On the Specification of Search Tree Ordering Heuristics by Pattern Matching in a Rule-Based Modeling Language

Julien Martin and Thierry Martinez and François Fages

EPI Contraintes, INRIA Paris-Rocquencourt,
BP105, 78153 Le Chesnay Cedex, France.
Julien.Martin@inria.fr Thierry.Martinez@inria.fr Francois.Fages@inria.fr
http://contraintes.inria.fr/

Abstract. In this paper, we show that in a rule-based modeling language, search tree ordering heuristics can be specified declaratively by pattern matching on left-hand sides of rule definitions. As opposed to other modeling languages able to express search heuristics, such as OPL and Comet, heuristics are expressed here purely declaratively. That eliminates the need of changing data structures or introducing intermediary objects. The price to pay for this ease of modeling is in the compilation process which we describe here with a formal system. We analyze the complexity of the transformation and present some performance figures on the compilation process and on the generated constraint programming code.

1 Introduction

Constraint programming is a programming paradigm which relies on two components: a constraint component which manages posting and checking satisfiability and entailment of constraints over some fixed computational domain, and a programming component which assembles the constraints of a given problem and expresses search procedures. To make constraint programming easier to use by non-programmers, a lot of work has been devoted to the design of front-end modeling languages using logical and algebraic notations instead of programming constructs for assembling constraints, e.g. OPL [10, 6], Zinc [8, 2], Essence [5] and Rules2CP [3, 4]. For efficiency reasons however, the search procedure needs be controlled and this part of programming can hardly be eliminated.

For instance, let us consider the following Comet model of the Bridge disjunctive scheduling problem [10] p. 209:

```
(...)
tuple Disjunction {Task first; Task second;}
int maxDuration = sum(t in Task) duration[t];
Scheduler<CP> cp(maxDuration);
Activity<CP> a[t in Task](cp, duration[t]);
UnaryResource<CP> tool[Resource](cp);
```

```
minimize<cp>
   a[stop].start()
subject to {
   forall(t in precedences)
      a[t.before].precedes(a[t.after]);
   forall(t in max_nf)
      cp.post(a[t.before].end() + t.dist >= a[t.after].start());
   forall(t in max_ef)
      cp.post(a[t.before].end() + t.dist >= a[t.after].end());
   forall(t in min_af)
      cp.post(a[t.before].start() + t.dist <= a[t.after].start());
   forall(t in min_sf)
      cp.post(a[t.before].start() + t.dist >= a[t.after].end());
   forall(t in min_nf)
      cp.post(a[t.before].end() + t.dist <= a[t.after].start());
   forall(r in Resource)
      forall(t in res[r])
        a[t].requires(tool[r]);
} using {
   forall(r in Resource)
      tool[r].rank();
   label(a[stop]);
}
```

The general strategy is to post first the (deterministic) precedence, distance and resource requirement constraints, and then to rank each unary resource by finding (non-deterministically) a total ordering of all tasks requiring the resource. Once the resources are ranked, the minimal starting dates of the activities provide a solution. Choosing which resource or task to rank next may be important for the size of the explored search tree and hence the efficiency of the search procedure. For instance, we may want to select the tasks by decreasing durations and discriminate equalities by preferring pairs of tasks which have the least domain lower bounds. To achieve that in Comet, some form of programming involving creation of intermediate arrays or sets, or changes of data-structures, is necessary:

```
using {
   forall(d in make_disjunct(res)) by (-duration[d.first] - duration[d.second],
                                       a[d.first].start().getMin() +
                                       a[d.second].start().getMin()) {
      try<cp>
             { a[d.first].precedes(a[d.second]); }
          |
             { a[d.second].precedes(a[d.first]); }
   }
   label(cp);
}
```

```
function set{Disjunction} make_disjunct(set{Task}[] res) {
   set{Disjunction} disj = {};
   forall(r in Resource)
      forall(t1 in res[r])
         forall(t2 in res[r]: t1 < t2)
            disj.insert(Disjunction(t1, t2));
   return disj;
}
```

We show that in a rule-based modeling language, search tree ordering heuristics can be specified declaratively by ordering criteria using pattern matching on the tree structure. These criteria apply to conjunctions and disjunctions of constraints involved in the search tree.

In the particular case of labeling, criteria on conjunctions encode variable ordering (*e.g.*, first-fail: among the operands of a conjunction, choose the variable with the smallest domain first) and criteria on disjunction encode value ordering (*e.g.*, middle-out: among the operands of a disjunction of assignments, choose the most centered value first). Ordering heuristics are generalized here to search trees with arbitrary nesting of conjunctions and disjunctions. Heuristics are driven by criteria to estimate which node to consider first. Every rule definition is of the form $p(\boldsymbol{x}) = e$, associating the left-hand side $p(\boldsymbol{x})$ to the right-hand side e. Patterns guarding criteria are matched against $p(\boldsymbol{x})$ which gives the scores used to reorder the subtree.

