Constraints (2013) 18:269–305
DOI 10.1007/s10601-012-9137-8

Search combinators

**Tom Schrijvers · Guido Tack · Pieter Wuille ·
Horst Samulowitz · Peter J. Stuckey**

Published online: 30 December 2012
© Springer Science+Business Media New York 2012

Abstract The ability to model search in a constraint solver can be an essential asset for solving combinatorial problems. However, existing infrastructure for defining search heuristics is often inadequate. Either modeling capabilities are extremely limited or users are faced with a general-purpose programming language whose features are not tailored towards writing search heuristics. As a result, major improvements in performance may remain unexplored. This article introduces *search combinators*, a lightweight and solver-independent method that bridges the gap between a conceptually simple modeling language for search (high-level, functional and naturally compositional) and an efficient implementation (low-level, imperative and highly non-modular). By allowing the user to define application-tailored search

T. Schrijvers (✉) · P. Wuille
Universiteit Gent, Gent, Belgium
e-mail: tom.schrijvers@ugent.be

G. Tack · P. J. Stuckey
National ICT Australia (NICTA), Victoria, Australia

G. Tack
Monash University, Victoria, Australia
e-mail: guido.tack@monash.edu

P. Wuille
Katholieke Universiteit Leuven, Leuven, Belgium
e-mail: pieter.wuille@cs.kuleuven.be, pieter.wuille@ugent.be

H. Samulowitz
IBM Research, New York, USA
e-mail: samulowitz@us.ibm.com

P. J. Stuckey
University of Melbourne, Victoria, Australia
e-mail: pjs@cs.mu.oz.au

strategies from a small set of primitives, search combinators effectively provide a rich *domain-specific language* (DSL) for modeling search to the user. Remarkably, this DSL comes at a low implementation cost to the developer of a constraint solver. The article discusses two modular implementation approaches and shows, by empirical evaluation, that search combinators can be implemented without overhead compared to a native, direct implementation in a constraint solver.

Keywords Search heuristics · Modeling language · Modularity · Implementation

1 Introduction

Search heuristics often make all the difference between effectively solving a combinatorial problem and utter failure. Heuristics make a search algorithm efficient for a variety of reasons, e.g., incorporation of domain knowledge, or randomization to avoid heavy-tailed runtimes. Hence, the ability to swiftly design search heuristics that are tailored towards a problem domain is essential for performance. This article introduces search combinators, a versatile, modular, and efficiently implementable language for expressing search heuristics.

1.1 Status quo

In CP, much attention has been devoted to facilitating the modeling of combinatorial problems. A range of high-level modeling languages, such as OPL [31], Comet [29], or Zinc [14], enable quick development and exploration of problem models. But there is substantially less support for high-level specification of accompanying search heuristics. Most languages and systems, e.g. ECLiPSe [20], Gecode [25], Comet [29], or MiniZinc [15], provide a set of predefined heuristics "off the shelf". Many systems also support user-defined search based on a general-purpose programming language (e.g., all of the above systems except MiniZinc). The former is clearly too confining, while the latter leaves much to be desired in terms of productivity, since implementing a search heuristic quickly becomes a non-negligible effort. This also explains why the set of predefined heuristics is typically small: it takes a lot of time for CP system developers to implement heuristics, too—time they would much rather spend otherwise improving their system.

1.2 Contributions

In this article we show how to resolve this stand-off between solver developers and users, by introducing a domain-specific modular search language based on combinators, as well as a modular, extensible implementation architecture.

For the user we provide a modeling language for expressing complex search heuristics based on an (extensible) set of primitive combinators. Even if the users are only provided with a small set of combinators, they can already express a vast range of combinations. Moreover, using combinators to program application-tailored search is vastly more productive than resorting to a general-purpose language.

For the system developer we show how to design and implement modular combinators. The modularity of the language thus carries over directly to modularity of the implementation. Developers do not have to cater explicitly for all possible combinator combinations. Small implementation efforts result in providing the user with a lot of expressive power. Moreover, the cost of adding one more combinator is small, yet the return in terms of additional expressiveness can be quite large.

We believe that there is potential for an additional group of beneficiaries, although this still has to be proven in practice.

For the community we propose that search combinators are an ideal starting point for a standard search language. The reason is that they have a low implementation complexity, and are not closely tied to the underlying solver architecture. Most CP systems have the ability to program arbitrary search, but in many cases this is an endeavour only for experts, and for each new system one must learn a new way to build a search strategy. Search combinators are expressive yet simple enough to add to a well-supported modelling language like MiniZinc [15] giving us a basis for a search language supported by multiple systems.

The technical challenge is to bridge the gap between a conceptually simple search language and an efficient implementation, which is typically low-level, imperative and highly non-modular. This is where existing approaches are weak; either the expressiveness is limited, or the approach to search is tightly tied to the underlying solver infrastructure.

The contribution is therefore the novel design of an expressive, high-level, compositional search language with an equally modular, extensible, and efficient implementation architecture.

1.3 Approach

We overcome the modularity challenge by implementing the primitives of our search language as *mixin* components [4]. As in Aspect-Oriented Programming [10], mixin components neatly encapsulate the *cross-cutting behavior* of primitive search concepts, which are highly entangled in conventional approaches. Cross-cutting means that a mixin component can interfere with the behavior of its sub-components (in this case, sub-searches). The combination of encapsulation *and* cross-cutting behavior is essential for systematic reuse of search combinators. Without this degree of modularity, minor modifications require rewriting from scratch.

An added advantage of mixin components is extensibility. We can add new features to the language by adding more mixin components. The cost of adding such a new component is small, because it does not require changes to the existing ones. Moreover, experimental evaluation bears out that this modular approach has no significant overhead compared to the traditional monolithic approach. Finally, our approach is solver-independent and therefore makes search combinators a potential standard for designing search.

This article is an extended version of a paper [22] that appeared in the proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP) 2011. That paper further developed the ideas laid out in our earlier paper [19], which was presented at ModRef 2010.

1.4 Plan of the article

The rest of the article is structured as follows. The next section discusses the motivation and challenges involved in defining search combinators. Section 3 defines the high-level search language in terms of basic heuristics and combinators. Section 4 shows how the modular language is mapped to a *modular design* of the combinator implementations. Section 5 presents two concrete implementation approaches for combinators and gives an overview of how we integrate search combinators into the MiniZinc toolchain. Section 6 verifies that combinators can be implemented with low overhead. Finally, Section 7 discusses related approaches, and Section 8 concludes the article.

2 Background and motivation

This section motivates the need for a search language, gives an overview of our approach, the main challenges and contributions, and introduces some terminology used in this article.

2.1 Problems and importance

Choosing a good search heuristic is an important aspect of solving many combinatorial problems, and many works in the literature are devoted to designing and evaluating search heuristics. Despite this awareness, most CP systems offer relatively little support for search heuristics. They usually offer one or both of the following two options.

1. The system offers a small number of predefined search heuristics.
2. The system offers users a general purpose programming language (e.g., C++ or Prolog) to program their own search heuristics.

Systems that combine both approaches are, for instance, Gecode and many CLP systems (e.g., ECLiPSe [20] or B-Prolog [33]). The former provides a few *search engines* and *branchers* and allows the user to program new ones in C++. The latter provide a limited set of search heuristics and enable the programmer to write their own search from scratch in Prolog.

Both approaches have substantial disadvantages. The first one puts a substantial implementation burden on the system developer. Implementing and maintaining a new search heuristic is a non-negligible task. This means that there is often little incentive for providing more than a handful of the most commonly used search heuristics. This is of course very confining for the system's users. Novice and intermediate users are not aware of potentially better alternatives than what the system offers and expert users simply do not access them. The second approach fares little better. It puts the burden on the user. Clearly, implementing their own search heuristics is beyond novice users, and poses a high threshold for intermediate users as well. Even for expert users, implementing a new search heuristic can be a time-consuming activity.

Both approaches hamper widespread adoption and reduce the potential impact of (even established) research results. Moreover, they prevent a wide range of search heuristics being considered and evaluated for a particular constraint problem. Hence,

feasible or better solutions are potentially not found. Moreover, models that rely on particular search heuristics, are not portable across systems and, based on the current state-of-the-art, standardization of search support is not to be expected in the near future. This exacerbates a general issue that users face. If they need two uncommon system features (e.g., particular global constraints or search heuristics) for solving their model, they often cannot find a system that provides both and have to make do with a suboptimal solution.

In summary, the high cost of developing and maintaining search heuristics has far-reaching consequences for many CP systems and users.

2.2 Approach

The general objective of search combinators is to reduce the effort of developing implementations of search heuristics. For that purpose, the search combinators approach applies well-known and widely researched tools from the fields of programming languages and software engineering: *modularity* and *reuse*. Modularity means that different aspects of a system can be developed (implemented, compiled, maintained) independently in software artifacts called modules or *components*. Components interact with one another through well-defined interfaces. If interfaces are sufficiently general and the means to compose components into systems are sufficiently flexible, the same component can be reused in different configurations to build different systems.

The above tools are obviously very abstract and apply to software systems in general. The key challenge is to make them concrete in the setting of the paper, search heuristics. Obviously, this paper is not the first to have observed the above problems and applied ideas of modularity and reuse to it. The essential difference lies in the degrees of granularity and orthogonality of modules. This paper provides a finer degree of modularity and a higher degree of orthogonality. Finer granularity means that modules are smaller, which lowers development and maintenance cost, and they capture more fine-grained concepts, which increases their potential for reuse and increases the number of meaningful configurations that can be built from the same number of given modules. Increased orthogonality means that the dependency on other system aspects is decreased, which lowers the effort of porting modules between systems and increases again the number of possible configurations that can be obtained with little effort.

A particularly important form of modularity that search combinators practice is *compositionality*. Compositionality means building a new component that implements a particular interface from other components that implement the same interface. If the logic of such a composition can be encapsulated in a separate component, that component is called a *combinator*.[1] Search combinators are combinators for building search heuristics. Combinators are very attractive for two reasons.

1. Theoretically, a system with n different roles and m different components implementing each role has m^n different possible configurations. If we consider only a single role, then m components yield only m different configurations.

[1]Combinator is a term from functional programming; in object-oriented programming it is known as the *Composite* design pattern.

However, when components are compositional, we need only a few primitive components and a few combinators to obtain an infinite number of possible configurations. While in practice the number of useful combinations is clearly not infinite, m compositional components are much more cost-effective than non-compositional ones.

