

Type Parametric Compilation of Algebraic Constraints

Marco Correia and Pedro Barahona

Centro de Inteligência Artificial, Departamento de Informática,
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
{mvc,pb}@di.fct.unl.pt

Abstract. This paper addresses the problem of propagating constraints involving arbitrary algebraic expressions. We formally describe previous approaches to this problem and propose a new model that does not decompose the expression thus avoiding introducing auxiliary data structures. We show how this compilation model fits naturally in a popular programming language supporting type parametricity, yielding significant speedups with respect to previous models.

1 Introduction

One of the key features in constraint programming is the ability to exploit problem structure. Traditionally, this has been achieved by designing efficient filtering algorithms for common constraints, and making them available to the user as modeling primitives. These algorithms remove inconsistent values from the domains of the variables subject to a constraint, often enforcing generalized arc-consistency (GAC) [13].

Unfortunately, enforcing GAC on an algebraic constraint is NP-complete in general [17]. In practice, constraints involving arbitrary algebraic expressions are decomposed into primitive subexpressions for which weaker filtering algorithms are available. However, since the decomposition consists of introducing auxiliary variables and propagators for each subexpression, propagating a large algebraic constraint may still be space and time inefficient.

This paper focuses on compiling (as opposed to decomposing) arbitrary algebraic expressions. We present two compilation models based on an abstraction called *view* [14,6] which overcome the limitations of decomposition based methods. The first model was first used in [12], but as far as we know, never formally described. The second model is original and improves over the first in a number of ways.

The output of the compilation process is a filtering algorithm which enforces some consistency on the entire expression. We will consider bounds(R) consistency algorithms [2], although the method works with other weak consistency types, namely node consistency.

The structure of the paper is the following. In section 2 we introduce the terminology used throughout the paper. Section 3 describes the classical approach to algebraic expression decomposition based on auxiliary variables and propagators. In section 4 we introduce views as the key for compiling expressions, and describe the compilation models for two different types of views. Theoretical and empirical comparisons of models using views and the classical decomposition are detailed in section 5 and 6 respectively. Section 7 points related work in the literature and state-of-the-art solvers, and the main conclusions are summarized in section 8.

2 Background and Notation

We will consider sentences from a first-order language $\mathcal{L}(\Sigma, V)$ involving the standard set of arithmetic and logical operators Σ and domain variables V (denoted by lowercase letters and the suffix : dvar). Additionally, we refer to subsets of $\mathcal{L}$ using *expression templates*, defined as follows:

Definition 1. [Expression template] *An expression template is an expression $t \in \mathcal{L}(\Sigma, V')$ where V' is V augmented with term variables (denoted by uppercase letters). It represents the largest language subset $\mathcal{L}^t \subseteq \mathcal{L}$ where for all $i \in \mathcal{L}^t$, i may be obtained from t by instantiating the term variables.*

The compilation process will be defined declaratively by means of action rules:

Definition 2. [Action Rule, Rule repository] *An action rule associated with a language $\mathcal{L}$ takes the form*

$$\frac{t}{e} \left[\begin{array}{c} a_1 \\ \vdots \\ a_n \end{array} \right]$$

and defines the rewriting of any input expression $i \in \mathcal{L}^t$ (i.e. matched by the template expression t) into output expression $e \in \mathcal{L}$, performing actions $a_1, \dots, a_n$ (the rule actions) as a side effect. A rule repository $R(\mathcal{L})$ is a collection of action rules associated with language $\mathcal{L}$.

Rule repositories will be used to perform derivations of expressions.

Definition 3. [Derivation] *A derivation (written $R[e]$) denotes the output of applying the most specific rule in the repository R matching e . A rule $r_1 \equiv \frac{t_1}{e_1}$ is more specific than a rule $r_2 \equiv \frac{t_2}{e_2}$ if $\mathcal{L}^{t_1} \subset \mathcal{L}^{t_2}$. For simplification, we will assume that there is no ambiguity for selecting the most specific rule for a given expression. In practice this can be enforced by adding extra rules to the repository.*

The following definition introduces the formalism for indexed collections, which will be used extensively.

Definition 4. [Indexed collection] *An indexed collection is a tuple $\langle C, X, M \rangle$ where C is a collection of arbitrary objects, X is a collection of indexing objects, and $M : X \rightarrow C$ is a mapping. We will denote an indexed collection by $C \langle X \rangle$ and will use $C[x]$ to refer to the element of the collection $c \in C \langle X \rangle$ mapped to by element $x \in X$.*

The following definitions introduce notions and standard terminology from the field of constraint programming. For detailed information see for example [13].