We consider a new version of the Rules2CP modeling language [3], called Cream (Constraints with Rules to EAse Modeling). The previous example can be written in Cream as follows:

```
(...)
end(T) = T:start + T:duration.
maxDuration = sum(map(T in tasks, T:duration)).
tasks_domain = domain(tasks, 0, maxDuration).
precedes(T1, T2) = end(T1) =< T2:start.
disjuncts(T1, T2) = precedes(T1, T2) or precedes(T2, T1).
precedences =
   forall(TaskPair in precedences_list,
      precedes(nth(1, TaskPair), nth(2, TaskPair))))).
distances =
   forall(T in max_nf_list, max_nf(nth(1, T), nth(2, T), nth(3, T))) and
   forall(T in min_sf_list, min_sf(nth(1, T), nth(2, T), nth(3, T))) and
   forall(T in max_ef_list, max_ef(nth(1, T), nth(2, T), nth(3, T))) and
   forall(T in min_nf_list, min_nf(nth(1, T), nth(2, T), nth(3, T))) and
   forall(T in min_af_list, min_af(nth(1, T), nth(2, T), nth(3, T))).
disjunctives =
   forall(Task in resource,
      forall(T1 in Task,
         forall(T2 in Task,
            T1:uid < T2:uid implies disjuncts(T1, T2)))).
```

```
? tasks_domain and precedences and distances and
  conjunct_ordering([
     greatest(T1:duration + T2:duration for disjuncts(T1, T2)),
     least(dmin(T1:start) + dmin(T2:start) for disjuncts(T1, T2))
  ]) and
  minimize(disjunctives, stop:start).
```

In Cream, heuristics on conjunctive or disjunctive formulae are stated declaratively by a vector of criteria:

$$[order_1(expression_1 \texttt{ for } pattern_1), \ldots, order_n(expression_n \texttt{ for } pattern_n]$$

where $\forall i \in \{1, \ldots, n\}$, $order_i \in \{\texttt{least}, \texttt{greatest}\}$, $expression_i$ is any Cream expression evaluating to an integer with variables bound to the arguments of $pattern_i$. Criteria give a vector of scores for each node of the search tree and every conjunction or disjunction group occurring in the search tree is reordered lexicographically according to these scores. Given a node v in the search tree, criteria are evaluated as follows: if v results from a call to a rule whose left-hand side can be matched against $pattern_i$, then the ith component of the associated vector of scores is given by $expression_i$. Otherwise, the ith component is assigned to the lowest possible value. As opposed to Rules2CP, Cream handles dynamic ordering criteria, *i.e.* criteria involving finite domain variables. In the above example, the second conjunctive criterion involves the domain lower bound of task starting date. Assume now that we want also to order disjunctions, for instance by trying to schedule the task with the greatest duration first. That composition of ordering heuristics would be difficult to program in Comet but simply consists in Cream in stating a disjunctive criterion as follows:

```
  disjunct_ordering([
     greatest(T1:duration for precedes(T1, T2))
  ])
```

Whereas *ad-hoc* data structures have to be explicited by the user in other modeling languages, the rule-based nature of Cream implicitely gives a structure to the search tree and allows the user to express directly heuristics by pattern matching.

Cream's rules are similar to Zinc's definitions and the pattern-matching language we define here to specify search tree ordering heuristics should be applicable to Zinc as well.

The price to pay for this ease of modeling is in the compilation process which we describe here with a formal system. We analyze the complexity of the transformation and present some performance figures on the compilation process and on the generated constraint programming code.

The rest of the paper is organized as follows. The next section defines the syntax of Cream. Section 3 presents the transformation of Cream models in constraint programs using a formal system to prove the correctness of the transformation. Section 4 evaluates the performances of our Cream compiler and of the generated code on a benchmark of scheduling and bin packing problems.

2 Cream Syntax

The language defined in this section covers the whole set of constructions for search trees and heuristics specifications in Cream. In the actual implementation, a layer of syntactic sugar offers facilities such as the separation between heuristic specifications and search tree constructions so as to let the programmer reuse heuristics throughout searches.

We suppose an infinite set $\mathcal{N}$ of labels for record fields, an infinite set $\mathcal{F}$ of predicates for user-defined rules, and a set $\mathcal{C}$ of predicates for constraints (which should map the underlying constraint theory).

A Cream program is a sequence of definitions ended by a query. $X^\star$ denotes a possibly empty sequence of X, delimited by commas.

$$\begin{array}{rll} \mathcal{P} ::= & \mathcal{D}\ \mathcal{P} \mid \mathcal{Q} & \textit{(program)} \\ \mathcal{D} ::= & \mathcal{F}\texttt{(}\mathcal{V}^\star\texttt{)} \texttt{ = } \mathcal{E}\texttt{.} & \textit{(rule definition)} \\ \mathcal{Q} ::= & \texttt{?}\ \mathcal{E}\texttt{.} & \textit{(query)} \end{array}$$

The central elements of the syntax are expressions $\mathcal{E}$, which cover definition calls, constraints, values $\mathcal{T}$ and search-tree directives $\mathcal{S}$ over an infinite set of variables $\mathcal{V}$. Operators follow usual priority rules.