2. By encapsulating the logic of combining components into combinators, the configuration of components becomes very easy. This means concretely that users can be gradually exposed to search heuristics: novices only use predefined configurations, intermediate users construct their own configurations, while expert users write new components. Note that by leveraging compositionality, even this last aspect becomes easier: only the missing functionality needs to be written, while existing functionality can be added by composition.

   An interesting result of combinators is that, with a little syntactic support, they have the look and feel of a special-purpose programming language, also known as a *domain-specific programming language* (DSL). The notable difference from traditional approaches to programming language design is the modularity. Traditional languages are designed and implemented as a whole, while the combinator approach is inherently extensional. This means that adding a new "language feature" has a very low cost, as it does not affect the implementations of the existing ones.

2.3 Technical challenges and contributions

Many previous works (e.g., Perron [16], IBM ILOG CP Optimizer, and Comet [29]) have realized that modularity and compositionality are key features for supporting CP tree search. In fact, nearly all systems, even the traditional CLP systems, offer two basic combinators: conjunction and disjunction. However, beyond that, there are few other combinators provided. Moreover, the general thrust is towards modularity only: a "textbook" search heuristic is written from scratch in a single module and can be reused many times. Finally, the design details of existing approaches are closely tied to a particular implementation platform and CP system, or simply not given (e.g., for closed-source systems).

The main contribution of this paper is a modular design that factorizes search heuristics into finer-grained less interdependent components than existing approaches. Arriving at such a highly compositional design is also the main technical challenge.

Although it is not always obvious, many search heuristics have common aspects that have a potential of being factorized out. For instance, iterative deepening, limited discrepancy search and restarting branch-and-bound and dichotomic search all share the aspect of repeatedly restarting their search. The reason why such commonality is not always apparent, and also why it is technically challenging to factor it out, is that their code is intermixed with that of other aspects of the search heuristics and not easily disentangled. In software engineering terms, such aspects are called *cross-cutting*. Obviously all existing systems can express in some way or another all search heuristics, the essential question here is whether they do so with the same degree of factorization. This paper claims that it goes further than existing work: Section 3 presents a set of search combinator components and shows how they can be combined into a range of well-known search heuristics, Section 4 explains

the underlying modular design of the components and Section 7 provides a detailed comparison with related work.

The degree of orthogonality is another important aspect. The search combinator approach makes minimal assumptions on the other aspects of the system or the underlying implementation platform.[2] This means that it can be integrated with relatively little effort in many existing systems and on many different platforms.[3] For different systems on the same implementation platform, it is often even possible to share the same search combinators implementation. Moreover, if multiple systems implement the search combinator approach on different platforms, it becomes easy to exchange definitions of search heuristics. In contrast, existing approaches do not consider the benefits of orthogonality and explicitly target only a single system. To support these claims, Section 5 summarizes different implementations of the search combinator approach and Section 6 shows that these implementations have competitive performance.

2.4 Scope and terminology

The words "search" and "search heuristic" are generally highly overloaded. In this paper they have a particular meaning.

Firstly, the kind of search that search combinators address is CP tree search. There are other important forms of CP search like local search and large neighborhood search. Each kind of search has its own strengths and weaknesses and for that reason some CP systems, like Comet, offer multiple forms of search.

Secondly, the word search heuristic refers to varying aspects of the dynamic traversal of a CP tree. Search combinators distinguish three different aspects:

1. The *labeling* strategy is concerned with splitting a node of the search tree into child nodes to enable further propagation on a set of constraint variables. Because it has a significant impact on the efficiency of search, labeling has already been widely studied in the literature and is often factored into variable and value selection strategies.
   This paper considers labeling strategies as primitive search heuristics, and builds on established results for them. As such, their particulars are not important for this paper.
2. The *queueing* strategy is concerned with the selection of a previously generated node of the search tree for further expansion. The best known and most widely applied queueing strategy is depth-first search. Alternatives are breadth-first search and best-first search.
   This paper is not concerned with the choice of a particular queueing strategy. The main point of interest is that search heuristics are orthogonal to the choice of queueing strategy. This means that a system is free to choose it or leave that choice to the user.

[2] E.g., first-class continuations like those used in Comet are not needed.

[3] E.g., on top of Comet continuations and Search Controllers.

3. The *search heuristic* proper is a controlling entity on top of one or more labeling strategies. It decides what nodes is processed by what labeling strategy, what node is processed again in possibly altered circumstances, and what node is not processed at all (i.e., is pruned). As parts of its business, it keeps track of all manner of information of the search process.

It is this last notion that is the central topic of interest of this paper.

Note that traditionally known search heuristics are made up of two or three of the above concepts. For instance, the traditional notion of *limited discrepancy* search (LDS) combines a specific queueing strategy with a control aspect that repeatedly expands a search tree from the root in a particular pattern. Search combinators enable the control aspect of LDS to be defined as a search heuristic and to be used with other queueing strategies.

3 High-level search language

This section introduces the syntax of our high-level search language and illustrates its expressive power and modularity by means of examples. The rest of the article then presents an architecture that maps the modularity of the language down to the implementation level.

The search language is used to define a *search heuristic*, which a *search engine* applies to each node of the search tree. For each node, the heuristic determines whether to continue search by creating child nodes, or to prune the tree at that node. The queuing strategy, i.e., the strategy by which new nodes are selected for further search (such as depth-first traversal), is determined separately by the search engine, it is thus orthogonal to the search language. The search language features a number of primitives, listed in the catalog of Fig. 1. These are the building blocks in terms of which more complex heuristics can be defined, and they can be grouped into *basic heuristics* (base_search and prune), *combinators* (ifthenelse, and, or, portfolio, and restart), and *state management* (let, assign, post). This section introduces the three groups of primitives in turn.

For many users, the given primitives will represent a simple and at the same time sufficiently expressive language that allows them to implement complex, problem-specific search heuristics. The examples in this section show how versatile this base language is. However, we emphasize that the catalog of primitives is open-ended.

s	::=	prune prunes the node	|	ifthenelse($cond, s_1, s_2$) perform s_1 until *cond* is false, then perform s_2
	|	base_search(*vars*, *var-select*, *domain-split*) label	|	and($[s_1, s_2, \ldots, s_n]$) perform $s1$, on success $s2$ otherwise fail, ...
	|	let(v, e, s) introduce new variable v with initial value e, then perform s	|	or($[s_1, s_2, \ldots, s_n]$) perform $s1$, on termination start $s2$, ...
			|	portfolio($[s_1, s_2, \ldots, s_n]$) perform $s1$, if not exhaustive start $s2$, ...
	|	assign(v, e) assign e to variable v and succeed	|	restart($cond, s$) restart s as long as *cond* holds
	|	post(c, s) post constraint c at every node during s		

Fig. 1 Catalog of primitive search heuristics and combinators

Advanced users may need to add new, problem-specific primitives, and Section 4 explains how the language implementation explicitly supports this.

The concrete syntax we chose for presentation uses simple nested terms, which makes it compatible with the *annotation* language of MiniZinc [15]. Section 5.3 discusses our implementation of MiniZinc with combinator support. However, other concrete syntax forms are easily supported (e.g., we support C++ and Haskell).

3.1 Basic heuristics

Let us first discuss the two basic primitives, base_search and prune.

base_search The most widely used method for specifying a basic heuristic for a constraint problem is to define it in terms of a *variable selection* strategy which picks the next variable to constrain, and a *domain splitting* strategy which splits the set of possible values of the selected variable into two (or more) disjoint sets. Common variable selection strategies are:

- firstfail: select the variable with the smallest current domain,
- smallest: select the variable which can take the smallest possible value,
- domwdeg [2]: select the variable with smallest ratio of size of current domain and number of failures the variable has been involved in, and
- impact [18]: select the variable that will (based on past experience) reduce the raw search space of the problem the most.

Common domain splitting strategies are:

- min: set the variable to its minimum value or greater than its minimum,
- max: set the variable to its maximum value or less than its maximum,
- median: set the variable to its median value, or not equal to this value, and
- split: constrain the variable to the lower half of its range of possible values, or its upper half.

The CP community has spent a considerable amount of work on defining and exploring the above and many other variable selection and domain splitting heuristics. The provision of a flexible language for defining new basic searches is an interesting problem in its own right, but in this article we concentrate on search combinators that combine and modify basic searches.

To this end, our search language provides the primitive base_search(*vars*, *var-select*, *domain-split*), which specifies a systematic search. If any of the variables *vars* are still not fixed at the current node, it creates child nodes according to *var-select* and *domain-split* as variable selection and domain splitting strategies respectively.

Note that base_search is a CP-specific primitive; other kinds of solvers provide their own search primitives. The rest of the search language is essentially solver-independent. While the solver provides few basic heuristics, the search language adds great expressive power by allowing these to be combined arbitrarily using combinators.

prune The second basic primitive, prune, simply cuts the search tree below the current node. Obviously, this primitive is useless on its own, but we will see shortly how prune can be used together with combinators.

3.2 Combinators

The expressive power of the search language relies on combinators, which combine search heuristics (which can be basic or themselves constructed using combinators) into more complex heuristics.

and/or Probably the most widely used combination of heuristics is *sequential composition*. For instance, it is often useful to first label one set of problem variables before starting to label a second set. The following heuristic uses the and combinator to first label all the xs variables using a first-fail strategy, followed by the ys variables with a different strategy:

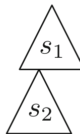
```
and([base_search(xs, firstfail, min),
     base_search(ys, smallest, max)])
```

As you can see in Fig. 1, the and combinator accepts a list of searches $s_1, \ldots, s_n$, and performs their and-sequential composition. And-sequential means, intuitively, that solutions are found by performing *all* the sub-searches sequentially down one branch of the search tree, as illustrated in Fig. 2(1).

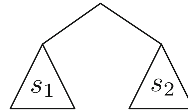
The dual combinator, or$([s_1, \ldots, s_n])$, performs a disjunctive combination of its sub-searches—a solution is found using *any* of the sub-searches (Fig. 2(2)), trying them in the given order.

