Haiku - a Scala combinator toolkit for semi-automated composition of metaheuristics

Zoltan A. Kocsis[1], Alexander E. I. Brownlee[1], Jerry Swan[1], Richard Senington[2]
[1]Computing Science and Mathematics, University of Stirling, FK9 4LA, UK.
[2]Data ductus AB, Skellefteå, Sweden.
Email: {zak,sbr,jsw}@cs.stir.ac.uk, richard.senington@dataductus.se

Abstract. There is an emerging trend towards the automated design of metaheuristics at the software component level. In principle, metaheuristics have a relatively clean decomposition, where well-known frameworks such as ILS and EA are parameterised by variant components for acceptance, perturbation etc. Automated generation of these frameworks is not so simple in practice, since the coupling between components may be implementation specific. Compositionality is the ability to freely express a space of designs 'bottom up' in terms of elementary components: previous work in this area has used combinators, a modular and functional approach to componentisation arising from foundational Computer Science. In this article, we describe HAIKU, a combinator toolkit written in the Scala language, which builds upon previous work to further automate the process by automatically composing the external dependencies of components. We provide examples of use and give a case study in which a programatically-generated heuristic is applied to the Travelling Salesman Problem within an Evolutionary Strategies framework.

1 Introduction

Early work in Search Based Software Engineering (SBSE) only needed to outperform manual approaches and in many cases random search and hillclimbing were sufficient for this. Now that the field is maturing and we wish to use SBSE to tackle more difficult problems, there is a need to employ more sophisticated search strategies. The difficulty facing the SBSE practitioner is the wealth of different metaheuristics available: e.g. can a software problem most usefully be solved with iterated local search, genetic alorithms, particle swarm or some hybridization of these techniques or their component parts?

A metaheuristic is instantiated for a particular problem domain via three domain-specific items, viz. a data structure for the representation of candidate solutions (e.g. bit-string, permutation etc); the ability to efficiently compare solution quality in order to guide the search process and lastly a collection of methods for transforming solutions. While metaheuristics can provide good results, operating at this level of abstraction offers no silver bullet. Rather, the family of techniques is ideally used as a toolbox, from which a practitioner can pick components and determine their effectiveness on a particular problem. Consequently, considerable development effort is focused on operator and parameter tuning for each new application (although it is encouraging to see increasing automation in this area [19]). It has also long been the norm to combine or *hybridise* methods,

for example using several in parallel, attempting to introduce the strengths of one method to others. As a concrete example of 'composition by hand', previous work [6] has applied Tabu search [14] and simulated annealing [16] at different points in a multi-stage local search algorithm. The desire for greater automation has led to approaches such as hyper-heuristics [3], which are the application of search to the problem of finding good heuristics ('heuristics for searching the space of heuristics'). Of particular interest for the automated design of algorithms are generative hyper-heuristics [5], which assemble basic components into more complex search algorithms. It is also worth mentioning algorithm portfolios [38], which use a group (portfolio) of different algorithms at the same time to solve a difficult problem. Fortunately for SBSE researchers, it is possible to express the problem of creating search strategies as one of software component assembly, thereby jointly incorporating knowledge from the domains of software engineering and metaheuristics. This paper introduces HAIKU, a tool-kit written in the Scala[1] language that facilitates the composition of metaheuristic components via combinators, extending previous work in the pure functional language Haskell [36].

Combinators are pure functions that depend exclusively on their input parameters. They are often higher-order functions, i.e. they can take other functions as parameters and (significantly) can return new functions, created dynamically from their inputs. A well-known example is function composition:

$$f \circ g = x \mapsto f(g(x))$$

The function $\circ$ takes two parameters, functions f and g, and returns a new function expressed in terms of these parameters. Functional programmers are in the habit of building reusable libraries using such functions because they encourage the expression of problems in terms of small building blocks. These building blocks can be combined and extended in a vast number of ways, with permissible combinations being enforced by the type system of the host programming language. Metaheuristics are a good fit for this pattern: individual metaheuristics can take functions (e.g. to provide an ordering of solutions or define acceptance criteria) as parameters but are themselves functions which can be passed to other metaheuristics (e.g. using an iterative improver as one component of a memetic algorithm [26]).

Recent work [34] on the use of combinators to build search heuristics notes that they have the look-and-feel of a Domain-Specific programming Language (DSL). Their modular nature allows new search algorithms to be developed for a specific application with reduced effort [25]. Further, their pure functional nature greatly simplifies the automated assembly of new search algorithms. In this context, the basic principles of modularity and re-use (well-established practices in software engineering) are fundamental to algorithm implementation. DSLs have already found uses in parameter control for evolutionary algorithms [18].