$$\begin{array}{rll} \mathcal{E} ::= & \mathcal{V} \mid \mathcal{T} \mid \mathcal{F}\texttt{(}\mathcal{E}^\star\texttt{)} \mid \mathcal{C}\texttt{(}\mathcal{E}^\star\texttt{)} \mid \mathcal{S} & \textit{(expression)} \\ & \mid \mathcal{E} \texttt{ and } \mathcal{E} \mid \mathcal{E} \texttt{ or } \mathcal{E} \mid \mathcal{E} \texttt{ implies } \mathcal{E} & \\ & \mid \texttt{let(}\mathcal{V} \texttt{ = } \mathcal{E} \texttt{ in } \mathcal{E}\texttt{)} \mid \texttt{minimize(}\mathcal{V} \texttt{ in } \mathcal{E}\texttt{)} & \end{array}$$

Cream values are numbers, lists and records. Numbers come with basic arithmetics and operators for indexicals over finite domains. `bottom` is $-\infty$. Lists are either constructed by extension, concatenation or as discrete interval between two numerical expressions. Lists can be reversed or projected to an element at a given position. Records can be projected to any fixed field.

$$\begin{array}{rll} \mathcal{T} ::= & \langle\, \textit{number} \rangle \mid \mathcal{E} \,\langle\, \textit{arithop} \rangle\, \mathcal{E} \text{ where } \langle\, \textit{arithop} \rangle \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{/}\} & \textit{(value)} \\ & \mid \texttt{bottom} \mid \texttt{dsize(}\mathcal{E}\texttt{)} \mid \texttt{dmin(}\mathcal{E}\texttt{)} \mid \texttt{dmax(}\mathcal{E}\texttt{)} & \\ & \mid \texttt{[}\mathcal{E}^\star\texttt{]} \mid \mathcal{E} \cdot \mathcal{E} \mid \mathcal{E} \texttt{..} \mathcal{E} \mid \texttt{nth(}\mathcal{E}\texttt{, } \mathcal{E}\texttt{)} & \\ & \mid \{\, (\mathcal{N}\texttt{=}\mathcal{E})^\star\} \mid \mathcal{E} : \mathcal{N} & \end{array}$$

Search strategies are defined through criteria, with respect to which search trees are reordered. Search trees are either constructed in extension (boolean operators of $\mathcal{E}$ give the $\wedge$ and $\vee$ nodes), or in intension by iterating over list values with `fold`. The parameters of a `fold` are respectively: a binary associative operator (which can be any user-defined predicate, with some fixed initial arguments), the neutral element of this operator, a variable, a list giving the values for the variable, and an expression depending on the variable.

$$\begin{array}{rll}\mathcal{S} ::= & \texttt{search(}\mathcal{E}\texttt{,disjunct(}\mathcal{O}^\star\texttt{),conjunct(}\mathcal{O}^\star\texttt{))} & \textit{(search)} \\ & \mid \texttt{fold(}\mathcal{F}\texttt{(}\mathcal{V}^\star\texttt{), }\mathcal{E}\texttt{, }\mathcal{V}\texttt{ in }\mathcal{E}\texttt{, }\mathcal{E}\texttt{)} & \end{array}$$

Each criterion applies on the sub-nodes of a definition distinguished by its left-hand side predicate. The score associated to the criterion is an arithmetic expression which possibly depends on the arguments of the definition. In the concrete Cream syntax, `greater` or `least` operators modify the sign of the expression.

$$\mathcal{O} ::= \quad \mathcal{E}\texttt{ for }\mathcal{F}\texttt{(}\mathcal{V}^\star\texttt{)} \qquad\qquad \textit{(criterion)}$$

Constraints $\mathcal{C}$ are plunged into values through reification. Reciprocally, values 1 and 0 are interpreted as the constraints true and false respectively. The use of other values as constraints is rejected.

Cream comes with a standard library of rule definitions. Here is an excerpt of those definitions:

```
domain(X, Lo, Hi) = Lo ≤ X and X ≤ Hi.
         and(E, A) = E and A.
          or(E, A) = E or A.
        cons(E, A) = [E] · A.
```

`exists`, `forall` and `map` are defined as syntactic sugar over `fold`:

$$\begin{array}{rl}\texttt{exists(}X\texttt{ in }L\texttt{, }E\texttt{)} & \equiv \texttt{fold(or, false, }X\texttt{ in }L\texttt{, }E\texttt{)} \\ \texttt{forall(}X\texttt{ in }L\texttt{, }E\texttt{)} & \equiv \texttt{fold(and, true, }X\texttt{ in }L\texttt{, }E\texttt{)} \\ \texttt{map(}X\texttt{ in }L\texttt{, }E\texttt{)} & \equiv \texttt{fold(cons, [], }X\texttt{ in }L\texttt{, }E\texttt{)}\end{array}$$

To illustrate Cream compilation, we will consider two rule definitions that constrain the shape of objects for a simple two-dimensional placement problem of thin sticks which can be either short (from 1 to 5 units), normal (from 11 to 15 units) or long (from 21 to 25 units). A stick is a 1-unit wide rectangle which can be either horizontal or vertical.

```
   shape_constraint(O) = exists( S, [1, 11 , 21],
                                      shape_stick(O, S, S + 4)).
shape_stick(O, Min, Max) = domain(O:w, Min, Max) and O:h = 1
                         or domain(O:h, Min, Max) and O:w = 1.
```

