Constraint-Based Combinators for Local Search

Pascal Van Hentenryck, Laurent Michel,
and Liyuan Liu

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-04-07
April 2004

Constraint-Based Combinators for Local Search

Pascal Van Hentenryck[1], Laurent Michel[2], and Liyuan Liu[2]

[1] Brown University, Box 1910, Providence, RI 02912
[2] University of Connecticut, Storrs, CT 06269-3155

Abstract. One of the most appealing features of constraint programming is its rich constraint language for expressing combinatorial optimization problems. This paper demonstrates that traditional combinators from constraint programming have natural counterparts for local search, although their underlying computational model is radically different. In particular, the paper shows that constraint combinators, such as logical and cardinality operators, reification, and first-class expressions can all be viewed as differentiable objects. These combinators naturally support elegant and efficient modelings, generic search procedures, and partial constraint satisfaction techniques for local search. Experimental results on a variety of applications demonstrate the expressiveness and the practicability of the combinators.

1 Introduction

Historically, most research on modeling and programming tools for combinatorial optimization has focused on systematic search, which is at the core of branch & bound and constraint satisfaction algorithms. However, in recent years, increased attention has been devoted to the design and implementation of programming tools for local search (e.g., [2, 20, 16, 7, 8, 12, 18]). This is motivated by the orthogonal strengths of the paradigms, the difficulty of obtaining efficient implementations, and the lack of compositionality and reuse in local search.

COMET [9, 17] is a novel, object-oriented, programming language specifically designed to simplify the implementation of local search algorithms. Comet supports a constraint-based architecture for local search organized around two main components: a declarative component which models the application in terms of constraints and functions, and a search component which specifies the search heuristic and meta-heuristic. Constraints and objective functions are natural vehicles to express combinatorial optimization problems and often capture combinatorial substructures arising in many practical applications. But constraints and objective functions have a fundamentally different computational model in COMET as they do not prune the search space. Rather they are *differentiable objects* that maintain a number of properties incrementally and provide algorithms to evaluate the effect of various operations on these properties. The search component then uses these functionalities to guide the local search using selectors and other high-level control structures [17]. The architecture enables local search algorithms to be high-level, compositional, and modular.

However, constraint programming languages and libraries also offer rich languages for combining constraints, including logical and cardinality operators, reification, and expressions over variables. These "combinators" are fundamental in practice, not only because they simplify problem modeling, but also because they lay the foundations for expressing various kinds of ad-hoc constraints that typically arise in complex applications, as well as generic search procedures.

This paper shows that traditional constraint programming combinators bring similar benefits for local search, although their underlying computational model differ significantly. In particular, the paper shows that arithmetic, logical, and cardinality operators, reification, and first-class expressions can all be viewed as differentiable objects, providing high-level and efficient abstractions for composing constraints and objectives in local search. These combinators naturally support very high-level local search models, partial constraint satisfaction techniques, and generic search procedures, which are independent of the application at hand and only rely on generic interfaces for constraints and objective functions. As a consequence, they foster the separation of concerns between modeling and search components, increase modularity, and favor compositionality and reuse. More generally, the paper shows that the rich language of constraint programming is conceptually robust and brings similar benefits to constraint programming and local search, despite their fundamentally different computational models.

The rest of this paper is organized as follows. Sections 2 and 3 discuss how to combine constraints and objective functions in Comet, Section 4 introduces first-class expressions, and Section 5 discusses generic search procedures. Applications and experimental results are presented together with the abstractions in these sections. Section 6 concludes the paper.

2 Constraints

Constraints in Comet are differentiable objects implementing the interface (partially) described in Figure 1. The interface gives access to the constraint variables (method `getVariables`) and to two incremental variables which represent the truth value of the constraint (method `isTrue`) and its violation degree (method `violations`). The violation degree is constraint-dependent and measures how much the constraint is violated. This information is often more useful than the constraint truth value as far as guiding the search is concerned. For instance, the violation degree of an arithmetic constraint $l \geq r$ is given by $\max(0, r - l)$. The violation degree of the combinatorial constraint `allPresent(R,x)`, which holds if all values in range `R` occur in array `x`, is given by $\#\{v \in R \mid \neg\exists i : x[i] = v\}$. The method `getViolations` returns the violations which may be attributed to a given variable, which is often useful in selecting local moves. The remaining methods provide the differentiable API of the constraint. They make it possible to evaluate the effect of an assignment, a swap, or multiple assignments on the violation degree. The differentiable API is fundamental in obtaining good performance on many applications, since the quality of local moves can be evalu-

```
Interface Constraint {
   inc{int}[] getVariables();
   inc{boolean} isTrue();
   inc{int} violations();
   int getViolations(inc{int} x);
   int getAssignDelta(inc{int} x,int v);
   int getSwapDelta(inc{int} x1,inc{int} x2);
   int getAssignDelta(inc{int}[] x,int[] v);
}
```

Fig. 1. The Constraint Interface in Comet (Partial Description).