Fig. 2 Primitive combinators

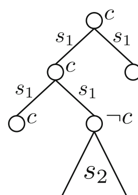
1) and$([s_1, s_2])$



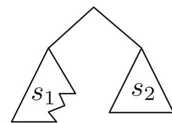
2) or$([s_1, s_2])$



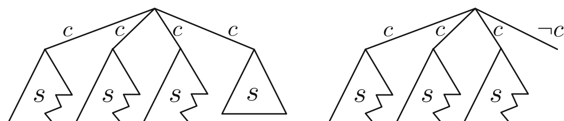
3) if(c, s_1, s_2)



4) portfolio$([s_1, s_2, s_3])$



5) restart(c, s)



Statistics and ifthenelse The ifthenelse combinator is centered around a conditional expression *cond*. As long as *cond* is true for the current node, the sub-search s_1 is used. Once *cond* is false, s_2 is used for the complete subtree below the current node (see Fig. 2(3)).

We do not specify the *expression language* for conditions in detail, we simply assume that it comprises the typical arithmetic and comparison operators and literals that require no further explanation. It is notable though that the language can refer to the constraint variables and parameters of the underlying model. Additionally, a condition may refer to one or more *statistics* variables. Such statistics are collected for the duration of a subsearch until the condition is met. For instance $\text{ifthenelse}(\text{depth} < 10, s_1, s_2)$ maintains the search depth statistic during subsearch s_1. At depth 10, the ifthenelse combinator switches to subsearch s_2.

We distinguish two forms of statistics: *Local statistics* such as depth and discrepancies express properties of individual nodes. *Global statistics* such as number of explored nodes, encountered failures, solution, and time are computed for entire search trees.

It is worthwhile to mention that developers (and advanced users) can also define their own statistics, just like combinators, to complement any predefined ones. In fact, Section 4 will show that statistics can be implemented as a *subtype* of combinators that can be queried for the statistic's value.

Abstraction Our search language draws its expressive power from the combination of primitive heuristics using combinators. An important aspect of the search language is *abstraction*: the ability to create new combinators by effectively defining macros in terms of existing combinators.

For example, we can define the limiting combinator $\text{limit}(cond, s)$ to perform s while condition *cond* is satisfied, and otherwise cut the search tree using prune:

$$\text{limit}(cond, s) \equiv \text{ifthenelse}(cond, s, \text{prune})$$

The $\text{once}(s)$ combinator, well-known in Prolog as `once/1`, is a special case of the limiting combinator where the number of solutions is less than one. This is simply achieved by maintaining and accessing the solutions statistic:

$$\text{once}(s) \equiv \text{limit}(\text{solutions} < 1, s)$$

Exhaustiveness and portfolio/restart The behavior of the final two combinators, portfolio and restart, depends on whether their sub-search was *exhaustive*. Exhaustiveness simply means that the search has explored the entire subtree without ever invoking the prune primitive.

The $\text{portfolio}([s_1, \ldots, s_n])$ combinator performs s_1 until it has explored the whole subtree. If s_1 was exhaustive, i.e., if it did not call prune during the exploration of the subtree, the search is finished. Otherwise, it continues with $\text{portfolio}([s_2, \ldots, s_n])$. This is illustrated in Fig. 2(4), where the subtree of s_1 represents a non-exhaustive search, s_2 is exhaustive and therefore s_3 is never invoked.

An example for the use of portfolio is the $\text{hotstart}(cond, s_1, s_2)$ combinator. It performs search heuristic s_1 while condition *cond* holds to initialize global parameters for a second search s_2. This heuristic can for example be used to initialize the widely

applied *Impact* heuristic [18]. Note that we assume here that the parameters to be initialized are maintained by the underlying solver, so we omit an explicit reference to them.

$$\text{hotstart}(cond, s_1, s_2) \equiv \text{portfolio}([\text{limit}(cond, s_1), s_2])$$

The restart($cond$, s) combinator repeatedly runs s in full. If s was not exhaustive, it is restarted, until condition *cond* no longer holds. Figure 2(5) shows the two cases, on the left terminating with an exhaustive search s, on the right terminating because *cond* is no longer true.

The following implements random restarts, where search is stopped after 1000 failures and restarted with a random strategy:

$$\text{restart}(\text{true}, \text{limit}(\text{failures} < 1000, \text{base_search}(\texttt{xs}, \text{randomvar}, \text{randomval})))$$

Clearly, this strategy has a flaw: If it takes more than 1000 failures to find the solution, the search will never finish. We will shortly see how to fix this by introducing user-defined search variables.

The prune primitive is the only source of non-exhaustiveness. Combinators propagate exhaustiveness in the obvious way:

- and($[s_1, \ldots, s_n]$) is exhaustive if all s_i are
- or($[s_1, \ldots, s_n]$) is exhaustive if all s_i are
- portfolio($[s_1, \ldots, s_n]$) is exhaustive if one s_i is
- restart($cond$, s) is exhaustive if the last iteration is
- ifthenelse($cond$, s_1, s_2) is exhaustive if, whenever *cond* is true, then s_1 is, and, whenever *cond* is false, then s_2 is

3.3 State access and manipulation

The remaining three primitives, let, assign, and post, are used to access and manipulate the state of the search:

- let(v, e, s) introduces a new search variable v with initial value of the expression e and visible in the search s, then continues with s. Note that search variables are distinct from the decision variables of the model.
- assign(v, e): assigns the value of the expression e to search variable v and succeeds.
- post(c, s): provides access to the underlying constraint solver, posting a constraint c at every node during s. If s is omitted, it posts the constraint and immediately succeeds.

These primitives add a great deal of expressiveness to the language, as the following examples demonstrate.

Random restarts Let us reconsider the example using random restarts from the previous section, which suffered from incompleteness because it only ever explored

1000 failures. A standard way to make this strategy complete is to increase the limit geometrically with each iteration:

$$\mathsf{geom_restart}(s) \equiv \mathsf{let}(maxfails, 100, \mathsf{restart}(\mathsf{true}, \mathsf{portfolio}([\mathsf{limit}(\mathsf{failures} < maxfails, s), \mathsf{and}([\mathsf{assign}(maxfails, maxfails * 1.5), \mathsf{prune}])])))$$

The search initializes the search variable *maxfails* to 100, and then calls search s with *maxfails* as the limit. If the search is exhaustive, both the portfolio and the restart combinators are finished. If the search is not exhaustive, the limit is multiplied by 1.5, and the search starts over. Note that assign succeeds, so we need to call prune afterwards in order to propagate the non-exhaustiveness of s to the restart combinator.

Branch-and-bound A slightly more advanced example is the branch-and-bound optimization strategy:

$$\mathsf{bab}(\mathtt{obj}, s) \equiv \mathsf{let}(best, \infty, \mathsf{post}(\mathtt{obj} < best, \mathsf{and}([s, \mathsf{assign}(best, \mathtt{obj})])))$$

It introduces a variable *best* that initially takes value ∞ (for minimization). In every node, it posts a constraint to bound the objective variable by *best*. Whenever a new solution is found, the bound is updated accordingly using assign.

The bab example demonstrates how search variables (like *best*) and model variables[4] (like `obj`) can be mixed in expressions. This makes it possible to remember the state of the search between invocations of a heuristic. All of the following combinators make use of this feature.

Restarting branch-and-bound This is a twist on regular branch-and-bound that restarts whenever a solution is found.

$$\mathsf{restart_bab}(\mathtt{obj}, s) \equiv \mathsf{let}(best, \infty, \mathsf{restart}(true, \mathsf{and}([\mathsf{post}(\mathtt{obj} < best), \mathsf{once}(s), \mathsf{assign}(best, \mathtt{obj})])))$$

Radiotherapy treatment planning The following search heuristic can be used to solve radiotherapy treatment planning problems [1]. The heuristic minimizes a variable k using branch-and-bound (bab), first searching the variables N, and then verifying the solution by partitioning the problem along the row_i variables for each row i one at a time (expressed as a MiniZinc array comprehension). Failure on one row must be caused by the search on the variables in N, and consequently search never backtracks into other rows.

This behavior is similar to the once combinator defined above. However, when a single solution is found, the search should be considered exhaustive. We therefore

[4] They are typeset in typewrite font to distinguish them from search variables.

need a committed-choice variant of once that is exhaustive when a solution is found. This exhaustive variant can be implemented by replacing prune with post(*false*):

$$\text{exh_once}(s) \equiv \text{ifthenelse}(\text{solutions} < 1, s, \text{post}(\mathit{false}))$$

This allows us to express the entire search strategy for radiotherapy treatment planning:[5]

$$\text{bab}(k, \text{and}([\text{base_search}(N, \ldots)]{+}{+} [\text{exh_once}(\text{base_search}(row_i, \ldots)) \mid i \text{ in } 1..n]))$$

For The for loop construct ($v \in [l, u]$) can be defined as:

$$\text{for}(v, l, u, s) \equiv \text{let}(v, l, \text{restart}(v \leq u, \text{portfolio}([s, \text{and}([\text{assign}(v, v+1), \text{prune}])])))$$

It simply runs $u - l + 1$ times the search s, which of course is only sensible if s makes use of side effects or the loop variable v. As in the geom_restart combinator above, prune propagates the non-exhaustiveness of s to the restart combinator.

Limited discrepancy search [8] with an upper limit of l discrepancies for an underlying search s.

$$\text{lds}(l, s) \equiv \text{for}(n, 0, l, \text{limit}(\text{discrepancies} \leq n, s))$$

The for construct iterates the maximum number of discrepancies n from 0 to l, while limit executes s as long as the number of discrepancies is smaller than n. The search makes use of the discrepancies statistic that is maintained by the search infrastructure. The original LDS [8] visits the nodes in a specific order. The search described here visits the same nodes in the same order of discrepancies, but possibly in a different individual order—as this is determined by the global queuing strategy.

The following is a combination of branch-and-bound and limited discrepancy search for solving job shop scheduling problems, as described in [8]. The heuristic searches the Boolean variables `prec`, which determine the order of all pairs of tasks on the same machine. As the order completely determines the schedule, we then fix the start times using exh_once.

$$\text{bab}(\mathit{makespan}, \text{lds}(\infty, \text{and}([\text{base_search}(\texttt{prec}, \ldots), \text{exh_once}(\text{base_search}(\texttt{start}, \ldots))])))$$

Fully expanded, this heuristic consists of 17 combinators and is 11 combinators deep.

[5] ++ denotes list concatenation.

