer, a helper class that allows for the exploration of a neighborhood in the style of simulated annealing. For each neighbour in a neighbourhood it will provide a boolean acceptance, based upon the current temperature.

```
virtual void operator()(EOT & _solution) {
  //Test if _solution has a Neighbor
  if (neighborhood.hasNeighbor(_solution)) {
    //init on the first neighbor: supposed to be
    //random solution in the neighborhood
    neighborhood.init(_solution, selectedNeighbor);

    //eval the _solution moved with the neighbor and
    //stock the result in the neighbor
    eval(_solution, selectedNeighbor);
  }
  else {
    //if _solution hasn't neighbor,
    isAccept=false;
  }
};

virtual bool accept(EOT & _solution) {
  if (neighborhood.hasNeighbor(_solution)) {
    if (solNeighborComparator(_solution, selectedNeighbor))
      // accept if the current neighbor is
      // better than the solution
      isAccept = true;
    else {
            double alpha=0.0;
            double fit1, fit2;
            fit1=(double)selectedNeighbor.fitness();
      fit2=(double)_solution.fitness();
      if (fit1 < fit2) // this is a maximization
        alpha = exp((fit1 - fit2) / temperature );
      else // this is a minimization
        alpha = exp((fit2 - fit1) / temperature );
      isAccept = (rng.uniform() < alpha) ;
    }
  }
  return isAccept;
};
```

E.2 Opt4J

E.2.1 TSP in Opt4J

This source code is based upon code drawn from the documentation of Opt4J at;

http://opt4j.sourceforge.net/documentation

The source code was modified to allow the loading of city positions from a file, rather than random construction of problems (see file *SalesmanProblem*). The file *City* is a problem specific helper class. The other files, while problem specific, would require equivalent implementations for any other problem.

City

```
public class City {
  protected final double x;
  protected final double y;

  public City(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public double getX() {
    return x;
  }

  public double getY() {
    return y;
  }
}
```

SalesmanRoute

```
@SuppressWarnings("serial")
public class SalesmanRoute
      extends ArrayList<City>
      implements Phenotype {}
```

SalesmanCreator

```
public class SalesmanCreator
        implements Creator<PermutationGenotype<City>>
{
  protected final SalesmanProblem problem;
```

```
  @Inject
  public SalesmanCreator(SalesmanProblem problem) {
    this.problem = problem;
  }

  public PermutationGenotype<City> create() {
    PermutationGenotype<City> genotype
      = new PermutationGenotype<City>();
    for (City city : problem.getCities()) {
      genotype.add(city);
    }
    Collections.shuffle(genotype);
    return genotype;
  }
}
```

SalesmanDecoder

```
public class SalesmanDecoder
      implements Decoder<PermutationGenotype<City>,
                 SalesmanRoute>
{
  public SalesmanRoute
      decode(PermutationGenotype<City> genotype)
  {
    SalesmanRoute salesmanRoute = new SalesmanRoute();
    for (City city : genotype) {
      salesmanRoute.add(city);
    }
    return salesmanRoute;
  }
}
```

SalesmanProblem

```
public class SalesmanProblem {
  protected Set<City> cities = new HashSet<City>();

  @Inject
  public SalesmanProblem() {
    // modified to load from file 417.tsp, a series of
    // floats as strings stored in recomputation form.
    try{
      Scanner lineScanner;
```

```
      lineScanner = new Scanner(new File("fl417.tsp"));
      while(lineScanner.hasNextLine())
      {
        Scanner fScan = new Scanner(lineScanner.nextLine());
        floatScanner.next();
        final City city = new City(fScan.nextDouble(),
                                   fScan.nextDouble());
        cities.add(city);
      }
    }catch(Exception e){e.printStackTrace(System.out);}
  }

  public Set<City> getCities() {
    return cities;
  }
}
```