A constraint network is a tuple $\langle X, D, C \rangle$, consisting of a set of variables X , a set of domains D , and a set of constraints C . Every variable $x \in X$ has an associated domain $D(x)$ representing its possible values. A k -ary constraint c over the set of variables $X(c) = \{x_1, \dots, x_k\}$ defines the subset of the cartesian product $D(c) = D(x_1) \times \dots \times D(x_k)$ that satisfy the constraint. The constraint satisfaction problem consists of finding an assignment of values to variables which satisfy all constraints.

A propagator associated with a constraint c is a contracting and monotone function $\pi : \wp(D(c)) \rightarrow \wp(D(c))$ ($\wp(X)$ is the powerset of X) which removes inconsistent values from the domains. The application of a propagator to a constraint network is called propagator execution.

Given a network $\langle X, D, C \rangle$ and a constraint $c \in C$, a bound support τ on c is a tuple that satisfies c and such that for all $x_i \in X(c)$, $\lfloor D(x_i) \rfloor \leq \tau[x_i] \leq \lceil D(x_i) \rceil$.

- A constraint c is bound(Z) consistent iff for all $x_i \in X(c)$, there exists two bound supports τ_1, τ_2 such that $\tau_1[x_i] = \lceil D(x_i) \rceil$ and $\tau_2[x_i] = \lfloor D(x_i) \rfloor$, and $\tau_1, \tau_2 \in \mathbb{Z}^k$.
- A constraint c is bound(R) consistent iff for all $x_i \in X(c)$, there exists two bound supports τ_1, τ_2 such that $\tau_1[x_i] = \lceil D(x_i) \rceil$ and $\tau_2[x_i] = \lfloor D(x_i) \rfloor$, and $\tau_1, \tau_2 \in \mathbb{R}^k$.

3 Auxiliary Variables

The decomposition process for the auxiliary variables model recursively rewrites an expression bottom-up, starting with higher precedence subexpressions. It is defined by the following repository, where the first rule appears for every $\oplus \in \Sigma$, and $\text{post } \pi$ inserts propagator π in the constraint store:

$$R_{\text{AUX}}(\pi) = \left\{ \bigvee_{\oplus \in \Sigma} \frac{X \oplus Y}{z : \text{dvar}} \left[\text{post } \pi [R_{\text{AUX}}[X] \oplus R_{\text{AUX}}[Y] = z] \right], \frac{x : \text{dvar}}{x : \text{dvar}} \right\}$$

We first remark that a constraint is a special case of an expression, specifically a comparison or logical expression. The decomposition process for an arbitrary constraint $e \in \mathcal{L}$ is triggered by the addition of the following propagator to the constraint store:

$$\pi [R_{\text{AUX}}[e] = \text{true}] \quad (1)$$

Note that, in the decomposition process, auxiliary variables are introduced for any subexpression, possibly using reification in the case of logical or comparison subexpressions.

Example 1. The decomposition of the constraint $a \times b + c \leq d$ using repository R_{AUX} yields the following derivation:

$$\frac{\frac{\frac{a : \text{dvar} \times b : \text{dvar} + c : \text{dvar} \leq d : \text{dvar}}{t_3 : \text{dvar} + c : \text{dvar} \leq d : \text{dvar}}}{t_2 : \text{dvar} \leq d : \text{dvar}}}{t_1 : \text{dvar}} \left[\begin{array}{l} t_1 \leftarrow \text{new dvar} \\ t_2 \leftarrow \text{new dvar} \\ t_3 \leftarrow \text{new dvar} \\ \text{post } \pi [a : \text{dvar} \times b : \text{dvar} = t_3 : \text{dvar}] \\ \text{post } \pi [t_3 : \text{dvar} + c : \text{dvar} = t_2 : \text{dvar}] \\ \text{post } \pi [(t_2 : \text{dvar} \leq d : \text{dvar}) = t_1 : \text{dvar}] \\ \text{post } \pi [t_1 : \text{dvar} = \text{true}] \end{array} \right]$$

This method requires a predefined repository of propagators for basic expressions involving domain variables, more specifically it assumes the existence of a collection of propagators $\pi \langle \mathcal{L}' \rangle$ indexed by expressions of a bounded language $\mathcal{L}' \subset \mathcal{L}$, typically defined as

$$\mathcal{L}' = \bigcup_{\oplus \in \Sigma} x : \text{dvar} \oplus y : \text{dvar} = z : \text{dvar}$$

Since rules in R_{AUX} may only generate $x : \text{dvar}$ expressions, created propagators will always have $x : \text{dvar}$ operands, i.e. $R_{\text{AUX}}(\pi)[e]$ only requires propagators in $\pi \langle \mathcal{L}' \rangle$, for any $e \in \mathcal{L}$.