Combinator	Violation Degrees
c_1 && c_2	$\upsilon(c_1) + \upsilon(c_2)$
$c_1 \parallel c_2$	$\min(\upsilon(c_1), \upsilon(c_2))$
$\tau(c)$	if $\upsilon(c_1) > 0$ then 1 else 0
exactly$(k, [c_1, \ldots, c_n])$	$\text{abs}(\sum_{i=1}^{n} \tau(c_i) - k)$
atmost$(k, [c_1, \ldots, c_n])$	$\max(\sum_{i=1}^{n} \tau(c_i) - k, 0)$
$k \times c$	$k \times \upsilon(c)$
satisfactionConstraint(c)	$\tau(c)$

Table 1. The Semantic of Some Constraint Combinators.

ated quickly. The rest of this section describes modeling abstractions to combine constraints. Table 1 summarizes the violation degrees of the various combinators.

Constraint Systems Constraint systems, a fundamental modeling abstraction in Comet, are container objects representing a conjunction of constraints. Constraint systems are constraints themselves and implement the `Constraint` interface. Hence they maintain their truth value and their violation degree, i.e., the sum of the violation degrees of their constraints. They also support the differentiable API. Figure 2 depicts a simple Comet program for the n-queens problem. Lines 1-4 describe the problem variables and data structures, lines 5-9 describe the modeling component, and lines 10-13 specify the search procedure. Line 3 creates a uniform distribution and line 4 declares the decision variables: variable `queen[i]` represents the row of the queen placed in column `i`. These incremental variables are initialized randomly using the uniform distribution. Line 5 declares the constraint system and lines 6-8 specify the problem constraints using the ubiquitous `allDifferent` constraint. More precisely, they specify that the queens cannot be placed on the same row, the same upper diagonal, and the same lower diagonal. Lines 10-13 describe a min-conflict search procedure [11]. Line 11 selects the queen `q` with the most violations and line 12 chooses a new value `v` for queen `q`. Line 13 assigns this new value to the queen, which has the effect of (possibly) updating the violation degree of the subset of affected constraints and of the constraint system. Lines 11-13 are iterated until a solution is found.

```
range Size = 1..1024;
LocalSolver m();
UniformDistribution distr(Size);
inc{int} queen[i in Size](m,Size) := distr.get();

ConstraintSystem S(m);
  S.post(allDifferent(queen));
  S.post(allDifferent(all(i in Size) queen[i] + i));
  S.post(allDifferent(all(i in Size) queen[i] - i));
m.close();

while (S.violations() > 0)
  selectMax(q in Size)(S.getViolations(queen[q]))
    selectMin(v in Size)(S.getAssignDelta(queen[q],v))
      queen[q] := v;
```

Fig. 2. A Comet Program for the Queens Problem.

Constraint systems provide a clean separation between the declarative and search components of a local search. Observe that it is possible to add new constraints to the constraint system without changing the search procedure. Similarly, it is possible to change the search procedure (e.g., adding a tabu list) without modifying the model. It is also important to stress that a single COMET program may use several constraint systems simultaneously. This is useful, for instance, for local search algorithms that maintain the feasibility of a subset of constraints (e.g., the hard constraints), while allowing others to be violated (e.g., the soft constraints).

Logical and Cardinality Operators One of the appealing features of constraint programming is its ability to combine constraints using logical and cardinality operators. COMET offers similar functionalities for local search. For instance

```
Constraint c = ((x != y) || (x != z));
```

illustrates a disjunctive constraint in COMET. *The disjunctive constraint is a differentiable object implementing the* `Constraint` *interface.* In particular, the violation degree of a disjunction $c = c_1 \| c_2$ is given by $\min(\upsilon(c_1), \upsilon(c_2))$, where $\upsilon(c)$ denotes the violation degree of c.