Iterative deepening [11] for an underlying search s is a particular instance of the more general pattern of restarting with an updated bound, which we have already seen in the geom_restart example. Here, we generalize this idea:

$$\begin{aligned}
&\mathsf{id}(s) \equiv \mathsf{ir}(\mathsf{depth}, 0, +, 1, \infty, s)\\
&\mathsf{ir}(p, l, \oplus, i, u, s) \equiv \mathsf{let}(n, l,\ \mathsf{restart}(n \leq u, \mathsf{and}([\mathsf{assign}(n, n \oplus i),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{limit}(p \leq n, s)])))
\end{aligned}$$

With let, bound n is initialized to l. Search s is pruned when statistic p exceeds n, but iteratively restarted by restart with n updated to $n \oplus i$. The repetition stops when n exceeds u or when s has been fully explored. The bound increases geometrically, if we supply $*$ for $\oplus$, as in the restart_flip heuristic:

$$\begin{aligned}
&\mathsf{restart_flip}(p, l, i, u, s_1, s_2) \equiv \mathsf{let}(\mathit{flip}, 1, \mathsf{ir}(p, l, *, i, u, \mathsf{and}([\mathsf{assign}(\mathit{flip}, 1 - \mathit{flip}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ifthenelse}(\mathit{flip} = 1, s_1, s_2)])))
\end{aligned}$$

The restart_flip search alternates between two search heuristics s_1 and s_2. Using this as its default strategy in the *free search* category, the lazy clause generation solver *Chuffed* scored most points in the 2010, 2011, and 2012 MiniZinc Challenges.[6]

Probe search Try out two searches s_1 and s_2 to a limited extent defined by condition *cond*. Then, for the remainder, use the search that resulted in the best solution so far.

$$\begin{aligned}
&\mathsf{probe}(\mathit{cond}, \mathit{obj}, s_1, s_2) \equiv \mathsf{let}(\mathit{best}_1, \infty,\\
&\qquad \mathsf{let}(\mathit{best}_2, \infty,\\
&\qquad\quad \mathsf{portfolio}([\ \mathsf{limit}(\mathit{cond}, \mathsf{and}([s_1, \mathsf{assign}(\mathit{best}_1, \mathit{obj})]))\\
&\qquad\qquad\qquad \mathsf{limit}(\mathit{cond}, \mathsf{and}([s_2, \mathsf{assign}(\mathit{best}_2, \mathit{obj})]))\\
&\qquad\qquad\qquad \mathsf{ifthenelse}(\mathit{best}_1 \leq \mathit{best}_2, s_1, s_2)])))
\end{aligned}$$

Dichotomic search [26] solves an optimization problem by repeatedly partitioning the interval in which the possible optimal solution can lie. It can be implemented by restarting as long the lower bound has not met the upper bound (line 2), computing the middle (line 3), and then using an or combinator to try the lower half (line 5). If it succeeds, $\texttt{obj} - 1$ is the new upper bound, otherwise, the lower bound is increased (line 6).

$$\begin{aligned}
&\mathsf{dicho}(s, \texttt{obj}, \mathit{lb}, \mathit{ub}) \equiv \mathsf{let}(l, \mathit{lb}, \mathsf{let}(u, \mathit{ub},\\
&\qquad \mathsf{restart}(l < u,\\
&\qquad\quad \mathsf{let}(h, l + \lceil (u - l)/2 \rceil,\\
&\qquad\quad \mathsf{once}(\mathsf{or}([\\
&\qquad\qquad \mathsf{and}([\mathsf{post}(l \leq \texttt{obj} \leq h), s, \mathsf{assign}(u, \texttt{obj} - 1)]),\\
&\qquad\qquad \mathsf{and}([\mathsf{assign}(l, h + 1), \mathsf{prune}])]))\\
&\qquad))))
\end{aligned}$$

[6] http://www.g12.csse.unimelb.edu.au/minizinc/

4 Modular combinator design

The previous section caters for the user's needs, presenting a high-level modular syntax for our combinator-based search language. To cater for advanced users' and system developers' needs, this section goes beyond modularity of syntax, introducing modularity of *design*.

Modularity of design is the one property that makes our approach practical. Each combinator corresponds to a separate module that has a meaning and an implementation independent of the other combinators. This enables us to actually realize the search specifications defined by modular syntax.

Modularity of design also enables growing a system from a small set of combinators (e.g., those listed in Fig. 1), gradually adding more as the need arises. Advanced users can complement the system's generic combinators with a few application-specific ones. Compared to creating new heuristics by just combining primitives, adding new combinators of course requires a deeper insight into the implementation details and therefore comes at a higher development cost. We believe that our architecture strikes the right balance with the split into a simple high-level language that caters for most users' needs, and a more complex but still compositional implementation for advanced users and system developers.

Solver independence is another notable property of our approach. While a few combinators access solver-specific functionality (e.g., base_search and post), the approach as such and most combinators listed in Fig. 1 are in fact generic (solver- and even CP-independent); their design and implementation is reusable.

The solver-independence of our approach is reflected in the minimal interface that solvers must implement. This interface consists of an abstract type `State` which represents a state of the solver (e.g., the variable domains and accumulated constraint propagators) which supports state restoration. Truly no more is needed for the approach or all of the primitive combinators in Fig. 1, except for base_search and post which require CP-aware operations for querying variable domains, solver status and posting constraints, and possibly interacting with statistics maintained by the solver. Note that state restoration can be implemented either by means of copying for copying solvers, or by means of recomputation techniques [16] for trailing-based solvers. Hence, there need not be a 1-to-1 correspondence between an implementation of the abstract `State` type and the solver's actual state representation. We have implementations of the interface based on both copying and trailing.

In the following we explain our design in detail by means of code implementations of most of the primitive combinators we have covered in the previous section.

4.1 The message protocol

To obtain a modular design of search combinators we step away from the idea that the behavior of a search combinator, like the and combinator, forms an indivisible whole; this leaves no room for interaction. The key insight here is that we must identify finer-grained steps, defining how different combinators interact at each node in the search tree. Interleaving these finer-grained steps of different combinators in an appropriate manner yields the composite behavior of the overall search heuristic, where each combinator is able to cross-cut the others' behavior.

Considering the diversity of combinators and the fact that not all units of behavior are explicitly present in all of them, designing this protocol of interaction is non-trivial. It requires studying the intended behavior and interaction of combinators to isolate the fine-grained units of behavior and the manner of interaction. The contribution of this section is an elegant and conceptually uniform design that is powerful enough to express all the combinators presented in this article.

The messages We present this design in the form of a *message protocol*. The protocol specifies a set of messages (i.e., an interface with one procedure for each fine-grained step) that have to be implemented by all combinators. In pseudo-code, this protocol for combinators consists of four different messages:

```
protocol combinator
  start(rootNode);
  enter(currentNode);
  exit(currentNode,status);
  init(parentNode,childNode);
```

The protocol concerns the *dynamic* behavior of a search combinator. A single static occurrence of a search combinator in a search heuristic may have zero or more dynamic *life cycles*. During a life cycle, the combinator observes and influences the search of a particular subtree of the overall search tree.

- The message `start(rootNode)` starts up a new life cycle of a combinator for the subtree rooted at `rootNode`. The typical implementation of this message allocates and initializes data for the life cycle.
- The message `enter(currentNode)` notifies the combinator that the node `currentNode` of its subtree is currently active. At this point the combinator may for instance decide to prune it.
- The message `exit(currentNode,status)` informs the combinator that the currently active node `currentNode` is a leaf node of its subtree. The node's status is one of **failure**, **success** or **abort** which denote respectively an inconsistent node, a solution and a pruned node.
- The message `init(parentNode,childNode)` registers with the combinator the node `childNode` as a child node of the currently active node `parentNode`.

Typically, during a life cycle, a combinator sees every node three times. The first time the node is included in the life cycle, either as a root with `start` or as the child of another node with `init`. The second time the node is processed with `enter`. The last time the node processing has determined that the node is either a leaf with `exit` or the parent of one or more other nodes with `init`.

The nodes All of the message signatures specify one or two search tree *nodes* as parameters. Each such node keeps track of a solver `State` and the information associated by combinators to that `State`.

We observe three different access patterns of nodes:

1. In keeping with the solver independence stipulated above, we will see that most combinators only query and update their associated information and do not access the underlying solver `State` at all.

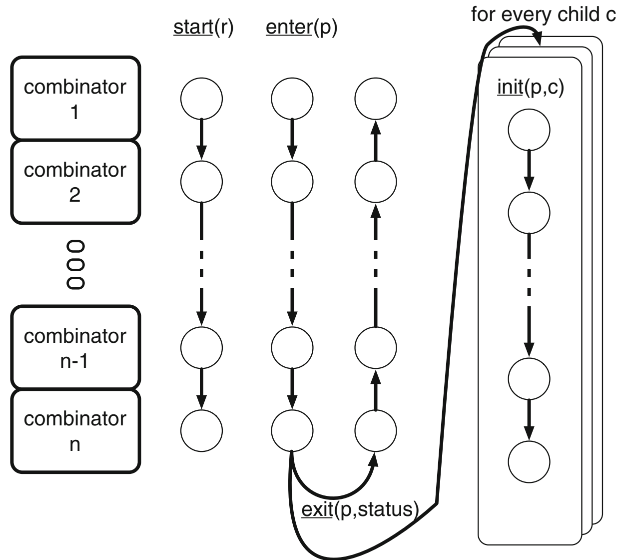


Fig. 3 The modular message protocol

2. Restarting-based combinators, like `restart` and `portfolio`, copy nodes. This means copying the solver's `State` representation and all associated information for later restoration.
3. Finally, selected solver-specific combinators like `base_search` do perform solver-specific operations on the underlying `State`, like querying variable domains and posting constraints.

The calling hierarchy In addition to the message signatures, the protocol also stipulates in what order the messages are sent among the combinators (see Fig. 3). While in general a combinator composition is tree-shaped, the processing of any single search tree node p only involves a stack of combinators. For example, given or([and$_1$([s_1, s_2]), and$_2$([s_3, s_4])]),[7] p is included in life cycles of [or, and$_1$, s_1], [or, and$_1$, s_2], [or, and$_2$, s_3] or [or, and$_2$, s_4]. We also say that the particular stack is *active* at node p. The picture shows this stack of active combinators on the left.

Every combinator in the stack has both a *super*-combinator above and a *sub*-combinator below, except for the *top* and the *bottom* combinators. The bottom is always a basic heuristic (base_search, prune, assign, or post). The important aspect to take away from the picture is the direction of the four different messages, either top-down or bottom-up.