Modularity means that components are self-contained and can be developed independently, communicating only through clearly-defined interfaces. If the in-

[1] for an introduction, see http://www.artima.com/scalazine/articles/steps.html

terfaces are sufficiently general, parts can be *re-used* and recombined in new ways. However, there is often a high degree of interdependence between algorithm components, reducing modularity and inhibiting re-use. This is known as *content coupling*, where the implementation of one component requires deep knowledge of (and in many cases, access to) the internal mechanisms and implementation details of another. This is a hindrance to the combination of different components and their substitutability within metaheuristic frameworks. The HAIKU tool-kit presented in this article is structured in such a way that modularity and re-usability are inherent in the component implementations.

The remainder of the paper is structured as follows: Section 2 summarises related work. Section 3 introduces combinators in more detail and Sections 4 and 5 describe the design and implementation of HAIKU. A simple example of HAIKU's use is provided in Sections 6 and 7, composing combinators for Tabu search and simulated annealing and applying all three to the Travelling Salesman Problem. Finally, Section 8 provides conclusions and future work.

2 Related work

There is a body of work applying combinators in the field of constraint programming. Perron [30] describes a compositional approach in which search heuristics are termed 'goals'. This does not seem intended to support additional combinators, and specifically targets depth-first search. The Comet system [39] features 'fully-programmable' search: in contrast to the composition approach of combinators, a search controller is used to determine the behaviour of the search heuristic [40]. Choi et al [7] describe a compositional framework for search that relies on composing search engines and Desouter [8] describes a Scala framework using combinators to build custom heuristics for constraint satisfaction problems. 'Monadic constraint programming' was introduced by Schrijvers et al. [33], describing 'stackable search transformers'. While these only provide a limited and low-level form of search control, the concept is extended by Schrijvers [34], who introduces the concept of search combinators. This bridges the gap between a high-level modelling language for search and its efficient implementation. The user is able to define application-specific search strategies by combining a small set of primitives, effectively providing a Domain-Specific Language (DSL). This also serves as the foundation of the work in [32], where a search algorithm is used to automate the composition process.

McGillicuddy et al [24] achieve rapid prototyping of combinatorial optimisation algorithms via functional implementation of DSLs, as applied to dynamic programming problems (unbounded knapsack and longest common substring). Senington [36] argues for a specific function signature as forming a good basis for building metaheuristics from combinators: metaheuristics are regarded as stream transformations (i.e. functions that take a stream of solutions and return an updated stream) which are composed into more complex search algorithms. The paper presents a toolkit for expressing metaheuristics in the pure functional language Haskell. Building on this toolkit, [35] describes the use of combinators to move between perturbation, recombination and neighbourhood methods in metaheuristics, demonstrated for the Travelling Salesman Problem (TSP).

Marmion et al. [22,23] propose a generic structure for stochastic local search (SLS) algorithms, represented in a text-based grammar. The productions of the grammar represent local search hybrids. In this structure, each SLS algorithm has a definition of perturbation, optional subsidiary SLS, and acceptance criterion. Hybridisation is possible by assembling algorithms via the subsidiary local search. In common with this article, most of the human effort required is in devising problem-specific components for neighbourhoods, perturbations and heuristics. There are also some major differences with our work: HAIKU defines a search using program code rather than via a grammar, with the attendant programmatic flexibility, compile-time checking and IDE support that this provides. HAIKU's automated mechanism for composing 'environmental' state (described in more detail subsequently) is both less onerous and less error prone that the requirement to manually embed information such as search trajectory within the algorithm itself.

In order to automate metaheuristic construction, we need to be able to unambiguously determine the contribution of a component. This is clearly essential for learning schemes involving reinforcement and/or credit assignment: if component state is hidden, then we cannot determine which changes contribute to the sucess of a metaheuristic. Popular metaheuristic frameworks such as ECJ or JMetal [11,21] etc. do not prevent components from making arbitrary changes to nonlocal state. In contrast, the various works on combinators due to Schrijvers and Senington (above) allow unambiguous component substitution because of their pure functional nature. Recall that the aspects of modularity that concern us include decomposability of the different components as well as their *recombinability*: the metaheuristic components such as acceptance, perturbation should all be equipped with suitable composition operations (compositors). This latter aspect of modularity is absolute (rather than quantitative). Existing metaheuristic frameworks don't achieve the level of modularity that is sufficient for recombinability purposes. The essential contribution of this work is to address this outstanding issue, as described in the following sections.

