Aspartame: Solving Constraint Satisfaction Problems with Answer Set Programming

M. Banbara[1], M. Gebser[2], K. Inoue[3], T. Schaub*[2], T. Soh[1], N. Tamura[1], and M. Weise[2]

[1] University of Kobe
[2] University of Potsdam
[3] National Institute of Informatics Tokyo

Abstract. Encoding finite linear CSPs as Boolean formulas and solving them by using modern SAT solvers has proven to be highly effective, as exemplified by the award-winning *sugar* system. We here develop an alternative approach based on ASP. This allows us to use first-order encodings providing us with a high degree of flexibility for easy experimentation with different implementations. The resulting system *aspartame* re-uses parts of *sugar* for parsing and normalizing CSPs. The obtained set of facts is then combined with an ASP encoding that can be grounded and solved by off-the-shelf ASP systems. We establish the competitiveness of our approach by empirically contrasting *aspartame* and *sugar*.

1 Introduction

Encoding finite linear Constraint Satisfaction Problems (CSPs; [1, 2]) as propositional formulas and solving them by using modern solvers for Satisfiability Testing (SAT; [3]) has proven to be a highly effective approach, as demonstrated by the award-winning *sugar*[4] system. The CSP solver *sugar* reads a CSP instance and transforms it into a propositional formula in Conjunctive Normal Form (CNF). The translation relies on the order encoding [4, 5], and the resulting CNF formula can be solved by an off-the-shelf SAT solver.

In what follows, we elaborate upon an alternative approach based on Answer Set Programming (ASP; [6]) and present the resulting CSP solver *aspartame*[5]. The major difference between *sugar* and *aspartame* rests upon the implementation of the translation of CSPs into Boolean constraint problems. While *sugar* implements a translation into CNF in the imperative programming language JAVA, *aspartame* starts with a translation into a set of facts.[6] In turn, these facts are combined with a general-purpose ASP encoding for CSP solving (also based on the order encoding), which is subsequently instantiated by an off-the-shelf ASP grounder. The resulting propositional logic program is then solved by an off-the-shelf ASP solver.

---

* Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada, and the Institute for Integrated and Intelligent Systems at Griffith University, Brisbane, Australia.

[4] `http://bach.istc.kobe-u.ac.jp/sugar`

[5] `http://www.cs.uni-potsdam.de/wv/aspartame`

[6] In practice, *aspartame* re-uses *sugar*'s front-end for parsing and normalizing CSPs.

The high-level approach of ASP has obvious advantages. First, instantiation is done by general-purpose ASP grounders rather than dedicated implementations. Second, the elaboration tolerance of ASP allows for easy maintenance and modifications of encodings. And finally, it is easy to experiment with novel or heterogeneous encodings. However, the intruding question is whether the high-level approach of *aspartame* matches the performance of the more dedicated *sugar* system. We empirically address this question by contrasting the performance of both CSP solvers, while fixing the back-end solver to *clasp*, used as both a SAT and an ASP solver.

From an ASP perspective, we gain insights into advanced modeling techniques for solving CSPs. The ASP encoding implementing CSP solving with *aspartame* has the following features:

- usage of function terms to abbreviate structural subsums
- avoidance of (artificial) intermediate Integer variables (to break sum expressions)
- order encoding applied to structural subsum variables (as well as input variables)
- encoding-wise filtering of relevant threshold values (no blind usage of domains)
- customizable "pigeon-hole constraint" encoding for alldifferent constraints
- "smart" encoding of table constraints, tracing admissible tuples along arguments

In the sequel, we assume some familiarity with ASP, its semantics as well as its basic language constructs. A comprehensive treatment of ASP can be found in [6], one oriented towards ASP solving is given in [7]. Our encodings are given in the language of *gringo* 3 [8]. Although we provide essential definitions of CSPs in the next section, we refer the reader to the literature [1, 2] for a broader perspective.

2 Background

A *Constraint Satisfaction Problem* (CSP) is given by a pair $(\mathcal{V},\mathcal{C})$ consisting of a set $\mathcal{V}$ of *variables* and a set $\mathcal{C}$ of *constraint clauses*. Every variable $x \in \mathcal{V}$ has an associated finite *domain* $D(x)$ such that either $D(x) = \{\top,\bot\}$ or $\emptyset \subset D(x) \subseteq \mathbb{Z}$; x is a *Boolean variable* if $D(x) = \{\top,\bot\}$, and an *Integer variable* otherwise. We denote the set of Boolean variables in $\mathcal{V}$ by $\mathcal{B}(\mathcal{V})$ and the set of Integer variables in $\mathcal{V}$ by $\mathcal{I}(\mathcal{V})$. A constraint clause $C \in \mathcal{C}$ is a set of literals over Boolean variables in $\mathcal{B}(\mathcal{V})$ as well as linear inequalities or global constraints on Integer variables in $\mathcal{I}(\mathcal{V})$. Any *literal* in C is of the form e or $\overline{e}$, where e is either a Boolean variable in $\mathcal{B}(\mathcal{V})$, a linear inequality, or a global constraint. A *linear inequality* is an expression $\sum_{1\leq i\leq n} a_i x_i \leq m$ in which m as well as all a_i for $1 \leq i \leq n$ are Integer constants and $x_1,\dots,x_n$ are Integer variables in $\mathcal{I}(\mathcal{V})$. A *global constraint* (cf. [9]) is an arbitrary relation over Integer variables in $\mathcal{I}(\mathcal{V})$; we here restrict ourselves to table and alldifferent constraints over subsets $\{x_1,\dots,x_n\}$ of the Integer variables in $\mathcal{I}(\mathcal{V})$, where a *table constraint* specifies tuples $(d_1,\dots,d_n) \in D(x_1) \times \cdots \times D(x_n)$ of admitted value combinations and *alldifferent* applies if $x_1,\dots,x_n$ are assigned to distinct values in their respective domains.[7]

[7] Linear inequalities relying on further comparison operators, such as $<$, $>$, $\geq$, $=$, and $\neq$, can be converted into the considered format via appropriate replacements [5]. Moreover, note that we here limit the consideration of global constraints to the ones that are directly, i.e., without normalization by *sugar*, supported in our prototypical ASP encodings shipped with *aspartame*.

Given a CSP $(\mathcal{V}, \mathcal{C})$, a *variable assignment* v is a (total) mapping $v : \mathcal{V} \rightarrow \bigcup_{x \in \mathcal{V}} D(x)$ such that $v(x) \in D(x)$ for every $x \in \mathcal{V}$. A Boolean variable $x \in \mathcal{B}(\mathcal{V})$ is *satisfied* w.r.t. v if $v(x) = \top$. Likewise, a linear inequality $\sum_{1 \leq i \leq n} a_i x_i \leq m$ is *satisfied* w.r.t. v if $\sum_{1 \leq i \leq n} a_i v(x_i) \leq m$. Table constraints $e \subseteq D(x_1) \times \cdots \times D(x_n)$ and alldifferent constraints over subsets $\{x_1, \ldots, x_n\}$ of $\mathcal{I}(\mathcal{V})$ are *satisfied* w.r.t. v if $(v(x_1), \ldots, v(x_n)) \in e$ or $v(x_i) \neq v(x_j)$ for all $1 \leq i < j \leq n$, respectively. Any Boolean variable, linear inequality, or global constraint that is not satisfied w.r.t. v is *unsatisfied* w.r.t. v. A constraint clause $C \in \mathcal{C}$ is *satisfied* w.r.t. v if there is some literal $e \in C$ (or $\overline{e} \in C$) such that e is satisfied (or unsatisfied) w.r.t. v. The assignment v is a *solution* for $(\mathcal{V}, \mathcal{C})$ if every $C \in \mathcal{C}$ is satisfied w.r.t. v.