The protocol initializes search by sending the `start(root)` message, where `root` is the root of the overall search tree, to the topmost combinator. This topmost combinator decides what child combinator to forward the message to, that child combinator propagates it to one of its children and so on, until a full stack of combinators is initialized.

Next, starting from the root node, nodes are processed in a loop. The `enter(`**`node`**`)` message is passed down through the stack of combinator stack to the

[7]The left and right and are subscripted to distinguish them.

primitive heuristic at the bottom, which determines whether the node is a leaf or has children. In the former case, the primitive heuristic passes the exit(**node**, status) message up. In the latter case, it passes the init(**node**, child) message down from the top for each child. These child nodes are added to the queue that fuels the loop. At any point, intermediate combinators can decide not to forward messages literally, but to alter them instead (e.g., to change the status of a leaf from **success** to **abort**), or to initiate a different message flow (e.g. to start a new subtree).

4.2 Basic setup

Before we delve into the interesting search combinators, we first present an example implementation of the basic setup consisting of a base search (base_search) and a search engine (dfs). This allows us to express overall search specifications of the form: dfs(base_search(*vars*, *var-select*, *domain-split*)).

Base search We do not provide full details on a base_search combinator, as it is not the focus of this article. However, we will point out the aspects relevant to our protocol.

The first line of base_search's implementation expresses two facts. Firstly, base_search implements the **combinator** protocol. Secondly, its constructor has three parameters (vars, var-select, domain-select) that can be referred to in its message implementations.

In the enter message, the node's solver state is propagated. Subsequently, the condition isLeaf(c,vars) checks whether the solver state is unsatisfiable or there are no more variables to assign. If either is the case, the exit status (respectively **failure** or **success**) is sent to the parent combinator. For now, the parent combinator is just the search engine, but later we will see how how other combinators can be inserted between the search engine and the base search.

If neither is the case, the search branches depending on the variable selection and domain splitting strategies. This involves creating a child node for each branch, determining the variable and value for that child and posting the assignment to the child's state. Then, the **top** combinator (i.e., the engine) is asked to initialize the child node. Finally the child node is pushed onto the search queue.

```
combinator base_search(vars,var-select,domain-select)
  enter(c):
    c.propagate
    if isLeaf(c,vars)
      parent.exit(c,leafstatus(c))
    pos = ...       // from vars based on var-select
    for each child: // based on domain-select
      val = ...     // from values of var based on domain-select
      child.post(vars[pos]=val)
      top.init(c,child)
      queue.push(child)
```

Note that, as the base_search combinator is a base combinator, its exit message is immaterial (there is no child heuristic of base_search that could ever call it). The start and init messages are empty. Many variants on and generalizations of the above implementation are possible.

Depth-first search engine The engine dfs serves as a pseudo-combinator at the **top** of a combinator expression heuristic and serves as the heuristic's immediate parent as well. It maintains the **queue** of nodes, a stack in this case. The search starts from a given root node by starting the heuristic with that node and then entering it. Each time a node has been processed, new nodes may have been pushed onto the queue. These are popped and entered successively.

```
combinator dfs(heuristic)
  start(root):
    top=this
    heuristic.parent=this
    queue=new stack()
    heuristic.start(root)
    heuristic.enter(root)
    while not queue.empty
      heuristic.enter(queue.pop())

  init(n,c):
    heuristic.init(n,c)
```

The engine's exit message is empty, the enter message is never called and the init message delegates initialization to the heuristic.

Other engines may be formulated with different queuing strategies.

4.3 Combinator composition

The idea of search combinators is to augment a base_search. We illustrate this with a very simple print combinator that prints out every solution as it is found. For simplicity we assume a solution is just a set of constraint variables *vars* that is supplied as a parameter. Hence, we obtain the basic search setup with solution printing with:

```
dfs(print(vars,base_search(vars,strategy)))
```

Print The print combinator is parametrized by a set of variables vars and a search combinator child. Implicitly, in a composition, that child's parent is set to the print instance. The same holds for all following search combinators with one or more children.

The only message of interest for print is exit. When the exit status is **success**, the combinator prints the variables and propagates the message to its parent.

```
combinator print (vars,child)
  exit(c,status):
    if status==success
      print c.vars
    parent.exit(c,status)
```

The other messages are omitted. Their behavior is default: they all propagate to the child. The same holds for the omitted messages of following unary combinators.

4.4 Binary combinators

Binary combinators are one step up from unary ones. They combine two complete search heuristics into a composite one. The most basic binary combinator is the binary version of and. For instance, if we need to label two sets of variables, we can do so with

$$\text{and}(\text{base_search}(vars_1, \ldots), \text{base_search}(vars_2, \ldots))$$

The principle shown here easily generalizes to n-ary combinators.

And The (binary) and combinator has two children, `left` and `right`. In order to keep track of what child combinator is handling a particular node, the and combinator associates with every node an `inLeft` Boolean variable. The **`local`** keyword indicates that every node has its own instance of that variable. We denote the instance of the `inLeft` variable associated with node `c` as `c.inLeft`.

When entering a node, it is delegated to the `left` or `right` combinator based on `inLeft`. At the start, the `root` node is delegated to the `left` combinator, so its `inLeft` variable is set to **`true`**. The value of `inLeft` is inherited in init from the current node to its children. Upon a successful exit for `left`, the leaf node becomes the root of a new subtree that is further handled by the `right` combinator.

```
combinator and(left,right) {
  local bool inLeft

  start(root):
    root.inLeft=true
    left.start(root)

  enter(c):
    if c.inLeft
      left.enter(c)
    else
      right.enter(c)

  exit(c,status):
    if c.inLeft and status==success
      c.inLeft=false
      right.start(c)
      right.enter(c)
    else
      parent.exit(c,status)

  init(p,c):
    c.inLeft=p.inLeft
    if c.inLeft
      left.init(p,c)
    else
      right.init(p,c)
```

Note that the `right` combinator is `start`ed repeatedly, once for each leaf node of `left`. In general, each combinator can be managing multiple subtrees of the search.

Multiple and combinators may be handling a search node at the same time. For instance in a heuristic of the form and(and(s_1, s_2), s_3), two and combinators are active at the same time. The scoping of the associated variables works in such a way that each and has its own instance of `inLeft` for each node.

4.5 Reusable combinators

Now we show how a *monolithic* combinator can be decomposed into more primitive combinators that can be reused for other purposes.

Monolithic combinator We start from the following limitsolutions combinator that prunes the search after `cutoff` solutions have been found. One new concept is the notion of a global variable associated with a (sub)tree: all descendants of `root` (implicitly) share the same instance of `count`. Hence, any update of `count` by one node is seen by all other nodes in the (sub)tree.

```
combinator limitsolutions(cutoff,child)
  global int count

  start(root):
    root.count = 0
    child.start(root)

  enter(c):
    if count == cutoff
      parent.exit(abort)
    else
      child.enter(c)

  exit(c,status):
    if status==success
      c.count++
    parent.exit(c,status)
```

Decomposition We can split up the above limitsolutions combinator into three different combinators: ifthenelse, solutionslimit and prune. They form a directed acyclic graph as depicted in Fig. 4 or denoted as an expression with sharing below:

limitsolutions(`cutoff,s`) = ifthenelse(`s',s'`,prune)

where the same heuristic object `s'` = solutionslimit(`cutoff,s`) is shared between the first and a second parameter of ifthenelse.

Here, `s'` is the heuristic `s` augmented with solutionslimit to monitor how many solutions are left to find until `cutoff` is reached. It is in the capacitiy of a heuristic

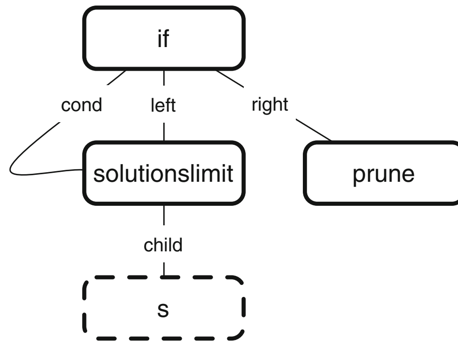


Fig. 4 The decomposition of the limitsolutions combinator

that `s′` occurs in the second parameter position of ifthenelse: Initially, ifthenelse makes `s′` is active child heuristic. If the `cutoff` is reached, then ifthenelse switches to prune as its active child, which discards the remaining nodes in the tree. This involves `s′` in its capacity as the first parameter of ifthenelse, where it tells ifthenelse how to evaluate its condition, namely by querying the solutionslimit object.

Note that the sharing in this example is somewhat odd at first sight, but perfectly natural on second thought. It is due to the generality of ifthenelse: its condition need not be determined by the topmost combinator of its then-child. Instead, the condition can be derived from a combination of sources in the active combinator stack. Hence, the condition parameter needs to specify how the condition value is obtained. In order to facilitate the common case of sharing for end users of the combinators, the sharing is easily hidden by more convenient syntax such as ifthenelse(`solutions <= cutoff`,`s`,prune) (e.g., see the examples in Section 3).

In the following we discuss the individual primitives in more detail.

Prune The prune combinator is a minimal base combinator that immediately exits every node with the **abort** status. The start message is empty, and the exit and init messages are never called.

```
combinator prune ()
  enter(c):
    parent.exit(c,abort)
```

Solutions count The solutionslimit combinator below illustrates how statistics gathering combinators are implemented. It implements both the **combinator** protocol and the additional **condition** protocol with an extra message eval that queries the current Boolean value:

```
protocol condition
  eval(currentNode);
```

In the case of solutionslimit, the returned Boolean value is whether a particular number (`cutoff`) of solutions has not yet been reached by its `child`.

For this purpose it maintains the number of solutions found so far in a global variable.

```
combinator & condition solutionslimit(cutoff,child)
  global int count

  start(root):
    root.count = 0
    child.start(root)

  exit(c,status):
    if status==success
      c.count++
    parent.exit(c,status)

  eval(c):
    return c.count <= cutoff
```

Ifthenelse The ifthenelse combinator is parametrized by one **condition** and two child combinators. It associates with every node whether it is handled by the `left` child (`inLeft`); this is the case for the `root` node. Whenever a node `c` is entered that is `inLeft`, the condition is checked. If the condition fails, `c` becomes the root of a subtree that is further handled by `right`.

```
combinator ifthenelse(cond,left,right)
  local bool inLeft

  start(root):
    root.inLeft=true
    left.start(root)

  enter(c):
    if not c.inLeft
      right.enter(c)
    else if cond.eval()
      left.enter(c)
    else
      c.inLeft=false
      right.start(c)
      right.enter(c)

  init(p,c):
    c.inLeft=p.inLeft
    if c.inLeft
      left.init(p,c)
    else
      right.init(p,c)
```

4.6 Restarting combinators

Restarting the search is common to several combinators; the mechanic is illustrated below in the portfolio combinator.

