Towards 'Metaheuristics in the Large'

Author(s) omitted for 'double blind' review.

ABSTRACT

There is a pressing need for a higher-level architectural perspective in metaheuristics research. This article proposes a purely functional collection of component signatures as a basis for the scalable and automatic construction of metaheuristics. We claim that this is an important step for scientific progress because:

i). It is increasingly accepted that newly-proposed metaheuristics should be grounded in terms of well-defined frameworks and components. Standardized descriptions help to distinguish novelty from minor variation.

ii). Greater reproducibility is needed, particularly to facilitate comparison with the state-of-the-art.

iii). Interoperable descriptions are a pre-requisite for a data model supporting large-scale knowledge discovery across frameworks and problems.

A key obstacle is that metaheuristic components suffer from an intrinsic lack of modularity, so we present some design options for dealing with this and use this to provide a roadmap for addressing the above issues.

Categories and Subject Descriptors

H.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*; D.1.1 [**Applicative (Functional) Programming**]

Keywords

Metaheuristics, algorithm selection, analysis, experimental framework, functional programming.

1. BACKGROUND

Metaheuristics methods are stochastic search algorithms that have been employed to address a wide range of problems. One of their major advantages is that they are abstract search methods: the basic search logic can be applied to any problem which can be decomposed into the elementary concepts of solution representation, solution quality and some notion of 'locality', i.e. the ability to generate 'neighboring' solutions that are (heuristically intended to be) a function of the quality of a given solution. Metaheuristics can be applied to a broad selection of very different problems across sciences, engineering, economics, business and logistics [1]. However, it is well-known that no 'Universal solver' exists (see e.g. [2]). The research community has responded to this in two main ways: Firstly, by seeking inspiration for new metaheuristics and secondly by exploiting problem domain specifics. For the former, the natural world has been a rich source of ideas and researchers continue to investigate natural phenomena in the hope of finding mechanisms with a degree of generality. The latter can be addressed either analytically or empirically. An analytic approach uses problem domain information to derive effective algorithms for search components (EAX crossover for the TSP being one such example [3]). The empirical approach performs configuration tuning (by hand, using statistical design or some machine learning technique) to create a metaheuristic biased to a target distribution of problem instances. There are however a number of issues with both these responses (some technical and some cultural), which we now describe.

Design Automation The desire to automate the exploitation of problem features has led to interest in algorithm selection methods [4] (including hyper-heuristics [5]). There are many aspects of a problem domain that can be made available in a domain-independent fashion. A trivial example is the notion of the inverse of some operator — more sophisticated approaches are given in [6, 7, 8]. Unfortunately, the notion of 'domain-independent domain knowledge' that is well-understood in the formal methods and constraint-programming communities is not prominent in metaheuristics. For example, the only features exposed in a popular selective hyper-heuristic framework [9] are opaque indices representing solutions and (effectively randomized) operators. This therefore treats what could otherwise be a 'white box' algorithm selection problem as a black box problem. A standardized format for declarative descriptions of problem domains, representations and operators would bring knowledge engineering into the metaheuristics mainstream, with significant research opportunities for increasing automation.

Communicability and Reproducibility Some metaphorically inspired approaches have recently suffered strong criticism for their lack of rigor: When the metaphor obscures the specific solution-domain mechanisms used [10], the nov-

GECCO'15, July 11-15, 2015, Madrid, Spain.

elty of the metaphorical contribution becomes difficult to determine. At worst, this can lead to re-invention or renaming of mechanisms that are already well-understood (e.g. the popular 'Harmony search' metaheuristic was determined to be a simple variant of the foundational 'Evolution Strategies' approach [11]). This unnecessarily fragments the field and makes it appear impenetrable to outsiders. Using more formal definitions helps avoid this issue, promotes principled decomposition of novel and existing frameworks [12], and allows a better identification of potentially novelty. Reproducibility is a stronger requirement and it is often a non-trivial matter to replicate pre-existing work. In particular, a metaheuristic may achieve success via (problem- or solution-domainspecific) mechanisms which cannot easily be ported to new architectures or applied to new problem domains. This can make it difficult to determine if a metaheuristic has simply been overfitted to a domain.

Scalability Historically, computing systems have tended to get faster. The situation is no longer so simple: rather, systems are able to do more and more work in parallel [13]. To take advantage of increasing parallel processing capabilities, computations must be subdivided into a set of interdependent tasks to be executed efficiently in parallel across multiple cores and/or across multiple networked machines. In computer science in general, much human effort has been invested in algorithm-specific parallelization strategies. Fortunately, many popular metaheuristics are 'embarrassingly parallel' and it is desirable to be able to embed such generic knowledge within a suitably structured workflow, so that parallelization strategies can be applied automatically on the target platform.