COMET also features a variety of cardinality operators. For instance, Figure 3 depicts the use of the cardinality operator $\texttt{exactly}(k, [c_1, \ldots, c_n])$, a differentiable constraint which holds if exactly k constraints hold in $c_1, \ldots, c_n$. The figure depicts a COMET program to solve the magic series problem, a traditional benchmark in constraint programming. A series $(s_0, \ldots, s_n)$ is magic if s_i represents the number of occurrences of i in $(s_0, \ldots, s_n)$. Lines 6-10 in Figure 3 specify the modeling component. Line 8 features the cardinality operator to express that there are `magic[v]` occurrences of `v` in the magic series and line 9 adds the traditional redundant constraint. Lines 11-19 implement a min-conflict search with a simple tabu-search component. Observe the modeling component in this program which is similar to a traditional constraint programming solution.

```
int n = 400;
range Size = 0..n-1;
LocalSolver m();
inc{int} magic[i in Size](m,Size) := 0;
int tabu[i in Size] = -1;

ConstraintSystem S(m);
forall(v in Size)
  S.post(exactly(magic[v],all(i in Size) magic[i] == v));
S.post(sum(i in Size) i * magic[i] == n);
m.close();

int it = 0;
while (S.violations() > 0) {
  selectMax(s in Size: tabu[s] < it)(S.getViolations(magic[s]))
    selectMin(v in Size: magic[s] != v)(S.getAssignDelta(magic[s],v)) {
      magic[s] := v;
      tabu[s] = it + 3;
    }
  it = it + 1;
}
```

Fig. 3. A Comet Program for the Magic Series Problem.

n	10	30	50	70	90	110	130	150	170	190	210
$best(T)$	0.00	0.03	0.09	0.21	0.41	0.57	0.84	1.09	1.44	2.09	3.20
$\mu(T)$	0.01	0.09	0.41	1.05	1.78	5.70	13.58	21.70	47.60	67.83	150.41
$worst(T)$	0.02	0.37	1.57	7.35	10.85	30.95	102.63	86.54	347.77	400.20	761.11
$\sigma(T)$	0.01	0.12	0.56	1.73	2.84	9.12	22.87	31.38	81.80	110.71	240.85

Table 2. Performance Results on the Magic Series Program.

Interestingly, local search performs reasonably well on this problem as indicated in Table 2. The table gives the best, average, and worst times in seconds for 50 runs on a 2.4Ghz Pentium, as well as the standard deviation.

The contributions here are twofold. On the one hand, Comet naturally accommodates logical and cardinality operators as differentiable objects, allowing very similar modelings for constraint programming and local search. On the other hand, implementations of logical/cardinality operators directly exploit incremental algorithms for the constraints they combine, providing compositionality both at the language and implementation level. The implementations can in fact be shown optimal in terms of the input/output incremental model [14], assuming optimality of the incremental algorithms for the composed constraints.

Weighted Constraints Many local search algorithms (e.g., [15, 4, 19]) use weights to focus the search on some subsets of constraints. Comet supports weight specifications which can be either static or dynamic. (Dynamic weights vary during the search). Weights can be specified with the $*$ operator. For instance the snippet

```
Constraint c = 2 * allDifferent(x)
```

```
1.  LocalSolver m();
2.  UniformDistribution distr(Hosts);
3.  inc{int} boat[Guests,Periods](m,Hosts) := distr.get();
4.  int tabu[Guests,Periods,Hosts] = -1;

5.  ConstraintSystem S(m);
6.  forall(g in Guests)
7.    S.post(2 * allDifferent(all(p in Periods) boat[g,p]));
8.  forall(p in Periods)
9.    S.post(2 * knapsack(all(g in Guests) boat[g,p],crew,cap));
10. forall(i in Guests, j in Guests : j > i)
11.   S.post(atmost(1,all(p in Periods) boat[i,p] == boat[j,p]));
12. m.close();
```

Fig. 4. The Modeling Part of a Comet Program for the Progressive Party Problem.

Hosts/Periods	6	7	8	9	10
1-12,16	0.32 (0.30)	0.41 (0.35)	0.60 (0.54)	1.01 (0.69)	3.74 (1.27)
1-13	0.41 (0.34)	0.77 (0.46)	3.15 (0.99)	42.22 (5.80)	
1,3-13,19	0.41 (0.33)	0.79 (0.47)	3.50 (0.92)	28.6 (6.39)	
3-13,25,26	0.44 (0.35)	0.93 (0.50)	4.53 (1.30)	65.32 (7.79)	
1-11,19,21	1.75 (0.54)	36.1 (2.86)			
1-9,16-19	2.81 (1.06)	95.4 (4.81)			

Table 3. Experimental Results for the Progressive Party Problem.

associates a constant weight of 2 to an `allDifferent` constraint and returns an object implementing the `Constraint` interface. Its main effect is to modify the violation degree of the constraint and the results of its differentiable API. Weights can also be specified by incremental variables, which is useful in many applications where weights are updated after each local search iterations. This feature is illustrated later in the paper in a frequency allocation application.

