# Differentiable Invariants

Pascal Van Hentenryck[1] and Laurent Michel[2]

[1] Brown University, Box 1910, Providence, RI 02912
[2] University of Connecticut, Storrs, CT 06269-2155

**Abstract.** Invariants that incrementally maintain the value of expressions under assignments to their variables are a natural abstraction to build high-level local search algorithms. But their functionalities are not sufficient to allow arbitrary expressions as constraints or objective functions as in constraint programming. Differentiable invariants bridge this expressiveness gap. A differentiable invariant maintains the value of an expression and its variable gradients, it supports differentiation to evaluate the effect of local moves. The benefits of differentiable invariants are illustrated on a number of applications which feature complex, possibly reified, expressions and whose models are essentially similar to their CP counterparts. Experimental results demonstrate their practicability.

## 1 Introduction

Local search algorithms approach the solving of combinatorial optimization problems by moving from solutions to solutions until a feasible solution or a high-quality solution is found. These algorithms typically maintain sophisticated data structures to evaluate or to propagate the effect of local moves quickly. Since, in general, the neighborhood does not vary dramatically when moving from one solution to one of its neighbors, these incremental data structures may significantly speed up local search algorithms.

Invariants were introduced in Localizer [3] to automate the tedious and error-prone implementation of incremental data structures. An invariant declaratively specifies a (numerical, set, or graph) expression whose value must be maintained incrementally under local moves. Invariants were shown to be instrumental in simplifying the implementation of many local search algorithms. However, the resulting algorithms were still not expressed at a similar level of abstraction as constraint programming (CP) approaches for the same problems. This recognition led to the concept of differentiable objects [5,8] which have emerged as the cornerstone of constraint-based local search (CBLS). In CBLS, objective functions and constraints, which are differentiable objects, not only maintain the value of an expression: they also maintain variable gradient (e.g., to determine how the expression value increases/decreases by changing a variable) and support differentiation (e.g., to assess the effect of a local move on the expression value). Although differentiable objects are often implemented using invariants (see [8] for some examples), it is still cumbersome, difficult, and repetitive to derive correct invariants for a given differentiable object.

This paper aims at bridging the expressiveness gap between invariants and differentiable objects by providing systematic ways of deriving differential objectives and constraints. *It proposes the concept of differentiable invariant that automatically lifts an arbitrarily complex expression into a differentiable objective function.* Like invariants, the resulting objective function incrementally maintains the value of the expression. Unlike invariants, it also maintains variable gradients (to determine how much a variable may increase/decrease the value of the expression) and supports the differentiability (to determine the effect of local moves on the expression value). Moreover, since a differentiable constraint can be seen as differentiable objective function maintaining its violations, *differentiable invariants automatically lift arbitrarily complex relations into differentiable constraints.* The resulting differentiable constraints maintain the violations of the relations, their variable violations (to determine how much a variable may increase/decrease the violations), and support differentiability.

As a consequence, differentiable invariants bring two main benefits for CBLS. First, expressions can now be used to state complex idiosyncratic constraints and objectives declaratively, a functionality that have accounted for much of the industrial success of constraint programming and that relieves programmers from deriving specific invariants and algorithms for each possible expressions. In other words, in the same way as CP languages perform domain reduction on arbitrary expressions, Comet now allows arbitrary expressions and relations as differentiable invariants, maintaining their values, their violations, and their variable gradients, as well as supporting differentiability. Second, differentiable invariants allow CBLS and CP models to be remarkably close since both now feature a similar, rich language for stating constraints and objectives.

The rest of this paper illustrates the concept of differentiable invariants, describes their implementation, and reports experimental results. Sections 2–4 show how differentiable invariants are a natural vehicle for modeling the spatially balanced latin square, scene allocation, and progressive party problems. Sections 5–7 show how to implement differentiable invariants in stepwise refinements, starting with their evaluations and gradients before presenting constraints and their reification. Section 8 presents the experimental results.

## 2 Totally Spatially Balanced Latin Squares

The first application consists of generating spatially balanced scientific experiment designs and, in particular, totally spatially balanced Latin squares [2].

*The Problem:* A latin square of size $n$ is an $n \times n$ matrix in which each number in $1..n$ appears exactly once in each row and column. The distance $d_r(v,w)$ of a pair $(v,w)$ in row $r$ is the absolute difference of the column indices in which $v$ and $w$ appear in row $r$. The total distance of a pair $(v,w)$ is given by

$$d(v,w) = \sum_{r=1}^{n} d_r(v,w).$$

A Latin square is totally spatially balanced if

$$d(v,w) = \frac{n(n+1)}{3} \quad (1 \leq k < l \leq n).$$

Gomes et al [2] introduced this problem to the community and proposed both local search and constraint programming solutions. Subsequently, Gomes [6] proposed a streamlined local search which solves large instances rapidly by permuting columns. This section only considers the local search in [2] since it raises more interesting modeling issues (for our purposes) and motivated our initial research on differentiable invariants.

*The Model:* The model uses a variable $col[r,v]$ to denote the column of value $v$ in row. The (latin square) constraint that a value $v$ appears in exactly one column is expressed by an *alldifferent* constraint $alldifferent(col[1,v],...,col[n,v])$. The (latin square) constraint that all the values in a row are different is an (implicit) hard constraint. It holds initially by assigning all rows to a permutation of $1..n$ and is maintained during the search by the local moves. The constraint that the Latin square be totally spatially balanced is soft and is transformed into an objective function. Hence the goal is to find a latin square minimizing

$$O = \sum_{1 \leq v < w \leq n} \big(d(v,w) - \frac{n(n+1)}{3}\big)^2.$$

Since the column constraint is a soft constraint as well, the overall problem can then viewed as minimizing the objection function $n \times viol(S) + O$ where *viol(S)* denotes the violations of the soft constraints and is weighted by $n$.