Knowledge Discovery It can be difficult to generalize from a paper that compares two frameworks on a handful of problems using 'engineering metrics' (such as CPU-time). There is increasing interest in knowledge discovery on problem and algorithm features [14, 5, 15, 16, 17]. Science is concerned with explanations rather than raw results, and in order to support this, it is necessary to combine, exchange, mine and reason about metaheuristics and their component parts (e.g. acceptance, selection, perturbation etc) on a far larger scale than has been possible to date. The goal is to transform sharing of results from relatively isolated data points communicated in via a paper into a large shared pool of experimental data.

Interoperability Underlying the above issues is the general lack of interoperability. Since there has been little community progress towards standardization, practitioners continue to create their own data formats. Fortunately, there is growing interest in 'metaheuristics in the large': a desire to create flexible and composite systems for solving not only individual problems but also integrated variants consisting of interrelated, but separated, problems. Realizing such systems requires interoperability of metaheuristic frameworks and components: perturbative vs constructive; single-state vs population-based; sequential vs parallel (multi-core/distributed), single- vs multi- objective etc. This in turn provides a foundation for generic tools (e.g. parameter-tuning and machine-learning libraries, GUIs for visualization and interaction etc).

In summary, whilst improving on the state of the art should always be a key driver for a research community, the relentless pursuit of novelty and the 'up-the-wall game' [10] is counterproductive. While researchers labor in relative isolation, the risk of misidentifying novelty and difficulty of achieving generalization remains present. Scientific progress is greatly hindered by the lack of a common vocabulary in machine-readable form. In this article, we propose a necessary step along this path, structured as follows: in Section 2 we discuss design issues and options for a generic suite of metaheuristic components. In Section 3 we give a functional and modular description of components, giving example frameworks in Section 4. In Section 5 we outline a roadmap for addressing the above issues and in Section 6 we provide conclusions and some open research questions.

2. DESIGN PRINCIPLES

In this section, we discuss design options, taking into account the desirable features of the elementary components of metaheuristic search. Since there is already a palette of heuristic components such as Exponential Monte Carlo (EMC) acceptance and tournament selection that are well-known to have good cross-domain performance, it is clearly desirable to be able to instantiate metaheuristics via automated component configuration. This can be considered as a special case of software component assembly and design issues can usefully be addressed from a software architecture perspective. Desirable design features include:

i). Rapid prototyping of new ideas, while readily expressing common use cases and maximizing ease-of-use.

ii). Instrumentable. For example, allow the user to plug-in 'algorithm visitors' to act as gauges or collect data on various 'observables'. For knowledge discovery, this includes the collection of traces decorated with any desired info (e.g. measures for instrumenting/predicting component quality).

iii). "Only pay for what you use". This means that the design should not mandate that calculations be performed (e.g. finding the objective value of a solution) unless and until this information is explicitly required. Meeting this criterion requires very considered design of component interfaces.

A significant obstacle to the automated composition of metaheuristic components is that they suffer from an intrinsic lack of modularity. Configuration can be defined in general terms by expressing frameworks as higher-order functions that take components as parameters. In order to 'plug-in' a newly-devised component into a pre-defined framework, we need to define a 'one-size fits all' signature for it. For example, a possible signature for acceptance (here parametrized by State, representing some generic notion of search state) is:

$$accept_{State} : \text{incumbent} : State \times \text{incoming} : State \rightarrow \mathbb{B}$$

As a concrete example of this, Listing 1 gives a simple local search framework. As is always the case when defining re-usable function signatures, there is a tension between generality and the information required by some arbitrary implementation. For example: EMC-acceptance needs to know about temperature and tabu-based approaches need a trace of search progress. These are just two motivating examples of *environmental state*, which forms the context for the current action of a search process. Since the key to

```
State iteratedPerturb(s: State,
  perturb: Perturb_State,
  accept: accept_State,
  isFinished: IsFinished_State) {
    while(not isFinished(s)) {
        newS = perturb(s)
        if(accept(s, newS))
            s = newS
    }
    return s
}
```

Listing 1: A framework using higher-order functions.

metaheuristic performance is the injection of detailed domain knowledge into the search process, we clearly cannot anticipate in advance what information will be required by some component yet to be devised and therefore need to manage this state information by some other means. Principled handling of environment state is key to metaheuristic modularisation, and is therefore essential both for component interoperability and the automated construction of metaheuristics.

2.1 Implementation Options

There are a number of options for handling environment state. The tradeoff we wish to achieve is a balance between modularity, simplicity and expressivity. For the reasons discussed above, the former is the most important requirement. Generality is also of great importance, since we do not wish to artificially restrict the range of heuristics we can express. Some possible design alternatives are as follows:

Global variables This approach is probably the most representative of current practice: metaheuristic construction is done 'by hand' and environment state is scattered throughout the implementation in an *ad hoc* fashion. This is of course completely counter to modularity and is additionally counter to automation: since the underlying state of the metaheuristic can be modified at any point, it is difficult to unambiguously assign credit (e.g. via reinforcement learning) to a specific component.