Example 1. Consider a CSP $(\mathcal{V}, \mathcal{C})$ with Boolean and Integer variables $\mathcal{B}(\mathcal{V}) = \{b\}$ and $\mathcal{I}(\mathcal{V}) = \{x, y, z\}$, where $D(x) = D(y) = D(z) = \{1, 2, 3\}$, and constraint clauses $\mathcal{C} = \{C_1, C_2, C_3\}$ as follows:

$$C_1 = \{\mathit{alldifferent}(x, y, z)\} \tag{1}$$

$$C_2 = \{b, 4x - 3y + z \leq 0\} \tag{2}$$

$$C_3 = \left\{\overline{b}, (x, y) \in \{(1, 3), (2, 2), (3, 1)\}\right\} \tag{3}$$

The alldifferent constraint in C_1 requires values assigned to x, y, and z to be mutually distinct. Respective assignments v satisfying the linear inequality $4x - 3y + z \leq 0$ in C_2 include $v(x) = 2$, $v(y) = 3$, and $v(z) = 1$ or $v(x) = 1$, $v(y) = 3$, and $v(z) = 2$, while the table constraint in C_3 is satisfied w.r.t. assignments v containing $v(x) = 1$, $v(y) = 3$, and $v(z) = 2$ or $v(x) = 3$, $v(y) = 1$, and $v(z) = 2$. In view of the Boolean variable b, whose value allows for "switching" between the linear inequality in C_2 and the table constraint in C_3, we obtain the following solutions $v_1, \ldots, v_4$ for $(\mathcal{V}, \mathcal{C})$:

	b	x	y	z
v_1	$\bot$	2	3	1
v_2	$\bot$	1	3	2
v_3	$\top$	1	3	2
v_4	$\top$	3	1	2

3 Approach

The *aspartame* tool extends the SAT-based solver *sugar* by an output component to represent a CSP in terms of ASP facts. The generated facts can then, as usual, be combined with a first-order encoding processable with off-the-shelf ASP systems. In what follows, we describe the format of facts generated by *aspartame*, and we present a dedicated ASP encoding utilizing function terms to capture substructures in CSP instances.

3.1 Fact Format

Facts express the variables and constraints of a CSP instance in the syntax of ASP grounders like *gringo* [8]. Their format is easiest explained on the CSP from Example 1, whose fact representation is shown in Listing 1. While facts of the predicate `var`/2

```
1  var(bool,b).  var(int,x;y;z,range(1,3)).

3  constraint(1,global(alldifferent,arg(x,arg(y,arg(z,nil))))).
4  constraint(2,b).
5  constraint(2,op(le,op(add,op(add,op(mul,4,x),op(mul,-3,y)),op(mul,1,z)),0)).
6  constraint(3,op(neg,b)).
7  constraint(3,rel(r,arg(x,arg(y,nil)))).

9  rel(r,2,3,supports).
10 tuple(r,1,1,1).  tuple(r,1,2,3).
11 tuple(r,2,1,2).  tuple(r,2,2,2).
12 tuple(r,3,1,3).  tuple(r,3,2,1).
```

Listing 1. Facts representing the CSP from Example 1.

provide labels of Boolean variables like b, the predicate `var`/3 includes a third argument for declaring the domains of Integer variables like x, y, and z. Domain declarations rely on function terms `range(`l`,`u`)`, standing for continuous Integer intervals $[l, u]$. While one term, `range(1,3)`, suffices for the common domain $\{1, 2, 3\}$ of x, y, and z, in general, several intervals can be specified (via separate facts) to form non-continuous domains. Note that the interval format for Integer domains offers a compact fact representation of (continuous) domains; e.g., the single term `range(1,10000)` captures a domain with 10000 elements. Furthermore, the usage of meaningful function terms avoids any need for artificial labels to refer to domains or parts thereof.

The literals of constraint clauses are also represented by means of function terms. In fact, the second argument of `constraint`/2 in Line 3 of Listing 1 stands for $\mathit{alldifferent}(x, y, z)$ from the constraint clause C_1 in (1), which is identified via the first argument of `constraint`/2. Since every fact of the predicate `constraint`/2 is supposed to describe a single literal only, constraint clause identifiers establish the connection between individual literals of a clause. This can be observed on the facts in Line 4–7, specifying literals belonging to the binary constraint clauses C_2 and C_3 in (2) and (3). Here, the terms `b` and `op(neg,b)` refer to the literals b and $\overline{b}$ over Boolean variable b, where `op(neg,`e`)` is the general notation of $\overline{e}$ for all (supported) constraint expressions e. The more complex term of the form `op(le,`Σ`,`m`)` in Line 5 stands for a linear inequality $\Sigma \leq m$. In particular, the inequality $4x - 3y + z \leq 0$ from C_2 is represented by nested `op(add,`Σ`,`ax`)` terms whose last argument ax and deepest Σ part are of the form `op(mul,`a`,`x`)`; such nesting corresponds to the precedence $(((4{*}x){+}(-3{*}y)){+}(1{*}z)) \leq 0$. The representation by function terms captures linear inequalities of arbitrary arity and, as with Integer intervals, associates (sub)sums with canonical labels. Currently, the order of arguments ax is by variable labels x, while more "clever" orders may be established in the future.

The function terms expressing table and alldifferent constraints both include an argument list of the form `arg(`x_1`,arg(`$\ldots$`,arg(`x_n`,nil)`$\ldots$`))`, in which $x_1, \ldots, x_n$ refer to Integer variables. In Line 3 of Listing 1, an alldifferent constraint over arguments $\boldsymbol{x}$ is declared via `global(alldifferent,`$\boldsymbol{x}$`)`; at present, `alldifferent` is a fixed keyword in facts generated by *aspartame*, but support for other kinds of global constraints can be added in the future. Beyond an argument list $\boldsymbol{x}$, function terms of the form `rel(`r`,`$\boldsymbol{x}$`)` also include an identifier r referring to a collection of table constraint tuples. For instance, the corresponding argument `r` in Line 7 ad-

dresses the tuples specified by the facts in Line 9–12. Here, `rel(r,2,3,supports)` declares that `r` is of arity 2 and includes 3 tuples, provided as white list entries via facts of the form `tuple(r,`t`,`i`,`d`)`. The latter include tuple and argument identifiers t and i along with a value d. Accordingly, the facts in Line 10, 11, and 12 specify the pairs $(1,3)$, $(2,2)$, and $(3,1)$ of values, which are the combinations admitted by the table constraint from C_3 in (3). The application of the table constraint to variables x and y is expressed by the argument list in Line 7, so that tuple declarations can be re-used for other variables subject to a similar table constraint.

3.2 First-Order Encoding

In addition to an output component extending *sugar* for generating ASP facts, *aspartame* comes along with alternative first-order ASP encodings of solutions for CSP instances. In the following, we sketch a dedicated encoding that, for one, relies on function terms to capture recurrences of similar structures and, for another, lifts the order encoding approach to structural subsum entities.