*The Search:* The local search is a tabu procedure swapping the position of two values on the same row. The best non-tabu move is selected at each step. The local search also uses an intensification component and a restart strategy.

*The Comet Program:* Figure 1 depicts the Comet statement. The declaration of data and decision variables are in lines 1–8. The soft constraints are specified in lines 9–11. The objective for spatial balance is in lines 12–14. The global objective is in line 15. The search procedure is in line 17–27. The Comet program is almost a one-to-one mapping of the informal description presented earlier and we review some of its components now. The matrix of decision variables is declared in line 4. All variables have a domain $1..n$ and each row is initialized by a random permutation (lines 5–8). A constraint system `S` is declared in line 9 and it contains all the soft constraints expressing that a value `v` appears atmost once in each column. The objective function for balancing the latin square spatially is specified in line 12. Variable `OS` (declared in line 12) is a sum of objectives, one for each pair of values `(v,w)` to express their relative balance, i.e.,

```
((sum(r in R) (abs(col[r,v] - col[r,w])))-balance)^2
```

Such an objective is a differentiable invariant involving absolute values, a subtraction, a square function, and an aggregate operator. The value of the expression is maintained incrementally under changes to its variables (i.e., all the decision variables `col[r,w]` and `col[r,w]` associated with values `v` and `w`). Moreover, the objective is differentiable and can be queried to evaluate the effect of local moves. Finally, it also maintains gradient information to estimate how much each variable may increase or decrease its value. Observe also that the expression involves absolutes values and a square function. The soft constraints are transformed into an objective function in line 15, specifying the overall objective `O` that combines the weighted violations of the soft constraints with the sum of the balance objectives.

(Part of) the tabu search is depicted in lines 17–27. As long as `O` evaluates to strictly positive value, the search selects the positions of the values $v$ and $w$ on row $r$ that are not tabu and whose swap produces the best value of the objective function. The call in line 22, i.e., `O.getSwapDelta(col[r,v],col[r,w])`, is particularly interesting, as it queries the soft constraints and the objectives to evaluate the candidate swap. This ability to estimate the effect of loval moves on arbitrary expressions is one of the novel contributions of differentiable invariants.

It is useful to emphasize that Comet enables a direct and natural formulation of the model. The constraints and objective functions are expressed declaratively. Their violations and evaluations are maintained incrementally and can be queried to assess the impact of local moves, providing the clean separation between model and search typically associated with CP and CBLS. The novelty for CBLS is the ability to use complex expressions as objectives.

## 3 Scene Allocation

The second application is the scene allocation problem [7].

*The Problem:* A scene allocation consists of deciding when to shoot scenes for a movie. Each scene involves a number of actors and each actor may appear in a number of different scenes. All actors of a scene must be present on the day the scene is shot and at most 5 scenes a day can be filmed. The actors have fees representing the amount to be paid per day they spent in the studio. The goal of the application is to minimize the production costs while satisfying the capacity constraints on the number of scenes per day.

*The Model:* The local search model is essentially the same as the CP model. It associates a variable $scene[s]$ with every scene $s$ to represent the day $s$ is filmed. The objective function uses reification to decide whether to pay an actor on a given day. The capacity on the scenes is an (implicit) hard constraint. It is satisfied by the initial assignment and maintained through local moves.

*The Local Search:* The local search is again a tabu procedure whose local moves swaps the days allocated to two scenes. Once again, the best non-tabu swap is selected at each step. A restarting strategy is also used.

```
int n = 8;
range R = 1..n;
int balance = n*(n+1)/3;
var{int} col[R,R](mgr,R);
forall(r in R) {
   RandomPermutation p(R);
   forall(v in R) col[r,v] := p.get();
}
ConstraintSystem S(mgr);
forall(v in R)
   S.post(alldifferent(all(r in R) col[r,v]));
ObjectiveSum OS(mgr);
forall(v in R, w in R: v < w)
   OS.post(((sum(r in R) (abs(col[r,v] - col[r,w])))-balance)^2);
Objective O = n * S + OS;
mgr.close();
int tabu[R,R,R] = -1;
int tabuLength = 10;
int it = 0;
while (O.evaluation() > 0) {
   selectMin(r in R,v in R, w in R: v < w && tabu[r,v,w] <= it)
            (O.getSwapDelta(col[r,v],col[r,w])) {
     col[r,v] :=: col[r,w];
     tabu[r,v,w] = it + tabuLength;
  }
  it++;
}
```

**Fig. 1.** A Simple Tabu-Search Algorithm for the Balanced Latin Square Problem

*The Comet Program:* The COMET program is (partially) depicted in Figure 2. For space reasons, the search procedure is omitted but is essentially the same as in the latin square application. Lines 1–8 declare and initialize the data. In particular, they declare the scenes, the days, the actors (line 4), the actors' fees (line 5), and the actors appearing in the scenes (line 6). Line 8 also specifies the scenes in which an actor appears, which is convenient to state the constraints.

The data and decision variables are declared in lines 9–17. A decision variable `scene[s]` specifies the day scene `s` is scheduled and the scene are allocated randomly initially (see lines 11–12). The most interesting part of the model is the objective function `O` (declared in line 14) which sums the fees of all actors on all days (lines 15–16). Each sub-objective thus represents the fee to be paid by an actor `a` on a day `d`. It is expressed by the differentiable expression

```
pay[a] * (or(s in which[a]) scene[s] == d)
```

which uses reification to determine whether actor `a` plays on day `d`. More precisely, if a scene `s` in `which[a]` is scheduled on day `d`, the disjunction holds and is reified to 1, in which case the amount is `pay[a]`. Otherwise, the disjunction does not hold and is reified to 0.

```
include "localSolver";
int maxScene = 19; range Scenes = 0..maxScene-1;
int maxDay = 5; range Days = 0..maxDay-1;
enum Actor = ...;
int pay[Actor] = ...;
set{Actor} appears[Scenes];
...
set{int} which[a in Actor] = setof(s in Scenes) member(a,appears[s]);
LocalSolver mgr();
var{int} scene[Scenes](mgr,Days);
RandomPermutation perm(Scenes);
forall(i in Scenes) scene[perm.get()] := i/maxDay;

ObjectiveSum O(mgr);
forall(a in Actor, d in Days)
   O.post(pay[a] * (or(s in which[a]) scene[s] == d));
```

