Under consideration for publication in J. Functional Programming

Combinators for Meta-heuristic Search

RICHARD SENINGTON, DAVID DUKE
University Of Leeds, Leeds LS2 9JT, UK
(*e-mail:* sc06r2s,d.j.duke@leeds.ac.uk)

Abstract

Metaheuristics are a group of iterative optimisation algorithms for combinatorial problems that have been widely studied for the last 25 years and this has resulted in a variety of different approaches. These algorithms and a number of frameworks for experimentation within the field are usually implemented in imperative languages, however little work has been done using functional languages for this task.

We develop a library of composable stream transformation functions in the pure functional language Haskell, and then show how a selection of well known, but quite different, metaheuristics can be built from this small set of combinators. This approach allows for the concise expression of the algorithms and provides a high degree of modularity and composability. This in turn allows for the rapid modification and hybridisation of algorithms to examine alternative strategies, an important provision for operational research and applications. We then illustrate the use of the library using the well known Travelling Salesperson Problem and give a brief comparison of the performance of algorithms constructed using the library with versions written in C.

Contents

1 Introduction

How do you best allocate finite resources? For example if you run a factory and have a number of machines, each machine is capable of doing a number of tasks at specific rates. At any given time the factory has a number of work tasks to complete. Which tasks should be done on which machines and in what order, such that all the tasks are completed as quickly as possible? Alternatively if you run a University, you will have timetables to design. There are many finite resources here (time slots, rooms and people) and the final timetable needs to satisfy a range of competing criteria.

There are many ways in which a set of jobs can be scheduled on a set of machines and many orders in which lectures can be assigned to rooms in a University. Commonly not all solutions are the same, with some using less of the finite resources (for example time) than others, or resulting in greater productivity. How can we find solutions that are better for these problems?

These are examples of combinatorial optimisation problems, a group of NP-hard problems that occur frequently in industry, engineering and science. In computer science many algorithms have been created for finding optimal solutions to these problems (for example depth first search and branch&bound) called *complete* algorithms. However as the size of the problems increases the size of the search space causes the runtime of the programs to increase exponentially.

In many tasks however time is limited, and finding higher quality solutions is more important than finding a provably optimal solution. In these cases meta-heuristic methods have been found to be effective in finding comparatively good solutions within practical time constraints, though they sacrifice the certainty that the optimal solution will ever be found (and are therefore *incomplete*).

Metaheuristics form a heterogeneous group of algorithms, so much so that they can be envisaged as being like a toolbox of concepts for the meta-heuristic designer to draw upon when encountering a new problem. No meta-heuristic is guaranteed to perform well on all problems, though most can be modified to improve performance on each new problem they are paired with, a process known as *tuning* (Birattari, 2005). This tuning can be a complex

process, involving the modification of static parameters, the modification of functional parameters and the integration of new concepts from the toolbox, known as *hybridisation*. For example, in genetic algorithms the process of experimentation can include; modifying the size of the populations, the rates of mutation and changing the method by which solutions are chosen for recombination (or breeding). In section 3.6 we will look at modifying a simple genetic algorithm meta-heuristic through the replacement of the mutation function and will see how this effects the quality of solutions discovered by the search process.

These considerations have led to the implementation of frameworks for creation, hybridisation and experimentation upon metaheuristics. However these frameworks, which are built on imperative programming abstractions, have had limited success. Masrom et al present the following conclusions:

> *"...they have limited predefined hybridisation. Indeed, some of them focus on either local search or evolutionary algorithms only. As a result, hybridisation is restricted within the limited metaheuristics."*

While many of the frameworks provide GUI support for non-expert control, for more sophisticated work, e.g. creating a new form of hybridisation, the same authors remark that *"it is imperative that the programmer has a deep understanding of the class libraries"* (Masrom *et˜al.*, 2011).

The contribution of this paper is to investigate the use of pure functional programming to express metaheuristics and hybridisation. *This is similar in intent to the work of Schrijvers et al on combinators enabling the construction of heuristics for the management of exhaustive tree based search algorithms* (Schrijvers *et˜al.*, 2011). Given the weakness of existing frameworks, can functional languages deliver a richer yet simpler framework for expressing and hybridising metaheuristics? Further, can a functional approach overcome the limits of predefined hybridisation and offer patterns of computation which link evolutionary and population based methods with local search or single solution methods? While such a 'combinatorial' approach to program solving should arguably be bread-and-butter for functional programming (Hughes, 1989), the diversity of local search problems confounds a simple approach to combinator design through:

- the heterogeneous nature of meta-heuristic algorithms and their components, which makes it difficult to capture or even identify common patterns;
- the absence of a unifying mathematical abstraction; for example the pattern of 'state + remaining input' model of functional parsers (Hutton & Meijer, 1998); and
- the interplay of different components of state that are not easily captured by Haskell's type system and current support for records.

Our solution is to model metaheuristics, at the top level, as *processes* which produce streams of solutions from seed data, such as seed solutions. The processes are constructed in a *data-flow style* through the composition and transformation of a number of functions that we think of as *stream transformers*. This simplifies the task of *hybridisation* by allowing the meta-heuristic designer to concentrate on the manipulation of finer-grained combinator building blocks. Our approach to metaheuristics is therefore solidly within the stream programming field of functional programming.

1.1 Travelling Sales Person Problem

The travelling salesman problem (TSP) is often used as a standard test and example application in combinatorial optimisation, due to it having a very simple description and direct real world application, and we will be using the problem in this paper. Here we will remind the readers of the specifications of the problem and discuss some of the variations that exist.

The task is to find a shortest Hamiltonian cycle in a connected graph, a cycle which goes through every vertex only once (Hoos & Stützle, 2005). The edges of the graph are weighted and the length of the cycle is the sum of the weights of the edges used. Specific instances of the problem can be symmetric, where the weight of an edge AB is always equal to the weight of BA, or asymmetric where this constraint is not present.

An instance of a TSP is defined by the number of vertices in the graph, and the weights of the edges between them. All the graphs that we have considered, both symmetric and asymmetric, have been drawn from the online repository TSPLIB (Reinelt, 1991) and are fully connected, though this is also an optional property in the wider context. For this paper we will only consider the problem `fl417.tsp` from the TSPLIB as our example. Where we report results this will be the example that has been used.

1.2 Paper Overview

The paper is structured as follows. Section˜2 introduces the four families of metaheuristics that will be examined in the paper, discusses how they interact with combinatorial problems, and looks at some existing styles of hybridisation. The examples that have been chosen are widely used and understood methods, covering both local search and evolutionary algorithms: iterative improvers, TABU, simulated annealing, and genetic algorithms. The TSP is then considered in light of the function types that are needed to interact with them, with the specific variants that we will use in our examples given.

Section˜3 gives the major content of the paper, looking at how functional languages can be used to break down processes into stream combinators, and the application of this method to metaheuristics. The section ends with a short series of example experiments upon the TSP, where a number of hybrids of the meta-heuristic methods discussed are compared.

In Section˜4 we step back, and examine the library design decisions, in particular *why* the stream transformation approach is preferable for this application than alternatives. Section˜5 shows a difficulty with our approach, related to thunk build up in lazy evaluation, and our solution to this problem.

Section˜6 provides a comparison between our functional metaheuristics and specialised versions written in C. Section˜7 concludes the paper, looking to future work and the deployment of functional methods as effective tools for meta-heuristic designers.

The major contributions of this paper are:

- a concerted attack on a class of related, yet heterogeneous algorithms, and demonstration as to how to present them in FP;
- the re-formulation of a number of well known metaheuristics into a data flow form;
- the creation of a set of combinators to capture the characteristics of these metaheuristics; and

- an examination of the performance issues that arise and a discussion of how they can be resolved.

Many of the combinators that are presented involve ordering solutions in some way. A convention of this paper while describing the combinators will be that when we refer one candidate solution being better than another we mean that the value of the better candidate is *lower*. This convention fits well with both the TSP example problem and the standard implementation of Haskell functions like *sort*.

2 Metaheuristics

Metaheuristic algorithms work by manipulating candidate solutions to combinatorial optimisation problems. These are *candidate* solutions in that (i) we cannot be sure they are optimal, and (ii) in some cases they may not even be valid solutions to the original problem. This second case arises in situations where the model of the problem being used cannot restrict, a-priori, the candidates being considered to those which are always valid. Superior models of problems that better restrict the candidates considered are always preferable. The TSP used in the remainder of the paper does not involve invalid candidates, and this issue will not be mentioned further in this paper.

The quality of a candidate solution is determined by an *objective function*. This function will usually provide a numerical value, or measure of the quality, however in some metaheuristics all that is needed is that the candidates form an ordered set. The underlying concept of metaheuristics is that, if we have a candidate solution of a given quality, small changes to that solution should yield other solutions of a 'similar' quality. A simple illustration can be drawn from the idea of a path through a graph, such as is found in the TSP. If the path is changed by removing a relatively small set of edges from a cycle and then adding a set of edges such that the result in a valid Hamiltonian cycle, the value of the new solution only differs from the old by the values of the edges involved in the change. Such a change is called a *perturbation*; many metaheuristics routinely apply perturbations to the candidate solutions as they execute.

Perturbation gives rise to the concept of a *neighbourhood*, a set of candidates that result from applying perturbation to a given candidate solution. Dually, perturbation is sometimes defined as the choice of a new candidate solution from within a given neighbourhood. Iterative local search typically involves moving from a current candidate to one within its neighbourhood.

A second method for creating new solutions from old is *recombination*, where a new candidate is produced from two or more existing candidates. This approach is often used with population based methods such as genetic algorithms (GA) (Goldberg, 1989).

Many variations of perturbation, neighbourhood and recombination exist for each problem that has been examined by the Operations Research (OR) community. Of the four families of algorithms chosen for this paper, iterative improvers (also known as hill climbers) and TABU search tend to use neighbourhoods to provide candidates. Simulated annealing tends to use perturbation, and genetic algorithms use recombination. The inspiration for each of these families is varied. Simulated annealing draws on concepts from physical simulations (in particular metallurgy); TABU is a variation on iterative improvers with the

addition of a memory; and genetic algorithms draw upon natural evolution and the study of it in artificial intelligence.