Assuming a method like hashing is used to index elements of the rule and propagator repositories, the runtime cost of the compilation algorithm R_{AUX} for an arbitrary constraint is linear on the number of binary operators in the constraint.

4 View Stores

Definition 5. [View store] A view store $\lambda \langle X \rangle$ is an indexed collection of objects which provides a common interface (a view) to each element of the entity X being viewed. We call $\lambda \langle X \rangle$ a view store λ over X , and $\lambda[x]$ the λ view for x .

For illustration purposes, we will detail a view store $\text{bnd} \langle \mathcal{L} \rangle$ over expressions from language $\mathcal{L}$. Specifically, all members of the bnd view store must expose the following interface¹:

$$\text{bnd} : \left\{ \begin{array}{l} \text{GETMIN} : \text{func} [\rightarrow r], \text{GETMAX} : \text{func} [\rightarrow r], \\ \text{UPDMIN} : \text{func} [i \rightarrow], \text{UPDMAX} : \text{func} [i \rightarrow] \end{array} \right\} \quad (2)$$

Defining a view store is achieved by defining views for concrete viewed objects, in our case expressions $e \in \mathcal{L}$. The bnd view for a domain variable is as follows, where i and r are respectively the input and output parameters of the functions defined above:

$$\text{bnd}[x : \text{dvar}] = \left\{ \begin{array}{l} \text{GETMIN} = \{r \leftarrow \lfloor D(x) \rfloor\} \\ \text{GETMAX} = \{r \leftarrow \lceil D(x) \rceil\} \\ \text{UPDMIN} = \{D(x) \leftarrow D(x) \setminus [-\infty \dots i - 1]\} \\ \text{UPDMAX} = \{D(x) \leftarrow D(x) \setminus [i + 1 \dots +\infty]\} \end{array} \right\}$$

Defining the full set of views for a collection of infinite size, such as language $\mathcal{L}$, is clearly infeasible. We now turn to this problem and present two possible solutions with a different compromise regarding efficiency as we will see later.

4.1 Finite View Stores (FVS)

The first solution follows from the fact that all views in a store have a common interface. They can all therefore be represented by a single semantic $x : \text{bnd}$ object which exposes the views's interface while abstracting its syntactic identity.

¹ This corresponds to the set of functions $\{\varphi, \varphi^-\}$ as defined in [15].

The compilation algorithm follows an approach similar to the auxiliary variables method, where the introduction of an auxiliary variable is replaced by the introduction of a $x : \text{bnd}$ object. This means that all bnd views may safely assume a finite set of operands, which in this case are $x : \text{bnd}$ objects.

Example 2. The following is a specialization of the bnd view for the addition of two arbitrary expressions:

$$\text{bnd}[x : \text{bnd} + y : \text{bnd}] : \left\{ \begin{array}{l} \text{GETMIN} = \{r \leftarrow x.\text{GETMIN}() + y.\text{GETMIN}()\} \\ \text{GETMAX} = \{r \leftarrow x.\text{GETMAX}() + y.\text{GETMAX}()\} \\ \text{UPDMIN} = \left\{ \begin{array}{l} x.\text{UPDMIN}(i - y.\text{GETMAX}()) \\ y.\text{UPDMIN}(i - x.\text{GETMAX}()) \end{array} \right\} \\ \text{UPDMAX} = \left\{ \begin{array}{l} x.\text{UPDMAX}(i - y.\text{GETMIN}()) \\ y.\text{UPDMAX}(i - x.\text{GETMIN}()) \end{array} \right\} \end{array} \right\}$$

The compilation algorithm is fully described by the following rule repository:

$$\mathbf{R}_{\text{BND}} = \left\{ \begin{array}{l} \bigvee_{\oplus \in \Sigma} \frac{X \oplus Y}{z : \text{bnd}} [z \leftarrow \text{new bnd}[\mathbf{R}_{\text{BND}}[X] \oplus \mathbf{R}_{\text{BND}}[Y]]] \\ \frac{x : \text{dvar}}{z : \text{bnd}} [z \leftarrow \text{new bnd}[x : \text{dvar}]] \end{array} \right\}$$

and may be used to define a specific propagator for an arbitrary constraint:

$$\pi[X] = \{\mathbf{R}_{\text{BND}}[X].\text{UPDMIN}(\text{true})\} \quad (3)$$

Propagator $\pi[e]$ instantiates the view for the constraint e and propagates it by setting its value to true .