**Fig. 2.** A Simple Tabu-Search Algorithm for the Scene Allocation Problem

What we find remarkable here is that the Comet model is almost identical to the OPL model in [7]: the only difference (besides syntactical details) is the fact that the cardinality constraint on the days is omitted since it holds initially and is maintained by local moves. This similarity is possible because differentiable invariants allows objective function to be complex reified expressions.

## 4 The Progressive Party Problem

Our last application is the progressive party problem which has been used several times to illustrate constraint-based local search [5,9]. The motivation here is to show that differentiable invariants can also be used to state constraints, providing the equivalent for CBLS of high-order or meta-constraint in CP. The key insight is to recognize that a constraint is nothing else but a differentiable invariant maintaining its violations.

*The Comet Program:* The Comet program is (partially) depicted in Figure 3. The search procedure can be found in earlier publications (e.g., [8]). The data is described in lines 1–7 and the decision variables are declared and initialized in lines 10–11. A decision variable `boat[g,p]` specifies the boat that group `g` visits in period `p`. The core of the model are the constraints in line 12–18.

The novelty here is in how the model expresses that no two groups meet more than once (in line 18). In [8], this constraint was expressed using a cardinality operator `atmost`. The model above uses a meta-constraint

```
sum(p in Periods) (boat[i,p] == boat[j,p]) <= 1
```

which is the way it would probably be expressed using a traditional constraint-programming tool such as OPL or Ilog Solver. The Comet implementation automatically derives the constraint violations of the constraints, i.e.,

```
include "LocalSolver";
int up = 6;
range Hosts = 1..13;
range Guests = 1..29;
range Periods = 1..up;
int cap[Hosts] = ...;
int crew[Guests] = ...;

LocalSolver m();
UniformDistribution distr(Hosts);
var{int} boat[Guests,Periods](m,Hosts) := distr.get();

ConstraintSystem S(m);
forall(g in Guests)
   S.post(2 * alldifferent(all(p in Periods) boat[g,p]));
forall(p in Periods)
   S.post(2 * knapsack(all(g in Guests) boat[g,p],crew,cap));
forall(i in Guests, j in Guests : j > i)
   S.post(sum(p in Periods) (boat[i,p] == boat[j,p]) <= 1);
```

**Fig. 3.** A COMET Model for the Progressive Party Problem

```
max(0,sum(p in Periods) (boat[i,p] == boat[j,p]) - 1)
```

which is a differentiable invariant involving, once again, reification. Note also that the COMET implementation must automatically derive the variable violations for these constraints, since the search procedure first selects variable with the most violations before choosing the value decreasing the violations the most. This highlights the benefits of differentiable invariants: they let programmers state constraints declaratively while systematically deriving their violations, their variable violations, and differentiation algorithms. How this is achieved is the topic of the next sections.

## 5 Expressions as Differentiable Objective Functions

As mentioned earlier, a differentiable invariant transforms an expression into a differentiable objective function. The syntax of the expressions used in this paper is given in Figure 4. Differentiable invariants must thus implement, for any such expression, the interface of objective functions depicted in 5. Method `evaluation` specifies the value of the objective function, which is maintained by invariants. Methods `increase` and `decrease` return gradient information for a decision variable `x`, i.e., they estimate by how much the objective may increase or decrease by re-assigning `x`. These gradients are also maintained incrementally. Note that the ability of determining both increasing and decreasing gradients is critical even if one is interested in minimization only. The next three methods specify how the objective value evolve under local moves, i.e., the assignment of a value to a variable, the swap of two variables, and the assignments of values

$v \in \mathcal{N}$; $x, y \in$ *Variable*; $e \in$ *Expression*;
$e ::= v \mid x \mid e+e \mid e-e \mid e \times e \mid \min(e,e) \mid \max(e,e) \mid -e \mid abs(e) \mid e^2 \mid (e) \mid c$

**Fig. 4.** The Syntax of Expressions (Partial Description)

```
interface Objective {
  var{int} evaluation();
  var{int} increase(var{int} x);
  var{int} decrease(var{int} x);
  int getAssignDelta(var{int} x,int v);
  int getSwapDelta(var{int} x,var{int} y);
  int getAssignDelta(var{int}[] x,var[] v);
  var{int}[] getVariables();
}
```

**Fig. 5.** The Objective Interface in Comet (Partial Description)

to a set of variables. The rest of this section shows how to implement these functionalities. Aggregate operations (e.g., for summation) can be viewed as shorthands for multiple applications of the same operators and are not discussed here for space reasons.

*Evaluations* Figure 6 shows how to evaluate an expression and how to maintain it through invariants. In the figure, $\mathbb{E}_\alpha[e]$ denotes the value of expression $e$ under variable assignment $\alpha$ and $i_e$ is the invariant maintaining $\mathbb{E}_\alpha[e]$. Both $\mathbb{E}_\alpha[e]$ and $i_e$ are defined by induction on the structure of expression $e$. In particular, there is one invariant associated with every sub-expression in $e$, which is important to implement gradients and differentiations efficiently. In this paper, a variable assignment is a function from variables to integers. Moreover, $\alpha[x/v]$ denotes the assignment behaving like $\alpha$ except that $x$ is now assigned to $v$. This notation may be generalized to multiple variables. These evaluations do not raise any difficulty and the algorithms to maintain these invariants efficiently are presented in [4]. Note that method `evaluation` in Figure 5 returns $i_e$ for the objective function associated with $e$.