The compilation scheme for `fold` described in the next section transforms the expression `shape_constraint(S)` into a code computing the same answers as the following unfolded expression:

```
((1≤S:w and S:w≤1+4) and S:h=1) or ((1≤S:h and S:h≤1+4) and S:w=1)
  or (((11≤S:w and S:w≤11+4) and S:h=1) or ((11≤S:h and S:h≤11+4) and S:w=1)
    or (((21≤S:w and S:w≤21+4) and S:h=1) or ((21≤S:h and S:h≤21+4) and S:w=1)
      or false)).
```
(1)

Recursion is prohibited in Cream: there should exist a topological order for dependency among rule definitions. A definition d depends on another definition e if d explicitly calls e (construction $\mathcal{F}\texttt{(}\mathcal{E}\texttt{)}$) or if e is used as a `fold` operator in d.

3 Cream Compilation

Compilation is defined as two transformations which produce intermediary code: $[\![\cdot]\!]^d$ expands a query to the deterministic code which adds the constraints and make calls to dynamic search parts. These search parts are transformed by $[\![\cdot]\!]^s$ to the non-deterministic code which handles reordering and searching. Intermediary code follows the syntax of Cream programs, but without the recursion restriction. In the intermediary code, there is no search-tree directives $\mathcal{S}$ (they are reformulated by $[\![\cdot]\!]^s$), operators `or` represent either reified $\vee$-constraints the deterministic code, or choice-points in the non-deterministic code. The syntactic construction $\texttt{delay(}p\texttt{(}\boldsymbol{X}\texttt{))}$ is introduced in intermediary code to denote the symbolic term $p\texttt{(}\boldsymbol{X}\texttt{)}$ (as opposed to a call to the definition $p\texttt{(}\boldsymbol{X}\texttt{)}$). Such an intermediary code is then straightforward to translate to a Prolog or Java program.

3.1 Transformation of the query to deterministic code

$[\![\cdot]\!]^d(\boldsymbol{V})$ reformulates search directives inductively over the structure of Cream expressions as follows. $\boldsymbol{V}$ is supposed to contain all the free variables appearing in the expression: $\boldsymbol{V}$ is used to pass the context to auxiliary definitions introduced by the translation.

Each definition $p\texttt{(}\boldsymbol{X}\texttt{)} \texttt{ = } e$ is translated in the intermediary code to the definition: $p_d\texttt{(}\boldsymbol{X}\texttt{)} \texttt{ = } [\![e]\!]^d(\text{fv}(e))$. The translation of calls follows then directly: $[\![p\texttt{(}\boldsymbol{X}\texttt{)}]\!]^d(\boldsymbol{V}) = p_d\texttt{(}\boldsymbol{X}\texttt{)}$.

Search directives rely on the search transformation (defined in the section 3.2).

$$[\![\texttt{search(}e\texttt{,disjunct(}\boldsymbol{o}_\vee\texttt{),conjunct(}\boldsymbol{o}_\wedge\texttt{))}]\!]^d(\boldsymbol{V}) = [\![e]\!]^s[\boldsymbol{o}_\wedge, \boldsymbol{o}_\vee](\boldsymbol{V})$$

Recursive predicates iterating on lists are generated for each `fold`:

$$[\![\texttt{fold(}p\texttt{(}\overrightarrow{e_i}\texttt{), }n\texttt{, }X\texttt{ in }l\texttt{, }e\texttt{)}]\!]^d(\boldsymbol{V}) = q\texttt{(}[\![l]\!]^d(\boldsymbol{V})\texttt{, }\overrightarrow{[\![e_i]\!]^d(\boldsymbol{V})}\texttt{, }\boldsymbol{V}\texttt{)}$$

with q a new predicate symbol described by the following definitions, where all variables are fresh with respect to $\boldsymbol{V}$:

$$q\texttt{([], }\overrightarrow{\texttt{E}_i}\texttt{, }\boldsymbol{V}\texttt{) = }[\![n]\!]^d(\boldsymbol{V})\texttt{.}$$
$$q\texttt{([H | T], }\overrightarrow{\texttt{E}_i}\texttt{, }\boldsymbol{V}\texttt{) = }p_d\texttt{(}\overrightarrow{\texttt{E}_i}\texttt{, }[\![e[\texttt{H/X}]]\!]^d(\boldsymbol{V})\texttt{, }q\texttt{(T, }\overrightarrow{\texttt{E}_i}\texttt{, }\boldsymbol{V}\texttt{)).}$$

Other cases for $[\![\cdot]\!]^d$ are defined homomorphically with respect to sub-expressions, taking care of scopes and name clashes: e.g.,