Static Extraction of Relevant Values To begin with, Listing 2 shows (relevant) instances of domain predicates, evaluated upon grounding, for the CSP from Example 1. While derived facts in Line 1 merely provide a projection of the predicate `var`/3 omitting associated domains, the instances of `look`/2 in Line 2 express that all values in the common domain $\{1,2,3\}$ of x, y, and z shall be considered. In fact, domain predicates extract variable values that can be relevant for the satisfiability of a CSP instance, while discarding the rest. The respective static analysis consists of three stages: (i) isolation of threshold values relevant to linear inequalities; (ii) addition of missing values for variables occurring in alldifferent constraints; (iii) addition of white/black list values for table constraints.

In the first stage, we consider the domains of Integer variables x in terms of corresponding (non-overlapping) intervals $I(x) = \{[l_1,u_1],\dots,[l_k,u_k]\}$. These are extended to multiplications by Integer constants a according to the following scheme:

$$I(ax) = \begin{cases} \{[a*l_1, a*u_1],\dots,[a*l_k, a*u_k]\} & \text{if } 0 \le a \\ \{[a*u_k, a*l_k],\dots,[a*u_1, a*l_1]\} & \text{if } a < 0 \end{cases}$$

For $4x - 3y + z \le 0$ from C_2 in (2), we get $I(4x) = \{[4,12]\}$, $I(-3y) = \{[-9,-3]\}$, and $I(1z) = \{[1,3]\}$. Such intervals are used to retrieve bounds for (sub)sums:

$$\begin{aligned} \overrightarrow{l}(ax) &= \min\{l \mid [l,u] \in I(ax)\} \\ \overrightarrow{u}(ax) &= \max\{u \mid [l,u] \in I(ax)\} \\ \overrightarrow{l}(a_1x_1 + a_2x_2) &= \overrightarrow{l}(a_1x_1) + \overrightarrow{l}(a_2x_2) \\ \overrightarrow{u}(a_1x_1 + a_2x_2) &= \overrightarrow{u}(a_1x_1) + \overrightarrow{u}(a_2x_2) \end{aligned}$$

Given $\overrightarrow{l}(4x) = 4$, $\overrightarrow{u}(4x) = 12$, $\overrightarrow{l}(-3y) = -9$, $\overrightarrow{u}(-3y) = -3$, $\overrightarrow{l}(1z) = 1$, and $\overrightarrow{u}(1z) = 3$, we derive $\overrightarrow{l}(4x - 3y) = -5$, $\overrightarrow{u}(4x - 3y) = 9$, $\overrightarrow{l}(4x - 3y + z) = -4$, and $\overrightarrow{u}(4x - 3y + z) = 12$.

In view of the comparison with 0 in $4x-3y+z \leq 0$, we can now "push in" relevant thresholds via:

$$
\begin{aligned}
\overleftarrow{l}(\textstyle\sum_{1\leq i\leq n} a_i x_i) &= \max\{m, \overrightarrow{l}(\textstyle\sum_{1\leq i\leq n} a_i x_i)\} && \text{for } \textstyle\sum_{1\leq i\leq n} a_i x_i \leq m \\
\overleftarrow{u}(\textstyle\sum_{1\leq i\leq n} a_i x_i) &= \min\{m, \overrightarrow{u}(\textstyle\sum_{1\leq i\leq n} a_i x_i)\} && \text{for } \textstyle\sum_{1\leq i\leq n} a_i x_i \leq m \\
\overleftarrow{l}(\textstyle\sum_{1\leq i\leq n-1} a_i x_i) &= \max\{\overleftarrow{l}(\textstyle\sum_{1\leq i\leq n} a_i x_i) - \overrightarrow{u}(a_n x_n), \overrightarrow{l}(\textstyle\sum_{1\leq i\leq n-1} a_i x_i)\} \\
\overleftarrow{u}(\textstyle\sum_{1\leq i\leq n-1} a_i x_i) &= \min\{\overleftarrow{u}(\textstyle\sum_{1\leq i\leq n} a_i x_i) - \overrightarrow{l}(a_n x_n), \overrightarrow{u}(\textstyle\sum_{1\leq i\leq n-1} a_i x_i)\}
\end{aligned}
$$

Such threshold analysis leads to $\overleftarrow{l}(4x-3y+z) = \overleftarrow{u}(4x-3y+z) = 0$, $\overleftarrow{l}(4x-3y) = -3$, $\overleftarrow{u}(4x-3y) = -1$, $\overleftarrow{l}(4x) = 4$, and $\overleftarrow{u}(4x) = 8$, telling us that subsums relevant for checking whether $4x-3y+z \leq 0$ satisfy $-3 \leq 4x-3y \leq -1$ and $4 \leq 4x \leq 8$. Note that maxima (or minima) used to construct $\overleftarrow{l}(\sum_{1\leq i\leq n} a_i x_i)$ (or $\overleftarrow{u}(\sum_{1\leq i\leq n} a_i x_i)$) serve two purposes. For one, they correct infeasible arithmetical thresholds to domain values; e.g., $\overleftarrow{l}(4x-3y) - \overrightarrow{u}(-3y) = -3+3 = 0$ tells us that 0 would be the greatest lower bound to consider for $4x$ (since $4x-3y+z \leq 0$ were necessarily satisfied when $4x \leq 0$), while the smallest possible value $\overrightarrow{l}(4x) = 4$ exceeds 0. For another, dominating values like $4x = 12$ are discarded, given that $4x-3y+z \leq 0$ cannot hold when $4x > \overleftarrow{u}(4x-3y) - \overrightarrow{l}(-3y) = -1+9 = 8$.

Letting $ub(0) = \{0\}$, the upper bounds for $\sum_{1\leq i\leq n} a_i x_i$ that deserve further consideration are then obtained as follows:

$$
\begin{aligned}
ub(\textstyle\sum_{1\leq i\leq n} a_i x_i) = \{&\max\{j + a_n{*}k, \overleftarrow{l}(\textstyle\sum_{1\leq i\leq n} a_i x_i)\} \mid j \in ub(\textstyle\sum_{1\leq i\leq n-1} a_i x_i), \\
&k \in \mathbb{Z}, [l,u] \in I(a_n x_n), l \leq a_n{*}k \leq \min\{u, \overleftarrow{u}(\textstyle\sum_{1\leq i\leq n} a_i x_i) - j\}\}
\end{aligned}
$$

Starting from the above thresholds, $ub(4x) = \{4, 8\}$, $ub(4x-3y) = \{-3,-2,-1\}$, and $ub(4x-3y+z) = \{0\}$ indicate upper bounds for subsums that are of interest in evaluating $4x-3y+z \leq 0$. Upper bounds in $ub(\sum_{1\leq i\leq n} a_i x_i)$ can in turn be related to "maximal" pairs of addends:

$$
\begin{aligned}
S(\textstyle\sum_{1\leq i\leq n} a_i x_i) = \{&(j, \max\{a_n{*}k \mid [l,u] \in I(a_n x_n), l \leq a_n{*}k \leq \min\{u, ub - j\}, \\
&k \in \mathbb{Z}\}) \mid j \in ub(\textstyle\sum_{1\leq i\leq n-1} a_i x_i), ub \in ub(\textstyle\sum_{1\leq i\leq n} a_i x_i), \overrightarrow{l}(a_n x_n) \leq ub - j\}
\end{aligned}
$$

In our example, we get $S(4x) = \{(0,4),(0,8)\}$, $S(4x-3y) = \{(4,-9),(4,-6),(8,-9)\}$, and $S(4x-3y+z) = \{(-3,3),(-2,2),(-1,1)\}$.