SalesmanModule

```
public class SalesmanModule extends ProblemModule {
  @Constant(value = "size")
  protected int size = 100;
  public int getSize() {
    return size;
  }
  public void setSize(int size) {
    this.size = size;
  }
  public void config() {
    bindProblem(SalesmanCreator.class,
                SalesmanDecoder.class,
                SalesmanEvaluator.class);
  }
}
```

E.2.2 Selected code from Opt4J

Drawn from MutatePermutationSwap. This is presumed to be the standard mutation strategy in an evolutionary algorithm.

```
public void mutate(PermutationGenotype<?> genotype, double p) {
  int size = genotype.size();

  if (size > 1) {
    for (int i = 0; i < size; i++) {
```

```
      if (random.nextDouble() < p) {
        int j;
        do {
          j = random.nextInt(size);
        } while (j == i);
        Collections.swap(genotype, i, j);
      }
    }
  }
}
```

Drawn from MatingCrossoverMutate. This is presumed to be the standard list crossover algorithm.

```
protected Pair<Individual> mate(Individual parent1,
                                Individual parent2,
                                boolean doCrossover) {
  Genotype p1 = parent1.getGenotype();
  Genotype p2 = parent2.getGenotype();
  Genotype o1, o2;

  if (doCrossover) {
    Pair<Genotype> offspring = crossover.crossover(p1, p2);
    o1 = offspring.getFirst();
    o2 = offspring.getSecond();
  } else {
    o1 = copy.copy(p1);
    o2 = copy.copy(p2);
  }

  mutate.mutate(o1, mutationRate.get());
  mutate.mutate(o2, mutationRate.get());

  Individual i1 = individualFactory.create(o1);
  Individual i2 = individualFactory.create(o2);

  Pair<Individual> individuals = new Pair<Individual>(i1, i2);
  return individuals;
}
```

From the function optimize, in EvolutionaryAlgorithm, this code shows the creation of a number of new solutions from a set of parents. All the new solutions are added to the population, and then the *lames*, presumed to be the weakest, are removed, restoring the population to a predefined maximum.

```
  if (offspringCount > 0) {
```

```
    // evaluate new individuals first
    if (offspringCount < lambda) {
      completer.complete(population);
    }

    Collection<Individual> parents;
    parents = selector.getParents(mu, population);
    Collection<Individual> offspring;
    offspring = mating.getOffspring(offspringCount, parents);
    population.addAll(offspring);
  }

  if (population.size() > alpha) {
    Collection<Individual> lames;
    lames = selector.getLames(population.size() - alpha,
                              population);
    population.removeAll(lames);
  }
```

From CrossoverListXPoint, this code shows the cross over of two lists.

```
public Pair<G> crossover(G p1, G p2) {
  ListGenotype<Object> o1 = p1.newInstance();
  ListGenotype<Object> o2 = p2.newInstance();

  int size = p1.size();

  if (x <= 0 || x > size - 1) {
    throw new RuntimeException(this.getClass() +
                               " : x is " +
                               x +
                               " for binary vector size " +
                               size);
  }

  SortedSet<Integer> points = new TreeSet<Integer>();

  while (points.size() < x) {
    points.add(random.nextInt(size - 1) + 1);
  }

  int flip = 0;
  boolean select = random.nextBoolean();

  for (int i = 0; i < size; i++) {
```

```
    if (i == flip) {
      select = !select;

      if (points.size() > 0) {
        flip = points.first();
        points.remove(flip);
      }
    }

    if (select) {
      o1.add(p1.get(i));
      o2.add(p2.get(i));
    } else {
      o1.add(p2.get(i));
      o2.add(p1.get(i));
    }
  }

  Pair<G> offspring = new Pair<G>((G) o1, (G) o2);
  return offspring;
}
```