3 Combinators

Formally speaking, a combinator is a 'pure' function (i.e. referentially transparent and without side-effects) with no free variables (i.e. they are self-contained, with no reference to external state). This modularity means that they can provide useful building blocks for describing a particular domain. Through the use of higher-order functions, combinators can combine their function parameters to provide more sophisticated control flow. Combinator libraries have been successfully employed in functional languages to provide clean and extensible capabilities for a diverse range of problems including real-time systems control [42] and expressing parser logic [15, 17]. These libraries capture patterns across diverse operations; provide mechanisms for combining these building blocks and allow extensibility via the provision of new constructs and control structures as different end-uses become apparent.

Parsers provide a good example of the power and mechanism of combinators since there is an obvious need to provide for many control structures, e.g. match-

ing a pattern *many* times in sequence; matching a single character or matching one pattern separated by another pattern. These can all be expressed as higher-order functions. In particular, most of this functionality can be defined so that parsers tend to act on other parsers, hence anything which is a parser can be passed to the library. This provides the high degree of customisability and extensibility that we desire from combinators. What follows is an example of a CSV parser written using parser combinators (this example is adapted from [28]), illustrating the creation of several user defined blocks of code (such as *cell*) built from library combinators and then reused.

```
val eol = Scanners.isChar('\n') // end of line
val cell = Scanners.notAmong(",\n").many()
// a cell is anything until , or \n
val line = cell.sepBy(Scanners.isChar(','))
// a line is a series of separated cells
val csvFile = line.endBy(eol);
// a csvfile is lines each ended by
```

The library of possible combinators can also be easily extended with user code, e.g. an operator that matches an identical symbol on either side of a given term. This could be coded by a user in the following manner and used in any expression which takes a parser as a parameter:

```
def surroundedBy( b : Parser, a : Parser ) : Parser = a.followedBy(b).endBy(a)
```

As discussed above, combinator libraries are essentially embedded DSLs and hence (unlike 'configuration-file' based approaches) can make use of the full power of the host language, as well as being customisable via the problem-specific code used to parametrise the system. In devising an appropriate metaheuristic, we have a toolbox of common patterns ('iterate until local optima', 'accept unimproved moves in inverse proportion to the number of iterations' etc.) and a desire to automatically combine different elements of this toolbox. Metaheuristics therefore share with combinators the essential notion of functionally parametrised and (recursively composable) control structures. The use of combinators is a natural fit for a generic metaheuristic library, allowing the problem-specific elements to be coded in the host language without limitations.

4 The design of Haiku

There are many popular metaheuristic software libraries (e.g. [9,11,12,20,21,41]), several of which abstract out common components such as acceptance, perturbation, recombination et c. It is typically the case that well-known metaheuristics such as iterated local search, evolutionary and swarm algorithms etc. then act as instances of the 'Template Method' design pattern [44], i.e. providing a pre-defined invocation sequence for the concrete instantiations of the abstract components with which they are (manually or automatically) configured. For example, a framework for iterated perturbation which is parametrised by the components for perturbation, acceptance and termination condition is given in Listing 1.1.

```
def iteratedPerturbation[Sol](incumbent : Sol,
  perturb : Sol => Sol,
  accept : (Sol,Sol) => Sol,
  isFinished : Sol => Boolean ) : Sol = {
    while( !isFinished(incumbent) ) {
      val incoming = perturb(incumbent)
      incumbent = accept(incumbent, incoming)
    }
    // the return keyword is implicit in Scala:
    incumbent
}
```

Listing 1.1: Iterated perturbation framework using polymorphic components

It is therefore desirable to be able to combinatorially configure such frameworks with different combinations of components. It is also known that different components can perform well at different points in the search (see e.g. [37]), which is particularly important in the case of dynamic environments [27]. One method of composition for components is to use the 'Composite' Design Pattern [13], i.e. to create a new component as a (perhaps dynamically-generated) function of existing components. As described by Woodward et al. [43], ensembles are a popular example of this approach. An elementary example would be to define a composite fitness function c as an aggregate of the fitness of a collection of surrogate functions $\{f_1, \ldots, f_k\}$, e.g. with c being given as a weighted sum of surrogates:

$$c : Sol \to \mathbb{R}$$
$$c : x \mapsto c_1 * f_1(x) + c_2 * f_2(x) + \ldots + c_k * f_k(x)$$

In order to ensure that our composite function can be plugged into the target framework, it needs to have the same signature as the abstract component that it instantiates. For the elementary generation of composites (e.g. weighted sum of fitness values, as above) this is straightforward: i.e. (in the case of a single-objective) the surrogates and the composite can all be defined in terms of functions from $\mathbb{R} \to \mathbb{R}$.