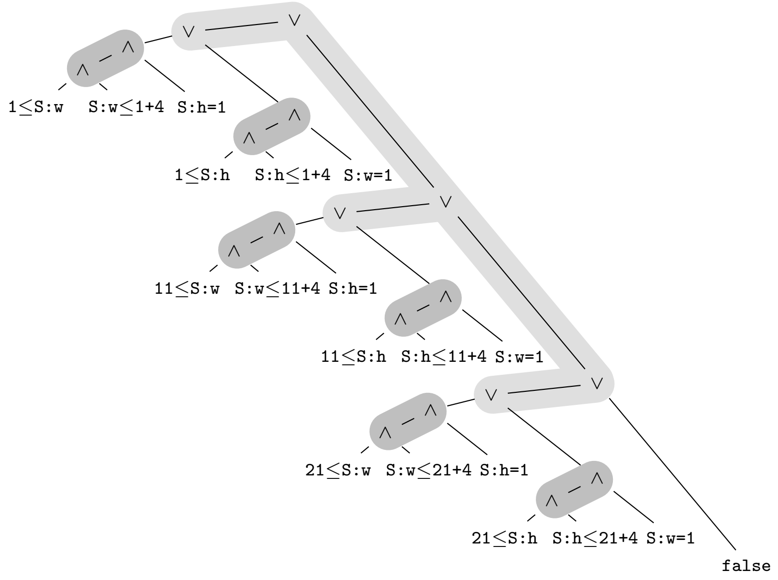
$$[\![\texttt{let(}v\texttt{ = }e\texttt{ in e')}]\!]^d(\boldsymbol{V}) = \texttt{let(}X\texttt{ = }[\![e]\!]^d(\boldsymbol{V})\texttt{ in }[\![e'[X/v]]\!]^d(\boldsymbol{V}\cdot X))$$

where X is a fresh variable.

3.2 Transformation of the search to non-deterministic code

Search-strategy compilation relies on the notion of *O-layers* of $\wedge/\vee$-trees: for $O \in \{\wedge, \vee\}$, we call *O-layer* of an $\wedge/\vee$-tree any maximal tree sub-graph with either only $\wedge$-nodes or only $\vee$-nodes. It is worth noticing that an O-layer is a tree but not necessarily a sub-tree of the considered $\wedge/\vee$-tree: branches are cut on operator changes.

The following $\wedge/\vee$-tree corresponds to the expression (1) given in the previous section, where layers have been circled:



The definition of O-layers is generalized for Cream expression syntax trees by letting layers go through let-bindings, definition calls, in the right-hand side of `implies` and through the tree intentionnaly constructed by `fold`. The children of a layer are children of a node in the layer without being themselve in the layer. The *root O-layer* is the O-layer containing the root node if it is not the dual of O, or the empty layer otherwise. By convention, the root node is the (only) child of the empty layer. Tree reordering is applied between all child nodes of each O-layer: criteria defined for $O \in \{\wedge, \vee\}$ associate a vector of scores to each child, and children are reordered according to their scores lexicographically (the score returned by the first criterion for O is considered first, then, in case of equality, the score of the second criterion for O, and so on).

Neither the tree (due to `fold` over arbitrary lists) nor the scores (due to indexicals) are supposed to be completely known at compile-time. Therefore, the transformation generates code for computing the reordering at execution-time rather than computing the reordering statically.

For a fixed pair of criteria $(\boldsymbol{o}_\wedge, \boldsymbol{o}_\vee)$, $[\![\cdot]\!]^s(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V})$ produces code which reorders the root O-layer of the tree and explores its children sequentially. $\boldsymbol{c}_\wedge$ and $\boldsymbol{c}_\vee$ are current score vectors (they have the same dimension than $\boldsymbol{o}_\wedge$ and

$\boldsymbol{o}_\vee$ respectively). Initially, scores are $\boldsymbol{c}_\wedge^{-\infty}$ and $\boldsymbol{c}_\vee^{-\infty}$, defined as vectors, where every component equals to `bottom`, since no criteria apply outside any definition. $[\![\cdot]\!]^s(\boldsymbol{V})$ is arbitrarily defined as $[\![\cdot]\!]^s_\wedge(\boldsymbol{c}_\wedge^{-\infty}, \boldsymbol{c}_\vee^{-\infty}, \boldsymbol{V})$ to initiate the transformation (the root layer, possibly empty, can always be considered as being an $\wedge$-layer). $[\![\cdot]\!]^s_O$ relies on the auxiliary transformation $[\![\cdot]\!]^l_O(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V})$ which produces code computing an associative list: this list contains an association for each child node of the O-layer, keys are score vectors and values are predicates to call to explore children recursively.

$$[\![e]\!]^s_O(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V}) = \texttt{iter\_predicates}_O\texttt{(}[\![e]\!]^l_O(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V})\texttt{)}$$

where $\texttt{iter\_predicates}_O\texttt{(}L\texttt{)}$ is an internal function which iteratively selects the item of L which has the best score, executes the associated definition, then consider the other items recursively, either in conjunction or in disjunction, according to O.