Partial function application The required state information is passed to components when they are constructed. In functional languages, this can be seen as equivalent to partial function application; it is also called *currying* and is indeed available in a wide range of scriptting languages including JavaScript, Ruby and Perl. While this is more principled than the 'global variables' approach, modularity is merely enabled by this means (i.e. via encapsulation), rather than enforced: nothing prevents this state information from being mutated by some arbitrary component to which it is visible.

Mixins are a behavioral factorization technique, supported in some object-oriented programming languages. They allow an entity to defer the implementation of behaviors to the other entities upon which it depends. Dependencies between components are explicitly specified via reference to interfaces providing the desired behaviors. The environment state problem can be addressed with mixins by dividing responsibility between 'state providers' and 'state clients'. When a component such as EMC acceptance is implemented with mixins, rather than having explicit direct access to a value of type Temperature, it would rather only *declare the dependency* on Temperature. This can be beneficial, as otherwise the implementer of EMC acceptance has to create the Temperature component to pass it to the constructor. This facilitates automated component assembly in 'just in time' mode, i.e. many components can be plugged together simultaneously (as opposed to step-by-step hierarchical construction). Mixins typically allow also *linearization*: several components *of the same type* can be combined to form a 'chain of responsibility', e.g. a composite acceptance condition or a sequence of search operators.

Combinators Combinators are pure functions (i.e. without side-effects) that combine one or more primitive values into a more complex form. They can be used similarly to partial applications, to pass environment state in specific ways from one primitive computation to another. However, they encapsulate state in a controlled way, ensuring that it is passed from one primitive operation to another without unwanted side effects or unexpected alteration. Previous work has demonstrated that it is possible to create expressive frameworks for constraint programming and local search with this approach [18, 19] and recent work has further increased the automated support for composition of components.

3. PROPOSED COMPONENTS

From the design options of Section 2, we now propose components which address the modularity issue. The proposed approach is 'pure functional', i.e. referentially transparent and without side effects. This approach provides a number of important advantages [20], of particular relevance to large-scale and automated design of metaheuristics: They make it easy to reason about functional equivalence; they provide a good abstraction mechanism; they make it easy to represent state explicitly; they ensure determinism, easing reproducibility of behavior on a per-component basis. As demonstrated by the presence of Java™ in one of the example listings below, it is entirely possible to express these components in non-functional languages (although much more compiler and library support/syntactic sugar is available in languages such as Haskell, Scala, F# or Clojure). As discussed in more detail in Section 5, a functional treatment of metaheuristics is important because this greatly facilitates architectures which can take advantage of abundant computing resources, e.g. thread-safe parallelism [21] or 'Software as a Service' (SaaS) implementation via stateless web-services. We therefore adopt the combinator-style approach of the previous section to give a purely functional formulation of metaheuristics. In the following, we present the signatures of some ubiquitous metaheuristic components.

3.1 Functional Formulation of Metaheuristics

We explicitly mandate the propagation of environment state in component signatures and therefore consider a metaheuristic framework to be a mapping

$$MH_{State,FEnv} : (State, FEnv) \rightarrow (State, FEnv)$$

where $FEnv$ is any environment state required by the framework itself (e.g. a source of randomness or a trace of solution history). State is a generic placeholder representing the current state of a search process, and may comprise one or more (partial or complete) solutions (or e.g. Pareto-fronts

thereof in the case of multi-objective optimization). Additional constraints can be added to this intentionally loose specification via the use of increasingly specialized components (e.g. PMX recombination requires the geneome to be a permutation), which effectively allows the injection of as (problem- or solution-) domain knowledge. For example, the specific type of State with which a component is implemented determines whether it is to be treated as constructive / perturbative / single-state / population-based.
Specific frameworks are instantiated via a *configuration*, which we can w.l.o.g. consider to be a tuple of components to be invoked by the framework. For example, the iterated perturbation metaheuristic given in Listing 1 can be configured with components for perturb, accept, and isFinished. In general, each heuristic H_i in the tuple may need to maintain its own environment, and can therefore be represented as:

$$H_{Env_i} : (I, Env_i) \rightarrow (O, Env_i)$$

where I and O are the input and output types for the heuristic. Hence a framework is configured via a tuple of heuristics $C = (H_{Env_1}, \ldots, H_{Env_n})$:

$$MF_{State,EnvF,C} : C \rightarrow MH_{State,(Env_F,Env_1,\ldots,Env_n)}$$

By defining a collection of components with appropriately-chosen signatures, it is possible to facilitate the creation of meta- and hyper- heuristics which meet the design criteria of Section 2. In the next section, we describe component signatures which have been designed with these aims in mind.