2.1 Hybridisation

In the absence of practical complete search methods the algorithm families addressed in this paper have been applied widely; and effort continues to tune algorithms to new problems. To this end a branch of study has developed which attempts to *hybridise* local search methods, combining different characteristics of each (Gendreau & Potvin, 2005; Birattari *et˜al.*, 2001; Raidl, 2006). The hope is that the hybrid meta-heuristic will have *stronger* search characteristics than its progenitors[1]. There has been substantial research into how to hybridise heuristics; Talbi(Talbi, 2002) for example, proposed two key concepts: *relay*, where metaheuristics are run in sequence, and *teamwork*, where they run in parallel and communicate findings. Talbi proposed two further subdivisions, *high* and *low level* hybridisation.

In both low level relay and teamwork hybrids the key concept is that the construction of the final strategy uses one meta-heuristic algorithm as a component or function of another. Talbi gives the following examples to illustrate this:

- *low level relay*; simulated annealing, using an iterative improver to yield the alternatives that it chooses between, rather than a random perturbation
- *low level teamwork*; a genetic algorithm, using an iterative improver to mutate solutions, rather than a random perturbation

High level hybridisation keeps the different strategies more clearly self-contained. For example, using simulated annealing for a time, then TABU search, where the TABU search begins from the final solution that the simulated annealing search produced.

While this paper starts with Talbi's low and high level hybridisation, one of our contributions is to move towards a uniform approach to hybridisation that eliminates this distinction.

2.2 Commonalities

The approaches described above suggest that local search algorithms consist of two parts. The first guides the application of perturbation and candidate selection, for example by iteration until a solution is deemed acceptable. The second part is the set of functions for generating, perturbing, selecting and recombining candidates. These functions tend to be highly problem-specific.

Our key insight is that a process which yields a sequences of solutions provides a suitable level of abstraction for meta-heuristic construction. This paper lays out a set of combinators for describing these processes, combining aspects of these processes and manipulating the resulting sequences of solutions. Before we consider the combinators we first present further details of the TSP problem that we use to demonstrate the ideas.

[1] Where stronger is understood as; finding better solutions to the combinatorial problems, in fewer iterations while continuing to find better solutions for longer.

2.3 Perturbation and Recombination for TSP

TSP provides a useful illustration for both our overall framework and the design of specific combinators. As a running example it demonstrates how a hard combinatorial problem can be addressed by the composition and construction of complex search strategies from simpler components. **Our aim is pedagogical, we do not intend nor claim that this represents fundamentally superior algorithms for solving instances of the TSP.** We shall therefore not look in detail at all known variations upon perturbation and recombination. To demonstrate how the example perturbation and recombination operations modify the structure of solutions we will provide example implementations.

2.3.1 Perturbation & Neighbourhoods

The basic perturbation operation is the swapping of two cities in the Hamiltonian cycle. From this swap operation we can define a second, more limited, form of perturbation, the adjacent exchange, where the two cities to be exchanged are adjacent in the sequence. The example code for these operations is provided in Figure 1.

```
type TSPSol a = [a]
type Perturb s = s → s
type Swap s   = (Int, Int) → Perturb s

tspSwap :: Swap (TSPSol a)
tspSwap (i,j) source = if i ≡ j then source else fr ++ b : md ++ a : bk
  where
    (fr, as)        = splitAt i source
    (a : md, b : bk) = splitAt (j − i) as

adjTspSwap :: Int → Int → Perturb (TSPSol a)
adjTspSwap len i | i ≡ len    = tspSwap (0, len)
                 | otherwise = tspSwap (i, i + 1)
```

Fig. 1: A list based implementation of TSP perturbation functions.

We define two neighbourhood functions, indicated by the suffix *N* in the function names, in terms of these perturbation operations; a deterministic neighbourhood derived from the *adjTspSwap* function, and a stochastic neighbourhood derived from the more general *tspSwap* function. The deterministic neighbourhood is constructed by applying the adjacent swap function to every adjacent pair of cities in the source solution.

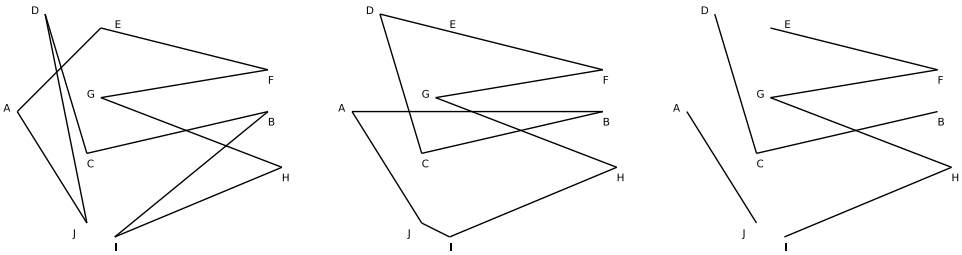
The second neighbourhood function we will use is the stochastic neighbourhood, which generates a neighbourhood by swapping randomly selected pairs of cities. This approach allows for the size of the neighbourhood to be controlled, while providing access to a wider overall range of candidate solutions. The code for these neighbourhood functions is found in figure 2, where the stochastic neighbourhood function takes a list of randomly generated integers (*rs*), assumed to be in the range of the problem. The size of the neighbourhood is controlled by the length of the list of randomly generated integers.

A trade off must be considered when designing a neighbourhood function relating to the size of the neighbourhood. A larger neighbourhood allows each step of a meta-heuristic

```
adjSwapN :: Neighbours (TSPSol a)
adjSwapN source = map (flip (adjTspSwap l) source) [0..l]
  where l = length source - 1

stochasticSwapN :: [Int] → Neighbours (TSPSol a)
stochasticSwapN rs source = map (flip tspSwap source) $ zip (map fst as) (map fst bs)
  where (as, bs) = partition ((≡ 0) ∘ snd) $ zip rs (cycle [0,1])
```

Fig. 2: A list based implementation of TSP neighbourhood functions.



(a) First parent (b) Second parent (c) Common Subsequences

Fig. 3: Identifying common subsequences in TSPs

to examine more solutions, however this requires more memory and time. While lazy evaluation can ameliorate this issue through generating elements of a neighbourhood only when they are required, in general it will only improve memory performance not the time issue.

2.3.2 Recombination

To implement recombination for TSP we use the concept that the new solution must contain all common subsequences of the parents, with the remainder of the sequence provided from a subsequent process. We limit our recombination process to operating only over two solutions at a time, although combining more than two is possible. Figure 3 gives a graphical example of the identification of common subsequences in a Euclidean TSP.

The recombination process can be divided into two parts, the first is identification of the common subsequences, the second putting them back together in a new order. This is seen in figure 4, where the *recomb* function is the composition of; (i) finding the common subsequences and (ii) shuffling them into a new order and concatenating them. Shuffling is provided by an auxiliary function that requires an external source of random values. The identification of common subsequences is provided by a function which processes one solution, comparing each pair of adjacent cities with a look up table provided by the other Hamiltonian cycle.

However even for pedagogical value this example is too naive, introducing too much chaotic behaviour into the metaheuristics. In the remainder of the paper we use a variant which is more respectful of edges present in the parents, even when not common to both parents.

```
recomb rs as = concat ∘ shuffle rs ∘ commonChunks as

shuffle :: Ord r ⇒ [r] → [a] → [a]
shuffle rs = map snd ∘ sortBy (λa b → compare (fst a) (fst b)) ∘ zip rs

commonChunks :: Eq a ⇒ [a] → [a] → [[a]]
commonChunks as (b : bs) = f bs [[b]]
  where
    aEdges = let as′ = cycle as in take (length as) $ zip as′(tail as′)
    f [ ] (cs : css)
      | elem (head cs, head $ last css) aEdges = (reverse cs ++ last css) : init css
      | otherwise = reverse cs : css
    f (x : xs) (cs : css)
      | elem (head cs, x) aEdges = f xs ((x : cs) : css)
      | otherwise = f xs ([x] : reverse cs : css)
```

Fig. 4: A list based implementation of a TSP recombination function.

2.3.3 Practical Implementation

The implementation presented here only deals with the structural changes to the Hamiltonian cycles. A full implementation would require a number of other features:

- pricing or evaluation, subject to a look up table of the edge weights for the specific TSP instances; and
- and explicit ordering over candidate solutions, often defined in terms of the value of the paths

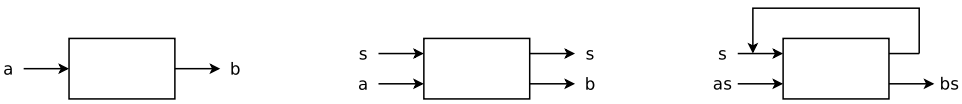
While not a requirement it is also faster to calculate the price of a new candidate in terms of the *differences* from its source candidate, rather than recalculating the cost of the whole route each time.

Lists are a poor implementation for this data structure. However the need to recall and reexamine previous solutions causes a problem for any data structure which relies upon in place updates. We therefore will use dictionaries to mimic the operation of an array, using the standard Haskell *Data.IntMap* structure, which will also provide the sharing functionality at minimal cost. Full details of this implementation using the dictionary, and providing the properties described above will not be provided, it being a rather mundane programming exercise.

Use of dictionaries is comparatively efficient for the swapping operations that we have described. Other data structures have been co-opted for representing TSP such as splay-trees (Fredman *et˜al.*, 1993) and specialised data structures have been created such as the two-level tree (Chrobak *et˜al.*, 1990).

3 Combinators For Metaheuristics

A direct translation of imperative approaches to metaheuristics results in monolithic state transformation systems. However we started from a “data-flow” perspective, structuring computations as a collection of evolving variables, or streams of data, that are generated, transformed and combined to yield solutions. This data-flow approach admits straightfor-



(a) A model of a simple function

(b) A model of a pure stateful function

(c) A Model of a stream transformation function with internal state

Fig. 5: Models of pure functions

ward expression as recursive Haskell functions, both problem specific and generic. It also facilitates the expression of more complex computations in a compositional style.

In this section we will first examine the generic aspects of the stream transformation approach, and in the later subsections describe the patterns of each of the metaheuristics and show how they can be expressed as stream transformation computations. The final subsections present a number of hybrid metaheuristics and their application to a TSP instance; and look at how our combinators can operate over other well combinatorial problems.