*Gradients.* Many search procedures choose local moves by a two-step approach, first selecting the variable to re-assign and then the new value. Typically, the variable selection uses gradients, i.e., information on how much the objective function or the violations may increase/decrease by changing the value of a variable. Since such a variable selection takes place at every iteration of the local search, such gradients are typically maintained incrementally in systems such as Comet. The section shows how to evaluate and maintain gradients for the expressions depicted earlier. The gradients must satisfy the following inequalities:

$$\overset{x}{\underset{\alpha}{\uparrow}} e \geq \max_{v \in D_x} \underset{\alpha[x/v]}{\mathbb{E}}[e] - \underset{\alpha}{\mathbb{E}}[e] \qquad \text{and} \qquad \overset{x}{\underset{\alpha}{\downarrow}} e \geq \underset{\alpha}{\mathbb{E}}[e] - \min_{v \in D_x} \underset{\alpha[x/v]}{\mathbb{E}}[e]$$

where $D_x$ denotes the domain of variable $x$. Variable gradients thus provide optimistic evaluations to the maximum increase/decrease of expression $e$ by

$$\begin{array}{ll|ll}
\underset{\alpha}{\mathbb{E}}[v] & = v & i_v & \leftarrow v \\
\underset{\alpha}{\mathbb{E}}[x] & = \alpha(x) & i_x & \leftarrow x \\
\underset{\alpha}{\mathbb{E}}[e_1 + e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] + \underset{\alpha}{\mathbb{E}}[e_2] & i_{e_1+e_2} & \leftarrow i_{e_1} + i_{e_2} \\
\underset{\alpha}{\mathbb{E}}[e_1 - e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] - \underset{\alpha}{\mathbb{E}}[e_2] & i_{e_1-e_2} & \leftarrow i_{e_1} - i_{e_2} \\
\underset{\alpha}{\mathbb{E}}[e_1 \times e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] \times \underset{\alpha}{\mathbb{E}}[e_2] & i_{e_1\times e_2} & \leftarrow i_{e_1} \times i_{e_2} \\
\underset{\alpha}{\mathbb{E}}[-e] & = -\underset{\alpha}{\mathbb{E}}[e] & i_{-e} & \leftarrow -i_e \\
\underset{\alpha}{\mathbb{E}}[abs(e)] & = abs(\underset{\alpha}{\mathbb{E}}[e]) & i_{abs(e)} & \leftarrow abs(i_e) \\
\underset{\alpha}{\mathbb{E}}[e^2] & = (\underset{\alpha}{\mathbb{E}}[e])^2 & i_{e^2} & \leftarrow (i_e)^2 \\
\underset{\alpha}{\mathbb{E}}[\min(e_1,e_2)] & = \min(\underset{\alpha}{\mathbb{E}}[e_1], \underset{\alpha}{\mathbb{E}}[e_2]) & i_{\min(e_1,e_2)} & \leftarrow \min(i_{e_1}, i_{e_2}) \\
\underset{\alpha}{\mathbb{E}}[\max(e_1,e_2)] & = \max(\underset{\alpha}{\mathbb{E}}[e_1], \underset{\alpha}{\mathbb{E}}[e_2]) & i_{\max(e_1,e_2)} & \leftarrow \max(i_{e_1}, i_{e_2})
\end{array}$$

**Fig. 6.** The Evaluation of Expressions and their Underlying Invariants

re-assigning variable $x$ only. It is critical to use optimistic evaluations since pessimistic evaluations may artificially reduce the connectivity of the neighborhood. Many of the gradients satisfy these relations at equality. However, for efficiency reasons, it may be beneficial to approximate the right-hand sides for complex nonlinear expressions with multiple occurrences of the same variables. In the following, the assignment $\alpha$ is implicit (unless specified otherwise). Similarly, the gradients are always taken with respect to variable $x$, and $y$ denotes a variable different from $x$. The minimum and maximum values in the domain $D_x$ of variable $x$ are denoted by $m_x$ and $M_x$.

Figure 7 depicts the evaluations of the gradients whose definitions are mutually recursive. It is useful to review some of the rules to convey the intuition. The increasing gradient for subtraction, i.e., $\uparrow[e_1 - e_2] = \uparrow e_1 + \downarrow e_2$, uses the increasing gradient on $e_1$ and the decreasing gradient on $e_2$. The rule for absolute value can be written as

$$\uparrow[abs(e)] = \max(abs(\mathbb{E}[e] + \uparrow e), abs(\mathbb{E}[e] - \downarrow e)) - \mathbb{E}[abs(e)].$$

It indicates that there are two ways to increase the absolute value of $e$: increase or decrease $e$. The definition captures the increase and subtracts the current value of $e$. The rule for square and for min/max are similar in spirit, while the multiplication has a more complex case analysis due to the possible signs of the underlying expressions. Observe also the base case for variable $x$ which returns the difference between $M_x$ and $\alpha(x)$. The decreasing gradient for absolute value

$$\downarrow[abs(e)] = \text{if } \mathbb{E}[e] \geq 0 \text{ then } \min(\mathbb{E}[e], \downarrow e) \text{ else } \min(-\mathbb{E}[e], \uparrow e)$$

is interesting. If $\mathbb{E}[e] \geq 0$, the gradient is obtained by decreasing $e$ but the decrease must be bounded by $\mathbb{E}[e]$ since zero is the smallest possible value. Observe that the maximum decrease of $abs(e)$ is not necessarily obtained by the maximum decrease of $e$ and hence the gradient is optimistic. The case $\mathbb{E}[e] < 0$ is symmetric and obtained by increasing $e$ up to $-\mathbb{E}[e]$.