3.2 Component signatures

The following descriptions of metaheuristic components can be seen to be 'Platonic' – as such, it is unsurprising that some similarity can be observed with various concepts in existing popular frameworks. However, it is notable that what *does not* appear in a signature is of as great importance as what *does*: for example, with the signature for acceptance, there is no explicit requirement for a fitness/objective value. This is in accordance with the 'don't pay for what you don't use' principle: any information required beyond the bare minimum is essentially an implementation detail of some concrete acceptance criterion and should not be mandated up-front. This parsimonious principle is put into effect in these signatures whenever possible.
As is quite common in modern metaheuristic libraries, constraints are loosely specified via the use of generic types: State is as given in Section 3.1, Sol denotes an individual candidate solution, Entity is a generic placeholder for some arbitrary type (e.g. solution, integer, String etc) and Value represents some partially-ordered type (or vector thereof). The context of the search process (i.e. the environment state) is represented by Env.

Perturbation We start off with the notion of perturbation. In its simplest form, perturbation is considered as a mapping between States but there are also other common use-cases:

P1 Some 'reactive search' variants have additional parameters representing the perturbation strength.

P2 Some variants (e.g. tabu) depend on the past trajectory of the search.

P3 Since the size of basins of attraction is not known *a priori*, it is desirable to express a 'kick' operation as a sequence of lower-level operations [22]. It is therefore useful to consider any form of iterated perturbation to itself be a kind of perturbation.

```
interface Perturb<State,Env> {
  Pair<State,Env> apply(
    State incumbent,Env env);
}
```

Listing 2: Java™ interface for Perturb component

Cases **P1** and **P2** can be expressed via the signature:

$$perturb_{State,Env} : State \times Env \rightarrow State \times Env$$

where *Env* is environmental state propagated by the invoking framework code, e.g. the parameter perturbStrength in the case of **P1** and a list of *State* representing the solution trace for **P2**. As discussed above, the idea is that this is a functional approach and side-effect free: i.e. any updates to environment state are observable in the result, but not the argument. The perturb component is exemplified in Java code in Listing 2. A major distinguishing factor between two different implementations of a single conceptual framework (e.g. local search, or the simple genetic algorithm) is the information recorded by components in order to guide subsequent decisions. It should be clear that this parameterization by environment allows the memoization of any desired information. We therefore include the environment parameter in all subsequent components without further comment.

Initialization Often one or more entities need to be created *ex nihilo* (e.g. initial populations, ephemeral random constants etc). The corresponding signature is:

$$create_{Entity,Env} : Env \rightarrow Entity \times Env$$

The Env parameter allows the invoking framework to pass any dynamic supplementary information required: for example when creating successive members of a population, the invoking framework might pass the 'population so far' as a parameter to maintain diversity.

Acceptance In single-point search, an acceptance criterion is used to choose between incumbent and incoming solutions. Popular schemes include threshold and Exponential Monte Carlo acceptance [23]. We define acceptance to have signature:

$$accept_{State,Env} : State \times State \times Env \rightarrow State \times Env$$

The more conventional approach of returning a boolean, rather than the accepted solution, requires consistency of interpretation between the calling code and the implementation, with greater margin for error. Also, this signature is *compositional* — it can form part of an operator pipeline.

Evaluation To our knowledge, all popular frameworks assume a 1-1 correspondence between a candidate solution and its value, and often rigidly bind them programmatically to each other. In general, one might wish to use different notions of value (e.g. surrogates) at different points in the search process, or in general guide the search using alternative *search drivers* [6]. It is therefore more useful to functionally abstract the notion of computing the value of some entity (e.g. be it a solution or a population as a whole):

$$evaluate_{Entity,Value,Env} : Entity \times Env \rightarrow Value \times Env$$

Although it might appear that this signature precludes the implementation of computational-efficiency schemes such as delta-evaluation , the general philosophy is to treat such schemes as an implementation issue wherever possible. If so desired, a concrete implementation of evaluate could (for example) rely on solutions carrying delta values with them by construction. Note also that there are no prior constraints on the type of *Value*: in particular nothing prevents it from being multi-objective.

Neighborhood Function This is an essential defining component of single-point search:

$$locality_{State,Env} : State \times Env \rightarrow [State] \times Env$$

$[State]$ denotes a list of search states. The Env parameter allows information to be passed in that might be of use for e.g. VNS-based approaches [24]. The 'only pay for what you use' maxim allows the resulting list of states to be a *lazy* list, so that only the states needed by the client/caller are generated. Infinite neighborhoods are thereby possible.

Selection To choose a single entity from a list:

$$choose_{Entity,Env} : [Entity] \times Env \rightarrow Entity_{\perp} \times Env$$

where $Entity_{\perp}$ denotes the possibility that none of the inputs are chosen. This concept can be represented directly in many programming languages via the use of the generic Option (a.k.a. Maybe) type. To select n entities from a list:

$$select_{Entity,Env} : [Entity] \times \mathbb{N} \times Env \rightarrow [Entity] \times Env$$

Recombination A mainstay of 'genetic algorithms' style approaches:

$$recombine2_{State,Env} : \quad State \times State \times Env \rightarrow State \times Env$$
$$recombineN_{State,Env} : \quad [State] \times Env \rightarrow [State] \times Env$$

Termination condition Determines when some iterated process should halt:

$$finished_{State,Env} : State \times Env \rightarrow \mathbb{B} \times Env$$

Generational succession This denotes the 'merging' of parent and child populations, e.g. expressing the gamut of strategies by which the end result of a single generation is determined.

$$merge_{State,Env} : State \times State \times Env \rightarrow State \times Env$$

This can be seen as presenting the same signature as acceptance and binary recombination. Large scale knowledge discovery efforts might therefore determine whether the prevailing intuition about the differing semantics of these three components is actually justified.