3.1 Stream Transformer Design

We begin with an example; imagine that you wish to add a random value to various different values in a pure functional program. This can be achieved by *threading* a random number generator through the program to the places where it is used, such as the following function;

$$f :: \mathit{RandomGen}\ g \Rightarrow \mathit{Float} \rightarrow g \rightarrow (\mathit{Float}, g)$$
$$f\ x\ g = \textbf{let}\ (a,b) = \mathit{random}\ g\ \textbf{in}\ (a+x,b)$$

An alternative is to gather the values that we wish to increment into a stream and apply a transformation over them all, such as this function;

$$f :: \mathit{RandomGen}\ g \Rightarrow g \rightarrow [\mathit{Float}] \rightarrow [\mathit{Float}]$$
$$f\ g = \mathit{zipWith}\ (+)\ (\mathit{randoms}\ g)$$

Figure 5 shows three models of functions diagrammatically; a similar set of diagrams is found in (Launchbury & Peyton Jones, 1995).

- 5a is the simplest form of a function;
- 5b illustrates the threading of state model;
- 5c corresponds to our stream transformation approach.

While simple, this forms the basis of how we deal with the stochastic perturbation technique later on. In section 4 we will discuss alternative ways in which metaheuristics and streams can be tackled in Haskell, including *arrowized streams* and *reactive programming* which have strong connections to this approach.

We have described metaheuristics as processes which give rise to a sequence of candidate solutions, where future solutions may be constructed from previous solutions. The model of stream transformation so far does not give rise to such evolving processes, but implement a map-like transformation. To “tie the knot” we provide;

$$loopP :: ([s] \to [s]) \to s \to [s]$$
$$loopP\ streamT\ seed = \textbf{let}\ sols = seed : streamT\ sols\ \textbf{in}\ sols$$

This function creates a stream of values from a seed value. The first value is the seed, the second the transformation of the seed, the third the transformation of the second value and so on. This is a well understood recursive technique used for efficient generation of the generating the Fibonacci numbers in functional languages.

A straight forward generalisation replaces the single seed with an initial stream segment;

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

These looping operations constrain the types of stream processors that are useful at this top level to;

$$\textbf{type}\ StreamT\ s = [s] \to [s]$$

All the metaheuristics that we will examine result in stream transformations of this type, however we have identified two other common functional patterns;

$$\textbf{type}\ ExpandT\ s = [s] \to [[s]]$$
$$\textbf{type}\ ContraT\ s = [[s]] \to [s]$$

ExpandT computations which gather information from a stream, or provide choices and the neighbourhood functions previously discussed in section 2, once lifted, are of this type. *ContraT* computations are usually contractions from a number of values, for example a choice from a neighbourhood, lifted top operation over streams.

Instances of these operations can be written in a number of ways, for example arrowized computations. However we have found that in practice a direct approach using functions from the standard libraries such as *map*, *zip*, *zipWith*, *scanl* and their variations are simpler, see section 4.4.

3.2 Iterative Improvers

Iterative Improvers are more commonly known as hill climbers and gradient decent. 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 two parts, the generation of the raw neighbourhood, and the processing of this into a neighbourhood of improving solutions, where improving means *less than the parent*, as previously stated in section 1.2. 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. We will first give a data-flow implementation of a deterministic first found iterative improver.

$$firstFoundii :: Ord\ s \Rightarrow (s \to [s]) \to s \to [s]$$
$$firstFoundii\ nf\ seed$$
$$= \textbf{let}\ sols = seed : map\ head\ improvements$$
$$neighbours = map\ nf\ sols$$
$$improvements = zipWith\ (\lambda a\ b \to filter\ (a >)\ b)\ sols\ neighbours$$
$$\textbf{in}\ sols$$

This function is defined in terms of three stream of data each created from a different transformation pattern.

- *sols* (*ContraT s*): the stream of solutions consisting of the initial solution followed by the stream of choices from the stream of improving neighbourhoods;
- *neighbours* (*ExpandT s*); the stream of neighbourhoods, found by applying the neighbourhood transformation to each element of *sols* :
- *improvements* (*StreamT* [*s*]); the stream of neighbourhoods such that for each neighbourhood, every element improves upon the solution from which it was created.

The stream of improving neighbourhoods is described as a *zipWith* operation over a filter taking both the stream of neighbourhoods and the stream of solutions. However it is really a modification of the neighbourhood function, to yield only improving neighbourhoods. So we propose a function called *improvement*, which is a transformation of a stream expander;

$$\begin{aligned}&\mathit{improvement} :: \mathit{Ord}\ s \Rightarrow \mathit{ExpandT}\ s \rightarrow \mathit{ExpandT}\ s\\&\mathit{improvement}\ \mathit{nf}\ \mathit{sols}\\&\quad = \mathit{zipWith}\ (\lambda a\ b \rightarrow \mathit{filter}\ (a >)\ b)\ \mathit{sols}\ (\mathit{nf}\ \mathit{sols})\end{aligned}$$

With this abstraction over the improvement process we can now provide a more generalised iterative improvement function;

$$\begin{aligned}&\mathit{iterativeImprover} :: \mathit{Ord}\ s \Rightarrow \mathit{ExpandT}\ s \rightarrow \mathit{ContraT}\ s \rightarrow s \rightarrow [s]\\&\mathit{iterativeImprover}\ \mathit{nf}\ \mathit{cf}\ \mathit{seed}\\&\quad = \mathbf{let}\ \mathit{sols} = \mathit{seed} : (\mathit{cf} \circ \mathit{improvement}\ \mathit{nf})\ \mathit{sols}\ \mathbf{in}\ \mathit{sols}\end{aligned}$$

The final change we must make is abstracting away the specification of the seed solution and the loop present in the *iterativeImprover*. Iterative improvement becomes a combinator of our library; it transforms neighbourhood functions and can be reused as a component of more complex hybrid metaheuristics.

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

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 *minimum* and *maximum*;

$$\begin{aligned}&\mathit{firstFoundii}, \mathit{maximalii}, \mathit{minimalii} :: \mathit{Ord}\ s \Rightarrow \mathit{ExpandT}\ s \rightarrow \mathit{StreamT}\ s\\&\mathit{firstFoundii}\ \mathit{nf} = \mathit{iterativeImprover}\ \mathit{nf}\ (\mathit{map}\ \mathit{head})\\&\mathit{maximalii}\ \mathit{nf} = \mathit{iterativeImprover}\ \mathit{nf}\ (\mathit{map}\ \mathit{minimum})\\&\mathit{minimalii}\ \mathit{nf} = \mathit{iterativeImprover}\ \mathit{nf}\ (\mathit{map}\ \mathit{maximum})\end{aligned}$$

To use one of these strategies on a problem they would first have to be looped and provided with a seed. At the *ghci* prompt, for a *tsp* problem, this could be done using the following;

```
> loopP (firstFoundii tsp_neighbourhood) tsp_seed
```

3.2.1 Stochastic Iterative Improvers

To build a *stochastic* iterative improver in terms of these combinators we need to provide a suitable contraction transformation. A stream of values, provided by a random number

generator, is encapsulated within the choice transformation and thus does not require any modification of the functions already shown. An example of a simple stochastic iterative improver is given below;

$$stochasticii :: Ord\ s \Rightarrow ([s] \to r \to s) \to [r] \to ExpandT\ s \to StreamT\ s$$
$$stochasticii\ rcf\ rs\ nf = iterativeImprover\ nf\ (zipWith\ (flip\ rcf)\ rs)$$

This encapsulation of a stochastic component in a stream transformer is an important tool in the construction of more sophisticated metaheuristics. We will revisit this approach in section 3.6.

3.3 TABU

TABU search (Glover, 1989; Glover, 1990) is an adaptation of the first found iterative improver algorithm, which makes use of memory to allow the strategy to try to make further progress once the iterative improver would have ended. Like iterative improvers, TABU search is usually based upon a neighbourhood function. Recently used moves are stored in a TABU list of fixed size and as the algorithm proceeds older moves are forgotten. Like an iterative improver the generated neighbourhood is pruned, but this pruning also takes into account the solutions in the TABU list. The following is the standard set of rules:

- If the solution is an improvement on the current solution, then this will be the choice
- If the solution is not an improvement and is on the TABU list, discard it
- If the solution is not an improvement but not on the TABU list, move to it if no other solution is found in the neighbourhood that improves upon the current solution

The implementation of TABU conceptually partitions a neighbourhood into three groups; those candidates that improver upon the current solution, those that do not but are not on the TABU list, and the ones that are. To implement this division in Haskell we lift the standard *partition* operation, so that it operates upon each neighbourhood with respect to the source candidate. As we often need to lift other operations to stream transformations we provide a general operation for this purpose;

$$lift :: (t \to b \to c) \to (a \to t) \to [a] \to [b] \to [c]$$
$$lift\ f\ g = zipWith\ (f \circ g)$$

Filters and partitions may each be lifted in this way;

$$lift\ filter\ (<)\ :: Ord\ a \Rightarrow [a] \to [[a]] \to [[a]]$$
$$lift\ partition\ (>)\ :: Ord\ a \Rightarrow [a] \to [[a]] \to [([a], [a])]$$

The *improvement* function can now be expressed succinctly in terms of *lift*;

$$improvement\ nf\ sols = lift\ filter\ (>)\ sols\ (nf\ sols)$$

Using lifted partitions and filters we can now take a stream of solutions and a stream of TABU information, and use this to divide a stream of neighbourhoods into two new streams; the stream of improving neighbourhoods and the stream of neighbourhoods which do not improve upon the source solution, but are not on the TABU list. The output of TABU is then defined in terms of these streams and the rules previously given;

```
tabuFilter :: Ord s ⇒ [[s]] → ExpandT s → ExpandT s
tabuFilter tabu nf sols
  = let (imp, notImp) = unzip $ lift partition (>) sols (nf sols)
        notTabu = lift filter (flip notElem) tabu notImp
        select [ ] [ ] c = c
        select [ ] b _ = b
        select a _ _ = a
    in zipWith3 select imp notTabu notImp
```

So far we have not specified where the TABU information comes from. As the list is a short term history, it can be thought of as a sliding window on the stream of candidate solutions that are being generated. Such a window can be produced by a queue data structure with limited size. An efficient queue functional implementation is known (Okasaki, 1998). Using this we can create the function *window* with the following data type;

```
window :: Int → ExpandT s
```

Implementing the queue is routine and can be found in the appendix A.6.

This window transformation has been limited to using list data types to keep this example short. A more flexible implementation can be achieved using Haskell classes. This would give opportunities for more sophisticated TABU behaviour, such as using *Sets* rather than lists for the TABU candidates.

We now define the generic TABU combinator;

```
tabu :: Ord s ⇒ ExpandT s →      --the window creation transformation
                ExpandT s →      --the neighbourhood transformation
                ContraT s →      --the final choice transformation
                StreamT s

tabu wf nf cf sols = cf $ tabuFilter (wf sols) nf sols
```

As with iterative improver this combinator takes a neighbourhood transformation and a choice transformation, but also takes a window transformation to create the TABU lists. Once again *loopP* would be used to tie the knot.