Definitions and calls For each definition $p\texttt{(}\boldsymbol{X}\texttt{) = }e$, the compilation produces two definitions in the intermediary code, one for each kind of layer:

$$\begin{aligned} p_\wedge\texttt{(}\boldsymbol{C}_\wedge\texttt{, }\boldsymbol{C}_\vee\texttt{, }\boldsymbol{V}\texttt{) = } & [\![e]\!]^s_\wedge(u(\boldsymbol{C}_\wedge, \boldsymbol{o}_\wedge, p\texttt{(}\boldsymbol{X}\texttt{)}), \boldsymbol{C}_\vee, \mathrm{fv}(e)) \\ p_\vee\texttt{(}\boldsymbol{C}_\wedge\texttt{, }\boldsymbol{C}_\vee\texttt{, }\boldsymbol{V}\texttt{) = } & [\![e]\!]^s_\vee(\boldsymbol{C}_\wedge, u(\boldsymbol{C}_\vee, \boldsymbol{o}_\vee, p\texttt{(}\boldsymbol{X}\texttt{)}), \mathrm{fv}(e)) \end{aligned}$$

where the function $u(\boldsymbol{c}, \boldsymbol{o}, p\texttt{(}\boldsymbol{X}\texttt{)})$ calculates the score vector $\boldsymbol{c}'$, where components corresponding to criteria matching $p\texttt{(}\boldsymbol{X}\texttt{)}$ are updated:

$$u(\overrightarrow{c_i}, \overrightarrow{e_i\ \texttt{for}\ p_i\texttt{(}\boldsymbol{X}_i\texttt{)}}, p\texttt{(}\boldsymbol{X}\texttt{)}) = \overrightarrow{c'_i}$$

where:

$$c'_i = \begin{cases} \sigma(e_i) & \text{if } \exists\sigma, \sigma(p_i\texttt{(}\boldsymbol{X}_i\texttt{)}) = p\texttt{(}\boldsymbol{X}\texttt{)} \\ c_i & \text{otherwise} \end{cases}$$

Calls select one of these two definitions, depending on the kind of the current layer.

$$[\![p\texttt{(}\boldsymbol{X}\texttt{)}]\!]^l_O(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V}) = p_O\texttt{(}\boldsymbol{c}_\wedge\texttt{, }\boldsymbol{c}_\vee\texttt{, }\boldsymbol{X}\texttt{)}$$

Boolean operators $[\![\cdot]\!]^l_\wedge$ aggregates lists in the root $\wedge$-layer. A new predicate q is introduced for each child node of the $\wedge$-layer.

$$[\![e\ \texttt{and}\ e']\!]^l_\wedge(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V}) = [\![e]\!]^l_\wedge(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V}) \cdot [\![e']\!]^l_\wedge(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V})$$

$$\begin{aligned} & [\![e\ \texttt{or}\ e']\!]^l_\wedge(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V}) = \\ & \qquad \texttt{[\{ scores = }\boldsymbol{c}_\wedge\texttt{, predicate = delay(}q\texttt{(}\boldsymbol{c}_\wedge\texttt{, }\boldsymbol{c}_\vee\texttt{, }\boldsymbol{V}\texttt{))\}]} \end{aligned}$$

where q applies the transformation recursively to the sub-$\vee$-layer (all variables are fresh with respect to $\boldsymbol{V}$):

$$q\texttt{(}\boldsymbol{C}_\wedge\texttt{, }\boldsymbol{C}_\vee\texttt{, }\boldsymbol{V}\texttt{) = }[\![e \vee e']\!]^s_\vee(\boldsymbol{C}_\wedge, \boldsymbol{C}_\vee, \boldsymbol{V})\texttt{.}$$

Dual definitions hold for $[\![\cdot]\!]^l_\vee$.

Filtering

$$[\![e \texttt{ implies } e']\!]^l_O(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V}) = \texttt{filter(}\boldsymbol{c}_O\texttt{, } [\![e]\!]^d(\boldsymbol{V})\texttt{, } [\![e']\!]^l_O(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V})\texttt{)}$$

where, $\texttt{filter(}\boldsymbol{c}\texttt{, } e\texttt{, } e'\texttt{)}$ is an internal function which returns e' if e is true, and returns the singleton list `[{ scores =` $\boldsymbol{c}$`, predicate = delay(true) }]` otherwise.

Let-binding

$$[\![\texttt{let(}X \texttt{ = } e \texttt{ in } e'\texttt{)}]\!]^l_O(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V}) = \\ \texttt{let(}Y \texttt{ = } [\![e]\!]^d(\boldsymbol{V})\texttt{, } [\![e'[Y/X]]\!]^l_O(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V}\cdot Y)\texttt{)}$$

where Y is a fresh variable.