$$
\begin{array}{ll}
\mathbb{E}[e]^{+} & = \mathbb{E}[e] + \uparrow e \\
\mathbb{E}[e]^{-} & = \mathbb{E}[e] - \downarrow e \\
\uparrow[v] & = 0 \\
\uparrow[y] & = 0 \\
\uparrow[x] & = M_x - \alpha(x) \\
\uparrow[e_1 + e_2] & = \uparrow e_1 + \uparrow e_2 \\
\uparrow[e_1 - e_2] & = \uparrow e_1 + \downarrow e_2 \\
\uparrow[-e] & = \downarrow e \\
\uparrow[abs(e)] & = \max(abs(\mathbb{E}[e]^{+}), abs(\mathbb{E}[e]^{-})) - \mathbb{E}[abs(e)] \\
\uparrow[e^2] & = \max((\mathbb{E}[e] + \uparrow e)^2, (\mathbb{E}[e] - \downarrow e)^2) - \mathbb{E}[e^2] \\
\uparrow[\max(e_1, e_2)] & = \max(\mathbb{E}[e_1]^{+}, \mathbb{E}[e_2]^{+}) - \mathbb{E}[\max(e_1, e_2)] \\
\uparrow[\min(e_1, e_2)] & = \min(\mathbb{E}[e_1]^{+}, \mathbb{E}[e_2]^{+}) - \mathbb{E}[\min(e_1, e_2)] \\
\uparrow[e_1 * e_2] & = \max(\mathbb{E}[e_1]^{+} * \mathbb{E}[e_2]^{+}, \mathbb{E}[e_1]^{+} * \mathbb{E}[e_2]^{-}, \mathbb{E}[e_1]^{-} * \mathbb{E}[e_2]^{+}, \mathbb{E}[e_1]^{-} * \mathbb{E}[e_2]^{-}) \\
 & \quad -\mathbb{E}[e_1 * e_2] \\
 & \\
\downarrow[v] & = 0 \\
\downarrow[y] & = 0 \\
\downarrow[x] & = \alpha(x) - m_x \\
\downarrow[e_1 + e_2] & = \downarrow e_1 + \downarrow e_2 \\
\downarrow[e_1 - e_2] & = \downarrow e_1 + \uparrow e_2 \\
\downarrow[-e] & = \uparrow e \\
\downarrow[abs(e)] & = \text{if } \mathbb{E}[e] \geq 0 \text{ then } \min(\mathbb{E}[e], \downarrow e) \text{ else } \min(-\mathbb{E}[e], \uparrow e) \\
\downarrow[e^2] & = \mathbb{E}[e]^2 - \text{if } \mathbb{E}[e] \geq 0 \text{ then } (\mathbb{E}[e] - \min(\mathbb{E}[e], \downarrow e))^2 \text{ else } (\mathbb{E}[e] + \min(-\mathbb{E}[e], \uparrow e))^2 \\
\downarrow[\max(e_1, e_2)] & = \mathbb{E}[\max(e_1, e_2)] - \max(\mathbb{E}[e_1]^{-}, \mathbb{E}[e_2]^{-}) \\
\downarrow[\min(e_1, e_2)] & = \mathbb{E}[\min(e_1, e_2)] - \min(\mathbb{E}[e_1]^{-}, \mathbb{E}[e_2]^{-}) \\
\downarrow[e_1 * e_2] & = \mathbb{E}[e_1 * e_2] - \\
 & \quad \min(\mathbb{E}[e_1]^{+} * \mathbb{E}[e_2]^{+}, \mathbb{E}[e_1]^{+} * \mathbb{E}[e_2]^{-}, \mathbb{E}[e_1]^{-} * \mathbb{E}[e_2]^{+}, \mathbb{E}[e_1]^{-} * \mathbb{E}[e_2]^{-})
\end{array}
$$

**Fig. 7.** The Evaluation for the Variable Gradients

Figure 8 depicts the invariants maintaining the gradients. There is an invariant $i^{\uparrow}_{e}$ and an invariant $i^{\downarrow}_{e}$) associated with each expression $e$ and each variable $x$ (which is implicit in the figure). These gradient invariants use both gradient invariants on the sub-expressions and evaluation invariants. For instance, the (increasing) gradient invariant for subtraction, i.e.,

$$i^{\uparrow}_{e_1 - e_2} \leftarrow i^{\uparrow}_{e_1} - i^{\downarrow}_{e_2}$$

uses increasing and decreasing gradient invariants on the sub-expressions. The gradient invariant for absolute value can be written as

$$i^{\uparrow}_{abs(e)} \leftarrow \max(abs(i_e + i^{\uparrow}_{e}), abs(i_e - i^{\downarrow}_{e}) - i_{abs(e)}$$

and illustrates the use of invariants $i_e$ and $i_{abs(e)}$ for accessing the current value of $e$ and $abs(e)$. Note that methods `increase` and `decrease` in Figure 5 returns $i^{\uparrow}_{e}$ and $i^{\downarrow}_{e}$ for the objective function associated with $e$.

*Differentiation* Differentiable methods can be evaluated directly. Indeed, given an expression $e$, a variable $x$, and a value $v$, method `e.getAssignDelta(x,v)` returns $\mathbb{E}_{\alpha[x/v]}[e] - \mathbb{E}_{\alpha}[e]$ where $\alpha$ is the current assignment. It is too costly to