Figure 4 depicts the modeling component of a COMET program to solve the progressive party problem. It illustrates both constant weights and the cardinality operator `atmost`. Line 7 expresses weighted `allDifferent` constraints to ensure that a party never visits the same boat twice. Line 9 posts weighted knapsack constraints to satisfy the capacity constraint on the boats. Finally, line 11 uses a cardinality constraint to specify that no two parties meet more than once over the course of the event. The cardinality operator makes for a very elegant modeling: it removes the need for the rather specific `meetAtmostOnce` constraint used in earlier version of the COMET program [9]. It also indicates the ubiquity of cardinality constraints for expressing, concisely and naturally, complex constraints arising in practical applications.

Table 3 describes experimental results for this modeling and the search procedure of [9] augmented with a restarting component that resets the current solution to a random configuration every 100,000 iterations in order to eliminate outlier runs. The table describes results for various configurations of hosts and various numbers of periods. The results report the average running times over 50 runs of the algorithm, as well as the best times in parenthesis. With the ad-

```
1.   SatisfactionSystem S(m);
2.   forall(d in DistanceCtrs)
3.     switch (d.ty) {
4.       case 1: S.post(abs(freq[d.v1]-freq[d.v2]) == d.rhs);break;
5.       case 2: S.post(abs(freq[d.v1]-freq[d.v2]) > d.rhs);break;
6.       case 3: S.post(abs(freq[d.v1]-freq[d.v2]) < d.rhs);break;
7.     }
```

Fig. 5. Partial Contraint Satisfaction in Frequency Allocation.

dition of a restarting component, the standard deviation for the hard instances is always quite low (e.g., configuration 1-9,16-19 with 7 periods has a mean of 95.4 and a deviation of 4.8) and shows that the algorithm's behavior is quite robust. The results were obtained on a 2.4Ghz Pentium 4 running Linux. The performance of the program is excellent, as the cardinality operator does not impose any significant overhead. Once again, the implementation can be shown incrementally optimal if the underlying constraints support optimal algorithms.

Partial Constraint Satisfaction Some local search algorithms do not rely on violation degrees; rather they reason on the truth values of the constraints only. This is the case, for instance, in partial constraint satisfaction [3], where the objective is the minimize the (possibly weighted) number of constraint violations. Comet provides an operator to transform an arbitrary constraint into a satisfaction constraint, i.e., a constraint whose violation degree is 0 or 1 only. For instance, the snippet

```
Constraint c = satisfactionConstraint(allDifferent(x));
```

assigns to `c` the satisfaction counterpart of the `allDifferent` constraint. The key contribution here is the systematic derivation of the satisfaction implementation in terms of the original constraint interface. The resulting implementation only induces a small constant overhead.

Comet also supports satisfaction systems that systematically apply the satisfaction operator to the constraints posted to them, simplifying the modeling and the declarative reading. Figure 5 illustrates satisfaction systems on an excerpt from a frequency allocation problem, where the objective is to minimize the number of violated constraints. Line 1 declares the satisfaction system `S`, while lines 4, 5, and 6 post the various types of distance constraints to the system.

3 Objective Functions

We now turn to objective functions, another class of differentiable objects in Comet. Objective functions may be linear, nonlinear, or may capture combinatorial substructures arising in many applications. A typical example is the objective function `MinNbDistinct(x[1],...,x[n])` which minimizes the number of distinct values in `x[1],...,x[n]` and arises in graph coloring, frequency allocation, and other resource allocation problems.

```
Interface Objective {
  inc{int}[] getVariables();
  inc{int} value();
  inc{int} cost();
  int getCost(inc{int} x);
  int getAssignDelta(inc{int} x,int v);
  int getSwapDelta(inc{int} x1,inc{int} x2);
  int getAssignDelta(inc{int}[] x,int[] v);
}
```

Fig. 6. The Objective Interface in Comet (Partial Description).

Objective	Value	Cost
MinNbDistinct(x[E])	$\#\{\ x[e] \ \mid\ e \in E\ \}$	$-\sum_i \lvert\{e \in E \mid x[e] = i\}\rvert^2$
MinNbDistinct(x[E],w[E])	$\#\{\ x[e] \ \mid\ e \in E\ \}$	$\sum_{e \in E} w[x[e]]$
MaxNbDistinct(x[E],w[V])	$\#\{\ x[e] \ \mid\ e \in E\ \}$	$\sum_{v \in V : \neg \mathrm{occur}(v,x)} w[v]$
condSum($b[E], c[E]$)	$\sum_{e \in E : b[e]} c[e]$	$\sum_{e \in E : b[e]} c[e]$
minAssignment($b[V], c[E, V]$)	$\sum_{e \in E} \max_{v \in V : b[v]} c[e, v]$	$\sum_{e \in E} \max_{v \in V : b[v]} c[e, v]$
constraintAsObjective(c)	$v(c)$	$v(c)$

Table 4. The Semantic of Some Objective Functions.