Unlike iterative improver, TABU search does not guarantee that later candidate solutions in the sequence are always better than earlier candidates because a user of a final system only really cares about the best solution that has been seen at any given point. Using the combinators defined and the following,

```
bestSoFar ∼ (x : xs) = scanl min x xs
```

We can now create a complete functional version of Glovers TABU algorithm (Glover, 1989; Glover, 1990);

```
gloverTABU :: Ord s ⇒ ExpandT s → Int → s → [s]
gloverTABU nf winSize
  = bestSoFar ∘ loopP (tabu (window winSize) nf
                            (map head))
```

A number of variations on TABU search exist. We will illustrate the flexibility of our combinators by implementing a part of Taillard's Robust Taboo search (Taillard, 1991). 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.

We first construct a stream transformation over windows which delivers stochastically varying window sizes;

```
varyWindow :: RandomGen g ⇒ (Int, Int) → g → StreamT [s]
varyWindow range g = zipWith take (randomRs range g)
```

Construction of Taillard's TABU is now straightforward;

```
taillardTABU nf winSize range g
  = bestSoFar ∘ loopP (tabu (varyWindow range g ∘ window winSize)
                            (map nf)
                            (map head))
```

3.4 Simulated Annealing

The simulated annealing (SA) meta-heuristic was proposed by (Kirkpatrick *et˜al.*, 1983) and later independently by (Černý, 1985). 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 random perturbation rather than deterministic selection.

At each step in SA a current solution is perturbed to generate an alternative solution. This alternative solution is then accepted or rejected by a decision process which is controlled by a real value *temperature*, which constrains the quality of a solution that is likely to be accepted. Temperature is determined by a temperature strategy which, like memory in TABU, is defined co-recursively with other parts of the system.

The choice between moving to a new solution, or remaining at the current solution is usually controlled by the following function;

$$saChoose(r,t,s,s') = \begin{cases} s', & \text{if } quality(s) \leq quality(s') \text{ or } exp(\frac{1}{t}quality(s) - quality(s')) \geq r \\ s, & \text{otherwise} \end{cases}$$

where r is a random number between 0 and 1, t is the current value of the temperature parameter, s and s' are solutions and *quality* is a function which gives the value of a solution. This is not the only function which could be used for this task, however it is the function suggested by Kirkpatrick and is still the most common. In Haskell this is implemented as follows;

```
saChoose :: (Floating v, Ord v) ⇒ (s → v) → v → v → s → s → s
saChoose quality r t s s'
  | d ≤ 0 || e > r = s'
  | otherwise = s
  where
    e = exp (−(d/t))
    d = (quality s') − (quality s)
```

This implementation takes an additional functional parameter which yields a quality value for a solution.

Simulated annealing may be implemented by lifting *saChoose* via *zipWith4*,

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

sa takes the following parameters; a function for evaluating the quality of a solution, a stream transformation that perturbs each solution in a stream, a supply of random numbers and a temperature strategy.

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 we will show and implement:

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

```
linCooling :: Floating b ⇒ b → b → [b]
linCooling tempChange startTemp = iterate (+tempChange) startTemp
```

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

```
geoCooling :: Floating b ⇒ b → b → [b]
geoCooling tempChange startTemp = iterate (*tempChange) startTemp
```

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

```
logCooling :: (Enum b, Floating b) ⇒ b → b → [b]
logCooling c d = map (λt → c/(log (t+d))) [1..]
```

We now have the appropriate combinators to show an example of their use. As with the previous strategies we must loop the stream transformer and provide it with a seed solution, and like TABU search simulated annealing does not always improve solution quality with time, so we will want to use *bestSoFar* once again.

```
exampleSA :: (Ord s, Floating v, Ord v) ⇒ (s → v) → StreamT s → v
                                                  → v → [v] → s → [s]
exampleSA quality perturbF startT propT rs
  = bestSoFar ∘ loopP (sa quality
                         perturbF
                         rs
                         (geoCooling propT startT))
```

3.4.1 Adaptive Simulated Annealing

The most common way to create variations upon the simulated annealing concept is through the cooling strategy. While simple patterns can be easily created in Haskell, here we will consider a more complex option which make use of feedback, a form of *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 minima and cease to change. In order to encourage further change the temperature must be raised. Common adaptive strategies include (i) restarting the temperature strategy and (ii) reheating the system gradually. These are often composed in terms of using one strategy until a particular trigger event is encountered before changing to an alternative strategy. This is almost identical to the concept of event driven changes to behaviours found in functional reactive programming (FRP) (Hudak *et˜al.*, 2003) and so we use a similar function.

```
until_ :: [a]          --stream of values, to place before trigger
       → [Bool]        --stream of triggers for switch over
       → [[a]]         --stream of potential futures
       → [a]
until_ (a : _) (True : _)  (_ : cs : _) = a : cs
until_ (a : as) (False : bs) (_ : cs)   = a : until_ as bs cs
```

The output of this function is a new stream consisting of, elements of the initial stream until the conditional stream contained a *True* value. At this point the replacement stream seen at that point is used to provide the remainder of the output.

Using this we can create the following function which takes a simple temperature strategy and a stream of conditionals. Each time a conditional value is found to be true, the temperature strategy will be reset.

```
restart :: [v] → [Bool] → [v]
restart basicS cs = until_ basicS cs $ map (restart basicS) (tails cs)
```

The stream of triggers can be provided in many ways, however a common method is to approximate the rate of improvement in the overall system. This will tend to zero as local minima are approached and the algorithm becomes stuck. An approximation of the rate of improvement can be found by comparing values over an interval of the process. The action of finding these intervals over a stream is the *window* function seen in TABU search, and so we will reuse it. While this is costly its simplicity makes it suitable for this example. Putting it together,

```
restartingSA :: (Ord s, Floating v, Ord v) ⇒ Int → v → v → (s → v) → StreamT s
                                                → [v] → s → [s]
restartingSA wSize startT propT getVal perturbF rs1 seed
  = let cs = map (λw → if null w then False else head w ≡ last w) $
             window wSize sols
        ts = restart (geoCooling propT startT) cs
        sols = loopP (sa getVal perturbF rs1 ts) seed
    in bestSoFar sols
```

3.5 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 we will show that stream transformations can be used here to tease apart the steps of the process into finer grained combinators.

A standard imperative description of a genetic algorithm is as follows:

- construct an initial population of candidate solutions
- repeat the following steps;
  - — randomly select pairs of solutions from the population (often called *parent solutions*), giving preference to the better solutions and recombine them to yield a new collection of candidate solutions
  - — randomly select some set of the new solutions and perturb them (in this context usually known as *mutation*)
  - — replace the previous population with the candidate solutions that have been created

Our reformulation of genetic algorithms is based upon the intuition that the stream of populations can be seen as a division of a stream of solutions into regular blocks. Conversely a stream of solutions can be seen as the concatenation of a stream of populations. The window operation seen in TABU could be used to create this regular chunking process, however we have found it is easier to create a new operation we will call chunk which simply divides an input stream into a stream of equally sized lists [2];

$$chunk :: Int \to ExpandT\ s$$
$$chunk\ sz = unfoldr\ (Just \circ splitAt\ sz)$$

It is possible to describe a genetic algorithm as a stochastic stream transformation over populations, and might have the following data type; $popTrans :: RandomGen\ g \Rightarrow g \to StreamT\ [s]$ This would give rise to the following sketch of a skeleton for genetic algorithms; $concat \circ popTrans\ g \circ chunk\ popSize$

However this approach places a great deal of emphasis upon the population transformation concept and we have found that breaking down the operations into a series of transformations provides greater flexibility of expression. We start with a sketch of a recombination function, which constructs a new solution from a collection of *parent* solutions. This has the type $[s] \to s$. To operate over stream the type of the recombination transformer becomes $ContraT\ s$, which takes a stream of collections of parents and gives back a stream of solutions. These functions will usually be problem specific.

Working backwards, we need a transformation which takes a stream of populations and provides a stream of collections of parents. The selection of a single parent from a collection is of the form $ContraT\ s$. To use a $ContraT\ s$ to select a group of parents we provide the function *manySelect*;

$$manySelect :: Int \to ContraT\ s \to StreamT\ [s]$$
$$manySelect\ sz\ f = chunk\ sz \circ f \circ concatMap\ (replicate\ sz)$$

[2] We are not sure why such a function is not present in Haskell's standard list manipulation libraries as it seems to us to be a common enough process.

The most common approach to selecting parents from collections in genetic algorithms is a stochastic process favouring the better candidate solutions. Many patterns could be used for the selection of candidates, such as uniform or skewed probability distributions, of which the standard approaches are only specialisations. We capture this generality using;

```
select :: Ord r ⇒ [r] → r → [s] → s
select dist r = snd ∘ head ∘ dropWhile ((r >) ∘ fst) ∘ zip dist
```

This is then lifted to operate over streams;

```
streamSelect :: Ord r ⇒ [r] → [r] → ContraT s
streamSelect dist = zipWith (select dist)
```

We compose these functions to give the function *gaSelect*, a stream transformation from populations to sets of parents, based upon a parameterising distribution.

```
gaSelect :: Ord r ⇒ Int → [r] → [r] → StreamT [s]
gaSelect sz dist rs = manySelect sz (streamSelect dist rs)
```

The use of distributions in this way has some echos of (Erwig & Kollmansberger, 2006) work in probabilistic modelling. It is still open whether there are opportunities for convergence between their approach and this library.

