Procedural Code Generation *vs* Static Expansion in Modelling Languages for Constraint Programming

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. To make constraint programming easier to use by the non-programmers, a lot of work has been devoted to the design of front-end modelling languages using logical and algebraic notations instead of programming constructs. The transformation to an executable constraint program can be performed by fundamentally two compilation schemas: either by a static expansion of the model in a flat constraint satisfaction problem (e.g. Zinc, Rules2CP, Essence) or by generation of procedural code (e.g. OPL, Comet). In this paper, we compare both compilation schemas. For this, we consider the rule-based modelling language Rules2CP with its static exansion mechanism and describe with a formal system a new compilation schema which proceeds by generation of procedural code. We analyze the complexity of both compilation schemas, and present some performance figures of both the compilation process and the generated code on a benchmark of scheduling and bin packing problems.

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 makes it possible to state the constraints of a given problem and define a search procedure for solving it. To make constraint programming easier to use by non-programmers, a lot of work has been devoted to the design of front-end modelling languages using logical and algebraic notations instead of programming constructs, e.g. OPL[14,7], Comet [10] , Zinc [11,3], Essence [6] or Rules2CP [4,5,2].

Such modelling languages for constraint programming offer a high-level of abstraction for stating constraint problems, and rely on default, possibly parameterized or adaptive, search strategies. The transformation to an executable constraint program can be performed by fundamentally two compilation schemas: either by a static expansion of the model in a flat constraint satisfaction problem, or by generation of procedural code. The first schema by static expansion

has been adopted by Zinc, Essence and Rules2CP, while the second schema by code generation has been implemented for OPL and Comet.

In this paper, we compare both compilation schemas. For this, we consider the rule-based modelling language Rules2CP with its static exansion mechanism described in [4], and introduce a new compilation schema which proceeds by generation of procedural code. With this new implementation, called Cream, we show that the code generation schema exhibits a time overhead of approximatively a factor 2 at runtime w.r.t. the statically expanded code. However, we show that the size of the procedural code is linear, which must be compared to the potentially exponential size of the expanded code. In particular, for problems where the search space is defined dynamically by values of variables at runtime, the code generation schema is the only viable one.

Furthermore, in a rule-based modelling language such as Rules2CP, the search tree is represented by a logical formula and search tree ordering heuristics can be expressed declaratively by pattern-matching on the rules' left-hand sides [5]. Compared with other modelling languages capable of expressing search heuristics, such as OPL/Comet for instance, rule-based pattern matching eliminates the need to program with lists and indices and to introduce data structures for defining the ordering criteria. Compared with Zinc, this mechanism provides a possible mean to define heuristics for the default search procedure. The price to pay for this expressivity however is in the compilation process which becomes more complicated. This was our original motivation for defining the transformations with a formal system.

The rest of the paper is organized as follows. The next section defines the syntax of Rules2CP, its polymorphic type system and the declarative semantics of the language. Section 3 defines the static expansion schema with a formal system that is reused in section 4 to define the code generation schema and to analyze their complexity. Section 5 evaluates the performance of both compilation schema and generated code on a benchmark of n-queens, scheduling and bin packing problems. Finally we conclude on the merits of each compilation schema.

2 Rules2CP Syntax and Declarative Semantics

2.1 Syntax

There are four data structures in Rules2CP:

- integer constants, with basic arithmetic operators and comparisons.
- finite domain variables, with indexicals and the equality constraint in addition to the operators shared with integer constants.
- lists, constructed by enumeration, interval between two integers and concatenation, and browsed with quantifiers and aggregators.
- records, with labeled fields used for projection.

The Rules2CP syntax is summarized in table 1. The non-terminal *variable* and *ident* range over a countable set of names. The non-terminal *integer* ranges

over a finite interval $\mathcal{D} \subseteq \mathbf{N}$ which includes at least the values 0 and 1. Underlined non-terminal $\underline{var}$ mark the binders which affect the underlined $\underline{expr}$.

The sets of bound and free variables in an expression e, denoted $\mathrm{bv}(e)$ and $\mathrm{fv}(e)$ respectively, are defined in the standard way: a variable is bound if it is in the scope of a binder (let or foldl) or if it appears in the left-hand side of a clause. Any assignment $\nu : var \to \mathcal{D}$ is homomorphically extended to a function $\tilde{\nu} : expr \to expr$.

program ::= *clause* ... *clause*
clause ::= `domain` *ident* `:=` { *ident*`,`...`,`*ident* }
| `object` *ident*`(`$\underline{var}$`,`...`,`$\underline{var}$`)` `:=` $\underline{expr}$
| `rule` *ident*`(`$\underline{var}$`,`...`,`$\underline{var}$`)` `:=` $\underline{expr}$
| `heuristics` *ident*`(`$\underline{var}$`,`...`,`$\underline{var}$`)` `:=` $\underline{heuristics}$
| `query` *expr*
expr ::= *variable* | *integer* | `error`
| *expr* ***op*** *expr* where ***op*** $\in$ {`+`, `-`, `*`, `/`}
| *expr* ***rel*** *expr* where ***rel*** $\in$ {`=`, `#`, `=<`, `<`, `>`, `>=`}
| *expr* ***logop*** *expr* where ***logop*** $\in$ {`and`, `or`, `implies`, `equiv`}
| `not` *expr*
| *ident*`(`*expr*`,`...`,`*expr*`)`
| `let(`$\underline{var}$ `:=` *expr*`,` $\underline{expr}$`)`
| `[`*expr*`,`...`,`*expr*`]` | `[`*expr* `..` *expr*`]` | *expr* `++` *expr*
| `length(`*expr*`)` | `nth(`*expr*`,` *expr*`)`
| {*ident*`:` *expr*`,`...`,`*ident*`:` *expr*} | *expr*`:`*ident*
| `foldl(`$\underline{var}$ `from` *expr*`,`$\underline{var}$ `in` *expr*`,`$\underline{expr}$`)`
| `minimize(`*expr*`,`*expr*`)` | `maximize(`*expr*`,`*expr*`)`
| `search(`*heuristics*`,`*expr*`)` | `constraint(`*expr*`)`
| `dynamic(`*expr*`)` | `static(`*expr*`)`
heuristics ::= `conjunctive(`$\underline{expr}$ `for` *ident*`(`$\underline{var}$`,`...`,`$\underline{var}$`))`
| `disjunctive(`$\underline{expr}$ `for` *ident*`(`$\underline{var}$`,`...`,`$\underline{var}$`))`
| *ident*`(`*expr*`,`...`,`*expr*`)`
| `nil` | *heuristics* `and` *heuristics*

Table 1. Rules2CP syntax

Free variables are not allowed in rule definitions. Free variables in object definitions are allowed and denote finite-domain variables. They are indexed by the head of the definition. For instance, the following definition introduces a new finite-domain variable in the field `row` for each value of `I`.

```
object queen(I) := { row : _, column : I }.
```

The concrete Rules2CP implementation introduces some syntactic sugar:

- the let-construction is recursively extended for multiple bindings. For all n, the let of $n+1$ bindings `let(`X_0`:=`e_0`,`...`,`X_n`:=`e_n`,`e`)` is defined with the simple let and the let of n bindings: `let(`X_0`:=`e_0`,let(`X_1`:=`e_1`,`...`,`X_n`:=`e_n`,`e`))`.