Unfortunately, the composition of many popular metaheuristic components is intrinsically not so straightforward. As a motivating example, consider an attempt to compose the well-known methods of Exponential Monte Carlo (EMC) [16] and Tabu acceptance [14]. EMC employs an *annealing schedule*, which tends to decrease the probability of accepting unimproved solutions as the search progresses. The Tabu scheme uses a *Tabu list* to prohibit the acceptance of recently-encountered solutions or operators. When attempting to automatically compose these components, a problem therefore arises because they depend on different notions of *component state*: e.g. EMC acceptance is dependant on the current state of the annealing schedule and Tabu Acceptance (TA) on the Tabu list.

Define acceptance to have signature:

$$State \times State \rightarrow State$$

where *State* is a generic type representing some combination of solution state *Sol* and component state. For solution state *Sol*, this means that EMC has signature:

$$\text{emc} : (Sol, Schedule) \times (Sol, Schedule) \rightarrow (Sol, Schedule)$$

and TA has:

$$\text{ta} : (Sol, TabuList) \times (Sol, TabuList) \rightarrow (Sol, TabuList)$$

Any attempt to build a combinator that composes EMC with TA would therefore need boilerplate to propagate component state information for *both* acceptance criteria:

$$\begin{aligned}\text{hybrid} :&(Sol, (Schedule, TabuList)) \times \\ &(Sol, (Schedule, TabuList)) \rightarrow \\ &(Sol, (Schedule, TabuList))\end{aligned}$$

This can be seen as a specific example of a general issue: i.e. whenever we compose stateful components (either within some composite component or as part of an operator pipeline), the combinator needs to be parametrised by the Cartesian product of the component states. This situation is particularly onerous for a metaheuristic designer since boilerplate code needs to be written for each specific combination of component states. What is therefore required is an automated means of dealing with Cartesian products of component states by 'lifting' pre-existing operations so that they correctly apply to the product state. The means by which HAIKU provides this functionality is described in more detail in the following section.

5 Haiku - Implementation

```
Tabu(diff,size=3)
transforms plainOldEMC
yielding hybridSearch
```

Actual HAIKU code

HAIKU is implemented in the Scala programming language. Scala was chosen because it has previously been used to implement combinator libraries [28], and its type system facilitates creating objects that behave like functions (e. g. fitness below). Scala runs on the Java Virtual Machine (JVM) and can call (and be called from) Java libraries and programs.

All HAIKU components are parametrised by the types Dec[Sol], where Sol is the type of solutions (as above) and Dec[Sol] (short for 'decorable') is a data type containing the solution along with the environment, i.e. the aggregated state of all composed components. The composition of components in HAIKU is simply illustrated in the context of the Evolutionary Strategy (ES) metaheuristic [2]. ES is a population-based metaheuristic that has been applied across a wide range of problem domains. In the general framework of ES, each iteration a set of one or more solutions ('parents') is selected from the population according to their

fitness. λ new solutions ('children') are generated from these by duplication, recombination and mutation. The children become members of the population, and the population is reduced back to its original size μ. EA approaches are classified into $(\mu + \lambda)$ and (μ, λ), according to the strategy used for reducing the population back to μ solutions (generational succession). With $(\mu + \lambda)$, the combined population of children and parents is ranked according to fitness, and the μ highest-ranking solutions are retained. Children only replace parents if they represent improvements. In (μ, λ) ES only the highest ranking μ children remain in the population. Parents are deleted, even if the children represent a decrease in fitness.

A single step of ES can be abstractly described by the composition of three operations neighbourhood, bias and select, with signatures as follows:

$$\begin{aligned} neighbourhood[Sol] = &\quad Dec[Sol] \rightarrow List[Dec[Sol]] \\ bias[Sol] = &\quad List[Dec[Sol]] \rightarrow List[Double] \\ select[Sol] = &\quad List[(Dec[Sol], Double)] \rightarrow Dec[Sol] \end{aligned}$$

For some solution state s, the output of the neighbourhood function is defined to consist of s together with its λ children. As explained in a subsequent section, this formulation makes it easy to generalise the 'plus' and 'comma' strategies described above for generational succession. The actual Scala code for a single-step of ES is given in Listing 1.2 and is depicted diagrammatically in Fig. 1.

```
case class ES[ Sol ] {
  type State = Dec[Sol]
  def update( currentState : State ) : State =
    select(neighbourhood(currentState) zip bias(neighbourhood(currentState)))
  // the zip function creates a list of pairs from the two list
}
```

Listing 1.2: Evolutionary Strategies update

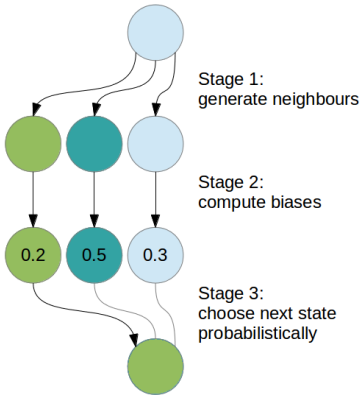

Fig. 1: A single step of ES described by the composition of the neighbourhood, bias and select operations