These selection combinators select from collections based upon probability distributions rather than upon solution quality. To capture the concept common in genetic algorithms, that good solutions should be more likely to be used, we require an appropriate probability distribution, and that the population is **sorted** by quality. We modify the process for creating the stream of populations, so that populations are ordered. We will also wish to generate several solutions from each population, where *gaSelect* will only provide one. This can be handled by replicate, in a similar way to the operation of *manySelect*.

```
makePop :: Ord s ⇒ Int → ExpandT s
makePop s = concatMap (replicate s ∘ sort) ∘ chunk s
```

The final component of genetic algorithms is mutation. This can be modelled as a stream transformation which only effects a substream of solutions. To achieve this we divide the stream of recombined solutions into two parts, apply the mutation transformation to one part and then reintegrate the two streams. We call the operator that applies a transformation to a substream *nest*, and indicate the substream by a stream of booleans.

```
nest :: [Bool] → StreamT s → StreamT s
nest bs tr = join bs ∘ zipWith ( $ ) [id, tr] ∘ divide bs

divide :: [Bool] → ExpandT s
divide bs xs = [[x | (b,x) ← zip bs xs, b ≡ i] | i ← [False, True]]

join :: [Bool] → ContraT s
join bs xss = unfoldr f (bs, xss)
  where
    f (False : ts, [x : xs, ys]) = Just (x, (ts, [xs, ys]))
    f (True : ts, [xs, y : ys]) = Just (y, (ts, [xs, ys]))
```

Let *tsp_perturb* be a perturbation stream transformation, it can be modified to only modify every other solution in an input stream using the following code fragment;

```
nest (cycle [False, True]) tsp_perturb
```

Alternatively we can modify solutions in a stream with a given probability p, with respect to a provided stream of randomly generated floats, like so;

$$nestWithProb :: (Ord\ r, Floating\ r) \Rightarrow [r] \rightarrow r \rightarrow StreamT\ s \rightarrow StreamT\ s$$
$$nestWithProb\ rs\ p = nest\ (map\ (< p)\ rs)$$

Our complete reformulation can be seen together in Figure 6. The application of *loopS* to this structure feedsback *evolved population one* into *population two*.

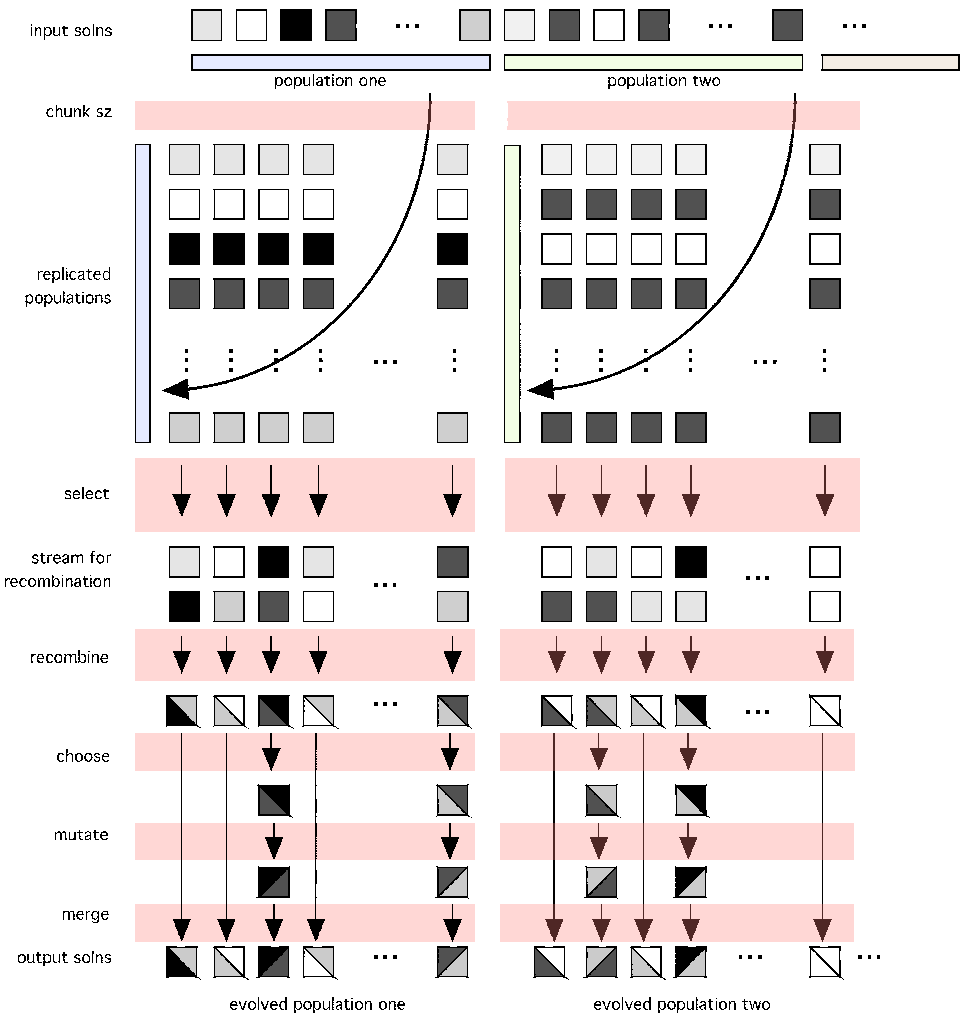


Fig. 6: Model of stream based implementation of genetic algorithms

Each of these groups of actions are composed to create the common skeleton of genetic algorithms;

$$ga :: ExpandT\ s \rightarrow ContraT\ s \rightarrow StreamT\ s \rightarrow StreamT\ s$$
$$ga\ mkPopulations\ recomb\ mutat = mutat \circ recomb \circ mkPopulations$$

We use the combinators described to give this implementation of genetic algorithms which is less general than the skeleton but provides more recognisable parameters;

$$
\begin{aligned}
&\mathit{gaConfig} :: \mathit{Ord}\ s \Rightarrow [\mathit{Float}] \to \mathit{Int} \to [\mathit{Float}] \to [\mathit{Bool}] \\
&\qquad \to \mathit{ContraT}\ s \to \mathit{StreamT}\ s \to \mathit{StreamT}\ s \\
&\mathit{gaConfig}\ \mathit{dist}\ \mathit{popSize}\ \mathit{rs}\ \mathit{bs}\ \mathit{recombine}\ \mathit{mutate} \\
&\quad = \mathit{ga}\ (\mathit{makePop}\ \mathit{popSize}) \\
&\qquad\quad (\mathit{recombine} \circ \mathit{gaSelect}\ 2\ \mathit{dist}\ \mathit{rs}) \\
&\qquad\quad (\mathit{nest}\ \mathit{bs}\ \mathit{mutate})
\end{aligned}
$$

These combinators provide a concise yet flexible way to build genetic algorithms. Variations upon the logic used for each of the combinator groups can easily be created, for example:

- the creation of populations using variable population sizes;
- mutation transformations being provided by iterative improvers from section 3.2; and
- the use of temperature strategies from simulated annealing to vary the mutation rate.

3.6 Application to TSP

We demonstrate the application of our combinators by presenting a short investigation into one form of hybrid metaheuristic for the TSP problem `fl417`[3]. As metaheuristic we adopt an approach used by (Suh & Van Gucht, 1987) which is a genetic algorithm where the standard mutation operation is replaced with an iterative improvement technique.

The ease of hybridisation is illustrated through a series of algorithms constructed using our combinators. The root strategy is a genetic algorithm with the following common parameters.

- a selection distribution such that the first solution in the ordered population will be selected with the highest likelihood, the second solution with some lower likelihood and so on; this has been provided using the geometric relationship $p_n = 0.25 * 1.0595^n$, encoded as an iteration;
- a population size of 100;
- a stream of booleans to indicate positions for mutation, and mutation will take place with a 40% likelihood;
- the stochastic recombination function from section 2.3, rewritten as a stream transformer;

These are fixed in order to limit the variations in configuration of each system, however in general these provide additional dimensions for tuning the metaheuristics.

Each of these hybrid algorithms is processed by the *bestSoFar* combinator to ensure that the solution being examined was the best seen. The seed solutions used are chosen with a uniform likelihood over all potential sequences.

The results presented are the quality (length of the Hamiltonian cycle) of the best solution discovered after a fixed number of iterations, rather than a fixed number of seconds of

[3] `fl417` is the example problem we have used throughout this paper, a symmetric TSP drawn from the TSPLIB(Reinelt, 1991)

processor time. This is to separate the evaluation of the metaheuristics from implementation issues which influence measurements using time.

Results were gathered by sampling the stream of constructed solutions at the following points; `0`, `1000`, `3000`, `5000`, `7000` and `10000`. Due to the stochastic nature of these algorithms, each test was run 25 times and the mean of the results at each sampling point is reported [4].

Each variation is shown below as a code fragment A complete program would require a harness for creating the random number generators, loading the problem file and creating an initial set of solutions for the algorithm to run over. These are problem specific and mundane tasks.

3.6.1 The Algorithms

We initially present a version of genetic algorithms which only recombines (we name this TS1) and has no form of mutation, illustrating the simplest parametrisation. In the following code the variables *gN where N* $\leftarrow [0..]$ are random number generators obtained from the calling context;

(TS1)

```
loopS $ gaConfig (iterate (*1.0595) 0.25)          --selection distribution
                 100                               --population size
                 (randoms g1)                      --random number source
                 (map (< (0.4)) ∘ randoms $ g2)    --mutation likelihood
                 (stochasticRecombine g3)          --recombination
                 id                                --mutation
```

and the standard version of genetic algorithms which uses a stochastic mutation process, which is a random swap of two cities in the Hamiltonian cycle;

(TS2)

```
loopS $ gaConfig (iterate (*1.0595) 0.25) 100
                 (randoms g1)
                 (map (< (0.4)) ∘ randoms $ g2)
                 (stochasticRecombine g3)
                 (stochasticMutate g4)
```

These two provide the baselines of the hybridisation experiments.

Following (Suh & Van Gucht, 1987) we replace the stochastic mutation operation with a stochastic iterative improvement operation;

(TS3)

```
loopS $ gaConfig (iterate (*1.0595) 0.25) 100
                 (randoms g1)
                 (map (< (0.4)) ∘ randoms $ g2)
                 (stochasticRecombine g3)
                 (stochasticii (!!) (randomRs (0,416) g4)
                                    (map adjacentExchangeN))
```

[4] The standard deviations of the results have been examined but found to be low and so are not shown.

However this was not found to be more effective than variant TS2.