4. EXAMPLE FRAMEWORKS

Listing 3 gives an example of a local search framework expressed in terms of the proposed signatures. As a simple illustration of modularity, note that LocalSearch is itself a perturbation operator, which facilitates the implementation of hybrid algorithms combining global and local search. By varying the components supplied to the LocalSearch constructor, it is possible to express many variants of common single-state algorithms, including random walks, hill-climbing, simulated annealing and tabu search. In the listings, the subscripts $_{1,2}$ denote the respective projection of the first and second elements of a pair. As an example of a population-based approach, Listing 4 gives the framework code for a generic Evolutionary Algorithm (EA). The notation ++ is a shorthand for list concatenation. While other EA framework variants are clearly possible, the example given can be configured to express a very wide range of common approaches.

```
localSearch(initial: State,env: Env)
: (State, Env) = {
  incumbent = (initial,env)
  best = incumbent
  (done,env) = isFinished(incumbent)
  while(not done) {
    incoming = perturb(incumbent)
    incoming = accept(incumbent_1, incoming_1,
        incoming_2)
    (done,env) = isFinished(incumbent)
    incumbent = (incoming_1,env)
    best = order(best,incoming)
  }
  return best
}
```

Listing 3: Local Search framework

```
EA1(initial: List<State>,env: Env):
  (List<State>,Env) = {
  incumbent = (initial,env)
  (done,env) = isFinished(incumbent)
  while(not done) {
    incoming = (Nil,incumbent_2);
    repeat( incumbent_1.length/2 times ) {
      incoming=select(incumbent_1,2,incoming_2)
      parents=recombineN(incoming_1,incoming_2)
      c1 = mutate((parents_1)_1,parents_2)
      c2 = mutate((parents_1)_2,c1_2)
      next = incoming_1 ++ c1_1 ++ c2_2
      incoming = (next,c2_2)
    }
    incumbent = merge(incumbent_1,incoming_1,
        incoming_2)
    (done,env) = isFinished(incumbent_1,
        incumbent_2)
    incumbent = (incumbent_1,env)
  }
  return incumbent
}
```

Listing 4: EA framework

5. IMPLICATIONS

Although ubiquitous, neither the components nor the frameworks presented above are intended to be exhaustive. In particular, this approach is entirely compatible with existing non-functional frameworks and components provided they are wrapped inside a functional interface that propagates environmental state. We now discuss how the proposed approach forms a basis for addressing the issues of Section 1.

5.1 Design Automation

By virtue of modularity, our proposed approach greatly facilitates automated assembly. In particular, this allows for bottom-up approaches, which are less subject to human bias

than the *a priori* prescription of a particular metaheuristic and which therefore has relevance to foundational knowledge discovery efforts. In other areas of design (e.g. manufacturing), standardization has allowed a shift from the design of integrated systems to the design of individual components within the system. In metaheuristics, this reflects the natural trend for incorporating specialized problem- or solution-domain knowledge i.e. a researcher can specialize in particular kind of components such as acceptance criteria and determine their cross-domain ubiquity. Of course, a particular domain might well benefit from a specific top-level metaheuristic: nothing in our proposed approach precludes this, and subordinate components can still be configured bottom-up as desired.

5.2 Communicability and Reproducibility

As discussed above, a pure functional description of frameworks parameterized by their environmental configuration makes explicit both the coupling between components and the state changes that they undergo. The clarity thus afforded is precisely why descriptions in terms of pure function is the language of mathematics and foundational computer science. Greater clarity obviously eases reproducibility via re-implementation, but a much stronger notion of reproducibility is the ability to invoke some (potentially remotely hosted) existing implementation. This leads to the notion of metaheuristic components as part of a Service Oriented Architecture (SOA) [25]. The modern approach to SOA is to employ web-services: self-contained software components that are network accessible via HTTP-based protocols. Web-service descriptions can be semantic or nonsemantic, according to the underlying goals of the service. Semantic web services expose machine-readable data structures, facilitating automated service discovery and composition.
The initial *de facto* standard of Simple Object Access Protocol (SOAP) has been widely supplanted by the REpresentational State Transfer (REST) approach, which exhibits superior simplicity, robustness and scalability. The stateless nature of the proposed approach marries well with that of REST.