$$\begin{array}{ll}
i_e^+ & \leftarrow i_e + i_e^\uparrow \\
i_e^- & \leftarrow i_e - i_e^\downarrow \\
\\
i_v^\uparrow & \leftarrow 0 \\
i_y^\uparrow & \leftarrow 0 \\
i_x^\uparrow & \leftarrow M_x - x \\
i_{e_1+e_2}^\uparrow & \leftarrow i_{e_1}^\uparrow + i_{e_2}^\uparrow \\
i_{e_1-e_2}^\uparrow & \leftarrow i_{e_1}^\uparrow + i_{e_2}^\downarrow \\
i_{-e}^\uparrow & \leftarrow i_e^\downarrow \\
i_{abs(e)}^\uparrow & \leftarrow \max(abs(i_e^+), abs(i_e^-)) - i_{abs(e)} \\
i_{e^2}^\uparrow & \leftarrow \max((i_e^+)^2, (i_e^-)^2) - i_{e^2} \\
i_{\max(e_1,e_2)}^\uparrow & \leftarrow \max(i_{e_1}^+, i_{e_2}^+) - i_{\max(e_1,e_2)} \\
i_{\min(e_1,e_2)}^\uparrow & \leftarrow \min(i_{e_1}^+, i_{e_2}^+) - i_{\min(e_1,e_2)} \\
i_{e_1*e_2}^\uparrow & \leftarrow \max(i_{e_1}^+ * i_{e_2}^+, i_{e_1}^+ * i_{e_2}^-, i_{e_1}^- * i_{e_2}^+, i_{e_1}^- * i_{e_2}^-) - i_{e_1*e_2} \\
\\
i_v^\downarrow & \leftarrow 0 \\
i_y^\downarrow & \leftarrow 0 \\
i_x^\downarrow & \leftarrow x - m_x \\
i_{e_1+e_2}^\downarrow & \leftarrow i_{e_1}^\downarrow + i_{e_2}^\downarrow \\
i_{e_1-e_2}^\downarrow & \leftarrow i_{e_1}^\downarrow + i_{e_2}^\uparrow \\
i_{-e}^\downarrow & \leftarrow i_e^\uparrow \\
i_{abs(e)}^\downarrow & \leftarrow \text{if } i_e \geq 0 \text{ then } \min(i_e, i_e^\downarrow) \text{ else } \min(-i_e, i_e^\uparrow) \\
i_{e^2}^\downarrow & \leftarrow i_{e^2} - \text{if } i_e \geq 0 \text{ then } (i_e - \min(i_e, i_e^\downarrow))^2 \text{ else } (i_e + \min(-i_e, i_e^\uparrow))^2 \\
i_{\max(e_1,e_2)}^\downarrow & \leftarrow i_{\max(e_1,e_2)} - \max(i_{e_1}^-, i_{e_2}^-) \\
i_{\min(e_1,e_2)}^\downarrow & \leftarrow i_{\min(e_1,e_2)} - \min(i_{e_1}^-, i_{e_2}^-) \\
i_{e_1*e_2}^\downarrow & \leftarrow i_{e_1*e_2} - \min(i_{e_1}^+ * i_{e_2}^+, i_{e_1}^+ * i_{e_2}^-, i_{e_1}^- * i_{e_2}^+, i_{e_1}^- * i_{e_2}^-)
\end{array}$$

**Fig. 8.** The Invariants of the Variable Gradients

maintain these evaluations incrementally for each pair $(x, v)$ in general. However, differentiable methods may exploit the fact that variables typically occur only in some sub-expressions and reuse the evaluations that are maintained incrementally. This is particularly important for aggregate operators. Consider an expression $e_1 + \ldots + e_n$ and assume that $x$ appears only in $e_1$. Then

$$\mathop{\mathbb{E}}_{\alpha[x/v]}[e_1 + \ldots + e_n] = \mathop{\mathbb{E}}_{\alpha[x/v]}[e_1] - \mathop{\mathbb{E}}_{\alpha}[e_1]$$

and the differentiable method only needs to evaluate $\mathbb{E}_{\alpha[x/v]}[e_1]$. By induction, differentiable methods then only evaluates the leaves containing the variables to be assigned and the branches from the root to these leaves.

## 6 Relational Expressions as Differentiable Constraints

This section shows how relational expressions can be translated into constraints. Recall that, in CBLS, a constraint is a differentiable object maintaining its violations and its variable violations, and supporting differentiation. In order to

$$\begin{array}{l} r \in \quad \textit{Relation.} \\ r ::= e = e \mid e \leq e \mid e \neq e \mid r \vee r \mid r \wedge r \mid \neg r \end{array}$$

$$\begin{array}{ll} \mathbb{V}[e_1 = e_2] & = abs(e_1 - e_2) \\ \mathbb{V}[e_1 \leq e_2] & = \max(e_1 - e_2, 0) \\ \mathbb{V}[e_1 \neq e_2] & = 1 - \min(1, abs(e_1 - e_2)) \\ & \\ \mathbb{V}[r_1 \wedge r_2] & = \mathbb{V}[r_1] + \mathbb{V}[r_2] \\ \mathbb{V}[r_1 \vee r_2] & = \min(\mathbb{V}[r_1], \mathbb{V}[r_2]) \\ \mathbb{V}[\neg r] & = 1 - \min(1, \mathbb{V}[r]) \end{array}$$

**Fig. 9.** Constraints as Objective Functions

```
interface constraint {
  var{int} violations();
  var{int} violations(var{int} x);
  int getAssignDelta(var{int} x,int v);
  int getSwapDelta(var{int} x,var{int} y);
  int getAssignDelta(var{int}[] x,var[] v);
  var{int}[] getVariables();
}
```

**Fig. 10.** The Constraint Interface in Comet (Partial Description)

translate a relation into a constraint, the key idea is to map the relation $r$ into an expression $\mathbb{V}[r]$ denoting its violations. Once such a mapping $\mathbb{V} : \textit{Relation} \rightarrow \textit{Expression}$ is available, the constraint interface depicted in Figure 10 can be naturally implemented. In particular,

– $\underset{\alpha}{\mathbb{E}}[\mathbb{V}[r]]$ denotes the violations of $r$ for $\alpha$, incrementally maintained by $i_{\mathbb{V}[r]}$ which is returned by method `violations()` of the constraint interface.
– $\overset{x}{\underset{\alpha}{\downarrow}}\mathbb{V}[r]$ is the variable violations of $x$ for $\alpha$, incrementally maintained by $i^{\downarrow}_{\mathbb{V}[r]}$ which is returned by method `violations(var{int} x)`.