We hypothesised that the issue lay with the neighbourhood function, which was constraining the potential of the iterative improver. Two further variations were devised;

- An alternative form of stochastic iterative improvement, which provides the stochastic aspect through a variable neighbourhood rather than from a fixed neighbourhood function. This variable neighbourhood is provided by generating a selection of random swaps of cities in a given solution. The iterative improvement is then replaced with a maximal iterative improver so that the mutation process will move to the best possible neighbour at each step;
  (TS4)

```
loopS $ gaConfig (iterate (*1.0595) 0.25) 100
                 (randoms g1)
                 (map (< (0.4)) ∘ randoms $ g2)
                 (stochasticRecombine g3)
                 (maximalii $ stochasticNeighbourhood 417 g4)
```

- An algorithm that uses two mutation operations independently. The first chosen was a maximal iterative improvement meta-heuristic to provide an improvement mutation. The second was the original purely stochastic mutation operation.
  In order to hybridise these, we ceased to use *gaConfig* and instead used the more general *ga* combinator. The mutators were given different likelihoods and then composed together to provide the final mutation operation. The construction of the full meta-heuristic follows the structure seen in section 3.5
  (TS5)

```
pattern1 = map (< (0.35) ∘ randoms $ g1
pattern2 = map (< (0.05)) ∘ randoms $ g2
iiMutator = nest pattern1 (maximalii (map adjacentExchangeN))
swapMutator = nest pattern2 $ stochasticMutate g3
loopS $ ga(makePop 100)
           (stochasticRecombine g4 ∘ gaSelect 2 (iterate (*1.095) 0.25)
                                                (randoms g5))
           (iiMutator ∘ swapMutator)
```

These ideas were found to provide stronger results than using a more traditional stochastic iterative improvement method, with the variable neighbourhood being found to be the strongest. The results from these experiments are shown in Figure 7 and summarised in Table 1.

3.7 Additional Combinatorial Problems

Application of our meta-heuristic combinators to different problems is a matter of designing a data structure for the problem which supports the *neighbourhood*, *perturbation* and *recombination* functions that the programmer wishes to work with. We have experimented with two other well known combinatorial problems, *satisfiability* (SAT-3) and *timetabling*. The libraries for loading the file formats used, storing and manipulating candidate solutions can be found in the Hackage library *combinatorial-problems*. The approach taken is similar

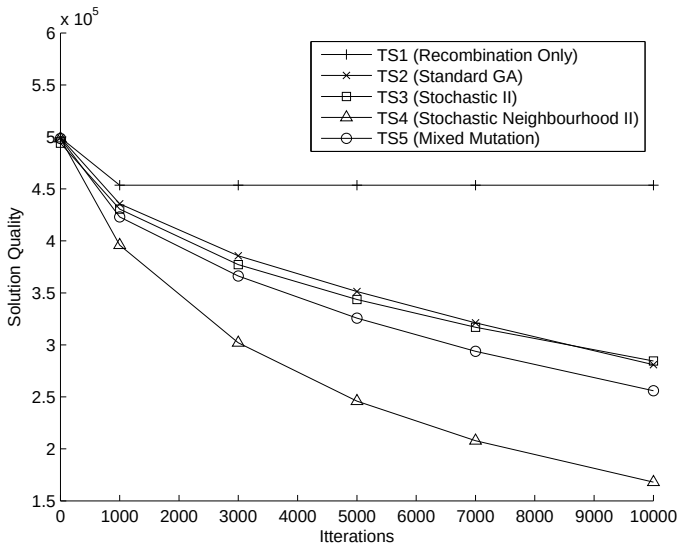


Fig. 7: A plot comparing the performance of the baseline strategies

	5000 iterations	10000 iterations
TS1 (Mutation Only Baseline)	453566	453566
TS2 (Standard GA)	351359	280998
TS3 (Stochastic II Mutate)	343620	284571
TS4 (Variable Neighbourhood)	245891	167938
TS5 (Mixed Mutate)	325671	255854

Table 1: A summary of the solution qualities discovered by various hybrid strategies

to that taken for TSP, using a number of dictionaries to store the data, which provides a balance of speed, memory sharing and stateless manipulation of the data structures.

3.7.1 SAT-3

Satisfiability is a well known problem and we feel it unnecessary to describe it in detail here. For our purposes it was reformulated from its usual description as a decision problem to an optimisation problem by seeking candidate solutions which satisfied greater and greater numbers of *clauses*. A large number of example problems can be found in the online repository SATLIB (Hoos & Stützle, 2000).

3.7.2 Timetabling

The timetabling problems considered are a specific type described by the international timetabling competition (Metaheuristic Network, 2007). Due to the large number of constraints and interactions between constraints in this form of problem, we found that a naive approach was not very successful. Many candidates considered were not valid and the metaheuristics made slow progress towards candidates which satisfied the constraints. We feel that a constraint based system, using metaheuristics to choose new constraints to add at each point, in a similar manner to that used by other authors may be of interest (Van Hentenryck & Michel, 2005), however this has not been explored at this time.

4 Design Perspectives

The approach that we have used for the expression of these metaheuristics is not the only approach that can be taken, nor was it our exclusive line of investigation. Here we will briefly describe the other methods that we experimented with and look at why we have settled on the stream transformation style.

4.1 Monolithic State

The traditional description of meta-heuristic algorithms is as stateful imperative systems. The most direct translation of such models into functional languages is to model them as state transformation systems. This can have the advantage of improving performance, especially when used in conjunction with monads that offer greater opportunities for compiler optimisations.

However monolithic state fits awkwardly with the desire for generic combinatorial design. Consider the following;

$$
\begin{aligned}
&\mathit{gloverTABU} :: \mathit{Ord}\ s \Rightarrow (s \to [s]) \to \mathit{Int} \to s \to [s] \\
&\mathit{gloverTABU}\ \mathit{nf}\ \mathit{sz} = \ \mathit{unfoldr}\ \mathit{innerTABU} \circ (\lambda x \to ([\,]\,, x)) \\
&\quad \textbf{where} \\
&\qquad \mathit{innerTABU}\ (\mathit{tabuList}, x) \\
&\qquad\quad = \textbf{let}\ \mathit{tabu}' = \ \mathit{take}\ \mathit{sz}\ \$\ [x]\ +\!\!+ \mathit{tabuList} \\
&\qquad\qquad\quad\ x' = \ \mathit{maximum}\ \$\ \mathit{filter}\ (\mathit{flip}\ \mathit{notElem}\ \mathit{tabu}')\ (\mathit{nf}\ x) \\
&\qquad\quad\ \textbf{in}\ \mathit{Just}\ (x, (\mathit{tabu}', x'))
\end{aligned}
$$

This implementation has two components that *could* be considered state, the seed solution at each step *x*, and the TABU list *tabuList*. Many variations make use of further factors to assist in decision making, for example as a statistical summary of the recently explored solutions as seen in Adaptive Simulated Annealing (see 3.4.1). The most common variations, which use random number generators, include; use of a stochastic neighbourhood, a stochastic choice of neighbour, and Talliard's *Robust Taboo search* which stochastically shortens the TABU list at each step. Each of these introduces additional state data which must be threaded through the process, and this requires the modification of the type signatures of the associated functions. This does not lend itself to the flexible expression and recombination of modular components

Some of the problems encountered with the naive approach can be alleviated by making use of Haskell's type classes to provide common interfaces to dissimilar state components.

However the programmer must still define their own data types for the monolithic state and provide the code for accessing each component.

These problems could be further alleviated by an extensible record system for Haskell. Such ideas have been considered and discussed but have not been finalised at the time of writing (has, 2011; Jones & Peyton Jones, 1999).

We feel the stream transformation approach that we have chosen, while not quite as fast as using a statically defined state in this way, does provide better composability and expressiveness.

4.2 Co-Monads

The stream transformation style that has been presented has evolved from, and encapsulates, an underlying breakdown of the meta-heuristic algorithms into a dataflow structure. (Uustalu & Vene, 2005) propose that co-monads provide a good framework for data flow programming, supported by category theoretic semantics. However although co-monads may provide a theoretically neat setting for meta-heuristic combinators, an implementation of code from (Uustalu & Vene, 2005) suffered from performance issues and space leaks.

4.3 Functional Reactive Programming

Functional Reactive Programming (Hudak, 2000; Hudak *et˜al.*, 2003) models systems as continuous behaviours and discrete events. We first considered FRP as a possibility while looking at simulated annealing, itself based upon continuous models of physical systems. While metaheuristics are better thought of as discrete step algorithms rather than behaviours over continuous time, we found that FRP was a good way to decompose monolithic algorithms into modular blocks.

For example, consider this mathematical model of simulated annealing;

$$r(i) = \begin{cases} randomSeed & \text{if } i \leq 0 \\ next(r(i-1)) & \text{otherwise} \end{cases}$$

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

$$sa(i) = \begin{cases} seedSolution & \text{if } i \leq 0 \\ sa(i-1) & \text{if not } accepted \\ permute(sa(i-1), r(i*2)) & \text{otherwise} \end{cases}$$

$$accepted = accept(permute(sa(i-1), r(i*2)), t(i), r(i*2+1))$$

This model provides modularity through the clean separation of the various behaviours. For example, to change the cooling strategy it is only required to modify function *t*, the other functions and the more general pattern of the simulated annealing algorithm remain unchanged.

A naive implementation of these functions will be inefficient due to the recomputation of intermediate values. Memoization can be used to fix this issue 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 as can be seen in (Elliott, 2009) and the *Reactive* library (Elliott, 2010). However at the present time this work is not mature enough for us to build upon.

4.4 Arrows

Arrows are a general model of computation proposed by Hughes (Hughes, 2000). We can implement our stream transformers as arrows operating over streams of values; i.e. $Arrow\ a \Rightarrow a\ [b]\ [c]$. There are difficulties in using them in our context; for example *loopS* is awkward to express in an arrowised form, however this can be done and is presented as *arrowFix*;

$$\begin{aligned}
&arrowFix :: ArrowLoop\ a \Rightarrow a\ [b]\ [b] \rightarrow a\ [b]\ [b] \\
&arrowFix\ f = loop\ (uncurryA\ (+)) \ggg f \ggg dup)
\end{aligned}$$

$$\begin{aligned}
&dup :: Arrow\ a \Rightarrow a\ s\ (s,s) \\
&dup = returnA\ \&\&\&\ returnA
\end{aligned}$$

$$\begin{aligned}
&uncurryA :: Arrow\ a \Rightarrow (b \rightarrow c \rightarrow d) \rightarrow a\ (b,c)\ d \\
&uncurryA = arr \circ uncurry
\end{aligned}$$

All the stream transformations previously seen can be lifted to an arrowised form, or rewritten in terms of arrow combinators, for example;

$$\begin{aligned}
&improvementA :: (Arrow\ a, Ord\ s) \Rightarrow a\ [s]\ [[s]] \rightarrow a\ [s]\ [[s]] \\
&improvementA\ nfa = returnA\ \&\&\&\ nfa \ggg \\
&\qquad\qquad uncurryA\ (zipWith\ (\lambda a\ b \rightarrow filter\ (<a)\ b))
\end{aligned}$$

$$\begin{aligned}
&iterativeImproverA :: (Arrow\ a, Ord\ s) \Rightarrow a\ [s]\ [[s]] \rightarrow a\ [[s]]\ [s] \rightarrow a\ [s]\ [s] \\
&iterativeImproverA\ nf\ fChoice = improvementA\ nf \ggg fChoice
\end{aligned}$$

$$\begin{aligned}
&firstFoundiiA :: (Arrow\ a, Ord\ s) \Rightarrow a\ [s]\ [[s]] \rightarrow a\ [s]\ [s] \\
&firstFoundiiA\ nf = iterativeImproverA\ nf\ (arr\ \$\ map\ head)
\end{aligned}$$

However in general we do not see clear benefits in using Arrows rather than the stream transformation functions previously presented, either in terms of algorithm expression or performance.