Finally, we associate each pair $(j, a_n{*}k) \in S(\sum_{1\leq i\leq n} a_i x_i)$ of addends with the upper bound $s(j, a_n{*}k) = \min\{ub \in ub(\sum_{1\leq i\leq n} a_i x_i) \mid j + a_n{*}k \leq ub\}$, thus obtaining $s(0,4) = 4$, $s(0,8) = 8$, $s(4,-9) = -3$, $s(4,-6) = -2$, $s(8,-9) = -1$, and $s(-3,3) = s(-2,2) = s(-1,1) = 0$.

The described analysis of thresholds for subsums is implemented via deterministic domain predicates in our ASP encoding. Variables' domain values underlying relevant addends are provided by the derived facts in Line 10–12 of Listing 2. Note that value 3 for x as well as 1 for y are ignored here, given that $4x = 12$ and $-3y = -3$ do not admit $4x-3y+z \leq 0$ to hold. The mapping of relevant addends to their associated

```
1  var(int,x;y;z).
2  look(x;y;z,1;2;3).

4  order(x;y;z,3,2).
5  order(x;y;z,2,1).

7  order(op(add,op(mul,4,x),op(mul,-3,y)),-1,-2).
8  order(op(add,op(mul,4,x),op(mul,-3,y)),-2,-3).

10 look(op(mul,4,x),1;2,1).
11 look(op(mul,-3,y),2;3,-1).
12 look(op(mul,1,z),1;2;3,1).

14 look(op(add,op(mul,4,x),op(mul,-3,y)),4,-6,-2).
15 look(op(add,op(mul,4,x),op(mul,-3,y)),4,-9,-3).
16 look(op(add,op(mul,4,x),op(mul,-3,y)),8,-9,-1).

18 look(op(add,op(add,op(mul,4,x),op(mul,-3,y)),op(mul,1,z)),-1,1,0).
19 look(op(add,op(add,op(mul,4,x),op(mul,-3,y)),op(mul,1,z)),-2,2,0).
20 look(op(add,op(add,op(mul,4,x),op(mul,-3,y)),op(mul,1,z)),-3,3,0).

22 bound(op(add,op(add,op(mul,4,x),op(mul,-3,y)),op(mul,1,z)),12).

24 difind(arg(x,arg(y,arg(z,nil))),x,1).
25 difind(arg(x,arg(y,arg(z,nil))),y,2).
26 difind(arg(x,arg(y,arg(z,nil))),z,3).
27 difmax(arg(x,arg(y,arg(z,nil))),3,1;2;3).
28 difall(arg(x,arg(y,arg(z,nil)))).

30 relind(r,arg(x,arg(y,nil)),x,1).
31 relind(r,arg(x,arg(y,nil)),y,2).
```

Listing 2. Domain predicates derived via stratified rules (not shown) from facts in Listing 1.

upper bound can be observed in Line 14–20 for the (sub)sums $4x - 3y$ and $(4x - 3y) + z$. The respective facts describe patterns for mapping assigned domain values to their multiplication results and then to upper bounds for subsums, which are eventually subject to a (non-trivial) comparison in some linear inequality. (Trivial comparisons are performed via the total upper bound for an addition result, as given in Line 22.) Notably, the static threshold analysis is implemented on terms representing the domains of variables, and outcomes are then mapped back to original variables. Thus, linear inequalities over different variables with the same domains are analyzed only once. The final function terms, however, mention the variables whose values are evaluated, where recurring substructures may share a common term with which all relevant threshold values are associated.

Although the analysis of the linear inequality $4x - 3y + z \leq 0$ identifies the values 3 for x and 1 for y as redundant, the presence of $alldifferent(x, y, z)$ leads to their "release" as relevant candidates for x and y. Accordingly, all values in the common domain $\{1, 2, 3\}$ of x, y, and z are put into (decreasing) order, given by the derived facts in Line 4–5. Beyond that, the order among relevant upper bounds in $ub(4x - 3y) = \{-3, -2, -1\}$ is reflected in Line 7–8; this is used to apply the order encoding to structural subsum variables (in addition to the input variables x, y, and z). The residual derived facts in Line 24–31 serve convenience by associating indexes to the arguments of $alldifferent(x, y, z)$ as well as to x and y considered in $(x, y) \in \{(1, 3), (2, 2), (3, 1)\}$. Furthermore, the fact in Line 27 indicates the index 3 of variable z in $alldifferent(x, y, z)$ as the final position at which either of the

```
1   % generate variable assignment

3   { less(V,E) : order(V,E,_) } :- var(int,V).
4    :- order(V,E1,E2), less(V,E2), not less(V,E1).
5   value(V,E) :- look(V,E), not less(V,E), less(V,EE) : order(V,EE,E).

7   { value(V,true) } :- var(bool,V).
8   value(V,false) :- var(bool,V), not value(V,true).

10  % evaluate linear inequalities

12  leq(op(mul,F,V),F*E) :- look(op(mul,F,V),E,1), less(V,EE) : order(V,EE,E).
13  leq(op(mul,F,V),F*E) :- look(op(mul,F,V),E,-1), not less(V,E).
14  leq(op(add,S1,S2),E) :- look(op(add,S1,S2),E1,E2,E), leq(S1,E1;;S2,E2).
15  leq(op(add,S1,S2),E) :- order(op(add,S1,S2),E,EE), leq(op(add,S1,S2),EE).

17  % evaluate alldifferent expressions

19  seen(A,I,E) :- difind(A,V,I), value(V,E).
20  seen(A,I,E) :- difind(A,_,I), seen(A,I-1,E), not difmax(A,I-1,E).

22  redo(A) :- difind(A,V,I), seen(A,I-1,E), value(V,E).
23  redo(A) :- difall(A), difmax(A,I,E), not seen(A,I,E).

25  % evaluate table expressions

27  rela(R,A,T,2)   :- relind(R,A,V,1), tuple(R,T,1,E), value(V,E).
28  rela(R,A,T,I+1) :- relind(R,A,V,I), tuple(R,T,I,E), value(V,E), rela(R,A,T,I).

30  rela(R,A,U) :- rela(R,A,_,I+1), rel(R,I,_,U).

32  % check constraint clauses

34  hold(C) :- constraint(C,V), value(V,true).
35  hold(C) :- constraint(C,op(neg,V)), value(V,false).

37  hold(C) :- constraint(C,op(le,S,E)), bound(S,U), leq(S,E) : E < U.
38  hold(C) :- constraint(C,op(neg,op(le,S,E))), bound(S,U), E < U, not leq(S,E).

40  hold(C) :- constraint(C,global(alldifferent,A)), not redo(A).
41  hold(C) :- constraint(C,op(neg,global(alldifferent,A))), redo(A).

43  hold(C) :- constraint(C,rel(R,A)), not rela(R,A,conflicts),
44             rela(R,A,supports) : rel(R,_,_,supports).
45  hold(C) :- constraint(C,op(neg,rel(R,A))), not rela(R,A,supports),
46             rela(R,A,conflicts) : rel(R,_,_,conflicts).

48  constraint(C) :- constraint(C,_).
49   :- constraint(C), not hold(C).

51  % display variable assignment

53  #hide.
54  #show value/2.
```

Listing 3. First-order encoding of solutions for finite linear CSPs.

values 1, 2, or 3 can possibly be assigned, and the fact in Line 28 expresses that all three values in $D(x) \cup D(y) \cup D(z) = \{1, 2, 3\}$ must be assigned in order to satisfy $\mathit{alldifferent}(x, y, z)$.