Example 3. The compilation of the expression $a \times b + c \leq d$ using repository $\mathbf{R}_{\text{BND}}$ yields the following derivation:

$$\frac{\frac{\frac{a : \text{dvar} \times b : \text{dvar} + c : \text{dvar} \leq d : \text{dvar}}{v_1 : \text{bnd} \times v_2 : \text{bnd} + c : \text{dvar} \leq d : \text{bnd}}}{v_3 : \text{bnd} + v_4 : \text{bnd} \leq v_6 : \text{bnd}}}{\frac{v_5 : \text{bnd} \leq v_6 : \text{bnd}}{v_7 : \text{bnd}}} \left[\begin{array}{l} v_1 \leftarrow \text{new bnd}[a : \text{dvar}] \\ v_2 \leftarrow \text{new bnd}[b : \text{dvar}] \\ v_3 \leftarrow \text{new bnd}[v_1 : \text{bnd} \times v_2 : \text{bnd}] \\ v_4 \leftarrow \text{new bnd}[c : \text{dvar}] \\ v_5 \leftarrow \text{new bnd}[v_3 : \text{bnd} + v_4 : \text{bnd}] \\ v_6 \leftarrow \text{new bnd}[d : \text{dvar}] \\ v_7 \leftarrow \text{new bnd}[v_5 : \text{bnd} \leq v_6 : \text{bnd}], \\ \pi = \{v_7.\text{UPDMIN}(\text{true})\} \\ \text{post } \pi \end{array} \right]$$

Like the previous model, this method requires a predefined collection of objects indexed by members of a bounded language $\mathcal{L}' \subset \mathcal{L}$. The difference is that these objects are now bnd views instead of propagators, and $\mathcal{L}'$ is defined as:

$$\mathcal{L}' = \{x : \text{dvar}\} \cup \bigcup_{\oplus \in \Sigma} x : \text{bnd} \oplus y : \text{bnd}$$

4.2 Infinite View Stores (IVS)

The decomposition model described above assumes the existence of a finite indexed collection of views. Replacing subexpressions by $x : \text{bnd}$ objects guarantees the existence of a view for each generated subexpression.

The present method takes the direct approach: it defines an infinite indexed collection of views. For this, it makes two key requirements. Firstly, a view $\lambda \langle X \rangle$ may be defined by providing specializations for partially instantiated members $x \in X$, also called *partial specializations*. Secondly, new specializations may be created not only in the compilation process as before, but also in the body of any specialization.

Example 4. The following is a partial specialization of the bnd view for addition of two arbitrary expressions:

$$\text{bnd}[X + Y] = \left\{ \begin{array}{l} \text{GETMIN} = \{r \leftarrow \text{bnd}[X].\text{GETMIN}() + \text{bnd}[Y].\text{GETMIN}()\} \\ \text{GETMAX} = \{r \leftarrow \text{bnd}[X].\text{GETMAX}() + \text{bnd}[Y].\text{GETMAX}()\} \\ \text{UPDMIN} = \left\{ \begin{array}{l} \text{bnd}[X].\text{UPDMIN}(i - \text{bnd}[Y].\text{GETMAX}()) \\ \text{bnd}[Y].\text{UPDMIN}(i - \text{bnd}[X].\text{GETMAX}()) \end{array} \right\} \\ \text{UPDMAX} = \left\{ \begin{array}{l} \text{bnd}[X].\text{UPDMAX}(i - \text{bnd}[Y].\text{GETMIN}()) \\ \text{bnd}[Y].\text{UPDMAX}(i - \text{bnd}[X].\text{GETMIN}()) \end{array} \right\} \end{array} \right\}$$

In this model, the compilation algorithm involves the recursive instantiation of views. This can be done simply by instantiating the view for the full expression:

$$\pi[X] = \{\text{bnd}[X].\text{UPDMIN}(\text{true})\} \quad (4)$$

Given that now the full expression is known to the view (i.e. operands are not abstractions like before), it is easy to design an algorithm which, for a specific instantiation, replaces all function calls with the corresponding definitions.