This section illustrates how objective functions, like constraints, can be combined naturally to build more complex objectives. Once again, this desirable functionality comes from the `Objective` interface depicted in Figure 6 and implemented by all objective functions. In particular, the interface gives access to their variables, to two incremental variables, and the differentiable API. The first incremental variable (available through method `value`) maintains the *value* of the function incrementally. The second variable (available through method `cost`) maintains the *cost* of the function which may, or may not, differ from its value. The cost is useful to guide the local search more precisely by distinguishing states which have the same objective value. Consider again the objective function `nbDistinct`. Two solutions may use the same number of values, yet one of them may be closer than the other to a solution with fewer values. To distinguish between these, the cost may favor solutions where some values are heavily used while others have few occurrences. For instance, such a cost is used for graph coloring in [6] and is shown in Table 4 that summarizes the objective functions discussed in this paper. The differentiable API returns the variation of the cost induced by assignments and swaps. The method `getCost` also returns the contribution of a variable to the cost and is the counterpart of method `getViolations` for objective functions.

Arithmetic Operators Objective functions can be combined using traditional arithmetic operators. For instance, the excerpt
`Objective f = condSum(open,fixed) + minAssignment(open,transportionCost);`
illustrates the addition of two objective functions capturing combinatorial substructures expressing the fixed and transportation costs in uncapacitated warehouse location. These functions are useful in a variety of other applications such as k-median and configuration problems. See also [10] for a discussion of efficient incremental algorithms to implement them.

Reification: Constraints as Objective Functions Some of the most challenging applications in combinatorial optimization feature complex feasibility constraints together with a "global" objective function. Some algorithms approach these problems by relaxing some constraints and integrating them in the objective function. To model such local search algorithms, COMET provides the generic operator `constraintAsObjective` that transforms a constraint into an objective function whose value and cost are the violation degree of the constraint. Moreover, traditional arithmetic operators transparently apply this operator to simplify the declarative reading of the model. The implementation of this combinator, which is also given in Table 4, is in fact trivial, since it only maps one interface in terms of the other.

Frequency Allocation To illustrate objective functions, consider a frequency allocation problem where feasibility constraints impose some distance constraints on frequencies and where the objective function consists of minimizing the number of frequencies used in the solution. We present an elegant COMET program implementing the Guided Local Search (GLS) algorithm proposed in [19].[3] The key idea underlying the GLS algorithm is to iterate a simple local search where feasibility constraints are integrated inside the objective function. After completion of each local search phase, the weights of the violated constraints are updated to guide the search toward feasible solutions. If the solution is feasible, the weights of some frequencies used in the solutions (e.g., the values that occur the least) are increased to guide the search toward solutions with fewer frequencies.

Figure 7 depicts the modeling part of the GLS algorithm. Line 2 declares the decision variables, while lines 3 and 4 create the incremental variables representing the weights that are associated with constraints and values respectively. Line 5 declares the satisfaction system `S` and lines 6-18 post the distance constraints in S, as presented earlier in the paper. The only difference is the use of weights which are useful to focus the search on violated constraints. Line 19 declares the objective function `nbFreq` which minimizes the number of distinct frequencies. Note that this function receives, as input, a dynamic weight for each value to guide the search toward solutions with few frequencies. Line 20 is particularly interesting: it defines the GLS objective function `obj` as the sum of the satisfaction system (a constraint viewed as an objective function) and the "actual" objective `nbFreq`. Of course, the search component uses the objective `obj`.

[3] The point is not to present the most efficient algorithm for this application, but to illustrate the concepts introduced herein on an interesting algorithm/application.

```
LocalSolver m();
inc{int} freq[RV](m);
inc{int} w[RC](m) := 0;
inc{int} fw[RF](m): = 0;
SatisfactionSystem S(m);
forall(d in DistanceCtrs)
  switch (d.ty) {
    case 1:S.post(abs(freq[d.v1]-freq[d.v2]) == d.rhs,w[d.id]);break;
    case 2:S.post(abs(freq[d.v1]-freq[d.v2]) > d.rhs,w[d.id]);break;
    case 3:S.post(abs(freq[d.v1]-freq[d.v2]) < d.rhs,w[d.id]);break;
  }
MinNbDistinct nbFreq(freq,fw);
Objective obj = S + nbFreq;
m.close();
```

Fig. 7. The Modeling Part of a Comet Program for Frequency Allocation.

id	$\mu(S)$	$\sigma(S)$	B(S)	W(S)	$\mu(O)$	$\sigma(O)$	B(O)	W(O)	$\mu(I)$	$\sigma(I)$
1	18.76	2.15	16	24	4.40	2.50	1.84	10.35	778.60	420.02
2	14.00	0.00	14	14	0.69	1.06	0.27	6.20	152.72	257.28
3	15.36	1.23	14	18	2.69	1.96	0.69	9.10	523.40	383.69

Table 5. Experimental Results for Frequency Allocation.