Portfolio Like the ifthenelse and and combinators, the portfolio combinator switches between child combinators. Only the logic for switching is more complex. In order to simplify presentation, we again restrict the code to the binary case; the n-ary variant is a straightforward generalization.

Firstly, portfolio keeps track of a global "reference" count `ref` of unprocessed nodes to be handled by the `s1` child. This count is incremented whenever a new child node is initialized, and decremented whenever a node is entered for actual processing.

When the last node of `s1` exits (witnessed by the reference count being 0) and the search was not exhaustive, the search starts over from the root, but now with the `s2` child. In order to decide about exhaustiveness, the portfolio combinator registers whether any `exit` with status **`abort`** occurred. At the same time it converts an **`abort`** inside `s1` into a `failure`, because the `s2` combinator may still perform an exhaustive search and avoid overall non-exhaustiveness. In order to restart from the `root`, a `copy` of the `root` node is made at the `start`.

Upon a successful `exit`, the leaf node becomes the root of a new subtree that is further handled by the `s2` combinator.

```
combinator portfolio(s1,s2)
  global node copy
  global bool inLeft
  global bool exhaustive
  global int ref

  start(root):
    copy=root.copy()
    root.inLeft=true
    root.exhaustive=true
    root.ref=1
    s1.start(root)

  enter(c):
    if c.inLeft
      ref--
      s1.enter(c)
    else
      s2.enter(c)

  exit(c,status):
    if not c.inLeft
      parent.exit(c,status)
    else
      if status==abort
```

```
        status=failure
        c.exhaustive=false
      if c.ref==0
        if c.exhaustive
          parent.exit(c,status)
        else
          copy.inLeft=false
          s2.start(copy)
          self.enter(copy)
      else
        parent.exit(c,status)

  init(p,c):
    ref++;
    if c.inLeft
      s1.init(p,c)
    else
      s2.init(p,c)
```

5 Modular combinator implementation

The message-based combinator approach lends itself well to different implementation strategies. In the following we briefly discuss two diametrically opposed approaches we have explored:

Dynamic composition implements combinators as objects that can be combined arbitrarily at runtime. It therefore acts like an *interpreter*. This is a lightweight implementation, it can be ported quickly to different platforms, and it does not involve a compilation step between the formulation and execution of a search heuristic.

Static composition uses a code generator to translate an entire combinator expression into executable code. It is therefore a *compiler* for search combinators. This approach lends itself better to various kinds of analysis and optimization.

As both approaches are possible, combinators can be adapted to the implementation choices of existing solvers. Section 6 shows that both implementation approaches have competitive performance.

5.1 Dynamic composition

To support dynamic composition, we have implemented our combinators as C++ classes whose objects can be allocated and composed into a search specification at runtime. The protocol events correspond to virtual method calls between these objects. For the delegation mechanism from one object to another, we explicitly encode a form of dynamic inheritance called *open recursion* or *mixin inheritance* [4].

In contrast to the OOP inheritance built into C++ and Java, this mixin inheritance provides two essential abilities: 1) to determine the inheritance graph *at runtime* and 2) to use multiple copies of the same combinator class at different points in the inheritance graph. In contrast, C++'s built-in static inheritance provides neither.

The C++ library currently builds on top of the Gecode constraint solver [25]. However, the solver is accessed through a layer of abstraction that is easily adapted to other solvers (e.g., we have a prototype interface to the Gurobi MIP solver). The complete library weighs in at around 2500 lines of code, which is even less than Gecode's native search and branching components.

5.2 Static composition

In a second approach, also on top of Gecode, we statically compile a search specification to a tight C++ loop. Again, every combinator is a separate module independent of other combinator modules. A combinator module now does not directly implement the combinator's behavior. Instead it implements a code generator (in Haskell), which in turn produces the C++ code with the expected behavior.

Hence, our search language compiler parses a search specification, and composes (in mixin-style) the corresponding code generators. Then it runs the composite code generator according to the message protocol. The code generators produce appropriate C++ code fragments for the different messages, which are combined according to the protocol into the monolithic C++ loop. This C++ code is further post-processed by the C++ compiler to yield a highly optimized executable.

As for dynamic composition, the mixin approach is crucial, allowing us to add more combinators without touching the existing ones. At the same time we obtain with the press of a button several 1000 lines of custom low-level code for the composition of just a few combinators. In contrast, the development cost of hand crafted code is prohibitive.

As the experiments in the next section will show, compiling the entire search specification into an optimised executable achieves better performance than dynamic composition. However, the dynamic approach has the big advantage of not requiring a compilation step, which means that search specifications can be constructed at runtime, as exemplified by the following application.

5.3 MiniZinc with combinators

As a proof of concept and platform for experiments, we have integrated search combinators into a complete MiniZinc toolchain. It translates a MiniZinc model together with a search annotation into FlatZinc, which is then interpreted and executed.

Our toolchain comprises a pre-compiler, which is necessary to support arbitrary expressions in annotations, such as the condition expressions for an ifthenelse. The expressions are translated into standard MiniZinc annotations that are understood by the FlatZinc interpreter. User-defined variables have type-inst `svar int` and can be introduced using the standard MiniZinc `let` construct. The `annotation` construct of MiniZinc has been extended to support simple function definitions. The

following example shows a MiniZinc version of the restart-based branch-and-bound heuristic from Section 3.3:

```
annotation limit(var bool: cond, ann: s) =
    ifthenelse(cond,s,prune);

annotation once(ann: s) = limit(solutions < 1, s);

annotation rbab(var int: obj, ann: s) =
  let { svar int: best = MAXINT } in
  restart(true, and([
            post(obj < best),
            once(s),
            assign(best,obj)]));

solve ::rbab(x,int_search(y,input_order,assign_lb)) satisfy;
```

The pre-compiler translates this code as follows:

```
solve :: sh_let(sh_letvar("best"), sh_int(MAXINT),
  sh_restart(sh_cond_true, sh_and([
    sh_post_succeed(sh_cond_lt(sh_intvar(objective),
                               sh_letvar("best"))),
    sh_let(sh_letvar("solutioncount"), 0,
      sh_ifthenelse(sh_cond_lt(sh_letvar("solutioncount"),
                               sh_int(1)),
        sh_solutioncount(sh_letvar("solutioncount"),
          sh_int_search(x, sh_var_input_order,
                           sh_val_assign_lb)),
        sh_prune)),
    sh_assign(sh_letvar("best"), sh_intvar(objective))])))
    satisfy;
```

All literals are quoted (e.g. `sh_int(1)`), user-defined search variables are turned into quoted strings (`sh_letvar("best")`), expressions like `obj < best` are translated into annotation terms (`sh_cond_lt ...`), and statistics are made explicit, introducing search variables and special combinators (`sh_solutioncount`). The result of the pre-compilation is valid, well-typed MiniZinc, which is then passed through the standard `mzn2fzn` translator to produce FlatZinc ready for solving. We intend to incorporate the translations done by the pre-compiler into the standard `mzn2fzn` in the future.

We extended the Gecode FlatZinc interpreter to parse the search combinator annotation and construct the corresponding heuristic using the Dynamic Composition approach described above. The three tools, pre-compiler, `mzn2fzn`, and the modified FlatZinc interpreter thus form a complete toolchain for solving MiniZinc models using search combinators. The source code including examples can be downloaded from http://www.gecode.org/flatzinc.html. If developers for other systems that support MiniZinc can be persuaded to implement search combinators for their

system, which we believe is not too difficult, then we can see search combinators as a basis for standardizing MiniZinc search.

5.4 Further implementations

We are in the process of implementing the search combinators approach on two more platforms:

Prolog Our Tor library [23] implements a subset of the search message protocol in Prolog. The library is currently available for SWI-Prolog [32] and B-Prolog [33], and extends the capabilities of their respective finite domain solver libraries. Among others, it provides all the search heuristics of ECLiPSe Prolog's [20] `search/6` predicate, but in a fully compositional way. The library implements the dynamic approach supplemented with load-time program specialization.

Scala Desouter [6] has implemented a preliminary library of search combinators for Scala [5] on the Java Virtual Machine. His implementation exploits Scala's built-in mixin mechanism (called *traits*) to further factorize the combinator implementations. The library's current backend is the JaCoP solver [12].

6 Experiments

This section evaluates the performance of the dynamic and static implementations. It establishes that a search heuristic specified using combinators is competitive with a custom implementation of the same heuristic, exploring exactly the same tree.

Section 4.1 introduced a message protocol that defines the communication between the different combinators *for one node of the search tree*. Any overhead of a combinator-based implementation must therefore come from the processing of each node using this protocol. All combinators discussed earlier process each message of the protocol in constant time (except for the base_search combinators, of course). Hence, we expect at most a constant overhead per node compared to a native implementation of the heuristic.

In the following, two sets of experiments confirm this expectation. The first set consists of artificial benchmarks designed to expose the overhead per node. The second set consists of realistic combinatorial problems with complex search strategies.

The experiments were run on a 2.26 GHz Intel Core 2 Duo running Mac OS X. The results are the means of 10 runs, with a coefficient of deviation less than 1.5 %.

Stress test The first set of experiments measures the overhead of calling a single combinator during search. We ran a complete search of a tree generated by 7 variables with domain $\{0, \ldots, 6\}$ and *no* constraints (1 647 085 nodes). To measure the overhead, we constructed a basic search heuristic s and a stack of n combinators:

$$\mathsf{portfolio}([\mathsf{portfolio}([\ldots\mathsf{portfolio}([s,\mathsf{prune}])\ldots,\mathsf{prune}]),\mathsf{prune}])$$

where n ranges from 0 to 20 (realistic combinator stacks, such as those from the examples in this article, are usually not deeper than 10). The numbers in the following

table report the runtime with respect to using the plain heuristic s, for both the static and the dynamic approach:

n	1	2	5	10	20
Static %	106.6	107.7	112.0	148.3	157.5
Dynamic %	107.3	117.6	145.2	192.6	260.9

A single combinator generates an overhead of around 7 %, and 10 combinators add 50 % for the static and 90 % for the dynamic approach. In absolute runtime, however, this translates to an overhead of around 17 ms (70 ms) per million nodes and combinator for the static (dynamic) approach. Note that this is a worst-case experiment, since there is no constraint propagation and almost all the time is spent in the combinators.