5.3 Scalability

In addition to the marriage of stateless components with efficient stateless web-services described in the preceding section, the recursive nature of the proposed signatures makes it easy to instantiate multi-level search. For example, the EA framework of Listing 4 is itself a perturbation operator (of lists of solutions), illustrating possibilities for: a) interoperability between single-state and population-based approaches and b) a scalable composition that can take advantage of the increasing availability of computing power. More significantly, with the proposed description, metaheuristics and components can be composed into a formal workflow using a functional mechanism known as a *monad*. Monads are a form of combinator pattern that follow well-defined laws to give strong mathematical properties [26]. By employing a monadic workflow, it is possible to facilitate the creation of parallelizable meta- and hyper- heuristics. Many modern languages (e.g. Haskell, F#, Scala) provide particular syntactic sugar for monads, which will additionally significantly simplify Listings 3 and 4. For example, we can encode a local search algorithm monadically as demonstrated by the Haskell code of Listing 5. We use a library function

```
localSearch :: State -> Env -> State

localSearch initial env =
  evalState ls (initState initial env 0)
   where ls = while not isFinished (
                 do incoming <- perturb
                  incoming <- accept incoming
                  checkFinished
                  increment)
                 best
```

Listing 5: Monadic formulation of Local Search

`evalState` to evaluate the stateful computation `ls` in the context of an internal state that is initialized from the internal search state, the environment and the initial value of an iterator.
Although the algorithm looks similar to its imperative counterpart, the internal state is fully encapsulated within the definition of `localSearch`. All state effects are explicit and type safe — there are no implicit side-effects. This brings major advantages of safety, including thread safety, and improves confidence in the correctness of the algorithm, as well as allowing efficient implementation. It also allows parallelism to be safely introduced as part of the metaheuristic, e.g. by embedding calls to the Par monad [27]. It might be thought that a monadic workflow requires metaheuristic researchers to become expert functional programmers, so it should be emphasized that is a consequence of our proposed formulation which can be exploited by those seeking the benefits, rather than a mandatory aspect. Another minor but pleasing property is that the internal state makes the parameter space of a component explicit, facilitating configuration via automated tools such as [28].

5.4 Knowledge Discovery

One key way in which the proposed approach facilitates knowledge is the ability to add arbitrary instrumentation to components via the generic environment representation. In particular, this allows for data mining on metaheuristic traces. In addition, by employing our generic notion of State (which denotes one or more solutions representing the current state of the search), the same framework can instantiate metaheuristics operating at different scales. For example, a 'composite' recombination operator can choose from different types of recombination strategies, putting meta and hyper-heuristics under the same framework in the manner of [29]. The generic notion of State is particularly useful in this respect, since State could even be represented by a model: once trained it can be used to predict which the best operator at a given moment of the search is. This supports very recent research on using data analytics within meta or hyper-heuristics, such as the algorithm proposed by Consoli et al. [15]. Having these different types of algorithms under a common framework greatly facilitates their extension and comparison.

5.5 Interoperability

It is useful to distinguish between syntactic and semantic interoperability. The former enables systems and system components to communicate/exchange data unambiguously and react accordingly. This requires specification of common data formats (e.g. XML), communication protocols etc.

Semantic interoperability builds upon the syntactic layer, enabling the meaning of the data to be shared, based on a pre-defined collection of ground terms specifying the interpretation of syntactic content. To achieve the goal of a common research platform, it is necessary to achieve interoperability of the following:

Component Implementations We discuss this most complex case first, since (depending on the level of abstraction at which we wish to interoperate), similar concerns can apply in all the other cases. One example of prior art in metaheuristics is the 'Timetabling Markup Language' (TTML), an XML data format representing input problem instances and solutions to a given timetabling problem instance [30]. TTML is based on the Mathematical Markup Language (MathML) which it uses to express the interpretation of the components of a timetabling problem, including constraints and how the solution is be evaluated. Another example is the Predictive Model Markup Language (PMML), which is a XML data format representing data mining and machine learning models. PMML represents not only the model also its attributes and the transformations defined on them.

Solution representations without associated constraints (e.g. a bitstring of arbitrary length) can be specified syntactically (e.g. via formats such as JSON, YAML or raw XML). If it is additionally necessary to pass constraints to the solver (e.g. to specify the notion of a permutation from first principles), then the full semantic power described above in 'Component Implementations' may be required. Regarding prior art, *Irace* is an automatic algorithm configuration algorithm [28] that defines an input data format for tuning a range of parameter types.

Component definitions The generic signatures of Section 3 provide a basis for syntactic interoperability of component definitions and simple serialization languages such as JSON suffice for data exchange of generic (i.e. representation independent) signatures. When specialized for particular solution representations (e.g. with specific associated constraints), the interoperability category is that of the representation. It is hopefully clear that the generality of what is proposed suffices to act as a 'Rosetta Stone' for intercommunication between popular metaheuristic frameworks (e.g. [31, 32, 33, 9, 34, 35, 36]).