Non-deterministic Encoding Part With the described domain predicates at hand, the encoding part in Listing 3 implements the non-deterministic guessing of a variable as-

signment along with the evaluation of constraint clauses. Following the idea of order encodings in SAT [4, 5], the choice rule in Line 3 permits guessing `less(V,E)` for all but the smallest (relevant) value `E` in the domain of an Integer variable `V`, thus indicating that `V` is assigned to some smaller value than `E`. The consistency among guessed atoms is established by the integrity constraint in Line 4, requiring `less(V,E1)` to hold if `less(V,E2)` is true for the (immediate) predecessor value `E2` of `E1`. The actual value assigned to `V`, given by the greatest `E` for which `less(V,E)` is false, is extracted in Line 5. For Boolean variables, the value `true` can be guessed unconditionally via the choice rule in Line 7, and `false` is derived otherwise via the rule in Line 8.

The dedicated extension of the order encoding idea to subsums of linear inequalities is implemented by means of the rules in Line 12–15 of Listing 3. To this end, upper bounds for singular multiplication results indicated as relevant by instances of `look(op(mul,F,V),E,G)` are directly derived from `less/2`. Thereby, the flag `G = F/|F|` provides the polarity of the actual coefficient `F`.[8] If `F` is positive, i.e., `G = 1`, the upper bound `F*E` is established as soon as `less(V,EE)` holds for the immediate successor value `EE` of `E` (or if `E` is the greatest relevant value in the domain of `V`). On the other hand, if `G = -1` indicates that `F` is negative, the upper bound `F*E` is derived from `not less(V,E)`, which means that the value assigned to `V` is greater than or equal to `E`. Relevant upper bounds `E` for subsums rely on maximal pairs (`E1`,`E2`) of addends, identified via static threshold analysis and readily provided by instances of `look(op(add,S1,S2),E1,E2,E)`. In fact, the rule in Line 14 derives `leq(op(add,S1,S2),E)`, indicating that $\texttt{S1} + \texttt{S2} \leq \texttt{E}$, from `leq(S1,E1)` and `leq(S2,E2)`. Although an established upper bound inherently implies any greater (relevant) upper bound to hold as well w.r.t. a total variable assignment, ASP (and SAT) solvers are not committed to guessing "input variables" first. Rather, structural variables like the instances of `leq(op(add,S1,S2),E)` may be fixed upon solving, possibly in view of recorded conflict clauses, before a total assignment has been determined. In view of this, the additional rule in Line 15 makes sure that an established upper bound `EE` propagates to its immediate successor `E` (if there is any). For instance, (simplified) ground instances of the rule stemming from $ub(4x - 3y) = \{-3, -2, -1\}$ include the following:

```
leq(op(add,op(mul,4,x),op(mul,-3,y)),-1) :-
    leq(op(add,op(mul,4,x),op(mul,-3,y)),-2).
leq(op(add,op(mul,4,x),op(mul,-3,y)),-2) :-
    leq(op(add,op(mul,4,x),op(mul,-3,y)),-3).
```

Unlike with the domains of Integer variables, we rely on a rule, rather than an integrity constraint, to establish consistency among the bounds for structural subsums. The reason for this is that upper bounds for addends `S1` and `S2`, contributing left and right justifications, may include divergent gaps, so that consistent value orderings for them are, in general, not guaranteed to immediately produce all relevant upper bounds for $\texttt{S1} + \texttt{S2}$. Encoding variants resolving this issue and using integrity constraints like the one in Line 4 are a subject to future investigation.

While linear inequalities can be evaluated by means of boundaries derived more or less directly from instances of `less(V,E)`, the evaluation of alldifferent and table

[8] Coefficients given in facts generated by *aspartame* are distinct from 0.

constraints in Line 19–23 and Line 27–30 of Listing 3 relies on particular instances of `value(V,E)`. The basic idea of checking whether an alldifferent constraint holds is to propagate assigned values along the indexes of participating variables. Then, a recurrence is detected when the value assigned to a variable with index `I` has been marked as already assigned, as determined from `seen(A,I-1,E)` in Line 22. Moreover, whenever `difall(A)` indicates that all domain values for the variables in argument list `A` must be assigned, the rule in Line 23 additionally derives a recurrence from some gap (a value that has not been assigned to the variable at the last possible index). Our full encoding further features so-called "pigeon-hole constraints" (cf. [10, 11]) to check that the smallest or greatest $1, \dots, n-1$ domain values for an alldifferent constraint with n variables are not populated by more than i variables for $1 \leq i \leq n-1$. Such conditions can again be checked based on instances of `less(V,E)`, and both counter-based (cf. [12]) as well as aggregate-based (cf. [13]) implementations are applicable in view of the native support of aggregates by ASP solvers like *clasp* (cf. [14]). In fact, the usage of rules to express redundant constraints, like the one in Line 23 or those for pigeon-hole constraints, as well as their ASP formulation provide various degrees of freedom, where comprehensive evaluation and configuration methods are subjects to future work.

The strategy for evaluating table constraints is closely related to the one for detecting value recurrences in alldifferent constraints. Based on the indexes of variables in a table constraint, tuples that are (still) admissible are forwarded via the rules in Line 27–28. The inclusion of a full tuple in an assignment is detected by the rule in Line 30, checking whether the arity `I` of a table constraint has been reached for some tuple, where a value `supports` or `conflicts` for `U` additionally indicates whether the included tuple belongs to a white or black list, respectively. Note that this strategy avoids explicit references to variables whose values are responsible for the exclusion of tuples, given that lack of inclusion is detected from incomplete tuple traversals.

Finally, the rules in Line 34–49 explore the values assigned to Boolean variables and the outcomes of evaluating particular kinds of constraints to derive `hold(C)` if and only if some positive or negative literal in `C` is satisfied or unsatisfied, respectively, w.r.t. the variable assignment represented by instances of `value(V,E)`. Without going into details, let us still note that our full encoding also features linear inequalities relying on the comparison operators $\geq$, $=$, and $\neq$, for which additional rules are included to derive `hold(C)`, yet sticking to the principle of upper bound evaluation via `leq`/2. In fact, the general possibility of complemented constraint expressions as well as of disjunctions potentially admits unsatisfied constraint expressions w.r.t. solutions, and our encoding reflects this by separating the evaluation of particular constraint expressions in Line 12–30 from further literal and clause evaluation in Line 34–49.