- `forall(`X `in` l`,` e`)` is a synonym for `foldl(`A `from 1,` X `in` l`,` A `and` e`)` where A is a fresh variable.
- `exists(`X `in` l`,` e`)` is `foldl(`A `from 0,` X `in` l`,` A `or` e`)` where A is a fresh variable.
- `map(`X `in` l`,` e`)` is `foldl(`A `from [],` X `in` l`,` A `++ [`e`])` where A is a fresh variable.
- `reverse(`l`)` is `foldl(`A `from [],` X `in` l`, [`X`] ++` A`)` where A and X are fresh variables.
- `foldr(`A `from` i`,` X `in` l`,` e`)` is `foldl(`A `from` i`,`X `in reverse(`l`),` e`)` where A is a fresh variable.

Example 1. The classical (unavoidable) n-queens problem can be modelled in Rules2CP as follows. First, the board of queens can be defined by the following object definitions:

```
object queen(I) := { row : _, column : I }.
object board(N) := map(I in [1 .. N], queen(I)).
```

For each integer `I`, `queen(I)` defines one record representing the queen in column `I`. Then, the goal is to post the constraints over the list of queens `B` and assign values to the free variables.

```
? let(N := 4, B := board(N),
    queens_constraints(B, N) and
    queens_labeling(variables(B), N)).
```

The predicate `queens_constraints` is defined by the following rules.

```
rule queens_constraints(B, N) := domain(B, 1, N) and safe(B).

rule safe(L) :=
  all_different(L) and
  forall(Q in L, forall(R in L,
    let(I := Q:column, J := R:column,
      I < J implies
      Q:row # J - I + R:row and
      Q:row # I - J + R:row))).
```

The rule `safe(L)` ensures that every queen in the list `L` is on a safe position: the global constraint `all_different` prevents row attacks and simple binary difference constraints prevents diagonal attacks.

The rule for `queens_labeling(Vars, N)` defines the search through a logical formula which induces a basic labeling search tree on variables `Vars`.

```
rule queens_labeling(Vars, N) :=
  search(mo(N), forall(Var in Vars, queens_labeling_var(Var, N))).

rule queens_labeling_var(Var) :=
  exists(Val in [1 .. N], queens_labeling_val(Var, Val)).

rule queens_labeling_val(Var, Val) := Var = Val.
```

The parameter `mo(N)` for search refers to the *middle out* heuristics which is defined by the following rule:

```
heuristics mo(N) :=
  disjunctive(least(abs(N/2 - Val))
              for queens_labeling_val(Var, Val)).
```

This statement specifies that the disjunctive formulae derived from `queens_labeling_val` must be ordered by increasing value of `abs(N/2 - Val)` (middle out ordering of values).

2.2 Type System

Rules2CP integrates a type system with five type constructors:

- `int` for integer values.
- `fd` for finite domain variables.
- `constraint` for first-order logic formulas.
- `[`τ`]` for (homogeneous) lists whose elements have type τ.
- `{`$\mathtt{f}_1$ `::` τ_1`, ...,` $\mathtt{f}_n$`::` τ_n`}` for records with the fields $\mathtt{f}_1$, ..., $\mathtt{f}_n$ carrying values of type τ_1, ..., τ_n respectively.

A free variable in a Rules2CP program is always an FD variable. The boolean values true and false are not distinguished from the integers 1 and 0.

The type system enjoys a type inference algorithm *à la* Hindley-Milner: typing rules are driven by the syntax of the expression and induce type equality constraints solved by unification. Type schemes with universal quantification are given to polymorphic definitions and rules where arguments are not completely specified.

Arithmetic operators and comparisons are overloaded to deal with both integer values and FD variables. For example, the addition operator is typed `int + int :: int` if both arguments are known to be of type `int`, otherwise it is typed `fd + fd :: fd`. It is worth noting that Hindley-Milner does not allow ad-hoc overloading in general. Here we made the choice to use `int` to follow statically known integer values and `fd` for model variables. Some constructions are specific to integer values: in particular, the list interval constructor has type `[int .. int] :: [int]`. Indexical built-ins transform FD variables into integer values: `min(fd) :: int` and `max(fd) :: int`.

Records are typed with *row types* [12], and two records are equal when they have the same set of fields and when fields of the same name carry values of equal types.

Example 2. Let us consider the two following rules defining the `area` and `volume` of an object. The argument `X` is only accessed by projection and can be of any record type containing at least the fields width and height (and depth for volume).

```
area(X) = X:width * X:height
volume(X) = area(X) * X:depth
```

In the inferred type,

```
area({ height: fd, width: fd, A }) :: fd
volume({ depth: fd, height: fd, width: fd, A }) :: fd
```

the unknown other fields are symbolized by a *row variable*. Such a row type containing a row variable is said to be open. Row types without row variables are closed.

In the following `shape` object definition

```
shape(Id) = { id: Id, width = _, height = _ }
```

the parameter `Id` can be of any type. The type inferred for `shape` is polymorphic and parameterized by a *type variable* `A` given to the argument `Id`. The other arguments are model variables (free variables at the right-hand side of a definition) and are therefore typed with `fd`.

```
shape(A) :: { id: A, width: fd, height: fd }
```

The Hindley-Milner type inference with row types is known to be decidable with a theoretical PSPACE-hard time complexity [8]. However, this worst-case time complexity does not exhibit in practice and the type inference algorithm is very efficient.

2.3 Declarative Semantics

Let $\mathcal{M}$ be a Rules2CP model. Let $O(\mathcal{M})$ be the set of all the objects of $\mathcal{M}$, let $R(\mathcal{M})$ be the set of all the rules, and $Q(\mathcal{M})$ be the set of all the queries of $\mathcal{M}$. Queries are interpreted conjunctively: the query associated to $\mathcal{M}$ is $q(\mathcal{M}) = \bigwedge_{q \in Q(\mathcal{M})} q$.

This section will characterize the solutions of the Rules2CP model $\mathcal{M}$. A solution is an assignment of all the free variables of $\mathcal{M}$ which satisfies all the constraints of $\mathcal{M}$. Free variables occurs in the query $q(\mathcal{M})$ and in object definitions. The free variables in object definitions are distinct for each instance of the object. The arguments of an object are restricted to belong to the following grammar.

$$\begin{aligned} \mathit{indexable} ::= {} & \mathit{integer} \\ \mid {} & \texttt{[}\mathit{indexable}\texttt{,}\ldots\texttt{,}\mathit{indexable}\texttt{]} \\ \mid {} & \{\mathit{ident}\texttt{:}\ \mathit{expr}\texttt{,}\ldots\texttt{,}\mathit{ident}\texttt{:}\ \mathit{expr}\}_{\mathit{uid}} \end{aligned}$$

Each indexable value v defines an index $id(v)$ which serves to index the free variables appearing in the object definition.

$$\begin{aligned} \mathit{id}: \quad & \mathit{indexable} \to \mathit{index} \\ & i \in \mathit{integer} \mapsto \texttt{constant(}i\texttt{)} \\ & \texttt{[}i_1\texttt{,}\ldots\texttt{,}i_n\texttt{]} \mapsto \texttt{[}\mathit{id}(i_1)\texttt{,}\ldots\texttt{,}\mathit{id}(i_n)\texttt{]} \\ & \{\mathit{ident}\texttt{:}\ \mathit{expr}\texttt{,}\ldots\texttt{,}\mathit{ident}\texttt{:}\ \mathit{expr}\}_{\mathit{uid}} \mapsto \texttt{uid(}\mathit{uid}\texttt{)} \end{aligned}$$

An assignment for $\mathcal{M}$ is a tuple (ν^Q, ν^O), where:

- $\nu^Q : \mathrm{fv}(q(\mathcal{M})) \to \mathcal{D}$
- ν^O is a family of assignments which maps every object $o \in O(\mathcal{M})$ and every tuple $(i_1, \ldots, i_n) \in \mathit{index}^n$, where n is the arity of the head of d, to an assignment $\nu^O_{o(i_1,\ldots,i_n)} : \mathrm{fv}(o) \to \mathcal{D}$

$$
\begin{array}{rl}
n \in \mathbf{N} \ \texttt{op}\ n' \in \mathbf{N} & \to n \mathit{\ op\ } n' \\
n \in \mathbf{N} \ \texttt{rel}\ n' \in \mathbf{N} & \to \delta(n \mathit{\ rel\ } n') \\
n \in \{0,1\} \ \texttt{logop}\ n' \in \{0,1\} & \to \delta(n = 1 \mathit{\ logop\ } n' = 1) \\
\texttt{not}\ n \in \{0,1\} & \to \delta(n = 0) \\
e & \to \tilde{\nu}^Q(e) \\
 & \quad \text{if } \texttt{query}\ e \in Q(\mathcal{M}) \\
o\texttt{(}e_1\texttt{,...,}e_n\texttt{)} & \to \tilde{\nu}^O_{o(id(e_1),\ldots,id(e_n))}(e)[X_1 := e_1, \ldots, X_n := e_n] \\
 & \quad \text{if } d = \texttt{object}\ o\texttt{(}X_1\texttt{,...,}X_n\texttt{)}\ \texttt{:=}\ e \in O(\mathcal{M}) \\
 & \quad \text{and } (e_1, \ldots, e_n) \in \mathit{indexable}^n \\
p\texttt{(}e_1\texttt{,...,}e_n\texttt{)} & \to e[X_1 := e_1, \ldots, X_n := e_n] \\
 & \quad \text{if } d = \texttt{rule}\ p\texttt{(}X_1\texttt{,...,}X_n\texttt{)}\ \texttt{:=}\ e \in R(\mathcal{M}) \\
\texttt{let(}x\ \texttt{:=}\ v\texttt{,}e\texttt{)} & \to e[x := v] \\
\texttt{[}n \in \mathbf{N}\ \texttt{..}\ n' \in \mathbf{N}\texttt{]} & \to \begin{cases} \texttt{[}n\texttt{,}\ n+1\texttt{,...,}n'\texttt{]} & \text{if } n \le n' \\ \texttt{[]} & \text{otherwise} \end{cases} \\
\texttt{[}e_1\texttt{,...,}e_n\texttt{]}\ \texttt{++}\ \texttt{[}e'_1\texttt{,...,}e'_n\texttt{]} & \to \texttt{[}e_1\texttt{,...,}e_n\texttt{,}e'_1\texttt{,...,}e'_n\texttt{]} \\
\texttt{length([}e_1\texttt{, ..., }e_n\texttt{])} & \to n \\
\texttt{nth(}i \in \{1,\ldots,n\}\texttt{, [}e_1\texttt{, ..., }e_n\texttt{])} & \to e_i \\
\{f_1\texttt{:}\ e_1\texttt{,...,}f_n\texttt{:}\ e_n\texttt{}\}\texttt{:}f_i & \to e_i \\
\texttt{foldl(}A\ \texttt{from}\ i\texttt{,}X\ \texttt{in}\ \texttt{[}e_1\texttt{,...,}e_n\texttt{],}e\texttt{)} & \to i \rhd_e e_1 \rhd_e \cdots \rhd_e e_n \\
 & \quad \text{where } u \rhd_e v = e[A := u, X := v] \\
\left.\begin{array}{r} \texttt{minimize(}g\texttt{,}k\texttt{)} \\ \texttt{maximize(}g\texttt{,}k\texttt{)} \\ \texttt{search(}h\texttt{,}g\texttt{)} \\ \texttt{constraint(}g\texttt{)} \\ \texttt{static(}g\texttt{)} \\ \texttt{dynamic(}g\texttt{)} \end{array}\right\} & \to g
\end{array}
$$

Table 2. Small-step reduction semantics defining the success semantics of Rules2CP (without distinguishing optimization from satisfaction predicates).

Let δ be the reification operator: $\delta(\top) = 1$ and $\delta(\bot) = 0$. A solution for a model $\mathcal{M}$ is an assignment (ν^Q, ν^O) for which the query of $\mathcal{M}$ is reduced to 1 by the small-step reduction described in table 2.

Definition 1. *The set of observables $\mathcal{O}_s(\mathcal{M})$ for the success semantics of $\mathcal{M}$ is the set of solutions of $\mathcal{M}$.*

$$\mathcal{O}_s(\mathcal{M}) = \{(\nu^Q, \nu^O) \mid \nu^Q(q(\mathcal{M})) \xrightarrow{*} 1\}$$

3 Static Expansion Schema

The static expansion schema is defined by two transformations, the first one producing intermediate code:

1. $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ expands a query to the deterministic code which adds the constraints
2. $\longrightarrow\!\!\langle \substack{\text{stc} \\ \text{srch}} \rangle\!\!\longrightarrow$ expands the search code.

The elimination of negations in formulae by descending them to the constraints with De Morgans laws are part of transformations, but are not presented.

3.1 Deterministic code generation

Built-in operators Reification transforms boolean values in integers and logical operators in artihmetic operators. Partial evaluation occurs on arithmetic, comparison and logical operators.

$$\frac{e_1 \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow e_1' \qquad e_2 \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow e_2'}{e_1 \ \texttt{op} \ e_2 \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow e_1' \ \texttt{op} \ e_2'}$$

$$\frac{e_1 \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow e_1' \qquad e_2 \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow e_2'}{e_1 \ \texttt{rel} \ e_2 \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow \texttt{reify(}e_1' \ \texttt{rel} \ e_2'\texttt{)}}$$

$$\frac{e_1 \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow e_1' \qquad e_2 \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow e_2'}{e_1 \ \texttt{logop} \ e_2 \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow \texttt{reify(}e_1' \texttt{ = 1 } \texttt{logop} \ e_2' \texttt{ = 1)}}$$

$$\frac{e \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow e'}{\texttt{not } e \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow \texttt{reify(}e' \texttt{ = 0)}}$$

Definitions and calls Within this static expansion schema, definitions are fully expanded. Free variables in object definitions are indexed and stored in a table ν^O.

$$\frac{\begin{array}{c} a_1 \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow a_1' \\ \cdots \\ a_n \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow a_n' \\ e[X_1 := a_1', \ldots, X_n := a_n'] \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow e' \end{array}}{p\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{)} \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow e'} \begin{cases} r = \texttt{rule } p\texttt{(}X_1\texttt{,}\ldots\texttt{,}X_n\texttt{) := } e \in R(\mathcal{M}) \\ \text{fv}(r) = \emptyset \end{cases}$$

$$\frac{\begin{array}{c} a_1 \longrightarrow_{\langle \mathrm{stc} \rangle} a'_1 \\ \dots \\ a_n \longrightarrow_{\langle \mathrm{stc} \rangle} a'_n \\ \sigma(e)[X_1 := a'_1, \dots, X_n := a'_n] \longrightarrow_{\langle \mathrm{stc} \rangle} e' \end{array}}{p(a_1, \dots, a_n) \longrightarrow_{\langle \mathrm{stc} \rangle} e'} \begin{cases} d = \texttt{object}\ o(X_1, \dots, X_n)\ \texttt{:=}\ e \in R(\mathcal{M}) \\ \sigma = \nu^{O}_{o(id(a'_1), \dots, id(a'_n))} \\ \mathrm{dom}(\sigma) = \mathrm{fv}(d) \end{cases}$$