E.2.3 Specialised code for Stream library, mimicking Opt4J functionality

$$
\begin{aligned}
&\mathit{selectFromMap} :: \mathit{Stream}\ (\mathit{Map}\ \mathit{Double}\ s) \to \mathit{Stream}\ b \\
&\mathit{selectFromMap} = \mathit{unsafePerformIO}\ \$ \\
&\quad \textbf{do}\quad \mathit{rs} \leftarrow \mathit{newStdGen} \gg\!= \mathit{return} \circ \mathit{randoms} \\
&\qquad\quad \mathit{return}\ \$\ \mathit{map}\ (\mathit{snd} \circ \mathit{fromJust}) \circ \mathit{zipWith}\ M.\mathit{lookupGT}\ \mathit{rs} \\
&\mathit{makeDistPop} :: \mathit{List}\ \mathit{Double} \to \mathit{List}\ s \to \mathit{Map}\ \mathit{Double}\ s \\
&\mathit{makeDistPop}\ \mathit{dist} \\
&\quad = \mathit{snd} \circ \\
&\qquad (\mathit{flip}\ (M.\mathit{mapAccum}\ (\lambda(a:\mathit{as})\ \_ \to (\mathit{as}, a)))\ \mathit{seedDist}) \\
&\quad \textbf{where} \\
&\qquad \mathit{seedDist} = M.\mathit{fromDistinctAscList}\ [(x, ()) \mid x \leftarrow \mathit{dist}] \\
&\mathit{mutate} :: \mathit{Double} \to \mathit{Stream}\ \mathit{TSPProblem} \to \mathit{Stream}\ \mathit{TSPProblem} \\
&\mathit{mutate}\ \mathit{mutateRate} = \mathit{unsafePerformIO}\ \$ \\
&\quad \textbf{do}\quad \mathit{is} \leftarrow \mathit{newStdGen} \gg\!= \mathit{return} \circ \mathit{randomRs}\ (0, 416) \\
&\qquad\quad \mathit{ds} \leftarrow \mathit{newStdGen} \gg\!= \mathit{return} \circ \mathit{randoms}
\end{aligned}
$$

```
        return $ mutate' (cycle [0..417] is ds
    where
      mutate' (v:vs) sk@(s:ss) (b:bs) ck@(c:cs)
        | v ≡ 417 = c : newMutate' vs sk bs cs
        | b ⩾ mutateRate = newMutate' vs sk bs ck
        | otherwise = c' `seq` newMutate' vs ss bs (c' : cs)
          where c' = swapCitiesOnIndex v s c

crossover :: Stream (List TSPProblem)
          → Stream (List TSPProblem))
crossover = unsafePerformIO $
  do rs ← newStdGen >>= return ∘ randomRs (0,416)
     return $ zipWith recombine rs
  where
    recombine i [a,b]
      = let bPath = getTSPPathAsList 0 b
            aPath = getTSPPathAsList 0 a
            (as,_) = splitAt i aPath
            (cs,_) = splitAt i bPath
            bs = filter (λx → S.notMember x (S.fromList as)) bPath
            ds = filter (λx → S.notMember x (S.fromList cs)) aPath
        in [setRoute (as ++ bs) a, setRoute (cs ++ ds) a]
```

Appendix F

Code comparison with MRFy

These examples are to illustrate the difference in complexity of the code between the implementation of metaheuristics using MRFy's original framework, and the stream combinator based framework, as compared in Chapter 10. Some values appear as named variables, to indicate their function but these values are not initialised. The stream examples require that they are applied to seed data, have convergence checking and final selection of the best solutions composed with them, while the MRFy examples would need to be processed and seeded with random number generators to be evaluated.