Benchmarks The second set of experiments shows that in practice, this overhead is dwarfed by the cost of constraint propagation and backtracking. Note that the experiments are not supposed to demonstrate the best possible search heuristics for the given problems, but that a search heuristic implemented using combinators is just as efficient as a native implementation.

Figure 5 compares Gecode's optimization search engines with branch-and-bound implemented using combinators. The column *Compiled* shows the absolute runtime of the Static Composition approach. The column *Interpreted* is the relative runtime of the Dynamic Composition approach compared to *Compiled*. The column *Gecode* is the relative runtime of the native Gecode search engines (i.e., not using combinators at all), compared to *Compiled*. For each problem instance, all three approaches use exactly the same search strategy and explore the same trees.

	Compiled	*Interpreted*	*Gecode*
Golomb 10	0.61 s	101.8%	102.5%
Golomb 11	12.72 s	102.9%	101.8%
Golomb 12	125.40 s	100.6%	101.9%
Radiotherapy 1	71.13 s	105.9%	107.3%
Radiotherapy 2	6.22 s	110.9%	110.1%
Radiotherapy 3	11.78 s	108.3%	108.1%
Radiotherapy 4	16.44 s	107.5%	106.9%
Radiotherapy 5	69.89 s	108.1%	98.7%
Radiotherapy 6	106.04 s	109.2%	99.1%
Job Shop G2	7.25 s	146.3%	101.2%
Job-Shop G4	6.96 s	164.0%	107.75%
Job-Shop H1	38.05 s	153.1%	103.81%
Job Shop H3	52.02 s	162.5%	102.8%
Job Shop H5	20.88 s	153.2%	107.0%
Job Shop ABZ1-5	2319.00 s	103.7%	100.1%
Job Shop mt10	2181.00 s	104.5%	99.9%

Fig. 5 Experimental results

On the well-known Golomb Rulers problem, both dynamic combinators and native Gecode are slightly slower than static combinators. Native Gecode uses dynamically combined search heuristics, but is much less expressive. That is why the static approach with its specialization yields better results.

On the radiotherapy problem (see Section 3.3), the dynamic combinators show an overhead of 6–11 %. For native Gecode, `exh_once` must be implemented as a nested search, which performs similarly to the dynamic combinators. However, in instances 5 and 6, the compiled combinators lose their advantage over native Gecode. This is due to the processing of `exh_once`: As soon as it is finished, the combinator approach processes all nodes of the `exh_once` tree that are still in the search queue, which are now pruned by `exh_once`. The native Gecode implementation simply discards the tree. We will investigate how to incorporate this optimization into the combinator approach.

The job shop scheduling examples, using the combination of branch-and-bound and discrepancy limit discussed in Section 3.3, show similar behavior. In ABZ1-5 and mt10, the interpreted combinators show much less overhead than in the short-running instances. This is due to more expensive propagation and backtracking in these instances, which spend almost 70 % more time per node than the short-running instances. Therefore, as the absolute time spent per combinator per node is constant, the relative overhead of executing the combinators is much lower.

In summary, the experiments show that the expressiveness and flexibility of a rich combinator-based search language can be achieved without any runtime overhead in the case of the static approach, and little overhead for the dynamic version.

7 Related work

This section explores and discusses previous work that is closely related to search combinators as presented in this article.

7.1 MCP

This work directly extends our earlier work on *Monadic Constraint Programming* (MCP) [21]. MCP introduces stackable search transformers, which are a simple form of search combinators, but only provide a much more limited and low level form of search control. In trying to overcome its limitations we arrived at search combinators.

7.2 Constraint logic programming

Constraint logic programming languages such as ECLiPSe [20] and SICStus Prolog [28] provide programmable search via the built-in search of the paradigm, allowing the user to define *goals* in terms of conjunctive or disjunctive sub-goals.

Prolog's limitation is that it does not permit cross-cutting between goals. For instance, disjunctions inside goals are too well encapsulated to observe them or interfere with them from outside that goal. Hence, combinators that inject additional behavior in disjunctions, i.e. to observe and/or prune the number of branches, cannot be expressed in a modular way. In contrast, cross-cutting is a crucial feature of our combinator approach, where a combinator higher up in the stack can interfere

with a sub-combinator, while remaining fully compositional. In summary, apart from conjunction and disjunction, Prolog's goal-based heuristics cannot be combined arbitrarily.

ECLiPSe copes with this limitation by combining a limited number of search heuristics into a monolithic `search/6` predicate. With various parameters the user controls which of the heuristics is enabled (e.g., depth-bounded, node-bounded or limited discrepancy search). A fixed number of compositions are supported, such as changing strategy when the depth bound finishes. The labeling itself is user programmable. If a user is not happy with the set of supported heuristics in `search/6`, they have to program his own from scratch.

7.3 Node evaluators, search selectors and search limits

Perron [16] describes a compositional approach to search where search heuristics are called *goals*. In addition to basic user-defined goals, he proposes five predefined combinators. These five combinators consist of the conventional binary `And` and `Or` combinators, as well as the unary combinators `Apply`, `SelectSearch` and `LimitSearch`. The three unary ones are parameterized by respectively a *node evaluator*, a *search selector* and a *search limit*:

- A node evaluator influences the position of the node in the queue.
- A search selector combines three roles: 1) management of branch-&-bound minimization, 2) determining whether a node is feasible, and 3) selection of solutions.
- A search limit (time or failure limit) aborts the remaining search when a global limit is exceeded.

This design is less uniform than our search combinators approach as it assigns different tasks to more specialized entities. At the same time, this approach does not seem intended to support additional combinators, such as our ifthenelse and restart, which enable random restarting and restarting branch-&-bound among others. The approach consequently does not cover aspects such as exhaustiveness which allow distinguishing between or and portfolio, which is necessary for restart. There is very little detail given about the implementation of the approach, and in particular how combinators interact.

Finally, this approach [16] caters specifically for depth-first search, based on trailing and recomputation, with a particular priority queue. Our approach is orthogonal to these choices. Nevertheless, it would be interesting to explore his concept of interaction between combinator and queue in the search combinator setting.

7.4 IBM ILOG CP optimizer

The *CP Optimizer* C++ library [9] of IBM ILOG offers support for fully programmable search in three different ways.

1. At one level, search heuristics are called `IlcGoal`. Programmers can write their own primitive `IlcGoals` and the library provides two combinators, `IlcAnd` and `IlcOr`, similar to our and and portfolio combinators.

2. At another level, search heuristics are called `IloGoal`. Again programmers can write their own primitive instances. The library also offers a number of primitives for labelling, including one based on dichotomic search. There are also three combinators, the counterparts of and, portfolio and limit.
3. Finally, there are *search monitors* (`IlcSearchMonitor`), that hook into the search and are notified of events (somewhat like our protocol messages). These search monitors are primarily meant to collect statistics about the search.

There are a two notable differences with search combinators. Firstly, by distinguishing between goals and search monitors, this approach lacks the uniformity of search combinators. Hence, the design is more complex than necessary. Secondly, the system aims at a very limited form of compositionality. Only three combinators are provided and extension is only promoted at the level of primitive goals. For instance, dichotomic search is presented as a primitive goal rather than as a combinator—moreover, it is written as a monolithic entity rather than as a composition.

The fact that the library has not been designed with much compositionality in mind obviously does not mean that compositionality cannot be achieved. On the contrary, we believe that all the necessary ingredients are available to implement the search combinator design.

7.5 The Comet language

The Comet [29] system features fully programmable search [30], built upon the basic concept of *continuations*, which make it easy to capture the state of the solver and write non-deterministic code.

The Comet library provides abstractions like the non-deterministic primitives `try` and `tryall` that split the search specification in two (orthogonal) parts: 1) the specification of the search tree which corresponds to our to our base_search heuristics, and 2) the exploration of that search tree by means of a *search controller*. In terms of our approach, the search controller determines both the queueing strategy and the behavior of the search heuristic (minus the base search) within a single entity. In other words, it defines what to do when starting or ending a search, failing, or adding a new choice.

Complex heuristics can be constructed as custom controllers, either by inheriting from existing controllers or implementing them from scratch.

Albeit at a different level of abstraction (e.g., compare the Comet definition of depth-bounded search in Fig. 6 to the combinator definition $\mathsf{dbs}(n, s) \equiv \mathsf{limit}(\mathsf{depth} \leq n, s)$), search controllers are quite similar to combinators as presented in this article. However, there is one essential difference. Our combinators are meant to be compositional, whereas search controllers are not. This difference in spirit is clearly reflected in 1) the design of the interface and its associated protocol, and 2) the instances:

1. The design of search controllers is simpler than that of search combinators because it does not take compositionality into account. While many of the messages in the two approaches are similar in spirit, the search combinator approach also stipulates the flow of messages within a search combinator composition. Notably, while most of the messages propagate top-down through a combinator stack, it is vital to compositionality that the <u>`exit`</u> message proceeds in a bottom-up

```
class DBS extends AbstractSearchController {
    stack{Continuation} s;
    int limit;
    DBS(SearchSolver so, int n) : AbstractSearchController(so) {
        s = new stack{Continuation}();
        limit = n;
    }
    void startTry() {
        if (s.getSize() > limit) fail();
    }
    void addChoice(Continuation f) {
        s.push(f);
    }
    void fail() {
        if (s.empty())
            exit();
        else
         call(s.pop());
    }
}
```

Fig. 6 Definition of depth-bounded search in Comet

manner. For instance, this bottom-up flow enables the inner and combinator in the composition and(and(s_1, s_2), s_3) to intercept leaf nodes of s_1 and start s_2 before its parent starts s_3. The other way around would clearly exhibit an undesirable semantics.
In Comet, this compositional protocol is entirely absent. All messages are directed at the single search controller.