Lists If its bounds are statically instantiated, a range is reduced to the list of integers that it contains by partial evaluation.

$$\frac{e_1 \longrightarrow_{\langle \mathrm{stc} \rangle} l \qquad \dots \qquad e_n \longrightarrow_{\langle \mathrm{stc} \rangle} u}{[e_1\ \texttt{..}\ e_2] \longrightarrow_{\langle \mathrm{stc} \rangle} [l,\ l+1, \dots,\ u]} \begin{cases} l, u \in \mathbf{N} \\ l \leq u \end{cases}$$

$$\frac{l_1 \longrightarrow_{\langle \mathrm{stc} \rangle} [d_1, \dots, d_n] \qquad l_2 \longrightarrow_{\langle \mathrm{stc} \rangle} [e_1, \dots, e_m]}{l_1\ \texttt{++}\ l_2 \longrightarrow_{\langle \mathrm{stc} \rangle} [d_1, \dots, d_n,\ e_1, \dots, e_n]}$$

$$\frac{e_1 \longrightarrow_{\langle \mathrm{stc} \rangle} e'_1 \qquad \dots \qquad e_n \longrightarrow_{\langle \mathrm{stc} \rangle} e'_n}{[e_1, \dots, e_n] \longrightarrow_{\langle \mathrm{stc} \rangle} [e'_1, \dots, e'_n]}$$

Records Record projection need the record to be statically instantiated.

$$\frac{e_i \longrightarrow_{\langle \mathrm{stc} \rangle} e'_i}{\{f_1\texttt{:}\ e_1, \dots, f_n\texttt{:}\ e_n\ \}\texttt{:}f_i \longrightarrow_{\langle \mathrm{stc} \rangle} e'_i}\ f_i \in \{f_1, \dots, f_n\}$$

$$\frac{e_1 \longrightarrow_{\langle \mathrm{stc} \rangle} e'_1 \qquad \dots \qquad e_n \longrightarrow_{\langle \mathrm{stc} \rangle} e'_n}{\{f_1\texttt{:}\ e_1, \dots, f_n\texttt{:}\ e_n\ \} \longrightarrow_{\langle \mathrm{stc} \rangle} \{f_1\texttt{:}\ e'_1, \dots, f_n\texttt{:}\ e'_n\ \}}$$

Let-binding Substitutions are implicitly operated modulo alpha-conversion.

$$\frac{v \longrightarrow_{\langle \mathrm{stc} \rangle} v'}{\texttt{let}(X\ \texttt{:=}\ v,\ e) \longrightarrow_{\langle \mathrm{stc} \rangle} e[X := v']}$$

Combinators Combinators are expanded and require their list and initial element arguments to be statically instantiated.

$$\frac{\begin{array}{c} i \longrightarrow_{\langle \mathrm{stc} \rangle} i_0 \qquad l \longrightarrow_{\langle \mathrm{stc} \rangle} [e_1, \dots, e_n] \\ i_0 \rhd_e e_1 \longrightarrow_{\langle \mathrm{stc} \rangle} i_1 \\ i_1 \rhd_e e_2 \longrightarrow_{\langle \mathrm{stc} \rangle} i_2 \qquad \dots \qquad i_{n-1} \rhd_e e_n \longrightarrow_{\langle \mathrm{stc} \rangle} i_n \end{array}}{\texttt{foldl}(A\ \texttt{from}\ i,\ X\ \texttt{in}\ l,\ e) \longrightarrow_{\langle \mathrm{stc} \rangle} i_n}$$

where $u \rhd_e v = e[A := u, X := v]$

Search By default, a logic formula f defines a reified constraint. In the context of a `search(`f`)` predicate, f defines a search tree.

$$\frac{f \longrightarrow\!\!\left\langle \begin{smallmatrix}\text{stc}\\ \text{srch}\end{smallmatrix}\right\rangle\!\!\longrightarrow f'}{\texttt{search(}f\texttt{)} \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow \texttt{search(}f'\texttt{)}} \qquad \frac{f \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow f'}{\texttt{constraint(}f\texttt{)} \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow \texttt{constraint(}f'\texttt{)}}$$

A predicate `minimize(`f`,` c`)` minimizes the value of the finite domaine variable V denoted by c following a branch and bound search. f is a formula implicitly interpreted as a search tree that constrain V to an assignment.

$$\frac{\texttt{search(}e\texttt{)} \longrightarrow\!\!\left\langle \begin{smallmatrix}\text{stc}\\ \text{srch}\end{smallmatrix}\right\rangle\!\!\longrightarrow e' \qquad c \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow \text{V}}{p\texttt{(}e\texttt{,}\ c\texttt{)} \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow p\texttt{(}e'\texttt{,}\ V\texttt{)}} \left\{ p \in \{\texttt{minimize}, \texttt{maximize}\} \right.$$

Dynamic mode It is possible to dynamically evaluate (see Sec. 4) an expression instead of statically expand it with the predicate `dynamic/1`.

$$\frac{e \longrightarrow\!\!\langle \text{dyn} \rangle\!\!\longrightarrow e'}{\texttt{dynamic(}e\texttt{)} \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow e'} \qquad \frac{e \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow e'}{\texttt{static(}e\texttt{)} \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow e'}$$

Example 3. For the n-queens model presented in example 1, the static expansion compilation schema procudes the following intermediate code for $n = 4$:

```
domain([Q_1_1,Q_2_1,Q_3_1,Q_4_1], 1, 4) and
all_different([Q_1_1,Q_2_1,Q_3_1,Q_4_1]) and
Q_1_1 # 1+Q_2_1 and Q_1_1 # -1+Q_2_1 and
Q_1_1 # 2+Q_3_1 and Q_1_1 # -2+Q_3_1 and
Q_1_1 # 3+Q_4_1 and Q_1_1 # -3+Q_4_1 and
Q_2_1 # 1+Q_3_1 and Q_2_1 # -1+Q_3_1 and
Q_2_1 # 2+Q_4_1 and Q_2_1 # -2+Q_4_1 and
Q_3_1 # 1+Q_4_1 and Q_3_1 # -1+Q_4_1 and

search(Q_1_1 = 2 or Q_1_1 = 3 or Q_1_1 = 1 or Q_1_1 = 4 and
       Q_2_1 = 2 or Q_2_1 = 3 or Q_2_1 = 1 or Q_2_1 = 4 and
       Q_3_1 = 2 or Q_3_1 = 3 or Q_3_1 = 1 or Q_3_1 = 4 and
       Q_4_1 = 2 or Q_4_1 = 3 or Q_4_1 = 1 or Q_4_1 = 4)
```

It is worth noting that the complete cartesian product of all queens is not generated for the binary difference constraints thanks to the partial evaluation mechanism.

3.2 Non-deterministic code generation

By the $\longrightarrow\!\!\left\langle \begin{smallmatrix}\text{stc}\\ \text{srch}\end{smallmatrix}\right\rangle\!\!\longrightarrow$ transformation, the conjunction operator `and` becomes a sequence operator and the disjunction operator `or` becomes a non-deterministic choice operator.

The formula f is expanded following $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ schema and the modification described above is operated giving a non-deterministic code $f' : f \longrightarrow\!\!\left\langle \begin{smallmatrix}\text{stc}\\ \text{srch}\end{smallmatrix}\right\rangle\!\!\longrightarrow f'$.