F.1 Genetic Algorithms

MRFy's original code for an implementation of Genetic Algorithms;

```
nss :: NewSS
nss hmm searchP query betas scorer = fullSearchStrategy
    (fmap (wrapBestScore ∘ map scorer) $ initialize hmm searchP query betas)
    (mutate searchP query betas scorer)
    scoreUtility
    (takeNGenerations (generations searchP))
    (unScored ∘ minimum)
type Population = [Scored Placement]
```

```
initialize :: RandomGen r ⇒
              HMM →
              SearchParameters →
              QuerySequence →
              [BetaStrand] →
              Rand r [Placement]
initialize hmm searchP query betas =
  sequence $ take n
    $ repeat
    $ projInitialGuess hmm (getSecPreds searchP) query betas
  where n = getSearchParm searchP populationSize
  -- invariant: len [SearchSolution] == 1
mutate :: SearchParameters
        → QuerySequence
        → [BetaStrand]
        → Scorer Placement
        → Scored Population
        → Rand StdGen (Scored Population)
mutate searchP query betas scorer (Scored placements _) =
  return ∘ wrapBestScore
  =≪ shuffle'
  =≪ return ∘ take (getSearchParm searchP populationSize)
     ∘ sort
     ∘ (placements++)
  =≪ progeny
  where progeny = parRandom $ map (λgs → scorer <$>
    randomizePlacement betas gs (V.length query))
    $ getPairings
    $ map unScored placements
getPairings :: [Placement] → [Placement]
getPairings [] = []
getPairings [p] = [p]
getPairings (p1 : p2 : ps) = crossover p1 p2 : getPairings ps
```

The stream implementation of Genetic Algorithms, following the original logic of MRFy.

```
loopS ( concat
      ∘ map (take popSize)
      ∘ map sortO
      ∘ (λchunks → zipWith (++) chunks
                 ∘ map (map (λ[a,b] → crossover a b))
                 ∘ map (chunk 2)    -- getPairings
                 $ chunks
        )
      ∘ chunk popSize
      )
```

F.2 Simulated Annealing

MRFy's original code for implementing the Simulated Annealing algorithm. The temperature strategy is a standard geometric cooling schedule and the acceptance criterion is the standard Simulated Annealing acceptance function. The cooling schedule and acceptance function are combined as a single function called *boltzmannUtility*.

```
nss :: NewSS
nss hmm searchP query betas scorer = fullSearchStrategy
  (fmap scorer $ RHC.initialize hmm searchP query betas)
  (RHC.mutate searchP query betas scorer)
  (boltzmannUtility searchP)
  (takeByCCostGap (acceptableCCostGap searchP))
  id

boltzmannUtility :: RandomGen r ⇒
                    SearchParameters →
                    Move a →
                    Rand r (Utility (Scored a))
boltzmannUtility searchP (Move {younger, older, youngerCCost})
  = do uniform ← getRandom
       return $ if boltzmann youngerCCost
         (scoreOf younger)
         (scoreOf older) ⩾ uniform
                then Useful younger
```

```
                    else Useless
  where boltzmann :: CCost → Score → Score → Double
        boltzmann cost (Score youngScore) (Score oldScore)
          = exp ((oldScore − youngScore) /
                 (constBoltzmann ∗ temperature))
          where temperature = (constCooling cost) ∗ constInitTemp
        constBoltzmann = getSearchParm searchP boltzmannConstant
        constInitTemp = getSearchParm searchP initialTemperature
        constCooling = getSearchParm searchP coolingFactor
```

The above search strategy implemented in terms of Streams;

```
loopP (λsols → zipWith4 (saChoose quality)
                        (randoms g)
                        (geoCooling changeRate initialTemp)
                        sols (perturb sols)
      )
```

F.3 Checking Placements

```
checkPlacement :: QuerySequence →
                  [BetaStrand] →
                  PricedSol →
                  Bool
checkPlacement qs = isValid 0
  where isValid lastGuess [ ] _ = True
        isValid lastGuess (b : bs) (o : os)
          | o < lastGuess = False
          | o ⩾ betaSum  = False
          | otherwise     = isValid (o + len b) bs os
          where
            betaSum = U.length qs − sum (map len (b : bs))
checkAllOptions :: (Int → PricedSol → PricedSol)
                   PricedSol →
                   List PricedSol
```

```
checkAllOptions changePlace p
  = let  as = map (λx → changePlace x p) [1..]
         bs = map (λx → changePlace x p) $ map (0−) [1..]
    in p : takeWhileDiff as ++ takeWhileDiff bs
  where
    takeWhileDiff x = map fst
                    ∘ takeWhile (λ(a,b) → a ≢ b)
                    $ zip (p : x) x
```