Figure 9 depicts the syntax of relations and a mapping $\mathbb{V}$ for a variety of relations and logical connectives. For instance, the violations of a relation $e_1 = e_2$ are specified by the expression $\mathbb{V}[e_1 = e_2] = abs(e_1 - e_2)$. The resulting expression can then be transformed into a differentiable objective which incrementally maintains the constraint violations using the invariant $i_{abs(e_1-e_2)}$ and the variable violations using the gradient invariant $i^{\downarrow}_{abs(e_1-e_2)}$. Observe how differentiable invariants preclude the need to derive specific variable violations (as in [1]), since variable violations are directly inherited from violation expressions.

## 7 Reification

Since expressions and relations can be both transformed into objectives, it becomes natural to support reification in expressions. Reification, a fundamental

$$\begin{array}{ll|ll}
\underset{\alpha}{\mathbb{E}}[r] & = \delta(\underset{\alpha}{\mathbb{B}}[r]) & i_r & \leftarrow \delta(b_r) \\
\underset{\alpha}{\mathbb{B}}[e_1 = e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] = \underset{\alpha}{\mathbb{E}}[e_2] & b_{e_1=e_2} & \leftarrow i_{e_1} = i_{e_2} \\
\underset{\alpha}{\mathbb{B}}[e_1 \leq e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] \leq \underset{\alpha}{\mathbb{E}}[e_2] & b_{e_1 \leq e_2} & \leftarrow i_{e_1} \leq i_{e_2} \\
\underset{\alpha}{\mathbb{B}}[e_1 \neq e_2] & = \underset{\alpha}{\mathbb{E}}[e_1] \neq \underset{\alpha}{\mathbb{E}}[e_2] & b_{e_1 \neq e_2} & \leftarrow i_{e_1} \neq i_{e_2} \\
\underset{\alpha}{\mathbb{B}}[r_1 \vee r_2] & = \underset{\alpha}{\mathbb{B}}[r_1] \vee \underset{\alpha}{\mathbb{B}}[r_2] & b_{r_1 \vee r_2} & \leftarrow b_{r_1} \vee b_{r_2} \\
\underset{\alpha}{\mathbb{B}}[r_1 \wedge r_2] & = \underset{\alpha}{\mathbb{B}}[r_1] \wedge \underset{\alpha}{\mathbb{B}}[r_2] & b_{r_1 \wedge r_2} & \leftarrow b_{r_1} \wedge b_{r_2} \\
\underset{\alpha}{\mathbb{B}}[\neg r] & = \neg \underset{\alpha}{\mathbb{B}}[r] & b_{\neg r} & \leftarrow \neg b_r
\end{array}$$

**Fig. 11.** The Evaluation of Reified Constraints and their Corresponding Invariants

$$\begin{array}{l|l}
\overset{x}{\underset{\alpha}{\downarrow}} r = \text{let } e = \mathbb{V}[r] \text{ in } \delta(\underset{\alpha}{\mathbb{B}}[r] \ \wedge \ \overset{x}{\underset{\alpha}{\uparrow}} e > 0) & i^{\downarrow}_r \leftarrow \delta(b_r \ \wedge \ i^{\uparrow}_{\mathbb{V}[r]} > 0) \\
\overset{x}{\underset{\alpha}{\uparrow}} r = \text{let } e = \mathbb{V}[r] \text{ in } \delta(\neg \underset{\alpha}{\mathbb{B}}[r] \ \wedge \ \overset{x}{\underset{\alpha}{\downarrow}} e \geq \underset{\alpha}{\mathbb{E}}[e]) & i^{\uparrow}_r \leftarrow \delta(\neg b_r \ \wedge \ i^{\downarrow}_{\mathbb{V}[r]} \geq i_{\mathbb{V}[r]})
\end{array}$$

**Fig. 12.** The Gradients of Reified Constraints and their Corresponding Invariants

technique in CP, was illustrated in the scene allocation and progressive party problems, in which expressions includes arithmetic operations over relations. It is different from, and more challenging than, the reification from differentiable constraints to differentiable objectives which already presented in [9]. To support reification in CBLS, it is necessary to specify how to evaluate reified expressions and their gradients. Figure 11 depicts the extensions of Figure 6 to handle reification in evaluations and their corresponding invariants. In the figure, $\mathbb{B}_\alpha[e]$ denotes the truth value of expression $e$ under assignment $\alpha$ and $b_e$ denotes the corresponding Boolean invariant. The figure also uses the Kronecker symbol $\delta$ to convert Boolean values into 0/1 values:

$$\delta(b) = \begin{cases} 1 & \text{if } b = \mathit{true}; \\ 0 & \text{otherwise.} \end{cases}$$

It remains to define how to evaluate the gradients of reified constraints, which is depicted in Figure 12. The definitions are specified generically using $\mathbb{B}$ and $\mathbb{V}$. The intuition is as follows: given an assignment $\alpha$, changing $x$ may decrease the evaluation of $r$ if $r$ holds for $\alpha$ (i.e., $\mathbb{B}_\alpha\, r$) and changing $x$ may violate $r$ (i.e., $i^{\uparrow}_{\mathbb{V}[r]} > 0$). Similarly, changing $x$ may increase the evaluation of $c$ if $c$ does not hold for $\alpha$ and changing $x$ may remove all violations of $c$, i.e.,

$$\overset{x}{\underset{\alpha}{\downarrow}} \mathbb{V}[r] \geq \underset{\alpha}{\mathbb{E}}[\mathbb{V}[r]].$$

Again, the invariants for maintaining gradients can be derived systematically from the evaluations.