It is important to remember that different neighbourhood, bias and select functions may in fact have different associated component states. Therefore, the requirement to form the Cartesian products of states is, as explained above, unavoidable if one wishes to compose components. HAIKU, uniquely among metaheuristic frameworks, frees the end-user of the burden of having to do this manually. In implementation terms, this is achieved by storing the relevant environments as entries in a map. To ensure that access to this map is both type-safe and documented in the component declarations, the public interface requires that callers implement the Uses[Env] interface[2]. Storing the aggregate component state in this manner has several benefits. In particular, alternative approaches to aggregate state (e.g. monad transformer stacks [33]) keep the ordering of the combinators explicit, so reordering combinators requires writing boilerplate (involving the $lift$ function that is well-known to functional programmers and type theorists). In contrast, the map-based approach requires no boilerplate for reordering combinators. The Scala definition of Dec can be seen in Listing 1.3[3].

```
trait Uses[Env] { } // marker interface
case class Dec[A]( extract : A, private val decor : Map[Uses[_],Any] ) {
  def get[Env](c : Uses[Env]) : Option[Env] =
    decor.get(c).map(_.asInstanceOf[Env])
  def set[Env](c : Uses[Env], value : Env) : Dec[A] =
    Dec(extract, decor.updated(c,value)))
}
```

Listing 1.3: The Dec class

6 Case study: TSP

In this section, we use HAIKU to create a hybrid metaheuristic for solving the well-known Travelling Salesman Problem (TSP) [1]. The techniques used in HAIKU are not specific to the TSP, but it is a suitable problem for illustration. First, an appropriate solution representation needs to be chosen. This example uses the simplest possible one: a Tour is a permutation, implemented as a list of nodes in the order they were visited.

```
type Node = Int
type Tour = Dec[List[Node]]
```

As described above, HAIKU uses a *bias function* to measure solution quality. This allows us to compose measures of solution quality in various ways (as described in more detail below), thereby facilitating the creation of surrogate fitness measures. A fitness function is a deterministic bias function, mapping the solution to an ordered set. The search algorithm then operates to minimise or

[2] For these purposes we can consider a Scala 'trait' to be equivalent to the more familiar concept of interface

[3] We would like to consider $Dec[A]$ to be a subtype of A. This is not expressible in Scala or any other mainstream language. Instead, we rely on Scala's implicit conversions to ensure that Dec[A] can be substituted for A.

to maximise its value accordingly. In the case of the TSP, the goal is to optimise the tour length associated with the solution. The following code defines the corresponding bias function.

```
def length : FitnessFunction[Tour,Double] =
  Minimise { x =>
    val xnext = x.tail ++ List( x.head )
    // the .zipped method turns a pair of lists into a list of pairs,
    // and map invokes the tsp.dist function on each resulting pair.
    val distances = ( x, xnext ).zipped map ( tsp.dist )
    distances.sum
  }
```

The search requires an initial state: the following code creates a random tour:

```
val seed : Tour = RNG.shuffle { (0 until tsp.size).toList }
```

The RNG singleton provides the sole point of access to HAIKU's random number generator. Since the combinator implementation is stateless, results are reproducible from a given random seed. The HAIKU ES implementation uses a neighbourhood function to move around the search space. The neighbourhood function takes the current state of the search, and returns a list consisting of the current state and its offspring. In the example below, *lambda* children are created by reversing a random segment of the parent.

```
def transition = NeighbourhoodFunction { (x : Tour, lambda : Int) =>
    val children = for( i ← 0 until lambda ) yield {
        val a = RNG.nextInt( tsp.size )
        val b = a + RNG.nextInt( tsp.size − a )
        val reversed = x.drop( a ).take( b ).reverse
        x.take( a ) ++ reversed ++ x.drop( a ).drop( b )
    }
    List( x ) ++ children
  }
```

Search objects encapsulate the information required for running a search, viz. the neighbourhood function, the bias function, and the environmental variables (if any). The following code creates a search object and executes 1000 iterations of the search:

```
val search : Search[Tour] = ES( seed, transition, length ) (†)
val result = search.run(1000)
```

6.1 Semi-automated Composition of Metaheuristics

We can use the combinator-decorator mechanisms of HAIKU to create a acceptance criterion as a composite of Tabu search and simulated annealing. A previous comprehensive study [29] indicates that acceptance criteria can have a strong effect on the cross-domain generalisability of a hyper-heuristic, so the ability to create such hybrids is likely to have general utility. In HAIKU, a combinator is an object with a transforms method, of signature $transforms : Search[A] \rightarrow$