5 Implementation issues

Our implementation is purely functional, however use of lazy evaluation can cause problems. Consider the following sketch of a simulated annealing program;

$$> bestSoFar\ (loopP\ sa)\ !!\ 10000$$

The runtime environment will attempt to construct the computation for the 10000^{th} element, which requires the computation for the previous element, and so on until it reaches the original seed. It therefore creates the entire computation before it begins the evaluation, leading to a very large (and unacceptable) thunk build up.

Various methods exist that can alleviate this problem. Indeed the problem was not encountered in our initial tests because we displayed each candidate as it was created, thus *pushing* the computation forward and avoiding the thunk build up. We therefore created

the following function, modelled after the implementation of *foldl'*, which acts as the list index operator but pushes the computation of the list elements.

$$(!!!) :: [a] \to Int \to a$$
$$(!!!) \sim (x : \_)\ 0 = x$$
$$(!!!) \sim (x : xs)\ n = x `seq` (xs `seq` xs!!!(n-1))$$

Given the dependencies between solutions in such a sequence, this is sufficient to resolve the problem. The following function has also been found to be of use;

$$indexWithRemainder :: [a] \to Int \to (a, [a])$$
$$indexWithRemainder \sim (x : xs)\ 0 = (x, xs)$$
$$indexWithRemainder \sim (x : xs)\ n = x `seq` (indexWithRemainder\ xs\ (n-1))$$

This pushes the computation as far as the n^{th} value, before returning the value and the remaining list. The list returned is not forced, so it is a computation that can be continued if desired, such as in an interactive system[5].

6 A Performance comparison with C

Previously when comparing metaheuristics in section 3.6, results were given in terms of quality of solution against the number of iterations. However when comparing to implementations built in C we wish to test *how long* it takes to perform a number of steps of the algorithm for each version. This provides a clear separation between comparing the logic of algorithms (previously) and the performance of the implementation.

Comparing performance of our Haskell programs with a more mainstream approach proved difficult. We struggled to find standard implementations of the metaheuristics that could be used to generate solutions using precisely the same logic as ours do, and so give a fair comparison.

In order to match our Haskell implementations we built our own versions of the test metaheuristics in C. The versions that we created were more specialised and much less flexible than those provided by the Haskell libraries.

The test problem was the TSP problem `fl417` again. We chose to use a version of simulated annealing and an iterative improver. The simulated annealing version had the following properties; geometric cooling strategy starting at the value `80000` with a reduction rate of `0.99`, the standard simulated annealing acceptance function and a perturbation function which randomly selected and swapped two cities in the sequence. The iterative improver was a maximal improver with an adjacent exchange neighbourhood and random restart when local minima were encountered. All parameters were set to the same values in both the C and Haskell versions.

The random number generator used for the C version was from the standard `stdlib` library; but it was found that using the *merzenne-pure64* library in Haskell was faster than the standard *System.Random* library. All generators were initialised from the system clock on each run. In order to improve the Haskell versions performance, some of the floating point

[5] Our functions can also be seen as more specialised implementations of strategies seen in the library *Control.Seq*.

computations in simulated annealing were rewritten to use the *Foreign.C.Types.CDouble* data type, and conversions were kept to a minimum.

The tests of the programs were monitored using the Unix time function. When executed each program was sampled at the following numbers of iterations: $1000 * 2^n$ where n in 0..8. Each test was carried out 25 times and the averages are reported[6].

All four programs showed linear time complexity in the number of iterations as expected. Figure 8 shows the ratios of the runtimes of the Haskell and C versions of the programs. The Haskell version of simulated annealing tends towards a factor of 8 times slower than the C version, while the Haskell maximal iterative improver is approximately 20 times slower than its counterpart. We note the odd shape of the graph for the ratio of the Iterative Improver implementations, and we suspect that it is caused by a combination of timing precision at very low numbers of iterations (where the C programs are very fast) and start-up costs. In general however the trend is towards a constant ratio between the implementations.

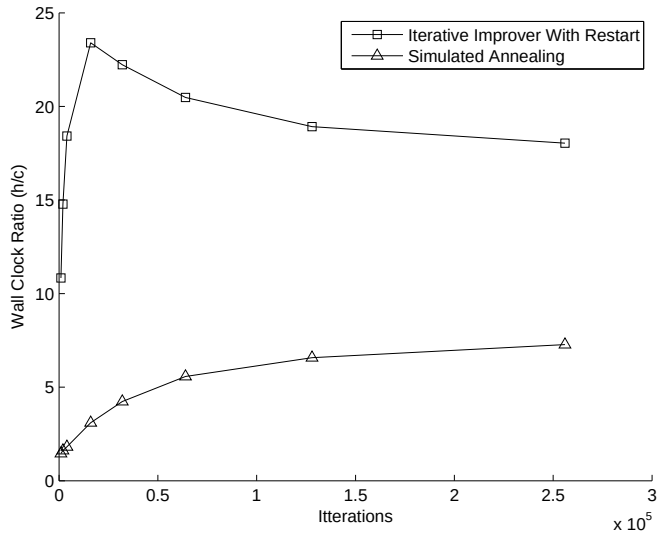


Fig. 8: Performance graph for simulated annealing on the `fl417` problem

While the aim of this work has not been raw performance, a preliminary analysis of the execution of the iterative improvement and simulated annealing programs has been done. This has indicated one key area for refinement, common to both demonstration programs; the perturbation of solutions.

One performance bottleneck is the data structure used for candidate solutions. Selecting a model involves a trade off between:

- time complexity of individual perturbation methods;

[6] The standard deviations of the results have been examined but found to be low and so are not shown.

- generality (the number and variety of recombination and perturbation functions that can be implemented with 'reasonable' efficiency);
- persistence, i.e. capturing history.

Specialisation of the state representation for specific perturbation functions is an obvious and important route to improving run-time performance, and for this reason have been a significant topic of interest within Operations Research for many years.

7 Further Work & Conclusion

There has been little previous concerted study of metaheuristics in Haskell, though on Hackage a number of implementations of specific metaheuristics can be found, e.g. Simulated annealing (Wasserman, 2010) and Genetic Algorithms (Hoste, 2011). In this paper we have selected a range of these algorithms and shown how they can be captured with a common pattern of computation, the stream transformer, and how this gives rise to a library of combinators.

We have then shown how each meta-heuristic may be rebuilt using the combinator library, and how variations upon them may be easily expressed. We have shown how the space leak problem that occurs when using basic Haskell functions such as !! for interacting with the streams can be avoided by using eager versions of the functions. The authors are not aware of any similar framework or library in a functional setting.

Other general frameworks for metaheuristics have been created in imperative languages, or as domain specific languages. (Masrom *et˜al.*, 2011) compares these other approaches and highlighted the lack of commonality and easy interoperability between population based methods (such as genetic algorithms) and single solution based techniques. Our approach uses standard transformation methods and combinations to express these different algorithm families, and hence allow interoperability, which we feel is a key advantage and topic worthy of further investigation.

There are trade-offs which must be made when choosing how to explore a combinatorial problem. Our approach suffers from being substantially slower than specialised C code for the same problem. It is quite possible to write faster code within Haskell itself, using in-place updates of data structures for example. However neither of these alternatives gives rise to a system with the same level of flexibility, which we feel gives our combinators a useful niche as a prototyping system for exploring the design space for metaheuristics. This is of great use in the operations research community where design of metaheuristics remains a labour intensive experimental process.

In the future we see this flexibility allowing the construction of a higher level framework for automated experimentation with metaheuristics. For example the automated evaluation of different combinations of transformers with regards to how they act upon a new problem. Further automation of the process would have the framework search the patterns of combinations of transformers for metaheuristics which provide superior performance upon a new class of problems. This approach to metaheuristics is known by the name of *hyper-heuristics* (Denzinger *et˜al.*, 1997), with work usually begin done using custom built systems, frameworks for other languages and domain specific languages.

The extension of the library to provide general methods for experimenting with metaheuristics, and testing sets of combinators forms one of two planned extensions to this

work. The second is the development of a new group of metaheuristics for the library, specifically ant-colony optimisation (Dorigo *et˜al.*, 1991).

It is also of some interest to investigate further improvements in expressiveness through the use of Template Haskell (Sheard & Peyton Jones, 2002; Lynagh, 2012). This would allow for the use of more general versions of *zip* and *zipWith* which adapt to the number of input streams they are provided with, known as *zipN* and *zipWithN*. We are also interested in the possibility of more sophisticated lifting operations, facilitated through Template Haskell.

Our other interests lie in improving the performance of our system. This can be achieved by looking for better rewriting and compilation rules so that our combinators give better serial performance, or investigation into parallelism, perhaps taking advantage of the pipeline like nature of our combinators to give more natural access to multi-core processors.

References

(2011). *Extensible Record: HaskellWiki*. Accessed 20.02.12. http://www.haskell.org/haskellwiki/Extensible_record.

Birattari, Mauro. (2005). *Tuning Metaheuristics: A Machine Learning Perspective*. First edn. Springer.

Birattari, Mauro, Paquete, Luis, Stützle, Thomas, & Varrentrapp, Klaus. (2001). *Classification of Metaheuristics and Design of Experiments for the Analysis of Components*. Tech. rept. AIDA-2001-05. Intellektik, Technische Universität Darmstadt, Germany.

Černý, V. (1985). A Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, **45**(1), 41–51.

Chrobak, M., Szymacha, T., & Krawczyk, A. (1990). A data structure useful for finding Hamiltonian cycles. *Theoretical Computer Science*, **71**, 419–424.