3.3 Correctness and complexity of the static expansion schema

Proposition 1. *Given a Rule2CP model $\mathcal{M}$, let $\mathcal{M}'$ such that $\mathcal{M} \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow \mathcal{M}'$, then $\mathcal{O}_\text{s}(\mathcal{M}) = \mathcal{O}_\text{s}(\mathcal{M}')$ (*i.e., *$\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$ preserves the model declarative semantics.)*

Proof. For every assignment (ν^Q, ν^O) for $\mathcal{M}$, we check inductively on the derivation of $\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$ and $\longrightarrow\!\langle \begin{smallmatrix}\text{stc}\\ \text{srch}\end{smallmatrix} \rangle\!\longrightarrow$ that $\nu^Q(\mathcal{M}) \xrightarrow{*} \nu^Q(\mathcal{M}')$. Most of derivations are independent from assignment and verify this property by definition. Calls to object definitions are restricted to indexable arguments and the table ν^O is used for indexation. $\longrightarrow\!\langle \begin{smallmatrix}\text{stc}\\ \text{srch}\end{smallmatrix} \rangle\!\longrightarrow$ schema does not change the set of solutions with respect to $\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$.

Definition 2. *Given a Rule2CP model $\mathcal{M}$, the* fold rank $\alpha(s)$ *of a symbol s is defined inductively by:*

> $\alpha(s) = 0$ *if s is not the head symbol of a declaration or rule in $\mathcal{M}$,*
> $\alpha(s) = max\{n + \alpha(s') \mid L = R \in \mathcal{M}$, *$s$ is the head symbol of L and R contains a nesting of n fold operators or quantifiers on an expression containing symbol s'*$\}$.

The fold rank of $\mathcal{M}$ is the maximum fold rank of the symbols in $\mathcal{M}$.

Definition 3. *the* definition rank $\rho(s)$ *of a symbol s is defined inductively by:*

> $\rho(s) = 0$ *if s is not the head symbol of a clause in $\mathcal{M}$,*
> $\rho(s) = n + 1$, *if s is the head symbol of a clause in $\mathcal{M}$ and n is the greatest definition rank of the symbols in the right hand side of the clause.*

The definition rank of $\mathcal{M}$ is the maximum definition rank of the symbols defined in $\mathcal{M}$.

Proposition 2. *[4] For any* `Rules2CP` *model $\mathcal{M}$, the size of the generated program is in $O(l^a * b^r)$, where l is the maximum length of the lists in $\mathcal{M}$ (or at least 1), a is the fold rank of $\mathcal{M}$, b is the maximum size of the declaration and rule bodies in $\mathcal{M}$, and r is the definition rank of $\mathcal{M}$.*

Example 4. The fold rank of the n-queens model presented in example 1 is 2. Therefore the size of the generated program is in $O(l^2)$. The bound is tight in this example.

Example 5. The exponential size of the generated code in the definition rank of the model can be reached with the following model:

```
rule c1(A) := c2(A+1) and c2(2*A)
rule c2(A) := c3(A+1) and c3(2*A)
...
rule cn(A) := c(A+1) and c(2*A)
rule c(A) := A # 666
```

In this example, the generated code for the query $c1(X)$ is of size 2^n.

4 The dynamic compilation schema

The dynamic compilation schema is defined by two transformations which produce intermediate code. The first transformation, noted $\longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow$, expands a query to a deterministic code which adds the constraints and calls the dynamic search part. The second transformation, noted $\longrightarrow\!\langle {}^{\text{dyn}}_{\text{srch}} \rangle\!\longrightarrow$, rewrites the search part to a non-deterministic code which performs the reordering and search. The intermediate code follows the syntax of Rules2CP programs but allows recursion. Search-tree directives $\mathcal{S}$ are eliminated and reformulated by $\longrightarrow\!\langle {}^{\text{dyn}}_{\text{srch}} \rangle\!\longrightarrow$.

It is worth noting that the operator `or` represents a reified $\vee$-constraint in the deterministic code, and a choice-point in the non-deterministic code. The syntactic construction `delay(`p`(`$\boldsymbol{X}$`))` is introduced in the intermediate code to denote the symbolic term p`(`$\boldsymbol{X}$`)` as opposed to a call to the definition p`(`$\boldsymbol{X}$`)`. Such an intermediate code is then straightforward to translate to a Prolog or Java program.

To illustrate dynamic compilation, let us consider two rule definitions that constrain the shape of objects in a simple two-dimensional placement problem of thin sticks, where the sticks 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)

4.1 Transformation of the query to deterministic code

$\boldsymbol{V} \vdash \cdot \longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow \cdot$ reformulates search directives inductively over the structure of Rules2CP 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`(`$\boldsymbol{X}$`) =` e is translated in the intermediate code to the definition: p_d`(`$\boldsymbol{X}$`) =` e', where $\text{fv}(e) \vdash e \longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow e'$. Then, translated calls rely on these definitions: $\boldsymbol{V} \vdash p(\boldsymbol{X}) \longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow p_d$`(`$\boldsymbol{X}$`)`

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

$$\frac{\boldsymbol{V} \vdash l \longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow l' \qquad \boldsymbol{V} \vdash i \longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow i'}{\boldsymbol{V} \vdash \texttt{foldl(}A\ \texttt{from}\ i\texttt{,}\ X\ \texttt{in}\ l\texttt{,}\ e\texttt{)} \longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow q\texttt{(}l'\texttt{,}\ i'\texttt{,}\ \boldsymbol{V}\texttt{)}}$$

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

$$q(\texttt{[]},\ I,\ \boldsymbol{V})\ \texttt{=}\ I\,.$$
$$q(\texttt{[}H\ \texttt{|}\ T\texttt{]},\ I,\ \boldsymbol{V})\ \texttt{=}\ q(T,\ e',\ \boldsymbol{V})\,. \qquad \boldsymbol{V} \vdash e[A := I, X := H] \longrightarrow\!\!\!\!\!\!\langle \text{dyn} \rangle\ e'$$

Other cases for $\longrightarrow\!\!\!\!\!\!\langle \text{dyn} \rangle$ are defined homomorphically with respect to sub-expressions, taking care of scopes and name clashes: e.g.,

$$\frac{\boldsymbol{V} \vdash d \longrightarrow\!\!\!\!\!\!\langle \text{dyn} \rangle\ d' \qquad \boldsymbol{V} \cdot X \vdash e[V := X] \longrightarrow\!\!\!\!\!\!\langle \text{dyn} \rangle\ e'}{\boldsymbol{V} \vdash \texttt{let(}V\ \texttt{=}\ d\ \texttt{in}\ e\texttt{)} \longrightarrow\!\!\!\!\!\!\langle \text{dyn} \rangle\ \texttt{let(}X\ \texttt{=}\ d'\ \texttt{in}\ e'\texttt{)}}$$

where X is a fresh variable.