4 The *aspartame* System

The architecture of the *aspartame* system is given in Figure 1. As mentioned, *aspartame* re-uses *sugar*'s front-end for parsing and normalizing CSPs. Hence, it accepts the same input formats, viz. XCSP[9] and *sugar*'s native CSP format[10]. We then implemented an

[9] http://www.cril.univ-artois.fr/CPAI08/XCSP2_1.pdf
[10] http://bach.istc.kobe-u.ac.jp/sugar/package/current/docs/syntax.html

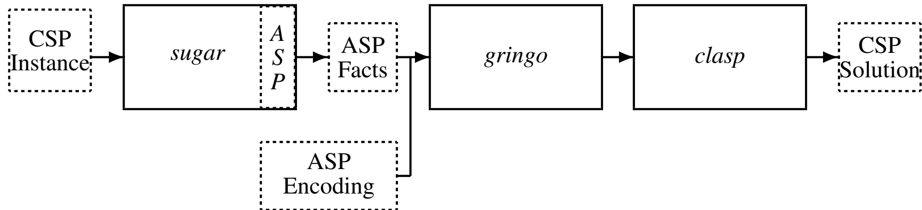

Fig. 1. Architecture of *aspartame*.

output hook for *sugar* that provides us with the resulting CSP instance in the fact format described in Section 3.1. These facts are then used for grounding the (full version of the) dedicated ASP encoding in Listing 3 or an alternative one (discussed below). This is done by the ASP grounder *gringo*. In turn, the resulting propositional logic program is passed to the ASP solver *clasp* that returns an assignment, representing a solution to the original CSP instance.

We empirically access the performance of *aspartame* relative to two ASP encodings, the dedicated one described in Section 3.2 as well as a more direct encoding inspired by the original CNF construction of *sugar* [5], and additionally consider the SAT-based reference solver *sugar* (2.0.0). In either case, we use the combined ASP and SAT solver *clasp* (2.1.0), and ASP-based approaches further rely on *gringo* (3.0.5) for grounding ASP encodings on facts generated by *aspartame*. We selected 60 representative CSP instances (that are neither too easy nor too hard), consisting of intensional and global constraints, from the benchmarks of the 2009 CSP Competition[11] for running systematic experiments on a cluster of Linux machines equipped with dual Xeon E5520 quad-core 2.26 GHz processors and 48 GB RAM. To get some first insights into suitable search options, we ran *clasp* with its default (berkmin-like) and the popular "vsids" decision heuristic; while SAT-based preprocessing (cf. [15]) is performed by default on CNF inputs, we optionally enabled it for (ground) ASP instances, leading to four combinations of *clasp* settings for ASP-based approaches and two for SAT-based *sugar*.

Table 1 reports runtime results in seconds, separated into conversion time of *aspartame* from CSP instances to ASP facts (first "convert" column) and of *sugar* from CSP instances to CNF, *gringo* times for grounding ASP encodings relative to facts, and finally columns for the search times of *clasp* with the aforementioned options. Each computational phase was restricted to 600 seconds, and timeouts counted in the last row of Table 1 are taken as 600 seconds in the second last row providing average runtimes. Looking at these summary rows, we observe that our two ASP encodings are solved most effectively when vsids decision heuristic and SAT preprocessing are both enabled; unlike this, neither decision heuristic dominates the other on CNF input. Apparently, *clasp* on CNFs generated by *sugar* still has a significant edge on facts by *aspartame* combined with either ASP encoding. In particular, we observe drastic performance discrepancies on some instance families (especially "fischer" and "queensKnights"), where *clasp* performs stable on CNFs from *sugar* but runs into trouble on corresponding ASP instances. Given that *aspartame* and its ASP encodings are prototypes, such behavior does not disprove the basic approach, but rather motivates future

[11] `http://www.cril.univ-artois.fr/CPAI09`

Table 1. Experiments comparing ASP encoding variants and the SAT-based solver *sugar*.