Problem definitions There are of course many well-known repositories of benchmark problems for specific domains (e.g. [37, 38]), but the motivation here is to work towards a universal description format for problem domains. For metaheuristic purposes, this is an area in which the tradeoffs between expressiveness and computational cost needs to be clearly delineated. There has been some previous work in this area [39, 40, 41] and TTML also provides a description of problem instances. Finally, it is pleasing to note that interoperability of trace information comes 'for free' given the facility to exchange solution representations and problem definitions.

6. CONCLUSIONS

In this article we have discussed some of the cultural and technical issues that we believe are impediments to progress in metaheuristic research. As a first step, we describe a set of purely functional signatures representing commonly-used metaheuristic components and designed to facilitate:

i). Interoperability at the signature level.
ii). The ability to instrument components for any desired data mining/knowledge discovery purpose.
iii). Self-assembly of metaheuristics (and ease of credit assignment for their component parts) by virtue of modularity.
iv). Scalability of implementations, from multi-level search to built-in support for parallelization.
v). The description of metaheuristics in terms of explicit mechanisms, rather than the language of metaphor.

The functional and interoperability criteria together form a basis for 'Software as a Service' implementations of metaheuristics via stateless web-services. The language and platform agnosticism of this approach helps in turn to address issues of reproducibility and scalability. A community initiative towards machine-readable descriptions of frameworks, components and experimental results will also motivate some important research questions, viz.

i). How can semantic descriptions best bring declaratively-driven approaches (e.g. such as those common in the scheduling and planning communities) into mainstream metaheuristics?
ii). More generally, what kind of declarative descriptions might be required for 'domain aware' algorithm generation that includes problem reformulation [42] and formal approaches to program synthesis [43].

In conclusion, we envision the emergence of a distributed, community driven suite of tools, providing an expanded repository of interoperable frameworks and components, bringing together researchers and practitioners across domains, unifying the field and closing the gap between scientific research and empirical practice.

7. REFERENCES

[1] H. Hoos and T. Stützle, *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann, 2005.

[2] D. Wolpert and W. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, 1997.

[3] Y. Nagata and S. Kobayashi, "Edge assembly crossover: A high-power genetic algorithm for the travelling salesman problem," in *Proceedings of the 7th International Conference on Genetic Algorithms, MI, USA*, 1997.

[4] L. Kotthoff, "Algorithm Selection for Combinatorial Search Problems: A Survey," *arXiv.org*, Oct. 2012.

[5] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall *et al.*, "Hyper-heuristics: A survey of the state of the art," *J. Oper. Res. Soc.*, vol. 64, no. 12, 2013.

[6] K. Krawiec and J. Swan, "Pattern-guided Genetic Programming," in *GECCO '13 Companion*, 2013.

[7] J. C. Ortiz-Bayliss, H. Terashima-Marín, S. E. Conant-Pablos, E. Özcan, and A. J. Parkes, "Improving the performance of vector hyper-heuristics through local search," in *GECCO '12*. NY, USA: ACM, 2012.

[8] A. J. Parkes, "Combined Blackbox and AlgebRaic Architecture (CBRA)," in *PATAT 2010*, 2010.

[9] G. Ochoa, M. Hyde *et al.*, "Hyflex: A benchmark framework for cross-domain heuristic search," in *Evolutionary Computation in Combinatorial Optimization*, ser. LNCS, J.-K. Hao and M. Middendorf, Eds. Springer, 2012, vol. 7245.

[10] K. Sörensen, "Metaheuristics - the metaphor exposed," *International Transactions in Operational Research*, vol. 22, no. 1, 2015.

[11] D. Weyland, "A rigorous analysis of the Harmony Search algorithm: How the research community can be misled by a "novel" methodology," *Int. J. Appl. Metaheuristic Comput.*, vol. 1, no. 2, 2010.

[12] M. López-Ibáñez, F. Mascia, M.-E. Marmion, and T. Stützle, "A template for designing single-solution hybrid metaheuristics," in *GECCO Comp '14*, New York, USA, 2014, pp. 1423–1426.

[13] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobbs Journal*, vol. 30, no. 3, 2005.

[14] F. Thabtah and P. Cowling, "Mining the data from a hyperheuristic approach using associative classification," *Expert Systems with Applications*, vol. 34, no. 2, 2008.

[15] P. Consoli, L. L. Minku, and X. Yao, "Dynamic selection of evolutionary algorithm operators based on online learning and fitness landscape metrics," in *10th International Conference on Simulated Evolution And Learning*, ser. LNCS, vol. 8886. Springer, 2014.

[16] K. Smith-Miles, D. Baatar, B. Wreford, and R. Lewis, "Towards objective measures of algorithm performance across instance space," *Computers & Operations Research*, vol. 45, 2014.

[17] A. Scheibenpflug, S. Wagner, E. Pitzer, and M. Affenzeller, "Optimization Knowledge Base: An open database for algorithm and problem characteristics and optimization results," in *GECCO '12*. NY, USA: ACM, 2012.

[18] T. Schrijvers, G. Tack, P. Wuille, H. Samulowitz, and P. Stuckey, "Search combinators," *Constraints*, vol. 18, no. 2, 2013.