Figure 8 depicts the search part of the algorithm. Function `GLS` iterates two steps for a number of iterations: a local search, depicted in lines 8-20, and the weight adjustment. The weight adjustment is not difficult: it simply increases the weights of violated constraints and, if the solution is feasible, the weights of the frequencies that are used the least in the solution (modulo a normalization factor [19]). The local search considers all variables in a round-robin fashion and applies function `moveBest` on each of them, until a round does not improve the objective function (lines 8-15). Function `moveBest` selects, for variable `freq[v]`, the frequency `f` that minimizes the values of the objective function (ties are broken randomly). It uses the differentiable API to evaluate moves quickly.

It is particularly interesting to observe the simplicity and elegance of the Comet program. The modeling component simply specifies the constraints and the objective function, and combines them to obtain the GLS objective. The search component is expressed at a high level of abstraction as well, only relying on the objective function and the decision variables. Table 5 depicts the experimental results on the first three instances of the Celar benchmarks. It reports quality and efficiency results for 50 runs of the algorithms on a 2.4Ghz Pentium IV. In particular, it gives the average, standard deviation, and the best and worst values for the number of frequencies and the time to the best solutions in seconds. It also reports the average number of iterations and its standard deviation. The preliminary results indicate that the resulting Comet program, despite its simplicity and elegance, compares well in quality and efficiency with specialized implementations in [19]. Note that there are many opportunities for improvements to the algorithm.

```
function void GLS() {
  while (it < maxIterations) {
    localSearch();
    updateWeights();
    it++;
  }
}
function void localSearch() {
  int old;
  do {
    old = obj.cost();
    forall(v in RV) moveBest(v);
  } while (old != obj.cost());
}
function void moveBest(int v) {
  if (obj.getCost(freq[v]) > 0)
    selectMin(f in Domain[v])(obj.getAssignDelta(freq[v],f))
      freq[v] := f;
}
```

Fig. 8. The Search Part of a Comet Program for Frequency Allocation.

4 First-Class Expressions

We now turn to first-class expression, another significant abstraction which is also an integral part of constraint programming libraries [5]. First-class expressions are constructed from incremental variables, constants, and arithmetic, logical, and relational operators. In COMET, first-class expressions are differentiable objects which can be evaluated to determine the effect of assignments and swaps on their values. In fact, several examples presented earlier feature first-class expressions. For instance, the COMET code

```
S.post(allDifferent(all(i in Size) queen[i] + i));
```

from the n-queens problem can be viewed as a shortcut for

```
expr{int} d[i in Size] = queen[i] + i;
S.post(allDifferent(d));
```

The first instruction declares an array of first-class integer expressions, element `d[i]` being the expression `queen[i] + i`. The second instruction states the `allDifferent` constraint on the expression array. As mentioned earlier, expressions are differentiable objects, which can be queried to evaluate the effect of assignments and swaps on their values. For instance, the method call `d[i].getAssignDelta(queen[i],5)` returns the variation in the value of expression `d[i]` when `queen[i]` is assigned the value 5.

First-class expressions significantly increase the modeling power of the language, since constraints and objective functions can now be defined over complex expressions, not incremental variables only. Moreover, efficient implementations of these enhanced versions can be obtained systematically by combining the differentiable APIs of constraints, objective functions, and first-class expressions.

```
1.  int n = 40;
2.  range Size = 1..n;
3.  range Domain = 0..n-1;
4.  range SD = 1..n-1;
5.  LocalSolver m();
6.  RandomPermutation perm(Domain);
7.  inc{int} v[Size](m,Domain) := perm.get();
8.  MaxNbDistinct obj(all(k in SD) abs(v[k+1]-v[k]));
9.  m.close();
10. while (obj.value() < n-1)
11.   select(i in Size: obj.getCost(v[i]) > 0)
12.     selectMax(j in Size: j != i)(obj.getSwapDelta(v[i],v[j]))
13.       v[i] :=: v[j];
```

Fig. 9. A Comet Program for the All-Interval Series Problem.

For instance, in the queens problem, the implementation can be thought of as (1) defining an intermediary set of incremental variables

```
inc{int} q[i in Size] <- queen[i] + i;
```

(2) specifying an `allDifferent(q)` constraint on these variables and (3) implementing the differentiable API as a composition of the differentiable APIs of the expressions `queen[i] + i` and of the `allDifferent` constraint.