		ASP Encoding 1 (dedicated)					*ASP Encoding 2 (SAT-inspired)*					*sugar*		
Benchmark	convert	ground	default	vsids	sat-pre	vsids/ sat-pre	ground	default	vsids	sat-pre	vsids/ sat-pre	convert	default	vsids
1-fullins-5-5	2.02	1.41	13.96	11.28	5.15	3.66	0.91	10.72	12.15	6.72	7.18	1.73	2.40	2.19
3-fullins-5-6	3.55	32.39	17.07	14.90	21.75	6.52	11.26	13.08	16.52	11.53	14.02	5.36	1.91	1.50
4-fullins-4-7	2.20	4.45	28.35	45.59	22.02	28.38	2.29	20.03	39.92	28.60	42.91	2.80	2.19	4.81
abb313GPIA-7	5.02	46.97	0.67	1.85	0.71	2.01	20.33	31.75	2.27	41.61	1.63	7.18	6.05	0.05
abb313GPIA-8	5.14	51.23	411.17	460.56	521.11	TO	21.97	173.06	451.02	433.31	180.33	7.23	131.41	70.57
abb313GPIA-9	5.96	52.79	TO	TO	TO	TO	24.68	243.72	TO	289.54	TO	8.49	0.59	5.73
bibd-8-98-49-4-21_glb	3.68	28.30	TO	20.70	4.47	1.42	28.59	10.52	18.11	5.36	8.41	11.28	1.65	1.49
bibd-10-120-36-3-8_glb	5.20	69.94	46.68	11.62	8.21	1.13	76.65	15.84	4.55	2.47	7.93	13.63	7.28	0.85
bibd-25-25-9-9-3_glb	7.34	84.68	TO	434.23	3.50	1.13	17.53	235.11	177.27	TO	59.41	8.90	28.86	41.12
bibd-31-31-6-6-1_glb	9.74	254.13	159.32	3.34	156.71	14.64	39.84	12.24	0.18	0.17	0.11	17.39	77.95	0.20
C2-3-15	0.88	3.30	0.21	0.13	0.15	0.08	9.74	6.03	4.88	6.63	8.24	1.64	7.37	1.05
C4-1-61	1.44	24.44	4.66	2.86	4.90	2.80	148.33	TO	TO	TO	TO	3.40	3.06	9.87
C4-2-61	2.96	26.04	6.26	1.98	6.16	2.04	152.33	222.52	69.47	227.54	165.80	3.22	7.05	3.21
C5-3-91	2.73	71.55	31.42	10.32	31.87	11.91	TO	TO	TO	TO	TO	7.89	261.39	TO
chnl-10-11	0.37	0.18	0.64	1.20	4.36	11.44	0.14	1.35	2.65	12.40	13.75	0.92	19.58	55.26
chnl-10-15	0.70	0.17	9.15	3.08	6.95	10.61	0.19	4.04	20.14	10.82	18.01	0.51	32.07	24.68
chnl-10-20	0.56	0.24	19.14	6.91	6.51	5.22	0.33	41.96	38.34	6.81	11.98	1.11	7.77	6.30
chnl10-15-pb-cnf-cr	0.42	0.42	9.28	3.05	6.93	10.74	0.20	4.00	20.05	10.90	16.43	1.12	31.85	24.87
costasArray-14	0.46	0.82	3.54	2.93	3.65	4.87	0.62	18.63	18.19	24.79	6.52	2.03	0.25	0.07
costasArray-15	0.52	0.94	17.96	60.17	75.50	17.95	0.78	62.10	136.40	4.96	8.83	1.59	18.41	8.16
costasArray-16	0.52	1.20	15.13	30.18	69.69	0.99	1.03	130.75	203.94	245.02	36.31	1.71	32.65	33.25
costasArray-17	0.56	1.50	TO	TO	TO	232.64	1.28	TO	TO	TO	TO	2.08	148.55	553.54
fischer-1-2-fair	1.44	304.21	11.21	92.81	11.42	18.79	28.50	23.81	7.81	TO	TO	3.42	0.97	0.02
fischer-2-3-fair	4.18	348.84	0.85	21.11	0.79	6.50	68.42	23.19	3.69	23.71	3.54	7.19	0.13	0.04
fischer-3-8-fair	30.18	515.69	TO	458.58	TO	60.32	376.33	TO	TO	TO	TO	35.16	22.05	37.27
fischer-4-6-fair	35.12	535.93	282.84	242.05	386.37	274.27	384.19	TO	TO	TO	TO	39.15	17.16	8.79
fischer-6-1-fair	4.40	457.55	0.91	12.16	0.90	14.68	150.78	TO	10.29	TO	5.46	13.20	1.79	2.69
magicSquare-6_glb	0.29	2.72	46.20	1.23	4.27	1.49	4.20	9.50	2.42	18.39	1.77	2.26	0.79	0.46
magicSquare-7_glb	0.28	8.04	7.54	78.17	1.15	30.18	15.63	51.08	9.59	51.00	7.41	3.00	4.10	2.20
magicSquare-8_glb	0.42	21.75	TO	318.69	477.11	47.22	53.61	331.09	19.14	124.79	36.60	6.29	5.50	3.72
mps-mzzv42z	4.84	166.01	2.74	1.54	2.61	1.02	395.20	8.56	5.44	8.75	5.20	24.84	5.44	3.28
mps-p2756	1.74	278.02	4.98	11.89	5.04	5.29	TO	TO	TO	TO	TO	161.39	1.32	1.20
mps-red-air06	11.59	272.57	139.86	381.61	34.78	25.27	TO	TO	TO	TO	TO	27.73	0.63	TO
mps-red-fiber	1.28	92.78	5.07	3.78	5.34	5.39	388.43	33.55	TO	33.43	TO	38.74	2.05	6.76
queensKnights-50-5-add	1.81	16.86	18.62	24.71	44.44	18.79	28.12	558.69	85.48	537.34	44.58	3.25	12.15	0.66
queensKnights-50-5-mul	3.50	17.31	40.78	48.54	38.57	20.64	30.33	221.08	67.39	TO	50.76	3.29	2.10	0.63
queensKnights-80-5-mul	2.47	77.52	181.70	354.74	172.33	383.73	126.32	TO	426.53	TO	414.97	5.81	7.61	2.51
queensKnights-100-5-add	3.49	163.50	TO	TO	TO	TO	243.06	TO	TO	TO	TO	6.92	18.45	4.39
ramsey-16-3	1.07	0.51	1.17	1.16	0.18	0.10	0.39	6.74	0.53	0.78	1.04	2.19	1.99	112.63
ramsey-30-4	2.13	5.91	150.64	180.79	28.56	26.47	2.75	198.83	98.97	82.69	32.39	2.35	8.09	8.67
ramsey-33-4	2.53	7.58	405.05	279.25	93.36	66.78	4.05	TO	193.49	173.67	109.45	3.13	32.02	39.82
ramsey-34-4	2.34	8.46	TO	TO	366.41	109.45	4.14	TO	TO	TO	317.08	2.78	67.41	38.71
ruler-34-9-a4	1.15	13.01	21.70	29.62	19.05	19.16	3.08	35.29	39.71	38.59	39.19	4.30	41.98	41.16
ruler-44-10-a4	1.56	37.60	367.84	233.65	302.54	280.79	8.39	567.20	542.78	TO	483.85	9.91	405.96	446.66
ruler-44-9-a4	1.04	23.41	182.48	124.14	172.79	118.37	6.18	351.14	13.41	102.36	34.25	6.61	TO	352.28
ruler-55-10-a3	1.08	5.89	TO	500.84	TO	TO	3.19	TO	TO	TO	TO	1.73	43.23	70.22
super-jobShop-e0ddr1-8	1.21	6.42	555.34	19.21	14.32	3.67	3.66	6.10	0.11	6.92	1.05	1.16	1.21	0.49
super-jobShop-e0ddr2-1	1.08	7.59	TO	32.94	8.38	12.14	3.81	16.08	0.84	2.56	3.87	1.51	5.41	0.76
super-jobShop-enddr2-3	1.00	8.60	TO	116.06	76.55	12.93	3.95	13.21	2.59	5.30	6.33	2.15	1.19	0.72
super-os-taillard-7-4	1.02	35.84	TO	493.02	TO	454.02	15.35	36.02	36.35	35.06	32.14	2.08	1.14	0.89
super-os-taillard-7-6	1.01	35.51	115.70	TO	112.53	TO	14.49	169.99	30.25	144.26	25.89	2.14	4.20	0.76
super-os-taillard-7-7	0.84	32.73	155.31	270.49	132.71	311.67	13.53	23.55	17.77	25.74	15.78	1.95	0.80	0.78
super-os-taillard-7-8	0.95	33.23	431.15	279.09	475.51	328.50	14.46	46.95	25.64	40.83	24.06	2.01	1.19	0.96
zeroin-i-1-10	1.97	2.41	20.92	27.16	14.97	16.18	1.74	22.09	28.81	26.62	18.56	2.17	3.48	2.57
zeroin-i-3-10	1.76	1.92	7.24	33.14	11.43	16.93	1.52	13.45	29.43	19.72	31.67	2.22	4.57	4.11
ii-32c4	3.73	21.92	0.24	0.02	0.03	0.03	56.69	0.02	0.01	0.03	0.04	21.67	8.77	0.76
ii-32d3	2.80	11.87	4.58	13.70	0.39	0.26	25.12	3.54	0.31	0.54	1.35	12.33	71.17	0.20
p2756	2.06	269.82	3.87	12.55	3.99	6.13	TO	TO	TO	TO	TO	162.72	4.26	2.41
ooo-burch-dill-3-accl-ucl	2.46	14.06	13.80	18.60	6.13	5.95	5.82	30.05	24.18	12.73	4.61	2.77	1.28	1.00
ooo-tag14	6.93	191.71	145.03	144.09	158.56	158.99	44.99	109.54	310.78	20.82	25.11	10.67	6.54	6.07
Average Time	3.51	80.21	188.67	149.80	129.06	103.87	91.49	209.46	184.50	218.43	169.76	12.56	37.47	54.27
Timeouts	0	0	12	5	6	5	4	14	13	17	13	0	1	2

investigations of the reasons for performance discrepancies. For one, we conjecture that normalizations of global constraints that are not yet supported by *aspartame* are primarily responsible for large instance sizes and long search times on some instance families. For another, we suppose that both of our ASP encodings are still quite naive compared to years of expertise manifested in *sugar*'s CNF construction. However, the observation that our dedicated ASP encoding has on edge the SAT-inspired one and yields significant performance improvements on some instance families ("C2-3-15"–"C5-3-91" and "mps") clearly encourages further investigations into ASP encodings of CSP instances.

5 Related Work

Unlike approaches to constraint answer set solving, e.g., [10, 16–18], which aim at integrating CSP and ASP solving (engines), the focus of *aspartame* lies on pure CSP solving. In fact, *aspartame*'s approach can be regarded as a first-order alternative to SAT-based systems like *sugar* [5], where the performance of the underlying SAT solver is crucial. However, it is now becoming recognized that the SAT encoding to be used also plays an important role [19]. There have been several proposals of encoding constraints to SAT: direct encoding [20, 21], support encoding [22, 23], log encoding [24, 25], log support encoding [26], regular encoding [27], order encoding [4, 5], and compact order encoding [28].