Example 5. Simplification of the GETMIN function of the bnd view over the $a + b + c$ expression:

$$\begin{aligned} \text{bnd}[a : \text{dvar} + b : \text{dvar} + c : \text{dvar}].\text{GETMIN} &= \\ &= \{r \leftarrow \text{bnd}[a : \text{dvar} + b : \text{dvar}].\text{GETMIN}() + \text{bnd}[c : \text{dvar}].\text{GETMIN}()\} = \\ &= \{r \leftarrow \text{bnd}[a : \text{dvar}].\text{GETMIN}() + \text{bnd}[b : \text{dvar}].\text{GETMIN}() + \lfloor D(c) \rfloor\} = \\ &= \{r \leftarrow \lfloor D(a) \rfloor + \lfloor D(b) \rfloor + \lfloor D(c) \rfloor\} \end{aligned}$$

In the present context, views are used very much like macros, effectively allowing creating specific propagators for arbitrary expressions which do not make use of function calls at execution.

Example 6. Propagator for the expression $a + b + c = d$ using repository R_{BND} :

$$\pi = \left\{ \begin{array}{l} D(d) \leftarrow D(d) \setminus [-\infty \dots \lfloor D(a) \rfloor + \lfloor D(b) \rfloor + \lfloor D(c) \rfloor - 1] \\ D(d) \leftarrow D(d) \setminus [\lceil D(a) \rceil + \lceil D(b) \rceil + \lceil D(c) \rceil + 1 \dots + \infty] \\ \text{etc.} \end{array} \right\}$$

5 Model Comparison

For the memory and runtime analysis below we will consider an arithmetic constraint involving n variables with uniform domain size d , with an unbalanced syntax tree, i.e. where each operator in the expression involves at least one variable. Figure 1 shows a fragment of the expression syntax tree.

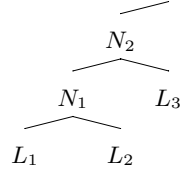


Fig. 1. A subtree of an unbalanced expression syntax tree. The internal nodes $N_1 \dots N_{n-1}$ represent operators and leafs $L_1 \dots L_n$ represent variables.

5.1 Memory

A view can be designed to expose just the subset of the expression's domain which is required for the view's client (e.g. the bounds of the expression). In contrast, a variable maintains the full domain of the expression, possibly containing regions which will always be ignored for propagation. For an expression containing $n - 1$ operators such as the expression in fig. 1, the auxiliary variables memory overhead is in $O(nD)$ compared to any of the views model, where D is the size of the largest domain of an auxiliary variable. For some expressions, namely an expression containing only multiplications, we can have $D = d^{n-1}$. In practice, the use of intervals to store the auxiliary variable's domains (i.e. $D = 2$) eliminates this problem.

5.2 Runtime

There is a fundamental operational difference between the views and the auxiliary variables model. A view computes its domain *on demand*, that is, it never updates its domain before it is needed by the view's client. This is the opposite of what happens in the auxiliary variable model, as the posted propagators will act on the auxiliary variable's domains independently.

This analysis focuses on the cost of accessing and updating the bounds of an expression, which may correspond to accessing/updating an auxiliary variable or a view, depending on the model. The cost is measured in terms of number of propagators executed, number of function calls, number of arithmetic operations, and number of domain updates.

We first note that enforcing bounds(R) consistency to an algebraic expression requires $O(nd)$ steps. As an example, consider a CSP with the constraint $x \times 2 = y \times 2 + 1$, where $D(x) = D(y) = [1 \dots d]$. See also [17].

We will first focus on the costs of updating the bounds of an expression whose syntax tree resembles fig. 1. The auxiliary variables model associates a propagator and an

auxiliary variable with each internal node N_i of the expression syntax tree. The propagator for a node N_1 involving two leafs may execute $O(2d)$ times and cause the same number of updates in the domain of the corresponding auxiliary variable. This means that the propagator for N_2 may execute $O(3d)$ times and so on for all the $n - 1$ internal nodes. The number of propagators executed for the auxiliary variables model is therefore $O(n^2d)$. Since each propagator execution requires a constant number of arithmetic operations, the number of operations for this model is also $O(n^2d)$. Finally, given that each propagation implies the update of the domain of an auxiliary variable, the number of domain updates is $O(n^2d)$.