$Search[B]$. The following code sets up EMC acceptance using the simulated annealing combinator SA:

```
val emcSearch = SA.EMCAccept( 100, t => 0.95*t, length ) transforms search (†)
```

The first two arguments determine the annealing schedule. The last argument is the fitness function to be optimised. Naturally, simulated annealing requires a real-valued fitness function. Tabu Search additionally requires a *difference function*, which yields the changes in the solution state from iteration to iteration. The following code defines such a function:

```
def changes(x : Tour, y : Tour) : List[Node] = {
  val diff = (x,y).zipped filter ((x,y) => x != y)
  diff._2
}
```

Using this difference function, the Tabu combinator can be invoked:

```
val hybridSearch = Tabu( changes, size = 2 ) transforms emcSearch (†)
```

Note that the change in environment (first from an empty environment to an environment with temperature, then to an environment with a temperature and a Tabu list) is handled without the end-user having to write any additional boilerplate. Since bias functions are List[Double]-valued, it is possible to define custom distributions over the neighbourhood. This can be seen as a generalisation of the standard ES generational succession strategy: as mentioned above, a neighbourhood consists of the current state and its offspring, so if we always assign a bias of 0 to the current state, we can obtain a $(1, \lambda)$-ES otherwise we obtain a generalisation of $(1 + \lambda)$-ES in which the incumbent succeeds to the next generation with some probability. The composition mechanism also allows for fine-grained control over how the hybridisation occurs. For example, the default compositor for Tabu is 'intersection' (i.e. takes the smaller of the two acceptance probabilities, which can be seen in Fig. 2). The Compositor object provides several built-in compositors, as can be seen in the following:

```
Tabu( changes, size = 2, Compositor.union ).transforms(emcSearch) (†)
```

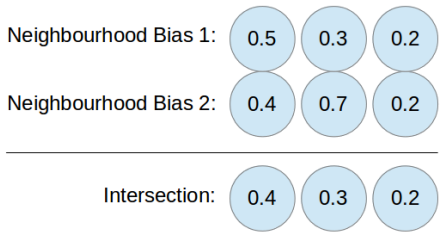

Fig. 2: The intersection composition operation

We used the Tabu and SA combinators to create the hybrid. Other elementary combinators provided by HAIKU for decorating a search via the 'transform' method include:

1. Threshold: Adds threshold acceptance capabilities to a search object.
2. Inspector: Allows observation of the search object while the search is in progress. Strict typing ensures that the search trajectory remains unchanged.

Notice that defining each hybrid (listings marked with †) for solving the TSP requires only a single line of code.

7 Experiments

As an illustration of the utility of the composition mechanism, we demonstrate that it is possible to find superior hybrids. We achieve this by performing a hyper-heuristic search in the space of composed bias functions. Recall from Section 5 that a bias function is given a neighbourhood and yields a *bias*, i.e. a list of corresponding non-negative values. A bias compositor (such as union, intersection etc as described above) takes a pair of biases and returns a new bias. A simple bias compositor that returns a list containing the weighted average of the corresponding input values is defined below. It contains a weight value $0.0 \leq m \leq 1.0$ as the sole hyper-heuristic parameter. This value essentially acts as an 'interpolator' between the contribution of the two input biases:

```
def weightedAverageCompositor(x: Bias, y: Bias): Bias = x*m + y*(1-m)
```

In order to show that the composition mechanism can yield useful hybrids, we performed a hyper-heuristic search for suitable m. Note that this search is over $[0, 1] \in \mathbb{R}$, despite the underlying space being permutation-based. The parameter values were as shown in Table 1, with input biases as given for the Tabu and EMC searches defined above. The heuristics were evaluated on 5 TSP instances from TSPLib [31] having less than 100 cities (att48,eil51,eil76,pr76,st70). In all of these instances, EMC significantly outperformed Tabu search (according to the Mann-Whitney U/Wilcox signed rank test with $p = 0.05$).

The top-level hyper-heuristic search was performed using the real-valued optimization method CMAES[4], an extension of ES which maintains a numerical approximation of the search gradient. The fitness value for the hyper-heuristic search was determined from an average of 21 runs of the $(1\{comma, plus\}1) - ES$ metaheuristic. In each case, CMAES converged quickly, taking resp. 117, 65, 57, 41 and 33 iterations [5].

In one of the five cases (st70), the hyper-heuristic search found a hybrid (the weight $m = 0.091$) that was significantly better (according to the Mann-Whitney U/Wilcox signed rank test with $p = 0.05$) than EMC search. Fig. 3 is box plot comparing the tour lengths found by the two algorithms.