Aggregators Aggregators use a special source symbol, `rec`, to handle recursion. `fold` is translated to a call to a recursive predicate:

$$[\![\texttt{fold(}p(\overrightarrow{e_i})\texttt{, } n\texttt{, } X \texttt{ in } e\texttt{, } e'\texttt{)}]\!]^l_O(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V}) = q_O\texttt{(}[\![e]\!]^d\texttt{, } \overrightarrow{[\![e_i]\!]^d}\texttt{, } \boldsymbol{c}_\wedge\texttt{, } \boldsymbol{c}_\vee\texttt{, } \boldsymbol{V}\texttt{)}$$

where $q_\wedge$ and $q_\vee$ are new predicate symbols described by the following definitions (all variables are fresh with respect to $\boldsymbol{V}$):

$$q_O\texttt{([], }\overrightarrow{\texttt{E}_i}\texttt{, } \boldsymbol{C}_\wedge\texttt{, } \boldsymbol{C}_\vee\texttt{, } \boldsymbol{V}\texttt{) = } [\![n]\!]^l_O(\boldsymbol{C}_\wedge, \boldsymbol{C}_\vee, \boldsymbol{V})\texttt{.}$$

$$q_O\texttt{([H | T], }\overrightarrow{\texttt{E}_i}\texttt{, } \boldsymbol{C}_\wedge\texttt{, } \boldsymbol{C}_\vee\texttt{, } \boldsymbol{V}\texttt{) =} \\ [\![p_O\texttt{(}\overrightarrow{\texttt{E}_i}\texttt{, rec(}q\texttt{, T, }\overrightarrow{\texttt{E}_i}\texttt{, }\boldsymbol{V}\texttt{), } e'[\texttt{H}/\texttt{X}]\texttt{)}]\!]^l_O(\boldsymbol{C}_\wedge, \boldsymbol{C}_\vee, \boldsymbol{V}\cdot\texttt{H})\texttt{.}$$

and `rec` is translated to a recursive call to q:

$$[\![\texttt{rec(}q\texttt{, T, }\overrightarrow{\texttt{E}_i}\texttt{, }\boldsymbol{V}\texttt{)}]\!]^l_O(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V}) = q_O\texttt{(T, }\overrightarrow{\texttt{E}_i}\texttt{, } \boldsymbol{c}_\wedge\texttt{, } \boldsymbol{c}_\vee\texttt{, } \boldsymbol{V}\texttt{)}$$

Constraints and sub-search directives Constraints and sub-search directives are children of the layer, therefore the transformation produces singleton lists associating their score to a fresh predicate q.

$$[\![e]\!]^l_O(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee, \boldsymbol{V}) = \texttt{[\{ scores = }\boldsymbol{c}_O\texttt{, predicate = delay(}q(\boldsymbol{V})\texttt{)\}]}$$

where q applies the transformation recursively (all variables are fresh with respect to $\boldsymbol{V}$):

$$q\texttt{(}\boldsymbol{V}\texttt{) = } [\![e]\!]^d\texttt{.}$$

Property 1 *There are* $\mathbf{O}(d\cdot s)$ p_d*-,* $p_\vee$*- and* $p_\wedge$*-definitions in intermediary code, where* d *is the number of definitions in the Cream code and* s *is the number of* `search` *clauses. Each definition in the intermediary code, including auxiliary definitions for* `fold` *and sub-layers, has a size linear in the size of the original Cream definition. In particular, if there is one* `search` *clause, the intermediary code has a size linear in the size of the original Cream code. The complexity of the transformation is linear in the size of the generated code.*

Proof. $[\![\cdot]\!]^d$ and $[\![\cdot]\!]^l$ are inductive transformations where each step linearly composes results of sub-transformations, either in auxiliary definitions or in-place expressions. Therefore, there exists a multiplicative constant factor between the size of the generated definitions and the size of the original Cream definition. For each Cream definition $p(\boldsymbol{X})$, there is one definition p_d in the intermediary code, plus two definitions $p_\vee$ and $p_\wedge$ by `search` clauses.

This complexity result contrasts with Rules2CP transformation complexity[3] where definition unfolding leads to exponential code size in the worst case.

4 Evaluation

We report here performances comparisons of the Rules2CP and Cream compilers and generated constraint programs as well as a brief explanation for observed differences. Performances are measured on the Bridge Scheduling, Open-Shop Scheduling and Optimal Rectangle Packing problems.

The Bridge problem consists in finding a schedule, involving 46 tasks subject to precedence, distance and resource requirement constraints, that minimizes the time to build a five-segment bridge [10] p. 209.

The Open-Shop problem consists in finding the non-preemptive schedule with minimal completion time of a set J of n jobs, consisting each of m tasks, on a set M of m machines. The processing times are given by a $m \times n$-matrix P, in which $p_{ij} \geq 0$ is the processing time of task $T_{ij} \in T$ of job J_j to be done on machine M_i. The tasks of a job can be processed in any order, but only one at a time. Similarly, a machine can process only one task at a time. Here, the j6-4 ($n = m = 6$) and j7-1 ($n = m = 7$) Open-Shop problem instances (Brucker *et* al. [1]) are considered.

The Rectangle Packing problem consists in finding the smallest rectangle containing n squares of sizes $S_i = i$ for $1 \leq i \leq n$. The model is based on the proposal of Simonis and O'Sullivan [9] implemented in SICStus Prolog. They tackled the Korf's benchmark [7] and improved best known runtimes up to a factor of 300. We consider here for evaluation the $n = 22$ instance of the problem.