Both models using views have only one propagator for the entire expression, which can be executed at most $O(nd)$ times. Unfortunately, a single update of the expression is now more costly because it may require evaluating the full tree, which is $O(n)$. This means that the total number of operations is still $O(nd \times n)$. However, given that there are no auxiliary variables in these models, the number of domain updates is only $O(nd)$. We remark that the only difference between both views is in the number of function calls.

Accessing the bounds of an expression is cheaper in the auxiliary variables model. This is because the domain of each subexpression is *cached* in the associated auxiliary variable. In contrast, both models involving views require evaluating the expression. Table 1 summarizes these results.

Table 1. Cost of accessing and updating an arbitrary expression represented by each of the described models

model	propagators	functions		operations		updates
vars	-	$O(n^2d)$	$O(1)$	-	$O(1)$	$O(n^2d)$
fvs	-	$O(nd)$	$O(n)$	$O(n^2d)$	$O(n)$	$O(n^2d)$
ivs	-	$O(nd)$	$O(1)$	-	$O(n)$	$O(n^2d)$

5.3 Propagation

The consistency level achieved by view based propagators is detailed in [15], namely in terms of $\text{bounds}(Z)$ and $\text{bounds}(R)$. Here, we remark that using a variable decomposition may improve propagation w.r.t. views, in particular, to achieve a pruning between $\text{bounds}(Z)$ and $\text{bounds}(R)$. This may be seen in the following example:

Example 7. Consider the constraint $x \times y \times 2 = z$, where $D(x) = D(y) = [2 \dots 3]$, and $D(z) = [9 \dots 15]$. Given that $x = z / (y \times 2)$, $y = z / (x \times 2)$, we have

$$\begin{aligned} D(x) &= D(y) = [2 \dots 3] \cap [9 / (3 \times 2) \dots 15 / (2 \times 2)] = [2 \dots 3] \\ D(z) &= [9 \dots 15] \cap [2 \times 2 \times 2 \dots 3 \times 3 \times 2] = [9 \dots 15] \end{aligned}$$

which means that the constraint is at a fixpoint according to $\text{bounds}(R)$ (note that a $\text{bounds}(Z)$ propagator for this constraint infers $z = 12$). Now let us assume the decomposition $e \equiv x \times y$ and $e \times 2 = z$, with $D(e) = [4 \dots 9]$. Propagating the latter constraint infers

$$D(e) \leftarrow [4 \dots 9] \cap \left[\frac{9}{2} \dots \frac{15}{2} \right] = \left[\frac{9}{2} \dots \frac{15}{2} \right]$$

Using the auxiliary variables model associates an integer variable with e , imposing an additional restriction: it must be an integer. This allows more pruning from the domain of z :

$$\begin{aligned} D(t_1) &\leftarrow [4 \dots 9] \cap [5 \dots 7] = [5 \dots 7] \\ D(z) &\leftarrow [9 \dots 15] \cap 2 \times [5 \dots 7] = [10 \dots 14] \end{aligned}$$

Even if stronger propagation leads to smaller search spaces, in our experiments it did not compensate the inefficiencies of the variable decomposition model.

6 Experiments

The first experiment applies previous compilation models to a system of linear equations. This problem is usually solved in constraint programming by using a specific propagator for linear equations. The goal of the experiment is therefore to assess the overhead of decomposing/compiling expressions using the presented models compared to a decomposition which uses a special purpose algorithm. The second experiment considers systems of nonlinear equations. These problems arise often in practice, and since the decomposition to special purpose propagators is not so direct as in previous case, it provides a realistic opportunity to apply the previously discussed models. The last experiment consists in solving the Golomb ruler problem (prob6 in [11]), a common benchmark for CSP solvers.

The code for all the following experiments was compiled with the gcc-4.2.3 C++ compiler, and executed on an Intel Core 2 Duo @ 2.20GHz, using Linux-2.6.24. Each experiment was repeated until the standard deviation of the runtime was below 1% of the average time, and then the minimum runtime was used. The search trees explored by all models are the same for each instance.

All previously described models were implemented in CaSPER [6]. Additionally, we used Gecode-2.2.0 [9] on one problem as a reference.

6.1 Modeling

Each system of linear equations is described by a tuple $\langle n, d, c, a, s|u \rangle$ where n is the number of variables in the problem, d is the uniform domain size, c is the number of linear equations, a is the number of terms in each equation, and the last term denotes if the problem is (s)atisfiable or (u)nsatisfiable. Problems were generated randomly, and the most difficult instances were selected (based on number of fails). In addition to the previously described models (vars, fvs, and ivs) for this problem we also tested a model using a propagator for linear equations (linear). Table 2 shows the results.