2. In terms of instantiation, because of their compositional nature, we promote many “small” combinator instances that each capture a single primitive feature. This approach provides us with a high-level modeling language for search, as the primitive combinators are conveniently assembled into many different search heuristics. In contrast, all Comet search controller instances we are aware of[8] are essentially monolithic implementations of a particular search heuristic; none of them takes other search controllers as arguments. Through a common abstract base class the instances share some basic infrastructure, but to implement a new search controller one basically starts from scratch.

The fact that search controllers have not been designed with compositionality in mind obviously does not mean that compositionality cannot be achieved in Comet. On the contrary, we believe that it is most easily achieved by integrating search controllers with the compositional design of our search combinators. In fact, because of Comet’s powerful primitives for non-determinism, this would lead to a particularly elegant implementation.

[8] i.e., those published in papers and shipped with the Comet library.

7.6 Other systems

The Salsa [13] language is an imperative domain-specific language for implementing search algorithms on top of constraint solvers. Its center of focus is a node in the search process. Programmers can write custom *Choice* strategies for generating next nodes from the current one; Salsa provides a regular-expression-like language for combining these Choices into more complex ones. In addition, Salsa can run custom procedures at the *exits* of each node, right after visiting it. We believe that Salsa's Choice construct is orthogonal to our approach and could be incorporated. Custom exit procedures show similarity to combinators, but no support is provided for arbitrary composition.

Oz [27] was the first language to truly separate the definition of the constraint model from the exploration strategy [24]. Computation spaces capture the solver state and the possible choices. Strategies such as DFS, BFS, LDS, Branch and Bound and Best First Search are implemented by a combination of *copying* and *recomputation* of computation spaces. The strategies are monolithic, there is no notion of search combinators.

Choi et al. [3] describe a compositional framework for search that relies on composing search engines. A search engine is a constraint store transformer which given an initial constraint store outputs a stream of constraint stores in a demand driven way. These are composed by plugging one search engine into another. This allows composition, and also simple filtering such as first solution (once) or last solution (e.g. in optimization to return only the best solution).

Zinc/MiniZinc [14, 15] lets the user specify search in its *annotation language*. There is a proposal for a more expressive search language for MiniZinc [19], but it is limited to basic variable ordering and domain splitting strategies. For Zinc, a language extension is available for implementing variable selection and domain splitting [17] but again it does not address more than basic search.

The original versions of the constraint modeling language OPL [31] provided programmable search using a `try` construct that creates the search tree. The tree could then be explored with a programmed strategy, or a built-in strategy such as DFS, LDS, BFS or Best First Search. Exploration strategies could be modified by limit strategies, which were effectively combinators.

7.7 Autonomous search

Autonomous search (AS) [7] addresses the challenge of providing complex application-tailored search heuristics in a different way. Rather than leaving the specification and tuning of search heuristics to the programmer, AS promotes systems that autonomously self-tune their performance while solving problems. Hence, while search combinators make writing search heuristics easier, AS takes it out of the hands of the programmer altogether. Well-known instances of this approach are Impact Based Search [18] or the *weighted degree* heuristic [2].

AS has advantages for 1) smaller problems where it produces a decent heuristic without programmer investment, and for 2) novice users who don't know how to obtain a decent heuristic. However, loss of programmer control is a liability for hard problems where AS can be ineffective and often only expert knowledge makes the difference.

8 Conclusion

We have shown how combinators provide a powerful high-level language for modeling complex search heuristics. To make this approach useful in practice, we devised an architecture in which the modularity of the language is matched by the modularity of the implementation. This relieves system developers from a high implementation cost and yet, as our experiments show, imposes no runtime penalty. Because the language is high-level and easy to implement, we believe it is an excellent starting point for standardizing search.

For future work, parallel search on multi-core hardware fits perfectly in our combinator framework. We have already performed a number of preliminary experiments and will further explore the benefits of search combinators in a parallel setting. We will also explore potential optimizations (such as the short-circuit of `exh_once` from Section 6) and different compilation strategies (e.g., combining the static and dynamic approaches from Section 5).

In addition we will consider applying search combinators in other problem domains like Mixed Integer Programming (MIP) and A^* where search strategies have a major impact on performance and no dominant default search exists. A combinator approach to local search and/or large area neighbourhood search is also possible, but since these searches typically require the description of nearly arbitrary functions on the solver state to specify neighbourhoods and evaluate moves it seems hard to avoid using a near full programming language

Finally, we note that combinators need not necessarily be heuristics that control the search. They may also monitor search, e.g., by gathering statistics or visualizing the search tree.

Acknowledgements NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council. This work was partially supported by Asian Office of Aerospace Research and Development grant 10-4123.

The authors are grateful for the constructive comments of the reviewers.

References

1. Baatar, D., Boland, N., Brand, S., Stuckey, P.J. (2011). CP and IP approaches to cancer radiotherapy delivery optimization. *Constraints, 16*(2), 173–194.
2. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L. (2004). Boosting systematic search by weighting constraints. In R.L. de Mántaras, & L. Saitta (Eds.), *Proceedings of the 16th Eureopean conference on artificial intelligence, ECAI'2004* (pp. 146–150). IOS Press.
3. Choi, C.W., Henz, M., Ng, K.B. (2001). A compositional framework for search. In *Proceedings of CICLOPS: Colloquium on implementation of constraint and logic programming systems, appeared as technical report TR-CS-003/2001*. New Mexico State University.
4. Cook, W.R. (1989). *A denotational semantics of inheritance*. Ph.D. thesis, Brown University.
5. Cremet, V., Garillot, F., Lenglet, S., Odersky, M. (2006). A core calculus for scala type checking. In R. Kralovic, & P. Urzyczyn (Eds.), *Mathematical foundations of computer science 2006, 31st international symposium, MFCS 2006. LNCS* (Vol. 4162, pp. 1–23). Springer.
6. Desouter, B. (2012). *Modular search heuristics in scala*. Master's thesis, Ghent University (in Dutch).
7. Hamadi, Y., Monfroy, E., Saubion, F. (Eds.) (2012). *Autonomous search*. Springer.
8. Harvey, W.D., & Ginsberg, M.L. (1995). Limited discrepancy search. In *Proceedings of the fourteenth international joint conference on artificial intelligence, IJCAI 95* (pp. 607–613). Morgan Kauufmann.

9. IBM Corporation (2011). IBM ILOG CP optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/. Accessed November 2012
10. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J. (1997). Aspect-oriented programming. In *ECOOP'97 - object-oriented programming, 11th European conference. LNCS* (Vol. 1241, pp. 220–242). Springer.
11. Korf, R.E. (1985). Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence, 27*, 97–109.
12. Kuchcinski, K., & Szymanek, R. (2012). JaCoP - Java constraint programming solver. http://www.jacop.eu/. Accessed November 2012
13. Laburthe, F., & Caseau, Y. (2002). SALSA: a language for search algorithms. *Constraints, 7*(3), 255–288.
14. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P., Garcia de la Banda, M., Wallace, M. (2008). The design of the zinc modelling language. *Constraints, 13*(3), 229–267.
15. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G. (2007). Minizinc: towards a standard CP modelling language. In C. Bessiere (Ed.), *Thirteenth international conference on principles and practice of constraint programming. LNCS* (Vol. 4741, pp. 529–543). Springer.
16. Perron, L. (1999). Search procedures and parallelism in constraint programming. In J. Jaffar (Ed.), *Fifth international conference on principles and practice of constraint programming. LNCS* (Vol. 1713, pp. 346–360). Springer.
17. Rafeh, R., Marriott, K., de la Banda, M.G., Nethercote, N., Wallace, M. (2008). Adding search to zinc. In P.J. Stuckey (Ed.), *Fourteenth international conference on principles and practice of constraint programming. LNCS* (Vol. 5202, pp. 624–629). Springer.
18. Refalo, P. (2004). Impact-based search strategies for constraint programming. In M. Wallace (Ed.), *Tenth international conference on principles and practice of constraint programming. LNCS* (Vol. 3258, pp. 557–571). Springer.
19. Samulowitz, H., Tack, G., Fischer, J., Wallace, M., Stuckey, P. (2010). Towards a lightweight standard search language. In *The 9th international workshop on constraint modelling and reformulation (ModRef)*. http://www.it.uu.se/research/group/astra/ModRef10/programme.html. Accessed November 2012
20. Schimpf, J., & Shen, K. (2012). ECLiPSe – from LP to CLP. *Theory and Practice of Logic Programming, 12*(1–2), 127–156.
21. Schrijvers, T., Stuckey, P.J., Wadler, P. (2009). Monadic constraint programming. *Journal of Functional Programming, 19*(6), 663–697.
22. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P. (2011). Search combinators. In J. Lee (Ed.), *Seventeenth international conference on principles and practice of constraint programming. LNCS* (Vol. 6876, pp. 774–788). Springer.
23. Schrijvers, T., Triska, M., Demoen, B. (2012). Tor: extensible search with hookable disjunction. In *Principles and practice of declarative programming (PPDP'12)*. ACM.
24. Schulte, C. (1997). Programming constraint inference engines. In G. Smolka (Ed.), *Third international conference on principles and practice of constraint programming. LNCS* (Vol. 1330, pp. 519–533). Springer
25. Schulte, C., et al. (2009). Gecode, the generic constraint development environment. http://www.gecode.org/. Accessed November 2012
26. Sellmann, M., & Kadioglu, S. (2008). Dichotomic search protocols for constrained optimization. In P.J. Stuckey (Ed.), *Fourteenth international conference on principles and practice of constraint programming. LNCS* (Vol. 5202, pp. 251–265). Springer.
27. Smolka, G. (1995). The Oz programming model. In J. van Leeuwen (Ed.), *Computer science today. LNCS* (Vol. 1000, pp. 324–343). Springer.
28. Swedish Institute of Computer Science (2008). SICStus prolog. http://www.sics.se/isl/sicstuswww/site/. Accessed November 2012
29. Van Hentenryck, P., & Michel, L. (2005). *Constraint-based local search*. MIT Press.
30. Van Hentenryck, P., & Michel, L. (2006). Nondeterministic control for hybrid search. *Constraints, 11*(4), 353–373.
31. Van Hentenryck, P., Perron, L., Puget, J.F. (2000). Search and strategies in OPL. *ACM Transactions on Computational Logic, 1*(2), 285–315.
32. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming, 12*(1–2), 67–96.
33. Zhou, N.F. (2012). The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming, 12*(1–2), 189–218.