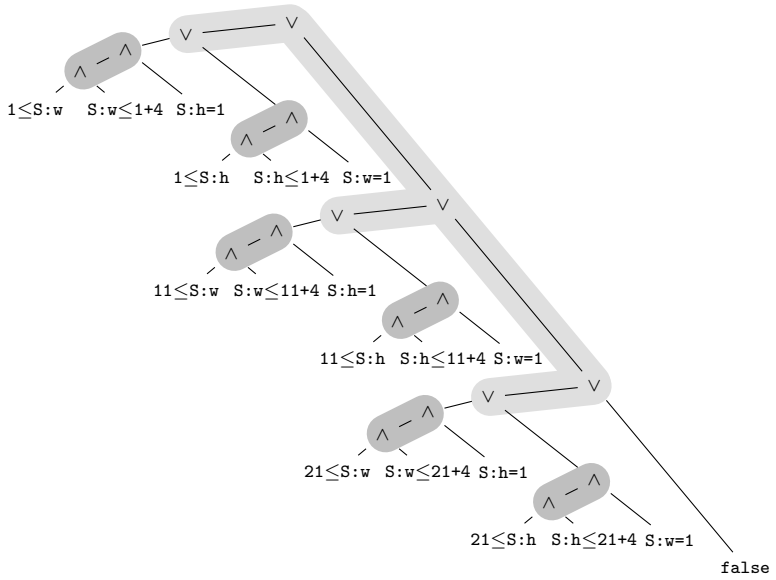
Search directives rely on the search transformation (defined in Sec. 4.2).

$$\frac{\begin{array}{c}\boldsymbol{V} \vdash h \longrightarrow\!\!\!\!\!\!\langle \text{dyn} \rangle\ \texttt{conjunctive(}o^1_\wedge\texttt{)}\ \ldots\ \texttt{and conjunctive(}o^n_\wedge\texttt{) and} \\ \texttt{disjunctive(}o^1_\vee\texttt{)}\ \ldots\ \texttt{and disjunctive(}o^m_\vee\texttt{)} \\ ([o^1_\wedge, \ldots, o^n_\wedge], [o^1_\vee, \ldots, o^m_\vee]); \boldsymbol{V} \vdash e \longrightarrow\!\!\!\!\!\!\langle \begin{smallmatrix}\text{dyn} \\ \text{srch}\end{smallmatrix} \rangle\ e'\end{array}}{\boldsymbol{V} \vdash \texttt{search(}h\texttt{,}\ e\texttt{)} \longrightarrow\!\!\!\!\!\!\langle \text{dyn} \rangle\ e'}$$

4.2 Transformation of the search to non-deterministic code

The compilation of the search-strategy relies on the notion of *O-layers* in a tree: 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.

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 Rules2CP expression syntax trees, by letting layers go through let-bindings, definition calls, and in the right-hand side of `implies` and through the tree intentionnaly constructed by `fold`. The child nodes of a layer are the nodes which are child 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 the 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 $(\boldsymbol{o}_\wedge, \boldsymbol{o}_\vee)$, $(\boldsymbol{o}_\wedge, \boldsymbol{o}_\vee); \boldsymbol{V} \vdash \cdot \multimap\!\langle {}^{\text{dyn}}_{\text{srch}} \rangle\!\longrightarrow \cdot$ 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 $\boldsymbol{c}_\vee^{-\infty}$, vectors whose every component equals to `bottom`, since no criteria apply outside any definition. $\boldsymbol{V} \vdash \cdot \multimap\!\langle {}^{\text{dyn}}_{\text{srch}} \rangle\!\longrightarrow \cdot$ is arbitrarily defined as $(\boldsymbol{c}_\wedge^{-\infty}, \boldsymbol{c}_\vee^{-\infty}); \boldsymbol{V} \vdash \cdot \multimap\!\langle {}^{\text{dyn}}_{\text{srch}(\wedge)} \rangle\!\longrightarrow \cdot$ to initiate the transformation (the root layer, possibly empty, can always be considered as being an $\wedge$-layer). $\cdot \multimap\!\langle {}^{\text{dyn}}_{\text{srch}(O)} \rangle\!\longrightarrow \cdot$ relies on the auxiliary transformation $(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash \cdot \multimap\!\langle {}^{\text{dyn}}_{\text{list}(O)} \rangle\!\longrightarrow \cdot$ which produces code computing an associative list: for each child node of the O-layer, the score vector of the node is associated to the definition to call to explore the child recursively.

$$\frac{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash e \multimap\!\langle {}^{\text{dyn}}_{\text{list}(O)} \rangle\!\longrightarrow e'}{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash e \multimap\!\langle {}^{\text{dyn}}_{\text{srch}(O)} \rangle\!\longrightarrow \texttt{iter\_predicates}_O(e')}$$

where $\texttt{iter\_predicates}_O(L)$ 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 intermediate code, one for each kind of layer:

$$\frac{(u(\boldsymbol{C}_\wedge, \boldsymbol{o}_\wedge, p\texttt{(}\boldsymbol{X}\texttt{)}), \boldsymbol{C}_\vee); \mathrm{fv}(e) \vdash e \multimap\!\!\left\langle {}^{\mathrm{dyn}}_{\mathrm{srch}(\wedge)} \right\rangle\!\!\longrightarrow e'}{(\boldsymbol{o}_\wedge, \boldsymbol{o}_\vee); \boldsymbol{V} \vdash \texttt{rule } p\texttt{(}\boldsymbol{X}\texttt{) := } e \multimap\!\!\left\langle {}^{\mathrm{dyn}}_{\mathrm{srch}(\wedge)} \right\rangle\!\!\longrightarrow p_\wedge(\boldsymbol{C}_\wedge, \boldsymbol{C}_\vee, \boldsymbol{X}) \texttt{ = } e'}$$

$$\frac{(\boldsymbol{C}_\wedge, u(\boldsymbol{C}_\vee, \boldsymbol{o}_\vee, p\texttt{(}\boldsymbol{X}\texttt{)})); \mathrm{fv}(e) \vdash e \multimap\!\!\left\langle {}^{\mathrm{dyn}}_{\mathrm{srch}(\vee)} \right\rangle\!\!\longrightarrow e'}{(\boldsymbol{o}_\wedge, \boldsymbol{o}_\vee); \boldsymbol{V} \vdash \texttt{rule } p\texttt{(}\boldsymbol{X}\texttt{) := } e \multimap\!\!\left\langle {}^{\mathrm{dyn}}_{\mathrm{srch}(\vee)} \right\rangle\!\!\longrightarrow p_\vee(\boldsymbol{C}_\wedge, \boldsymbol{C}_\vee, \boldsymbol{X}) \texttt{ = } e'}$$

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:

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

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

$$(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash p\texttt{(}\boldsymbol{X}\texttt{)} \multimap\!\!\left\langle {}^{\mathrm{dyn}}_{\mathrm{list}(O)} \right\rangle\!\!\longrightarrow p_O\texttt{(}\boldsymbol{c}_\wedge\texttt{, } \boldsymbol{c}_\vee\texttt{, } \boldsymbol{X}\texttt{)}$$

Boolean operators $\multimap\!\!\left\langle {}^{\mathrm{dyn}}_{\mathrm{list}(\wedge)} \right\rangle\!\!\longrightarrow$ aggregates lists in the root $\wedge$-layer. A new predicate q is introduced for each child node of the $\wedge$-layer.

$$\frac{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash a \multimap\!\!\left\langle {}^{\mathrm{dyn}}_{\mathrm{list}(\wedge)} \right\rangle\!\!\longrightarrow a' \qquad (\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash b \multimap\!\!\left\langle {}^{\mathrm{dyn}}_{\mathrm{list}(\wedge)} \right\rangle\!\!\longrightarrow b'}{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash a \texttt{ and } b \multimap\!\!\left\langle {}^{\mathrm{dyn}}_{\mathrm{list}(\wedge)} \right\rangle\!\!\longrightarrow \texttt{append(}a'\texttt{, } b'\texttt{)}}$$

$$(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash a \texttt{ or } b \multimap\!\!\left\langle {}^{\mathrm{dyn}}_{\mathrm{list}(\wedge)} \right\rangle\!\!\longrightarrow \texttt{[\{ costs = } \boldsymbol{c}_\wedge\texttt{,}$$
$$\texttt{predicate = delay(}q\texttt{(}\boldsymbol{c}_\wedge\texttt{, } \boldsymbol{c}_\vee\texttt{, } \boldsymbol{V}\texttt{)) \}]}$$

