An Introduction to Search Combinators

Tom Schrijvers[1], Guido Tack[2], Pieter Wuille[1,3],
Horst Samulowitz[4], and Peter J. Stuckey[5]

[1] Universiteit Gent, Belgium
{tom.schrijvers,pieter.wuille}@ugent.be
[2] National ICT Australia (NICTA) and Monash University, Victoria, Australia
guido.tack@monash.edu
[3] Katholieke Universiteit Leuven, Belgium
pieter.wuille@cs.kuleuven.be
[4] IBM Research, New York, USA
samulowitz@us.ibm.com
[5] National ICT Australia (NICTA) and University of Melbourne, Victoria, Australia
pjs@cs.mu.oz.au

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 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.

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 [1], Comet [2], or Zinc [3], 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 [4], Gecode [5], Comet [2], or MiniZinc [6], 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.

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 [7]. As in Aspect-Oriented Programming [8], 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

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

Fig. 1. Catalog of primitive search heuristics and combinators

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 provides on overview of the work on search combinators, that appeared in the proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP) 2011 [9] and the Special Issue on Modeling and Reformulation of the Constraints journal [10].

2 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. Advanced users may need to add new, problem-specific primitives, and Sect. 3 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 [6]. However, other concrete syntax forms are easily supported (e.g., we support C++ and Haskell).

2.1 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.

The CP community has spent a considerable amount of work on defining and exploring many such 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.

2.2 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:

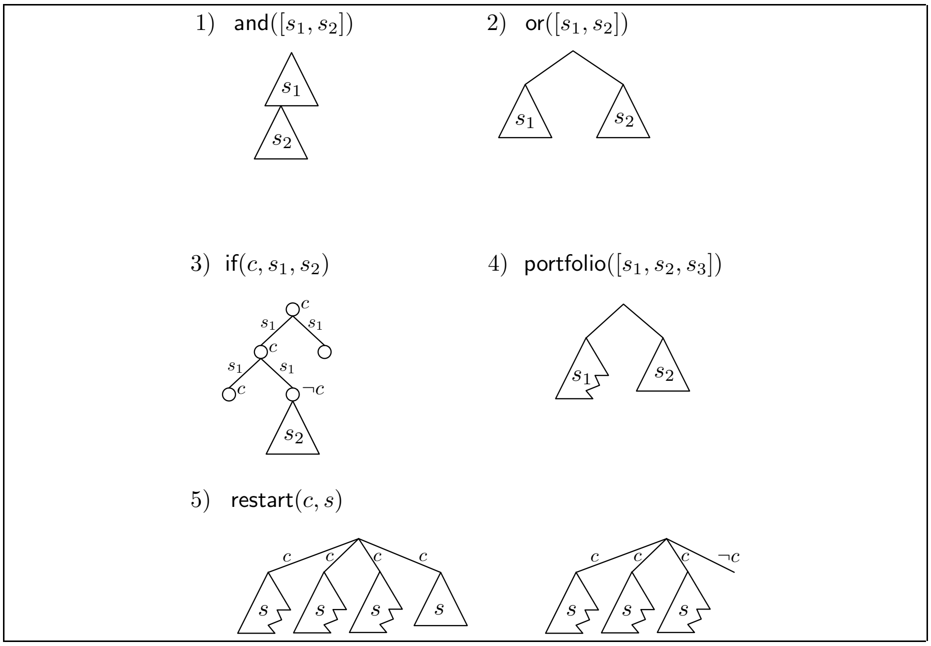

Fig. 2. Primitive combinators

```
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.

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 ifthenelse(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, Sect. 3 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 $\mathsf{limit}(cond, s)$ to perform s while condition *cond* is satisfied, and otherwise cut the search tree using prune:

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

The $\mathsf{once}(s)$ 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:

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

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 $\mathsf{portfolio}([s_1, \ldots, s_n])$ 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 $\mathsf{portfolio}([s_2, \ldots, s_n])$. 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 $\mathsf{hotstart}(cond, s_1, s_2)$ 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 [11]. 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.

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

The $\mathsf{restart}(cond, s)$ combinator repeatedly runs s in full. If s was not exhaustive, it is restarted, until condition *cond* no longer holds. Fig. 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:

$$\boxed{\mathsf{restart}(\text{true}, \mathsf{limit}(\mathsf{failures} < 1000, \mathsf{base\_search}(\texttt{xs}, \text{randomvar}, \text{randomval})))}$$

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

2.3 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:

```
geom_restart(s) ≡ let(maxfails, 100,
                      restart(true, portfolio([limit(failures < maxfails, s),
                                               and([assign(maxfails, maxfails * 1.5),
                                                    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.