Denzinger, Jörg, Fuchs, Marc, & Fuchs, Matthias. (1997). High performance ATP systems by combining several AI methods. *Pages 102–107 of: Proceedings of the 15th International Joint Conference on Artifical Intelligence*, vol. 1. Morgan Kaufmann.

Dorigo, Marco, Maniezzo, V., & Colorni, Alberto. (1991). *Positive Feedback as a Search Strategy*. Tech. rept. Politecnico di Milano.

Elliott, Conal M. (2009). Push-pull functional reactive programming. *Pages 25–36 of: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. Haskell '09. ACM.

Elliott, Conal M. (2010). *Hackage DB: Reactive package*. Accessed 20.02.12. http://hackage.haskell.org/package/reactive.

Erwig, Martin, & Kollmansberger, Steve. (2006). Functional pearls: Probabilistic functional programming in haskell. *J. funct. program.*, **16**, 21–34.

Fredman, M. L., Johnson, D. S., McGeoch, L. A., & Ostheimer, G. (1993). Data structures for traveling salesmen. *Pages 145–154 of: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*. SODA '93. Society for Industrial and Applied Mathematics.

Gendreau, Michel, & Potvin, Jean-Yves. (2005). Metaheuristics in Combinatorial Optimization. *Annals of Operations Research*, **140**(1), 189–213.

Glover, F. (1989). Tabu Search, Part I. *ORSA Journal on Computing*, **1**(3), 190–206.

Glover, F. (1990). Tabu Search, Part II. *ORSA Journal on Computing*, **2**(1), 4–32.

Goldberg, David E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.

Hoos, Holger, & Stützle, Thomas. (2000). SATLIB: An Online Resource for Research on SAT. IOS Press. SATLIB is available online at www.satlib.org.

Hoos, Holger, & Stützle, Thomas. (2005). *Stochastic local search: Foundations & applications*. Morgan Kaufmann Publishers Inc.

Hoste, Kenneth. (2011). *Hackage DB: GA package*. Accessed 20.02.12. http://hackage.haskell.org/package/GA.

Hudak, Paul. (2000). *The haskell school of expression - learning functional programming through multimedia*. New York: Cambridge University Press.

Hudak, Paul, Courtney, Antony, Nilsson, Henrik, & Peterson, John. (2003). Arrows, Robots, and Functional Reactive Programming. *Pages 159–187 of: Advanced Functional Programming, 4th International School, volume 2638 of LNCS*. Springer-Verlag.

Hughes, John. (1989). Why functional programming matters. *The computer journal*, **32**, 98–107.

Hughes, John. (2000). Generalising monads to arrows. *Science of Computer Programming*, **37**, 67–111.

Hutton, Graham, & Meijer, Erik. (1998). Monadic Parsing in Haskell. *Journal of Functional Programming*, **8**(4), 437–444.

Jones, Mark P., & Peyton Jones, Simon. (1999). Lightweight Extensible Records for Haskell. *In haskell workshop*.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. *Science*, **220**(4598), 671–680.

Launchbury, John, & Peyton Jones, Simon. (1995). State in Haskell. *Pages 293–341 of: LISP and Symbolic Computation*, vol. 8. Kluwer Academic Publishers.

Lynagh, Ian. (2012). *Hackage DB: template-haskell package*. Accessed 20.02.12. http://hackage.haskell.org/package/template-haskell.

Masrom, S., Abidin, Siti Z.Z., Hashimah, P. N., & Rahman, A. S. Abd. (2011). Towards Rapid Development of User Defined Metaheuristics Hybridisation. *International Journal of Software Engineering and Its Applicatons*, **5**.

Metaheuristic Network. (2007). *International Time Tabling Competition*. Organised by eventMAP research Group at Queen's University with partners from Cardiff University, Napier University, University of Nottingham and the University of Udine, Accessed 20.02.12. http://www.cs.qub.ac.uk/itc2007/.

Okasaki, Chris. (1998). *Purely functional data structures*. Cambridge University Press.

Raidl, Günther R. (2006). A Unified View on Hybrid Metaheuristics. *Pages 1–12 of:* Almeida, Francisco, Aguilera, María˜J. Blesa, Blum, Christian, Moreno-Vega, J. Marcos, Pérez, Melquíades Pérez, Roli, Andrea, & Sampels, Michael (eds), *Hybrid metaheuristics, third international workshop, hm 2006, gran canaria, spain, october 13-15, 2006, proceedings*. Lecture Notes in Computer Science, vol. 4030. Springer-Verlag.

Reinelt, Gerhard. (1991). TSPLIB - A Traveling Salesman Problem Library. *INFORMS Journal on Computing*, **3**, 376–384. http://comopt.ifi.uni-heidelberg.de/ software/TSPLIB95/.

Schrijvers, Tom, Tack, Guido, Wuille, Pieter, Samulowitz, Horst, & Stuckey, Peter. (2011). Search Combinators. *Pages 774–788 of: Principles and Practice of Constraint Programming, 17th International conference, Proceedings*. Springer.

Sheard, Tim, & Peyton Jones, Simon. (2002). Template meta-programming for Haskell. *ACM SIGPLAN Notices: PLI Workshops*, **37**(12), 60–75.

Suh, Jung Y., & Van Gucht, Dirk. (1987). Incorporating heuristic information into genetic search. *Pages 100–107 of: Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. L. Erlbaum Associates Inc.

Taillard, E. (1991). Robust Taboo Search for the Quadratic Assignment Problem. *Parallel Computing*, **17**, 443–455.

Talbi, E.G. (2002). A Taxonomy of Hybrid Metaheuristics. *Journal of Heuristics*, **8**, 541–564.

Uustalu, Tarmo, & Vene, Varmo. (2005). The Essence of Dataflow Programming. *Pages 2–18 of: Lecture Notes in Computer Science*. Springer-Verlag.

Van Hentenryck, Pascal, & Michel, Laurent. (2005). *Constraint-based local search*. The MIT Press.

Wasserman, Louis. (2010). *Hackage DB: concurrent-sa package*. Accessed 20.02.12. http://hackage.haskell.org/package/concurrent-sa.

A Combinators of the library

$type\ StreamT\ s = [s] \to [s]$	page 11
$type\ ExpandT\ s = [s] \to [[s]]$	page 11
$type\ ContraT\ s = [[s]] \to [s]$	page 11
$loopS :: StreamT\ s \to StreamT\ s$	page 11
$loopP :: StreamT\ s \to s \to [s]$	page 11
$bestSoFar :: Ord\ s \Rightarrow StreamT\ s$	page 14
$chunk :: Int \to ExpandT\ s$	page 18
$lift :: (t \to b \to c) \to (a \to t) \to [a] \to [b] \to [c]$	page 13
$until\_ :: [a] \to [Bool] \to [[a]] \to [a]$	page 17
$nest :: [Bool] \to StreamT\ s \to StreamT\ s$	page 19
$divide :: [Bool] \to ExpandT\ s$	page 19
$join :: [Bool] \to ContraT\ s$	page 19
$manySelect :: Int \to ContraT\ s \to StreamT\ [s]$	page 18

A.1 Iterative Improver Combinators

$improvement :: Ord\ s \Rightarrow ExpandT\ s \to ExpandT\ s$	pages 12 and 13
$iterativeImprover :: Ord\ s \Rightarrow ExpandT\ s \to ContraT\ s \to StreamT\ s$	page 12
$firstFoundii, maximalii, minimalii :: Ord\ s \Rightarrow ExpandT\ s \to StreamT\ s$	page 12
$stochasticii :: Ord\ s \Rightarrow ([s] \to r \to s) \to [r] \to ExpandT\ s \to StreamT\ s$	page 13

A.2 TABU Search Combinators

$tabu :: Ord\ s \Rightarrow ExpandT\ s \to ExpandT\ s \to ContraT\ s \to StreamT\ s$	page 14
$window :: Int \to ExpandT\ s$	page 14
$varyWindow :: RandomGen\ g \Rightarrow (Int, Int) \to g \to StreamT\ [s]$	page 15
$tabuFilter :: Ord\ s \Rightarrow [[s]] \to [s] \to [[s]] \to [[s]]$	page 14

A.3 Simulated Annealing Combinators

$linCooling :: Floating\ b \Rightarrow b \to b \to [b]$	page 16
$geoCooling :: Floating\ b \Rightarrow b \to b \to [b]$	page 16
$logCooling :: Floating\ b \Rightarrow b \to b \to [b]$	page 16
$saChoose :: (Floating\ v, Ord\ v) \Rightarrow (s \to v) \to v \to v \to s \to s \to s$	page 15
$sa :: (Floating\ v, Ord\ v) \Rightarrow (s \to v) \to StreamT\ s \to [v] \to [v] \to StreamT\ s$	page 16
$restart :: [v] \to [Bool] \to [v]$	page 17

A.4 Genetic Algorithms Combinators

$makePop :: Ord\ a \Rightarrow Int -> ExpandT\ s$	page 19
$gaSelect :: Ord\ r \Rightarrow Int \to [r] \to [r] \to StreamT\ [s]$	page 19
$nestWithProb :: (Ord\ r, Floating\ r) \Rightarrow [r] \to r \to StreamT\ s \to StreamT\ s$	page 20
$ga :: ExpandT\ s \to ContraT\ s \to StreamT\ s \to StreamT\ s$	page 20
$gaConfig :: Ord\ s \Rightarrow [Float] \to Int \to [Float]$ $\to [Bool] \to ContraT\ s \to StreamT\ s \to StreamT\ s$	page 21

A.5 Eager Combinators

$(!!!) :: [a] \to Int \to a$	page 28
$indexWithRemainder :: [a] \to Int \to (a, [a])$	page 28

A.6 Queue Based Window

```
data Queue a = Queue [a] [a] Int

initQ :: Queue a
initQ = Queue [ ] [ ] 0

sizeQ :: Queue a → Int
sizeQ (Queue _ _s) = s

append :: Queue a → a → Queue a
append (Queue fr bk sz) x = Queue fr (x : bk) (1 + sz)

remove :: Queue a → Queue a
remove q@(Queue [ ] [ ] _) = q
remove (Queue [ ] bk sz) = remove (Queue (reverse bk) [ ] sz)
remove (Queue as bk sz) = Queue (tail as) bk (sz − 1)

toList :: Queue a → [a]
toList (Queue fr bk _) = fr ++reverse bk

window :: Int → [a] → [[a]]
window sz = (map toList) ∘ (scanl fappend initQ)
  where
    fappend q v | sizeQ q ≡ sz = append (remove q) v
                | otherwise    = append q v
```