[4] the default implementation in the Apache Commons Math 3.3 library

[5] execution time: 117.7s, 162.7s, 80.8s, 57.8s and 59.4s on an Intel Xeon 2.13 GHz with 4 GB RAM

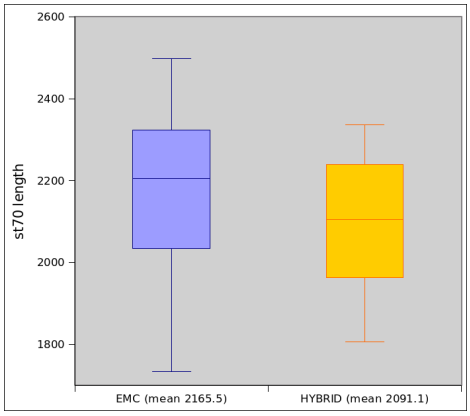

Fig. 3: Comparison of the tours found by EMC and Hybrid searches on instance st70

Parameter	Value
Tabu List Size	3
Max MH ((1{plus,comma}1)-ES) iter	500
Num MH runs per HH iter	21
Max HH (CMAES) iter	1000

Table 1: Parameter values

8 Conclusion

We described HAIKU, a combinator tool-kit written in Scala, for semi-automated hybridisation of metaheuristics. HAIKU addresses an intrinsic issue in the automated assembly of metaheuristic components, viz. the composition of component state. Experiments on instances of the Travelling Salesman Problem reveal that such hybridisation can indeed be useful: we implemented a real-valued hyper-heuristic which acts as an interpolator between a pair of acceptance criteria and used this to demonstrate the existence of a Tabu-annealing hybrid which significantly outperforms the base components. The addition of a wider palette of components (e.g. Great Deluge [10] and Late Acceptance [4] criteria) will allow a larger number of hybridizations to be explored.

Acknowledgment

Work funded by UK EPSRC grant EP/J017515 (DAASE).

References

1. Applegate, D.L., Bixby, R.E., Chvatal, V., Cook, W.J.: The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics). Princeton University Press, Princeton, NJ, USA (2007)
2. Beyer, H.G., Schwefel, H.P.: Evolution strategies - a comprehensive introduction 1(1), 3–52 (May 2002)
3. Burke, E.K., Gendreau, M., et al.: Hyper-heuristics: A survey of the state of the art. J. Oper. Res. Soc. 64, 1695–1724 (2013)
4. Burke, E.K., Bykov, Y.: A late acceptance strategy in hill-climbing for examination timetabling problems. In: Proc. PATAT (2008)

5. Burke, E.K., Hyde, M.R., et al.: Exploring hyper-heuristic methodologies with genetic programming. Computational Intelligence pp. 177–201 (2009)
6. Cambazard, H., Hebrard, E., OŚullivan, B., Papadopoulos, A.: Local search and constraint programming for the post enrolment-based course timetabling problem. Annals of Operations Research 194, 111–135 (2012)
7. Choi, C.W., Henz, M., Ng, K.B.: A compositional framework for search. In: Pontelli, E. (ed.) Proc. CICLOPS: Colloquium on Implementation of Constraint and LOgic Programming Systems, appeared as Tech. Rep. TR-CS-003/2001, New Mexico State University. Paphos, Cyprus (Nov 2001)
8. Desouter, B.: Modular search heuristics in Scala. Master's thesis, Ghent University, Belgium (2012), http://bdsouter.github.io/thesis/thesis.pdf
9. Di Gaspero, L., Schaerf, A.: Easylocal++: An object-oriented framework for the flexible design of local-search algorithms. Softw. Pract. Exper. 33(8) (2003)
10. Dueck, G.: New optimization heuristics: The great deluge algorithm and the record-to-record travel. J of Computation Physics 104, 86–92 (1993)
11. Durillo, J.J., Nebro, A.J.: jMetal: A Java framework for multi-objective optimization. Adv. in Engineering Software 42, 760–771 (2011)
12. Fink, A., Voß, S.: Hotframe: A heuristic optimization framework. In: Voß, S., Woodruff, D. (eds.) Optimization Software Class Libraries. pp. 81–154. OR/CS Interfaces Series, Kluwer Academic, Boston (2002)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
14. Glover, F., Laguna, M.: Tabu Search. Kluwer Academic, Norwell, MA, USA (1997)
15. Hutton, G., Meijer, E.: Monadic Parsing in Haskell. Journal of Functional Programming 8(4), 437–444 (1998)
16. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983)
17. Leijen, D., Meijer, E.: Parsec: Direct Style Monadic Parser Combinators For The Real World. Tech. Rep. UU-CS-2001-27, Dep. of Comp. Sc., Univ. Utrecht (2001)
18. Liu, S., Bryant, B., Mernik, M., Črepinšek, M., Zubair, M.: PPCea: A Domain-Specific Language for Programmable Parameter Control in Evolutionary Algorithms. INTECH Open Access (2011)
19. López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., Birattari, M.: The irace package, Iterated Race for Automatic Algorithm Configuration. Tech. Rep. TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium (2011)
20. Lukasiewycz, M., Glaß, M., Reimann, F., Teich, J.: Opt4J - A Modular Framework for Meta-heuristic Optimization. In: Proc. GECCO. pp. 1723–1730. Dublin (2011)
21. Luke, S.: The ECJ Owner's Manual (Oct 2010)
22. Marmion, M.É., Mascia, F., López-Ibáñez, M., Stützle, T.: Automatic design of hybrid stochastic local search algorithms. In: Hybrid Metaheuristics, Ischia, Italy. LNCS, vol. 7919, pp. 144–158. Springer Berlin / Heidelberg (2013)
23. Marmion, M.É., Mascia, F., López-Ibáñez, M., Stützle, T.: Towards the automatic design of metaheuristics. In: Lau, H.C., Raidl, G., Hentenryck, P.V. (eds.) MIC 2013, Singapore (Aug 2013)
24. McGillicuddy, D., Parkes, A.J., Nilsson, H.: An investigation into the use of Haskell for dynamic programming. Proc. PATAT (2014)
25. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (Dec 2005)
26. Moscato, P.: New ideas in optimization. chap. Memetic Algorithms: A Short Introduction, pp. 219–234. McGraw-Hill Ltd., Maidenhead, UK (1999)