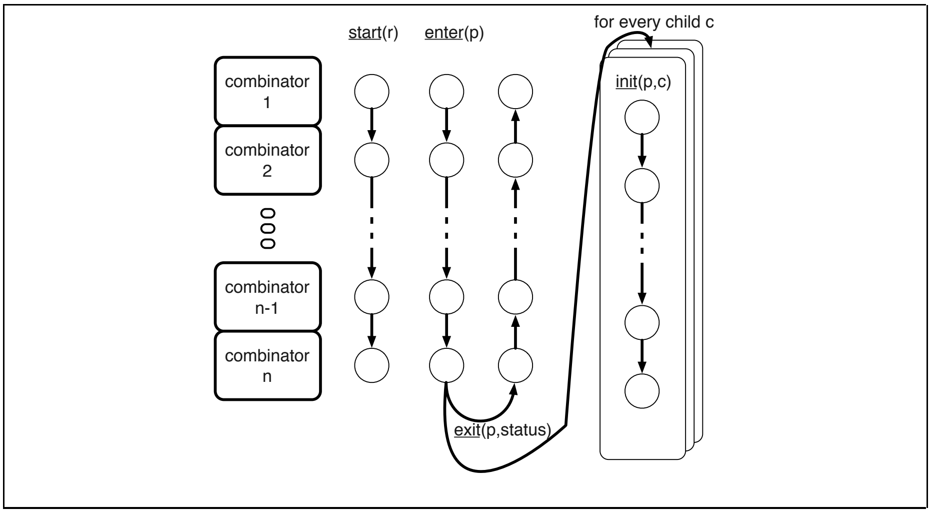

Fig. 3. The modular message protocol

Other Heuristics. Many more heuristics can be implemented with the primitive combinators: branch-and-bound, restarting branch-and-bound, limited discrepancy search, iterative deepening, dichotomic search, . . . See [10] for the details of these heuristics.

3 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.

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.

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.

3.1 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 of the fact 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.
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 $\mathsf{or}([\mathsf{and}_1([s_1,s_2]),\mathsf{and}_2([s_3,s_4])])$,[1] p is included in life cycles of $[\mathsf{or},\mathsf{and}_1,s_1]$, $[\mathsf{or},\mathsf{and}_1,s_2]$, $[\mathsf{or},\mathsf{and}_2,s_3]$ or $[\mathsf{or},\mathsf{and}_2,s_4]$. 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 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).

3.2 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

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

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, it states that the 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 = ...      // 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.

3.3 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.

We refer to [10] for the definitions of all the primitive search combinators.

4 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. Experimental evaluation [10] has shown that both approaches have competitive performance.

4.1 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* [7]. 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 [5]. 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.

4.2 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.

4.3 Further Implementations

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

MiniZinc. As a proof of concept and platform for experiments, we have integrated search combinators into a complete MiniZinc toolchain:[2] The 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. We extended the Gecode FlatZinc interpreter to parse the search combinator annotation and construct the corresponding heuristic using the Dynamic Composition approach described above.

Prolog. Our Tor library [13] implements a subset of the search message protocol in Prolog. The library is currently available for SWI-Prolog [14] and B-Prolog [15], and extends the capabilities of their respective finite domain solver libraries. Among others, it provides all the search heuristics of ECLiPSe Prolog's [4] `search/6` predicate, but in a fully compositional way. The library implements the dynamic approach supplemented with load-time program specialization.

Scala. Desouter [16] has implemented a preliminary library of search combinators for Scala [17] 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 [18].

5 Conclusion

Search combinators provide a powerful high-level language for modeling complex search heuristics. To make this approach useful in practice, the architecture matches the modularity of the language with the modularity of the implementation. This relieves system developers from a high implementation cost and yet, as experiments show, imposes no runtime penalty.

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.

[2] The source code including examples can be downloaded from `http://www.gecode.org/flatzinc.html`

References

1. Van Hentenryck, P., Perron, L., Puget, J.F.: Search and strategies in OPL. ACM TOCL 1(2), 285–315 (2000)
2. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. MIT Press (2005)
3. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P., Garcia de la Banda, M., Wallace, M.: The design of the Zinc modelling language. Constraints 13(3), 229–267 (2008)
4. Schimpf, J., Shen, K.: ECLiPSe – From LP to CLP. Theory and Practice of Logic Programming 12(1-2), 127–156 (2012)
5. Schulte, C., et al.: Gecode, the generic constraint development environment (2009) `http://www.gecode.org/` (accessed November 2012)
6. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007)
7. Cook, W.R.: A denotational semantics of inheritance. PhD thesis, Brown University (1989)
8. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
9. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P.J.: Search combinators. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 774–788. Springer, Heidelberg (2011)
10. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P.: Search combinators. Constraints, 1–37 (2012)
11. Refalo, P.: Impact-based search strategies for constraint programming. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 557–571. Springer, Heidelberg (2004)
12. Perron, L.: Search procedures and parallelism in constraint programming. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 346–361. Springer, Heidelberg (1999)
13. Schrijvers, T., Triska, M., Demoen, B.: Tor: Extensible search with hookable disjunction. In: Principles and Practice of Declarative Programming, PPDP 2012. ACM (2012)
14. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming 12(1-2), 67–96 (2012)
15. Zhou, N.F.: The language features and architecture of B-Prolog. Theory and Practice of Logic Programming 12(1-2), 189–218 (2012)
16. Desouter, B.: Modular Search Heuristics in Scala. Master's thesis, Ghent University (2012) (in Dutch)
17. Cremet, V., Garillot, F., Lenglet, S., Odersky, M.: A core calculus for Scala type checking. In: Královič, R., Urzyczyn, P. (eds.) MFCS 2006. LNCS, vol. 4162, pp. 1–23. Springer, Heidelberg (2006)
18. Kuchcinski, K., Szymanek, R.: JaCoP - Java Constraint Programming solver (2012), `http://www.jacop.eu/` (accessed November 2012)