The order encoding, where Boolean variables represent whether $x \leq i$ holds for variables x and values i, showed good performance for a wide range of CSPs [4, 11, 27, 29–34]. Especially, the SAT-based constraint solver *sugar* became a winner in global constraint categories at the 2008 and 2009 CSP solver competitions [35]. Moreover, the SAT-based CSP solver BEE [36] and the CLP system B-Prolog [37] utilize the order encoding. In fact, the order encoding provides a compact translation of arithmetic constraints, while also maintaining bounds consistency by unit propagation. Interestingly, it has been shown that the order encoding is the only existing SAT encoding that can reduce tractable CSP to tractable SAT [38].

6 Conclusion

We presented an alternative approach to solving finite linear CSPs based on ASP. The resulting system *aspartame* relies on high-level ASP encodings and delegates both the grounding and solving tasks to general-purpose ASP systems. We have contrasted *aspartame* with its SAT-based ancestor *sugar*, which delegates only the solving task to off-the-shelf SAT solvers, while using dedicated algorithms for constraint preprocessing. Although *aspartame* does not fully match the performance of *sugar* from a global perspective, the picture is fragmented and leaves room for further improvements. This is to say that different performances are observed on distinct classes of CSPs, comprising different types of constraints. Thus, it is an interesting topic of future research to devise more appropriate ASP encodings for such settings. Despite all this, *aspartame* demonstrates that ASP's general-purpose technology allows to compete with state-of-the-art constraint solving techniques, not to mention that *aspartame*'s intelligence is driven by an ASP encoding of less than 100 code lines (for non-deterministic predicates subject

to search). In fact, the high-level approach of ASP facilitates extensions and variations of first-order encodings for dealing with particular types of constraints. In the future, we thus aim at more exhaustive investigations of encoding variants, e.g., regarding alldifferent constraints, as well as support for additional kinds of global constraints.

Acknowledgments This work was partially funded by the Japan Society for the Promotion of Science (JSPS) under grant KAKENHI 24300007 as well as the German Science Foundation (DFG) under grant SCHA 550/8-3 and SCHA 550/9-1. We are grateful to the anonymous reviewers for many helpful comments.

References

1. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers (2003)
2. Rossi, F., van Beek, P., Walsh, T., eds.: Handbook of Constraint Programming. Elsevier Science (2006)
3. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. IOS Press (2009)
4. Crawford, J., Baker, A.: Experimental results on the application of satisfiability algorithms to scheduling problems. In Hayes-Roth, B., and Korf, R., eds.: Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94), AAAI Press (1994) 1092–1097
5. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. Constraints **14**(2) (2009) 254–272
6. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
7. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan and Claypool Publishers (2012)
8. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`.[12]
9. Beldiceanu, N., Simonis, H.: A constraint seeker: Finding and ranking global constraints from examples. In Lee, J., ed.: Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'11), Springer-Verlag (2011) 12–26
10. Drescher, C., Walsh, T.: A translational approach to constraint answer set solving. Theory and Practice of Logic Programming **10**(4-6) (2010) 465–480
11. Metodi, A., Codish, M., Stuckey, P.: Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. Journal of Artificial Intelligence Research **46** (2013) 303–341
12. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. In van Beek, P., ed.: Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP'05), Springer-Verlag (2005) 827–831
13. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
14. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: On the implementation of weight constraint rules in conflict-driven ASP solvers. [39] 250–264
15. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In Bacchus, F., Walsh, T., eds.: Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05), Springer-Verlag (2005) 61–75

[12] `http://potassco.sourceforge.net`

16. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. [39] 235–249
17. Balduccini, M.: Representing constraint satisfaction problems in answer set programming. In Faber, W., Lee, J., eds.: Proceedings of the Second Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09), (2009) 16–30
18. Ostrowski, M., Schaub, T.: ASP modulo CSP: The clingcon system. Theory and Practice of Logic Programming **12**(4-5) (2012) 485–503
19. Prestwich, S.: CNF encodings. [3] 75–97
20. de Kleer, J.: A comparison of ATMS and CSP techniques. In Sridharan, N., ed.: Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI'89), Morgan Kaufmann Publishers (1989) 290–296
21. Walsh, T.: SAT v CSP. In Dechter, R., ed.: Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP'00), Springer-Verlag (2000) 441–456
22. Kasif, S.: On the parallel complexity of discrete relaxation in constraint satisfaction networks. Artificial Intelligence **45**(3) (1990) 275–286
23. Gent, I.: Arc consistency in SAT. In van Harmelen, F., ed.: Proceedings of the Fifteenth European Conference on Artificial Intelligence (ECAI'02), IOS Press (2002) 121–125
24. Iwama, K., Miyazaki, S.: SAT-variable complexity of hard combinatorial problems. In Pehrson, B., Simon, I., eds.: Proceedings of the Thirteenth IFIP World Computer Congress (WCC'94), North-Holland (1994) 253–258
25. Van Gelder, A.: Another look at graph coloring via propositional satisfiability. Discrete Applied Mathematics **156**(2) (2008) 230–243
26. Gavanelli, M.: The log-support encoding of CSP into SAT. In Bessiere, C., ed.: Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP'07), Springer-Verlag (2007) 815–822
27. Ansótegui, C., Manyà, F.: Mapping problems with finite-domain variables into problems with Boolean variables. In Hoos, H., Mitchell, D., eds.: Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04), Springer-Verlag (2004) 1–15
28. Tanjo, T., Tamura, N., Banbara, M.: Azucar: A SAT-based CSP solver using compact order encoding (tool presentation). In Cimatti, A., Sebastiani, R., eds.: Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12), Springer-Verlag (2012) 456–462
29. Bailleux, O., Boufkhad, Y.: Efficient CNF encoding of Boolean cardinality constraints. In Rossi, F., ed.: Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03), Springer-Verlag (2003) 108–122
30. Gent, I., Nightingale, P.: A new encoding of AllDifferent into SAT. In Frisch, A., Miguel, I., eds.: Proceedings of the Third International Workshop on Modelling and Reformulating Constraint Satisfaction Problems (ModRef'04), (2004) 95–110
31. Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M., Tamura, N.: A competitive and cooperative approach to propositional satisfiability. Discrete Applied Mathematics **154**(16) (2006) 2291–2306
32. Soh, T., Inoue, K., Tamura, N., Banbara, M., Nabeshima, H.: A SAT-based method for solving the two-dimensional strip packing problem. Fundamenta Informaticae **102**(3-4) (2010) 467–487
33. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints **14**(3) (2009) 357–391
34. Banbara, M., Matsunaka, H., Tamura, N., Inoue, K.: Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers. In Fermüller, C., Voronkov, A., eds.: Proceedings of the Seventeenth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10), Springer-Verlag (2010) 112–126

35. Lecoutre, C., Roussel, O., van Dongen, M.: Promoting robust black-box solvers through competitions. Constraints **15**(3) (2010) 317–326
36. Metodi, A., Codish, M.: Compiling finite domain constraints to SAT with BEE. Theory and Practice of Logic Programming **12**(4-5) (2012) 465–483
37. Zhou, N.: The SAT compiler in B-prolog. The Association for Logic Programming Newsletter, March 2013 (2013)[13]
38. Petke, J., Jeavons, P.: The order encoding: From tractable CSP to tractable SAT. In Sakallah, K., Simon, L., eds.: Proceedings of the Fourteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'11), Springer-Verlag (2011) 371–372
39. Hill, P., Warren, D., eds.: Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09), Springer-Verlag (2009)

[13] http://www.cs.nmsu.edu/ALP/2013/03/the-sat-compiler-in-b-prolog