The all-interval series problem [1] illustrates the richness of first-class expressions in Comet. (The n-queens problem only illustrates a simple use of first-class expressions, since every variable occurs in exactly one expression.) The problem, which is a well-known exercise in music composition, is extremely challenging for constraint programming. It consists of finding a sequence of notes such that all notes in the sequence, as well as tonal intervals between consecutive notes, are different. The all-interval series problem can thus be modeled as the finding of a permutation of the first n integers such that the absolute difference between two consecutive pairs of numbers are all different.

Figure 9 depicts a Comet program solving the all-interval series problem. The basic idea behind the modeling is to maximize the number of different distances in `abs(v[2]-v[1]),...,abs(v[n]-v[n-1])`. Line 7 declares the variables `v[i]` in the series, which are initialized to a random permutation of $0..n-1$. This guarantees that all variables have distinct values, a property maintained by the search procedure whose local moves swap the values of two variables. Line 8 specifies the objective function `MaxNbDistinct` which maximizes the number of distinct distances in `abs(v[2]-v[1]),...,abs(v[n]-v[n-1])`. It is important to observe that almost all variables occur in two expressions. *As a consequence, it is non-trivial to evaluate the impact of a swap on the objective function, since this may involve up to 4 specific expressions.* The Comet implementation abstracts this tedious and error-prone aspect of the local search through first-class expressions and the combinatorial function `MaxNbDistinct`.

The performance of the algorithm can be improved upon by associating weights to the distances (as suggested in [1]). The justification here is that larger

n	$\mu(Time)$	$best(Time)$	$worst(Time)$	$\sigma(Time)$	$\mu(Iter)$	$\sigma(Iter)$
10	0.00	0.00	0.01	0.00	25.18	38.27
15	0.01	0.00	0.06	0.02	203.92	276.25
20	0.04	0.00	0.26	0.07	859.02	1235.77
25	0.07	0.00	0.52	0.10	1183.30	1419.42
30	0.39	0.01	2.41	0.64	6092.78	7985.98
35	0.86	0.03	6.75	1.49	11864.52	16792.77
40	2.62	0.09	18.68	4.05	32669.46	38641.32
45	6.63	0.12	36.79	9.24	78961.70	75349.35
50	34.09	1.27	165.41	52.56	355816.56	416601.45
55	130.05	9.30	278.75	160.56	1277633.38	906189.43

Table 6. Performance Results on the All-Interval Series Program.

```
function void minConflictSearch(ConstraintSystem S) {
  inc{int}[] var = S.getVariables();
  range Size = var.getRange();
  while (S.violations() > 0)
    selectMax(i in Size)(S.getViolations(var[i]))
      selectMin(v in var[i].getDomain())(S.getAssignDelta(var[i],v))
        var[i] := v;
}
```

Fig. 10. A Generic Min-Conflict Search in Comet.

distances are much more difficult to obtain and it is beneficial to bias the search toward them. To accommodate this enhancement, it suffices to replace line 8 by

```
MaxNbDistinct obj(all(k in SD) abs(v[k+1]-v[k]),all(k in SD) k^3);
```

Table 6 gives the experimental results for various values of n and for the Comet program, extended to restart the computation after 1,000 iterations if no solution was found. The resulting Comet significantly outperforms existing programs. The Java implementation from [1] takes an average time of 63 seconds for $n = 30$ on a 733MHz Pentium III (instead of 0.39 seconds for Comet on a 2.4GHz machine). The gain probably comes from the better incrementality of Comet which uses differentiation to evaluate moves quickly. Indeed, our first Comet program for this problem, which did not use first-class expressions and did not support the differentiable API for the objective function, took 10.44 seconds in average over 50 runs for $n = 30$. The incrementality of the Comet program presented here is obtained directly from the composition of differentiable objects, while it is tedious to derive manually.

5 Generic Search Procedures

The combinators presented in this paper have an additional benefit: they provide the foundation for writing generic search procedures in Comet. Indeed, search procedures are now able to interact with declarative components only through

```
1.   function int cdSearch(ConstraintSystem S) {
2.     UniformDistribution noise(0..99);
3.     range C = S.getRange();
4.     Constraint c[i in C] = S.getConstraint(i);
5.     while (S.violations() > 0)
6.       select(i in C: !c[i].isTrue()) {
7.         inc{int}[] var = c[i].getVariables();
8.         range RV = var.getRange();
9.         selectMax(v in RV)(c[i].getViolations(var[v]))
10.          selectMin(val in var[v].getDomain())
                      (c[i].getAssignDelta(var[v],val))
11.            if (S.getAssignDelta(var[v],val) < 0)
12.              var[v] := val;
13.            else if (noise.get() < 10) {
14.              var[v] := val;
15.      }
16.  }
```

Fig. 11. A Generic Constraint-Directed Search in Comet.

the `Constraint` and `Objective` interfaces, abstracting away the actual details of constraints and objective functions. These generic search procedures do not depend on the application at hand, yet they exploit the differentiable APIs to implement heuristics and meta-heuristics efficiently. Moreover, these APIs are implemented by efficient incremental algorithms exploiting the structure of the applications. In other words, although the search procedures are generic and do not refer to specificities of the applications, they exploit their underlying structure through the combinators.