[19] R. Senington and D. Duke, "Decomposing metaheuristic operations," in *Implementation and Application of Functional Languages*, ser. LNCS, R. Hinze, Ed. Springer Berlin Heidelberg, 2013.

[20] J. Hughes, "Why functional programming matters," *The Computer Journal*, vol. 32, 1984.

[21] K. Hammond and G. Michaelson, *Research Directions in Parallel Functional Programming*. Springer, 1999.

[22] L. Di Gaspero and A. Schaerf, "EasyLocal++: An object-oriented framework for the flexible design of local-search algorithms," *Softw. Pract. Exper.*, vol. 33, no. 8, 2003.

[23] E. Özcan, B. Bilgin, and E. E. Korkmaz, "A comprehensive analysis of hyper-heuristics," *Intell. Data Anal.*, vol. 12, no. 1, 2008.

[24] P. Hansen and N. Mladenovic, "Variable Neighborhood Search: Principles and applications." *European Journal of Operational Research*, vol. 130, no. 3, 2001.

[25] P. García-Sánchez, J. González, P. A. Castillo, M. G. Arenas, and J. J. M. Guervós, "Service oriented evolutionary algorithms," *Soft Comput.*, vol. 17, no. 6, pp. 1059–1075, 2013. [Online]. Available: http://dx.doi.org/10.1007/s00500-013-0999-5

[26] P. Wadler, "The essence of functional programming," in *Proceedings of POPL '92*, New York, USA, 1992.

[27] S. Marlow, R. Newton, and S. Peyton Jones, "A monad for deterministic parallelism," *SIGPLAN Not.*, vol. 46, no. 12, 2011.

[28] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari, "The irace package," IRIDIA, Université Libre de Bruxelles, Belgium, Tech. Rep. TR/IRIDIA/2011-004, 2011.

[29] J. Woodward, J. Swan, and S. Martin, "The 'Composite' Design Pattern in metaheuristics," in *GECCO '14 Companion*, New York, USA, 2014.

[30] E. Özcan, "Towards an XML-based standard for timetabling problems: TTML," in *Multidisciplinary Scheduling: Theory and Applications*, G. Kendall, E. K. Burke, S. Petrovic, and M. Gendreau, Eds. Springer US, 2005, pp. 163–185.

[31] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich, "Opt4J - A Modular Framework for Meta-heuristic Optimization," in *GECCO '11*, Dublin, Ireland, 2011.

[32] S. Wagner, G. Kronberger, A. Beham, Kommenda *et al.*, *Advanced Methods and Applications in Computational Intelligence*. Springer, 2014, vol. 6, ch. Architecture and Design of the HeuristicLab Optimization Environment.

[33] J. Swan, E. Özcan, and G. Kendall, "Hyperion - a recursive hyper-heuristic framework," in *LION*, ser. LNCS, C. Coello, Ed. Springer, 2011, vol. 6683.

[34] S. Cahon, N. Melab, E.-G. Talbi, and M. Schoenauer, "ParaDisEO-Based Design of Parallel and Distributed Evolutionary Algorithms," in *Evolution Artificielle*, ser. 29, vol. 36. Marseille, France: Springer, 2003.

[35] S. Luke, "The ECJ owner's manual," http://www.cs.gmu.edu/~eclab/projects/ecj, 2010.

[36] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, no. 10, 2011.

[37] G. Reinelt, "TSPLIB: A traveling salesman problem library," *ORSA Journal of Computing*, vol. 3, no. 4, 1991.

[38] H. Hoos and T. Stutzle, "SATLIB: An Online Resource for Research on SAT," in *SAT2000*, H. van Maaren I. P. Gent and T. Walsh, Eds. IOS Press, 2000.

[39] A. Guazzelli, M. Zeller, W.-C. Lin, and G. Williams, "PMML: An open standard for sharing models," *The R Journal*, vol. 1, no. 1, 2009.

[40] M. Ghallab, C. K. Isi, S. Penberthy, D. E. Smith *et al.*, "PDDL - The Planning Domain Definition Language," Yale Center for Computational Vision and Control, Tech. Rep. CVC TR-98-003/DCS TR-1165, 1998.

[41] G. Kronberger, M. Kommenda, S. Wagner, and H. Dobler, "GPDL: A framework-independent problem definition language for grammar-guided genetic programming," in *GECCO '13 Companion*, 2013.

[42] E. K. Burke, J. Mareček, A. J. Parkes, and H. Rudová, "Decomposition, reformulation, and diving

in university course timetabling," *Comput. Oper. Res.*, vol. 37, no. 3, Mar. 2010.

[43] E. Kitzelmann, "Inductive programming: A survey of program synthesis techniques," in *Approaches and Applications of Inductive Programming*, ser. LNCS, U. Schmid, E. Kitzelmann, and R. Plasmeijer, Eds. Springer Berlin Heidelberg, 2010, vol. 5812, pp. 50–73.