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

$$q\texttt{(}\boldsymbol{C}_\wedge\texttt{, } \boldsymbol{C}_\vee\texttt{, } \boldsymbol{V}\texttt{) = } e. \qquad (\boldsymbol{C}_\wedge, \boldsymbol{C}_\vee); \boldsymbol{V} \vdash a \texttt{ or } b \multimap\!\!\left\langle {}^{\mathrm{dyn}}_{\mathrm{srch}(\vee)} \right\rangle\!\!\longrightarrow e$$

Dual definitions hold for $\multimap\!\!\left\langle {}^{\mathrm{dyn}}_{\mathrm{list}(\vee)} \right\rangle\!\!\longrightarrow$

Filtering

$$\frac{\boldsymbol{V} \vdash a \longrightarrow\!\!\!\!\!\!\langle \text{dyn} \rangle\!\!\!\!\!\!\longrightarrow a' \qquad (\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash b \longrightarrow\!\!\!\!\!\!\langle {}^{\text{dyn}}_{\text{list}(O)} \rangle\!\!\!\!\!\!\longrightarrow b'}{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash a \texttt{ implies } b \longrightarrow\!\!\!\!\!\!\langle {}^{\text{dyn}}_{\text{list}(O)} \rangle\!\!\!\!\!\!\longrightarrow \texttt{filter(}\boldsymbol{c}_O\texttt{, } a'\texttt{, } b'\texttt{)}}$$

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

Let-binding

$$\frac{\boldsymbol{V} \vdash v \longrightarrow\!\!\!\!\!\!\langle \text{dyn} \rangle\!\!\!\!\!\!\longrightarrow v' \qquad (\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \cdot Y \vdash e[X := Y] \longrightarrow\!\!\!\!\!\!\langle {}^{\text{dyn}}_{\text{list}(O)} \rangle\!\!\!\!\!\!\longrightarrow e'}{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash \texttt{let(}X\texttt{ = }v\texttt{ in }e\texttt{)} \longrightarrow\!\!\!\!\!\!\langle {}^{\text{dyn}}_{\text{list}(O)} \rangle\!\!\!\!\!\!\longrightarrow \texttt{let(}Y\texttt{ = }v'\texttt{, }e'\texttt{)}}$$

where Y is a fresh variable.

Aggregators Aggregators use a special source symbol, `rec`, to handle recursion.

$$\frac{\boldsymbol{V} \vdash \texttt{reverse(}l\texttt{)} \longrightarrow\!\!\!\!\!\!\langle \text{dyn} \rangle\!\!\!\!\!\!\longrightarrow l'}{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash \texttt{foldl(}A\texttt{ from }i\texttt{, }X\texttt{ in }l\texttt{, }e\texttt{)} \longrightarrow\!\!\!\!\!\!\langle {}^{\text{dyn}}_{\text{list}(O)} \rangle\!\!\!\!\!\!\longrightarrow q_O\texttt{(}l'\texttt{, }\boldsymbol{c}_\wedge\texttt{, }\boldsymbol{c}_\vee\texttt{, }\boldsymbol{V}\texttt{)}}$$

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

$$\frac{}{q_O\texttt{([],}\boldsymbol{C}_\wedge\texttt{,}\boldsymbol{C}_\vee\texttt{,}\boldsymbol{V}\texttt{) = }i'\texttt{.}} \qquad \frac{(\boldsymbol{C}_\wedge, \boldsymbol{C}_\vee); \boldsymbol{V} \vdash i \longrightarrow\!\!\!\!\!\!\langle {}^{\text{dyn}}_{\text{list}(O)} \rangle\!\!\!\!\!\!\longrightarrow i' \quad (\boldsymbol{C}_\wedge, \boldsymbol{C}_\vee); \boldsymbol{V} \cdot H \vdash e[A := \texttt{rec(}q\texttt{, }T\texttt{, }\boldsymbol{V}\texttt{)}, X := H] \longrightarrow\!\!\!\!\!\!\langle {}^{\text{dyn}}_{\text{list}(O)} \rangle\!\!\!\!\!\!\longrightarrow e'}{q_O\texttt{([}H\texttt{ | }T\texttt{],}\boldsymbol{C}_\wedge\texttt{,}\boldsymbol{C}_\vee\texttt{,}\boldsymbol{V}\texttt{) = }e'\texttt{.}}$$

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

$$(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \cdot H \vdash \texttt{rec(}q\texttt{, }T\texttt{, }\boldsymbol{V}\texttt{)} \longrightarrow\!\!\!\!\!\!\langle {}^{\text{dyn}}_{\text{list}(O)} \rangle\!\!\!\!\!\!\longrightarrow q_O\texttt{(}T\texttt{, }\boldsymbol{c}_\wedge\texttt{, }\boldsymbol{c}_\vee\texttt{, }\boldsymbol{V}\texttt{)}$$

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.

$$(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash e \longrightarrow\!\!\!\!\!\!\langle {}^{\text{dyn}}_{\text{list}(O)} \rangle\!\!\!\!\!\!\longrightarrow \texttt{[\{ costs = }\boldsymbol{c}_O\texttt{,}\\ \texttt{predicate = delay(}q\texttt{(}\boldsymbol{V}\texttt{)) \}]}$$

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

$$\frac{}{q\texttt{(}\boldsymbol{V}\texttt{) = }e'\texttt{.}} \qquad \frac{\boldsymbol{V} \vdash e \longrightarrow\!\!\!\!\!\!\langle \text{dyn} \rangle\!\!\!\!\!\!\longrightarrow e'}{}$$

Property 1. There are $\mathbf{O}(d \cdot s)$ p_d-, $p_\vee$- and $p_\wedge$-definitions in intermediate code, where d is the number of definitions in the Rules2CP code and s is the number of `search` clauses. Each definition in the intermediate code, including the auxiliary definitions for `fold` and sub-layers, has a size linear in the size of the original Rules2CP definition. In particular, if there is one `search` clause, the intermediate code has a size linear in the size of the original Rules2CP code. The complexity of the transformation is linear in the size of the generated code.

Proof. $\longrightarrow\!\!\langle_{\text{dyn}}\!\!\rangle\longrightarrow$ and $\longrightarrow\!\!\langle^{\text{dyn}}_{\text{list}}\!\!\rangle\longrightarrow$ are inductive transformations where each step linearly composes results of the 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 Rules2CP definition. For each Rules2CP definition $p(\boldsymbol{X})$, there is one definition p_d in the intermediate code, plus two definitions $p_\vee$ and $p_\wedge$ by `search` clauses.

This complexity result contrasts with Rules2CP transformation complexity[4] where definition unfolding leads to exponential code size in the worst case.