Figure 10 depicts a min-conflict search in Comet. The code is essentially similar to the search procedure in the queens problem: it is only necessary to collect the variable array and its range in lines 2-3, and to use the variable domains in line 6. As a consequence, lines 10-13 in the queens problem can simply be replaced by a call `minConflictSearch(S)`. Similarly, Figure 11 implements a constraint-directed search inspired by search procedures used in WSAT [15, 20] and DragonBreath [13]. The key idea is to select a violated constraint first (line 6) and then a variable to re-assign (line 9). Once again, lines 10-13 in the queens problem can simply be replaced by a call `cdSearch(S)`. Such constraint-oriented local search procedures are particularly effective on a variety of problems, such as the ACC sport-scheduling problem. The actual search procedure in [20] was also successfully implemented in Comet and applied to their integer formulation.

Observe that these search procedures are generic and do not depend on the actual shape of the constraints, which can be propositional, linear, nonlinear, combinatorial, or any combination of these. Moreover, generic search procedures bring another interesting benefit of constraint programming to local search: the ability to provide a variety of (parameterized) default search procedures, while exploiting the specific structure of the application.

6 Conclusion

This paper aimed at demonstrating that constraint-based combinators from constraint programming are natural abstractions for expressing local search algorithms. In particular, it showed that logical and cardinality operators, reification, and first-class expressions can all be viewed as differentiable objects encapsulating efficient incremental algorithms. These combinators, and their counterparts for objective functions, provide high-level ways of expressing complex ad-hoc constraints, generic search procedures, and partial constraint satisfaction. They were also shown to be amenable to efficient implementations. As a consequence, this paper, together with earlier results, indicates that the rich language of constraint programming is a natural vehicle for writing a wide variety of modular, extensible, and efficient local search programs.

References

1. C. Codognet and D. Diaz. Yet Another Local Search Method for Constraint Solving. In *AAAI Fall Symposium on Using Uncertainty within Computation*, 2001.
2. L. Di Gaspero and A. Schaerf. *Optimization Software Class Libraries*, chapter Writing Local Search Algorithms Using EasyLocal++. Kluwer, 2002.
3. E. Freuder. Partial Constraint Satisfaction. Artificial Intelligence, 58, 1992.
4. P. Galinier and J.-K. Hao. A General Approach for Constraint Solving by Local Search. In *CP-AI-OR'00*, Paderborn, Germany, March 2000.
5. Ilog Solver 4.4. Reference Manual. Ilog SA, Gentilly, France, 1998.
6. D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. *Operations Research*, 37(6):865–893, 1989.
7. F. Laburthe and Y. Caseau. SALSA: A Language for Search Algorithms. In *CP'98*, Pisa, Italy, October 1998.
8. L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5:41–82, 2000.
9. L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. In *OOPSLA-02*.
10. L. Michel and P. Van Hentenryck. A Simple Tabu Search for Warehouse Location. *European Journal of Operational Research*, 2004. in press.
11. S. Minton, M.D. Johnston, and A.B. Philips. Solving Large-Scale Constraint Satisfaction and Scheduling Problems using a Heuristic Repair Method. In *AAAI-90*.
12. A. Nareyek. Constraint-Based Agents. Springer Verlag, 1998.
13. A. Nareyek. DragonBreath. www.ai-center.com/projects/dragonbreath/, 2004.
14. G. Ramalingam. *Bounded Incremental Computation*. PhD thesis, University of Wisconsin-Madison, 1993.
15. B. Selman, H. Kautz, and B. Cohen. Noise Strategies for Improving Local Search. In *AAAI-94*, pages 337–343, 1994.
16. P. Shaw, B. De Backer, and V. Furnon. Improved local search for CP toolkits. *Annals of Operations Research*, 115:31–50, 2002.
17. P. Van Hentenryck and L. Michel. Control Abstractions for Local Search. In *CP'03*, Cork, Ireland, 2003. (Best Paper Award).
18. S. Voss and D. Woodruff. *Optimization Software Class Libraries*. Kluwer Academic Publishers, 2002.
19. C. Voudouris and E. Tsang. Partial constraint satisfaction problems and guided local search. In *PACT'96*, pages 337–356, 1996.
20. J. Walser. *Integer Optimization by Local Search*. Springer Verlag, 1998.