Each system of nonlinear equations is described by a tuple $\langle n, d, c, a_1, a_2, s|u \rangle$ where a_1 is the number of terms in each equation, each term is composed of a product of a_2 factors, and all remaining variables have the same meaning as before. In this experiment we also tested a model where each product is decomposed using auxiliary variables, projected to a variable x_i , and a linear propagator is used to enforce $\sum_{i=1}^{a_1} x_i$ for each equation (linear+vars). Table 3 show the results.

In the Golomb ruler problem, the use of views allows saving a quadratic number of variables (one for each difference between marks). The results are shown in table 4.

Table 2. Runtime for finding all solutions or proving that no solution exists for problems consisting of systems of linear equations. For each result the table presents the runtime in seconds and the speed up compared to the worst performing model in that problem.

	20-7-7-4-s		20-30-6-8-u		40-7-12-20-u		40-7-10-40-u	
casper-vars	21.1	1.00	135	1.00	128	1.00	137	1.02
casper-fvs	20.5	1.03	90.9	1.48	87.0	1.47	140	1.00
casper-ivs	19.5	1.08	40.0	3.37	17.4	7.34	13.2	10.6
casper-linear	19.5	1.08	43.7	3.09	18.8	6.81	13.3	10.5

Table 3. Runtime for finding the first solution or proving that no solution exists for problems consisting of systems of nonlinear equations

	20-20-10-4-2-s		50-10-19-4-2-s		50-10-28-4-3-u	
casper-vars	49.1	1.00	126	1.00	85.2	1.37
casper-fvs	48.4	1.01	117	1.08	117	1.00
casper-ivs	22.9	2.14	55.7	2.26	43.1	2.70
casper-linear+vars	40.1	1.22	106	1.19	93.4	1.25
gecode-vars	36.6	1.34	98.2	1.28	72.7	1.60
gecode-linear+vars	26.8	1.83	70.2	1.79	65.7	1.77
	50-5-20-4-3-u		50-6-26-4-4-u		60-4-24-4-4-5-u	
casper-vars	47.1	1.05	63.4	1.84	1.29	1.86
casper-fvs	49.6	1.00	116	1.00	2.40	1.00
casper-ivs	18.8	2.64	31.2	3.73	0.68	3.52
casper-linear+vars	49.3	1.01	66.4	1.75	1.46	1.64
gecode-vars	40.4	1.23	52.2	2.23	1.18	2.03
gecode-linear+vars	32.4	1.53	47.2	2.47	1.11	2.16

Table 4. Number of propagations (x1000) (p), runtime (t), and speedup (s) for finding the solution to the Golomb ruler problem

	golomb-10			golomb-11			golomb-12		
	p	t	s	p	t	s	p	t	s
casper-vars	0.38	0.14	1	2.12	0.73	1	116	40.3	1
casper-fvs	0.05	0.11	1.26	0.24	0.65	1.11	11.9	35.8	1.13
casper-ivs	0.05	0.1	1.36	0.24	0.5	1.46	11.9	30.19	1.33

6.2 Discussion

As expected, the FVS model has a good performance on small problems, but it degrades for larger expressions. It can be one order of magnitude slower than IVS on linear equations, and two times slower than the auxiliary variables decomposition on other problems. The performance of the IVS model is significantly better than the other methods. In fact it achieves similar performance of the model using global propagators on linear equations (table 2), which shows the potential of the IVS model to seemingly derive propagators for other global constraints over algebraic expressions.

Comparing to Gecode we notice a small difference regarding the runtimes for the same models favoring the Gecode implementation. We think this discrepancy is probably due to some optimization of Gecode not present in CaSPER, and that applying the IVS model to Gecode should yield similar gains in performance. We note that the IVS model still outperforms the best model we could implement in Gecode using the available modeling primitives in some instances.

7 Related Work

The idea of using type polymorphism for defining generic propagators is incipient in [6] and [14]. The former presents a general overview of a constraint solver incorporating type polymorphism, and gives a first description on some of the data structures required for creating generic propagators from arbitrary expressions. The latter coins the term *view* (we originally called them *polymorphic constraints*), and describes how it can be used for creating generic propagator implementations, that is, propagators which can be reused for different constraints. [15] provides further analysis, in particular, it is shown for which consistencies using views achieves less pruning than a specialized propagator.