27. Nguyen, T.T., Yao, X.: Continuous dynamic constrained optimization: The challenges. IEEE T Evolut Comput 16(6), 769–786 (Dec 2012)
28. O'Sullivan, B., Goerzen, J., Stewart, D.: Real World Haskell. O'Reilly (2008)
29. Özcan, E., Bilgin, B., Korkmaz, E.E.: A comprehensive analysis of hyper-heuristics. Intell. Data Anal. 12(1) (2008)
30. Perron, L.: Search procedures and parallelism in constraint programming. In: Jaffar, J. (ed.) Principles and Practice of Constraint Programming, LNCS, vol. 1713, pp. 346–360. Springer Berlin Heidelberg (1999)
31. Reinelt, G.: TSPLIB - A T.S.P. library. Tech. Rep. 250, Universität Augsburg, Institut für Mathematik, Augsburg (1990)
32. Samulowitz, H., Sabharwal, A., Schrijvers, T., Tack, G., Stuckey, P.: Automated design of search with composability (2013), 27th AAAI Conf. on Artificial Intelligence
33. Schrijvers, T., Stuckey, P., Wadler, P.: Monadic constraint programming. J of Functional Programming 19, 663–697 (2009)
34. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P.: Search combinators. Constraints 18(2), 269–305 (2013)
35. Senington, R., Duke, D.: Decomposing metaheuristic operations. In: Hinze, R. (ed.) Implementation and Application of Functional Languages, pp. 224–239. LNCS, Springer Berlin Heidelberg (2013)
36. Senington, R.J.: Hybrid meta-heuristic frameworks: a functional approach. Ph.D. thesis, University of Leeds (2013)
37. Soria-Alcaraz, J.A., Ochoa, G., Swan, J., Carpio, M., Puga, H., Burke, E.K.: Effective learning hyper-heuristics for the course timetabling problem. Euro J Oper Res 238(1), 77 – 86 (2014)
38. Tang, K., Peng, F., Chen, G., Yao, X.: Population-based algorithm portfolios with automated constituent algorithms selection. Inform. Sciences 279, 94–104 (2014)
39. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. MIT Press (2005)
40. Van Hentenryck, P., Michel, L.: Nondeterministic control for hybrid search. Constraints 11(4), 353–373 (2006)
41. Wagner, S., Kronberger, G.: Algorithm and experiment design with heuristic lab: An open source optimization environment for research and education. In: Proc. GECCO Companion. pp. 1287–1316. ACM, New York, NY, USA (2012)
42. Wan, Z.: Functional Reactive Programming for Real-Time Reactive Systems. Ph.D. thesis, Department of Computer Science, Yale University (December 2002)
43. Woodward, J., Swan, J., Martin, S.: The 'Composite' design pattern in metaheuristics. In: Proc. GECCO Comp. pp. 1439–1444. ACM, New York, USA (2014)
44. Woodward, J.R., Swan, J.: Template method hyper-heuristics. In: Proc. GECCO Companion. pp. 1437–1438. ACM (2014)