**Table 1.** The Overhead of Differential Invariants

| $n$ | 6 | 7 | 8 | 9 |
|---|---|---|---|---|
| PP(Atmost) | 0.85 | 1.01 | 7.46 | 145.89 |
| PP(DI) | 2.18 | 2.46 | 12.46 | 213.29 |
| %Overhead | 256.47 | 243.56 | 67.20 | 46.20 |

It is also interesting to illustrate the expressions obtained for the reified constraints in the progressive party problem. These constraints are of the form

$$(x_1 = y_1) + \ldots + (x_p = y_p) \leq 1$$

where $x_1, \ldots, x_p, y_1, \ldots, y_p$ are all distinct variables. The invariants maintaining the violations are of the form

$$i_c \leftarrow \max(i_m, 0) \qquad\qquad i_m \leftarrow i_{\delta(x_1=y_1)} + \ldots + i_{\delta(x_p=y_p)} - 1$$

The gradient for variable $x_1$ is maintained through invariants of the form

$$i_c^{\downarrow} \leftarrow \max(i_m + i_m^{\downarrow}, 0) - i_c \qquad\qquad i_m^{\downarrow} \leftarrow \delta(b_{x_1=y_1} \wedge i^{\uparrow}_{abs(x_1-y_1)} > 0)$$

Observe how differentiable invariants abstract away the complexity behind these constraints, elegantly encompass reification, and allow constraint-based local search to support a constraint language as rich as in traditional CP languages.

## 8 Experimental Evaluation

This section provides preliminary evidence of the practicability of differentiable invariants. It studies the cost of differentiable invariants and the benefits of gradients invariants, and gives results on the applications.

*The Cost of Differentiable Invariants.* Table 1 reports the cost of differentiable invariants. It measures the time in seconds for finding solutions to the progressive party problem of increasing sizes (from 6 to 9 periods) when the hosts are boats 1–13. The table compares the Comet program with differentiable invariants (PP(DI) shown in Figure 3) with the same program is replaced by the cardinality operator proposed in [9]. Since there are a quadratic number of these constraints, this is where most of the computation time is spent. Both programs are compared using the deterministic mode of Comet so that they execute exactly the same local moves. The results indicate that the overhead of using differential invariants decreases as the problem size grows and goes down to 46% for the largest instance. Differentiable invariants thus introduces a reasonable overhead compared to a tailored cardinality operator. This overhead should be largely compensated by the simplicity of expressing complex idiosyncratic constraints, which frees programming from implementing special-purpose constraints, objective functions, or combinators.

**Table 2.** The Benefits of Gradient Invariants

| $n$ | 6 | 7 | 8 | 9 |
|---|---|---|---|---|
| PP(DI-G) | 6.05 | 6.98 | 122.76 | 2777.74 |
| PP(DI) | 2.18 | 2.46 | 12.46 | 213.29 |
| Speedup | 2.77 | 2.83 | 9.85 | 13.02 |

**Table 3.** Experimental Results on Scene Allocation and Balanced Latin Squares

| Bench | min(S) | $\mu(S)$ | max(S) | $\mu(TS)$ | $\sigma(S)$ | $\sigma(TS)$ |
|---|---|---|---|---|---|---|
| scene | 334144.00 | 335457.38 | 343256.00 | 1.10% | 0.72 | 0.06 |
| balance(8) | 0 | 0 | 0 | 0.0 | 13.85 | 14.74 |
| balance(9) | 0 | 0 | 0 | 0.0 | 61.26 | 51.78 |

*The Benefits of Gradient Invariants.* Table 2 reports experimental results on the benefits on maintaining variable gradients incrementally. It compares the results of the model in Figure 3 when the implementation incrementally updates (PP(DI)) or evaluates (PP(DI-G)) variable gradients. The results highlight the importance of gradient invariants as the speed-ups increase with the problem size to reach a 13-fold improvement on the largest instance.

*Other Experimental Results.* For completeness, Table 3 reports the experimental results on scene allocation and spatially balanced latin squares. The first three columns report the min, average, and maximal values of the objective function, the fourth column reports the average CPU time (in seconds), and the last two columns show the standard deviation. The scene allocation program, despite its simplicity, performs extremely well and does not need the symmetry-breaking required in the CP solution for good performance (see [7]). The COMET program for latin square is very competitive with the local search algorithms presented in [2] which use a similar neighborhood (but a different search strategy which is not specified precisely enough for reproduction).

Overall, these results show that differentiable invariants are an effective high-level abstraction to bridge the gap between invariants and differentiable objects. They allow programmers to express complex, idiosyncratic constraints declaratively, while leaving the system deriving invariants and incremental algorithms so important in constraint-based local search.

## References

1. M. Agren, P. Flener, and J. Pearson. Inferring Variable Conflicts for Local Search. In *CP'06*, September 2006.
2. C. Gomes, M. Sellmann, C. van Es1, and H. van Es. The Challenge of Generating Spatially Balanced Scientific Experiment Designs. In *CP-AI-OR'04*, Nice, 2004.
3. L. Michel and P. Van Hentenryck. Localizer: A Modeling Language for Local Search. In *CP'97)*, October 1997.

4. L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5:41–82, 2000.
5. L. Michel and P. Van Hentenryck. A Constraint-Based Architecture for Local Search. In *OOPSLA-02*, November 2002.
6. C. Smith, C. Gomes, and C. Fernàndez. Streamlining Local Search for Spatially Balanced Latin Squares. In *IJCAI-05*, Edinburgh, Scotland, July 2005.
7. P. Van Hentenryck. Constraint and Integer Programming in OPL. *Informs Journal on Computing*, 14(4):345–372, 2002.
8. P. Van Hentenryck. *Constraint-Based Local Search.* The MIT Press, 2005.
9. P. Van Hentenryck, L. Michel, and L. Liu. Constraint-Based Combinators for Local Search. In *CP'04*, October 2004.