Example 6. Consider the result of transforming the 4-queens Rules2CP model by the dynamic compilation schema. Instead of expanding rule definitions as in the static schema, the dynamic schema generates one definition of the intermediate code for each definition, *e.g.* `safe/1` and `queens_constraints/1`, as follows:

```
safe(L) =
  all_different(rcp_variables(L)) and safe_foldl_2(L, 1, []).

queens_constraints(B, N) =
  domain(rcp_variables(B), 1, N) and safe(B).
```

Similarly, one (recursive) definition is generated for each aggregator of the model, *e.g.* the two nested universal quantifiers:

```
safe_foldl_2([], I_safe_foldl_2, _) = I_safe_foldl_2.
safe_foldl_2([Q_2 | Tail_2], I_safe_foldl_2, []) =
  safe_foldl_2(Tail_2, I_safe_foldl_2 and
  safe_foldl_3(L,1,Q_2), []).

safe_foldl_3([], I_safe_foldl_3, _) = I_safe_foldl_3.
safe_foldl_3([R_3 | Tail_3], I_safe_foldl_3, [Q_2]) =
  safe_foldl_3(Tail_3,
               (I_safe_foldl_3 and
                let(I := rcp_att(Q_2, column),
                    J := rcp_att(R_3, column),
                 I < J implies
                 rcp_att(Q_2, row) # J - I + rcp_att(R_3, row) and
                 rcp_att(Q_2, row) # I - J + rcp_att(R_3, row))),
               [Q_2]).
```

As for the search component, all rules in the scope of a `search` predicate generate two definitions of the intermediate code, one for a use in a conjunctive context and one for the disjunctive context.

When compiled with the dynamic schema, the model presented in the example 1 can be advantageously modified by writing the rule `queens_labeling_var` as follows:

```
rule queens_labeling_var(Var) :=
  exists(Val in [domain_min(Var) .. domain_max(Var)],
         queens_labeling_val(Var, Val)).
```

Here, the existential quantifier ranges over the actual bounds of queen variables instead of `[1 .. N]` as in the static version, thus allowing the search to benefit from propagation.

Similarly, the search tree ordering heuristics can be written with a dynamic criterion as follows:

```
heuristics mo :=
  disjunctive(
    least(abs((domain_max(Var) - domain_min(Var))/2 - Val))
    for queens_labeling_val(Var, Val)).
```

5 Evaluation

In this section, we first compare the compilation times and run times of Rules2CP and Cream. The performances are measured on classical N-Queens, Bridge Scheduling, and Open-Shop Scheduling problems. Then, we report performances of Cream on the Optimal Rectangle Packing problem which illustrates the need for dynamic search strategies that cannot be compiled with the static expansion schema.

5.1 Comparison of both compilation schemes

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 [14] 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.

Table 3 compares the compilation and execution runtimes in seconds in Cream with those obtained in Rules2CP.

	Rules2CP		Cream	
	Compilation	Solving	Compilation	Solving
8-Queens	0.070	0.000	0.020	0.000
16-Queens	0.290	0.000	0.020	0.020
32-Queens	1.840	0.005	0.020	0.080
64-Queens	15.430	0.030	0.020	0.340
96-Queens	58.510	0.060	0.020	0.740
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

Table 3. Rules2CP and Cream programs runtimes in seconds.

In all N-Queens instances, the "first-fail variables selection heuristics" is applied. In Rules2CP, first-fail is handled by the SICStus `labeling/2` built-in, whereas in Cream selection is handled by generated code (leaning on the `domain_size/1` predicate in this case).

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. 4, and no effort has been made yet to improve performances.

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

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.

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.

5.2 Dynamic search strategies

Our work on rule-base modelling languages for constraint programming originates from the EU project Net-WMS[1] which aims at solving real-size non-pure bin packing problems of the automotive industry. Three-dimensional Packing problems tackled in this project involve many business constraints, in addition to pure containment and non-overlapping constraints. To solve efficiently these problems, it is mandatory to benefit as much as possible from propagation during the search. Hence the need of expressing dynamic search strategies which depend on the values or domains of variables at runtime.

The Optimal Rectangle Packing problem, also known as Korf's benchmark [9], consists in finding the smallest rectangle containing n squares of sizes $S_i = i$ for $1 \leq i \leq n$. In [13], Simonis and O'Sullivan have provided a simple but efficient dynamic search strategy for solving this problem in SICStus Prolog, improving the best known runtimes obtained by Korf up to a factor of 300. We have transposed their model in Cream and report the performance figures obtained with the same SICStus Prolog system in table 4

n	Compilation	Solving	
	Cream	Cream	Reference
18	0.650	17	9
19	0.700	17	8
20	0.780	30	17
21	0.810	100	63
22	0.870	430	297
23	0.930	2700	1939
24	0.980	3900	2887
25	1.060	27020	20713

Table 4. Optimal Rectangle Packing problem runtimes in seconds (Linux / Intel Core2 CPU, 2.83GHz).

Table 4 shows that with fast compilation times, Cream generates SICStus Prolog code nearly as efficient as the hand-written SICStus Prolog program of [13] for the different instances of the problem.

6 Conclusion

Modelling languages for stating combinatorial optimization problems can be interpreted to produce executable constraint programs by fundamentally two

[1] `http://net-wms.ercim.org`

compilation schemas: the static expansion schema and the procedural code generation schema. We have shown that the static expansion schema may generate constraint programs of exponential size in the level of nesting of definitions (which remains limited in practice), while the code generation schema generates code of linear size. In our implementation of both schemas for the rule-based modelling language Rules2CP, we have shown that the code generation schema exhibits a time overhead of approximatively a factor 2 at runtime w.r.t. the statically expanded code. Furthermore the code generation schema makes it possible to benefit from propagation during search by executing dynamical search strategies specified in Rules2CP with a good efficiency as shown on optimal rectangle packing problems. All these results thus militate in favor of the procedural code generation schema which should probably be preferred to the static expansion schema.

The declarative specification of ordering heuristics by pattern matching on rules' left-hand sides introduced in Rules2CP should also be applicable to other modelling languages that use definitions, such as Zinc [11,3] for instance. A natural extension for future work is the specification of more complex search procedures which are currently limited in our system to depth-first backtracking and branch and bound search.

Acknowledgements. We acknowledge support from European FP6 Strep project Net-WMS, and discussions with the partners of this project. Special thanks go to Sylvain Soliman for his insights and to Sunrinderjeet Singh who developped the 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. Mats Carlsson, Nicolas Beldiceanu, and Julien Martin. A geometric constraint over k-dimensional objects and shapes subject to business rules. In Peter J. Stuckey, editor, *Proceedings of CP'08*, volume 5202 of *LNCS*, pages 220–234. Springer-Verlag, 2008.
3. 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.
4. 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'08*, volume 5655 of *Lecture Notes in Artificial Intelligence*, pages 66–83. Springer-Verlag, 2009.

5. François Fages and Julien Martin. Modelling search strategies in Rules2CP. In *Proceedings of CPAIOR'09*, volume 5547 of *Lecture Notes in Computer Science*, pages 321–322. Springer-Verlag, 2009.
6. 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.
7. Pascal Van Hentenryck, Laurent Perron, and Jean-Francois Puget. Search and strategies in OPL. *ACM Transactions on Compututational Logic*, 1(2):285–320, 2000.
8. P. C. Kanellakis and J. C. Mitchell. Polymorphic unification and ml typing. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 105–115, New York, NY, USA, 1989. ACM.
9. Richard E. Korf. Optimal rectangle packing: New results. In *ICAPS*, pages 142–149, 2004.
10. Laurent Michel and Pascal Van Hentenryck. The comet programming language and system. In *CP*, pages 881–881, 2005.
11. 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.
12. Didier Rémy. Records and variants as a natural extension of ML. In *Sixteenth Annual Symposium on Principles Of Programming Languages*, 1989.
13. 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.
14. Pascal Van Hentenryck. *The OPL Optimization programming Language*. MIT Press, 1999.