	Rules2CP		Cream	
	Compilation	Solving	Compilation	Solving
Bridge	0.360	0.150	0.200	0.370
Open-Shop j6-4	1.370	160	0.790	325
Open-Shop j7-1	2.150	1454	1.310	2327
Rectangle-Packing n22	0.490	284	0.490	284

Table 1. Rules2CP and Cream programs runtimes in seconds.

Table 1 compares the compilation and execution runtimes in seconds in Cream with those obtained in Rules2CP. In all scheduling problem instances, the same heuristics on disjunctive formulae with static criterion "schedule first the task that has the greatest duration" was used. The implementation of the Cream compiler is a proof of concept of the transformations presented in Sec. 3, and no effort has been made yet to improve performances.

On the one hand, Cream yields structured constraint programs including (recursive) clauses as a programmer would have written the model in Prolog. On the other hand, Rules2CP produces optimized flatten constraint programs by complete expansion of definitions and record projections with partial evaluation.

In the Rectangle Packing model, there is no heuristics stated, thus the same runtimes between Rules2CP and Cream.

When heuristics on formulae are involved, that is in the scheduling problems, the compilation in Cream is about twice faster than in Rules2CP because ordering is delayed to execution time and partial evaluation does not occur.

Solving runtimes of constraint programs generated by Cream are twice slower than those generated by Rules2CP. This overhead is explained by the following reasons: (a) in both Rules2CP and Cream, finite domain variables are global variables. But in constraint programs generated with Cream, they are handled by a backtrackable table associating names with actual variables. Whereas programs generated by Rules2CP does not need such a mechanism because of the complete expansion scheme; (b) In Rules2CP, partial evaluation at compile-time avoids the need of Prolog tests for handling logical implication as it is the case in programs generated with Cream; (c) record projections, finite domain arithmetic expressions computation, and goal calls in general are yet other sources of overhead. As we considered optimization problems, this aggregation of overheads for one call of the search goal is to multiply by the number of iterations of the branch and bound algorithm; (d) finally, priority queues could advantageously substitute for lists of pairs to enumerate children of layers.

It is worth noticing that these points are mainly implementation details and should be avoided in future work by an optimizing compiler.

5 Conclusion

A rule-based modeling language provides interesting features for specifying search tree and labeling ordering heuristics declaratively by pattern matching on rule left-hand sides' derivation. We have described our experience with Cream and Rules2CP modeling languages. While in Rules2CP the static expansion of the goals [3] was combined with calls to the Rules2CP interpreter for expressions depending on variable domains that can only be evaluated at runtime, in Cream, models are compiled in constraint programs that select the continuation for search according to the heuristic criteria, at runtime.

We have described the transformation of Cream models to constraint programs with a formal system and showed with a benchmark of examples that in the static case, the Cream overhead at runtime was limited *w*.r.t. the static expansion of Rules2CP.

This approach to specifying ordering heuristics by pattern matching should be applicable to other modeling languages that use definitions, such as Zinc [8, 2] for instance. A natural extension for future work is the specification of search procedures which are currently limited in Cream to depth-first backtracking and branch and bound search.

Acknowledgements. We acknowledge partial support from the European FP6 Strep project Net-WMS, and discussions with the partners of this project, and in our group with Sylvain Soliman and with Sunrinderjeet Singh who provided us with his Rules2CP models for Open-Shop.

References

1. Peter Brucker, Johann Hurink, Bernd Jurisch, and Birgit Wöstmann. A branch & bound algorithm for the open-shop problem. In *GO-II Meeting: Proceedings of the second international colloquium on Graphs and optimization*, pages 43–59, Amsterdam, The Netherlands, The Netherlands, 1997. Elsevier Science Publishers B. V.
2. Maria Garcia de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language Zinc. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming* CP*'06)*, pages 700–705. Springer-Verlag, 2006.
3. François Fages and Julien Martin. From rules to constraint programs with the Rules2CP modelling language. In *Recent Advances in Constraints, Revised Selected Papers of the 13th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2008*, Lecture Notes in Artificial Intelligence, pages 66–83. Springer-Verlag, 2008.
4. François Fages and Julien Martin. Modelling search strategies in Rules2CP. In *Proceedings of CPAIOR'09*, Lecture Notes in Computer Science. Springer-Verlag, 2009.
5. Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martinez-Hernandez, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.
6. Pascal Van Hentenryck, Laurent Perron, and Jean-Francois Puget. Search and strategies in opl. *ACM Transactions on Compututational Logic*, 1(2):285–320, 2000.
7. Richard E. Korf. Optimal rectangle packing: New results. In *ICAPS*, pages 142–149, 2004.
8. Reza Rafeh, Maria Garcia de la Banda, Kim Marriott, and Mark Wallace. From Zinc to design model. In *Proceedings of PADL'07*, pages 215–229. Springer-Verlag, 2007.
9. Helmut Simonis and Barry O'Sullivan. Using global constraints for rectangle packing. In *Proceedings of the first Workshop on Bin Packing and Placement Constraints BPPC'08, associated to CPAIOR'08*, May 2008.
10. Pascal Van Hentenryck. *The OPL Optimization programming Language.* MIT Press, 1999.