[1] focus on the complexity of propagation for different decomposition models, specifically through symbolic manipulation of expressions, such as replacing common subexpressions and factorizing. In this context, the authors show that the propagation resulting from the decomposition using auxiliary variables is stronger than propagation on the original expression. It turns out that introducing auxiliary variables may strengthen propagation also without any symbolic manipulation (see section 5.3).

Constraint solvers deal with constraints over arbitrary algebraic expressions in different ways. Most prolog systems [16,8,7] automatically decompose them, apparently using the auxiliary variables model. The use of indexicals and goal expanded constraints [3] is advised as a way to avoid introducing auxiliary variables, and should be comparable to the IVS model (without the optimizations available to strongly typed languages). ILog Solver processes arbitrary expressions bottom-up, converting its subexpressions to *constrained expressions* [12], following the FVS model. Choco [5] features automatic decomposition using auxiliary variables.

The problem of creating propagators for arbitrary expressions has been approached in a different way using knowledge compilation techniques [10,4]. These methods create a compact extensional representation of the set of solutions to the constraint and apply a general propagation algorithm which filters tuples not found in this set. Applying these algorithms for arithmetic expressions is possible, and not so inefficient in practice concerning runtime. However even the most compact extensional representation is exponential in the number of variables in the worst case, while all the decompositions we presented above are linear.

8 Conclusion

This paper presents an approach for creating view based propagators for arbitrary algebraic expressions *at the modeling phase*. The performance of the resulting model compares favourably, both theoretically and experimentally, with models resulting from

other decomposition/compilation methods. The method may be integrated in any constraint solver implemented in a type parametric programming language, which is often the case (e.g. [5,12,9]).

References

1. Apt, K.R., Zoetewij, P.: A comparative study of arithmetic constraints on integer intervals. In: Apt, K.R., Fages, F., Rossi, F., Szeredi, P., Váncza, J. (eds.) CSCLP 2003. LNCS(LNAI), vol. 3010, pp. 1–24. Springer, Heidelberg (2003)
2. Bessiere, C.: Constraint Propagation. In: Handbook of Constraint Programming. Foundations of Artificial Intelligence, pp. 29–83. Elsevier Science, Amsterdam (2006)
3. Carlson, B.: Compiling and executing finite domain constraints. PhD thesis, Uppsala University (1995)
4. Cheng, K.C.K., Yap, R.H.C.: Maintaining generalized arc consistency on ad hoc r -ary constraints. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 509–523. Springer, Heidelberg (2008)
5. Choco. Choco constraint programming system (2003), <http://www.emn.fr/x-info/choco-solver/doku.php?id=>
6. Correia, M., Barahona, P., Azevedo, F.: Casper: A programming environment for development and integration of constraint solvers. In: Azevedo, F. (ed.) Proceedings of the First International Workshop on Constraint Programming Beyond Finite Integer Domains, BeyondFD 2005 (2005)
7. Diaz, D., Codognet, P.: Design and implementation of the gnu prolog system. Journal of Functional and Logic Programming 2001(6) (October 2001)
8. ECLiPSe. ECLiPSe prolog (2008), <http://www.eclipse-clp.org/>
9. Gecode. Gecode: Generic constraint development environment (2006), <http://www.gecode.org>
10. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: AAAI, pp. 191–197. AAAI Press, Menlo Park (2007)
11. Gent, I.P., Walsh, T.: Csplib: a benchmark library for constraints. Technical report, APES-09-1999 (1999), <http://www.csplib.org/>
12. ILOG. Ilog solver 6.0: Reference manual, 2003.
13. Rossi, F., Van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Foundations of Artificial Intelligence. Elsevier Science, Amsterdam (2006)
14. Schulte, C., Tack, G.R.: Views and iterators for generic constraint implementations. In: Hnich, B., Carlsson, M., Fages, F., Rossi, F. (eds.) CSCLP 2005. LNCS (LNAI), vol. 3978, pp. 118–132. Springer, Heidelberg (2006)
15. Schulte, C., Tack, G.R.: Perfect derived propagators. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 571–575. Springer, Heidelberg (2008)
16. SICStus. SICStus Prolog 3.12 User's Manual, 3.12 edn. (October 2006), <http://www.sics.se/sicstus/>
17. Yuanlin, Z., Yap, R.H.C.: Arc consistency on n -ary monotonic and linear constraints. In: Principles and Practice of Constraint Programming, pp. 470–483 (2000)