Noname manuscript No.
(will be inserted by the editor)

MiniBrass: Soft Constraints for MiniZinc

Alexander Schiendorfer · Alexander Knapp · Gerrit Anders · Wolfgang Reif

Received: date / Accepted: date

Abstract Over-constrained problems are ubiquitous in real-world decision and optimization problems. Plenty of modeling formalisms for various problem domains involving soft constraints have been proposed, such as weighted, fuzzy, or probabilistic constraints. All of them were shown to be instances of algebraic structures. In terms of modeling languages, however, the field of soft constraints lags behind the state of the art in classical constraint optimization. We introduce MiniBrass, a versatile soft constraint modeling language building on the unifying algebraic framework of partially ordered valuation structures (PVS) that is implemented as an extension of MiniZinc and MiniSearch.

We first demonstrate the adequacy of PVS to naturally augment partial orders with a combination operation as used in soft constraints. Moreover, we provide the most general construction of a c-semiring from an arbitrary PVS. Both arguments draw upon elements from category theory. MiniBrass turns these theoretical considerations into practice: It offers a generic extensible PVS type system, reusable implementations of specific soft constraint formalisms as PVS types, operators for complex PVS products, and morphisms to transform PVS. MiniBrass models are compiled into MiniZinc to benefit from the wide range of solvers supporting FlatZinc. We evaluated MiniBrass on 28 "softened" MiniZinc benchmark problems with six different solvers. The results demonstrate the feasibility of our approach.

Keywords Soft Constraints · Modeling Languages · MiniZinc

1 Introduction: From Algebraic Soft Constraints to Practical Solvers

Many (perhaps most) industrial combinatorial optimization problems in practice tend to be over-constrained, according to [33]. A most common remedy is to iteratively refine the initial constraint model by manually weakening or dropping constraints until a solution can be found. However, this approach does not work if the actual problem instance, i.e., all

The theoretical foundation of this work draws upon previous papers presented at ICTAI [57] and in an LNCS-Festschrift [55]. The research is partly sponsored by the German Research Foundation (DFG) in the project "OC-Trust" (FOR 1085).

Alexander Schiendorfer, Alexander Knapp, Gerrit Anders, Wolfgang Reif
University of Augsburg, Institute for Software & Systems Engineering
E-mail: {schiendorfer,knapp,anders,reif}@isse.de

input parameters, is only available at runtime. This is the case, for instance, when a system is intended to act autonomously (e.g., smart homes or smart grids). Simply failing with `unsatisfiable` is not an option – instead, a compromise solution is necessary. To accomplish this, a constraint model has to be written with the intention of *graceful degradation* in the first place. Some constraints have to be *softened* if necessary or even ignored.

A second driving force is the need to model users' *preferences* to discriminate the set of feasible solutions. In discrete optimization and decision theory, this role is assumed by the objective (or, utility), a function mapping variable assignments to some numeric codomain such as the rational numbers. The goal is to minimize or maximize the function's value. While many optimization problems feature a natural choice for a numeric objective (e.g., minimizing the makespan in scheduling, or minimizing the encompassing area in packing), adequately eliciting and modeling preferences in real-world settings is generally more involved and spans its own area of research [38]. The *essence* is an *ordering* relation over the available options, i.e., solutions.

Directly assessing this relation is impractical due to the exponential number of choices in a combinatorial setting [38]. Instead, several more compact preference formalisms for various use cases have been proposed. For example, preferences could be stated numerically as penalties incurred for violated soft constraints (as in weighted constraints [3]) or as degrees ranging from 0% to 100% satisfaction (as in fuzzy constraints [53]). Alternatively, one could state comparative *constraint preferences* [56], such as that constraint "two nurses should be on night shift" is more important than constraint "nurse Jones should be off-duty" if no solution satisfies both desirable constraints simultaneously (see Figure 1 for an example), or that the value "white wine" is more desirable than "red wine" for a variable "drink" if another variable "meal" is assigned value "fish", as in CP-nets [16]. For these comparative formalisms, inventing a numeric objective function that represents the modeler's ordering relation is a cognitively demanding task in its own right. If users fail to easily assign numeric values, this quantification should be distinct from the elicitation of preferences. Adequately modeling preferences then amounts to an exercise in "objective engineering".

Both motivations have been recognized for many years (see Section 2), leading to a unified theory of *soft constraints* that subsumes over-constrained problems and preferences [44]. It offers a more general treatment of satisfaction (or violation) degrees as an ordered set ac-

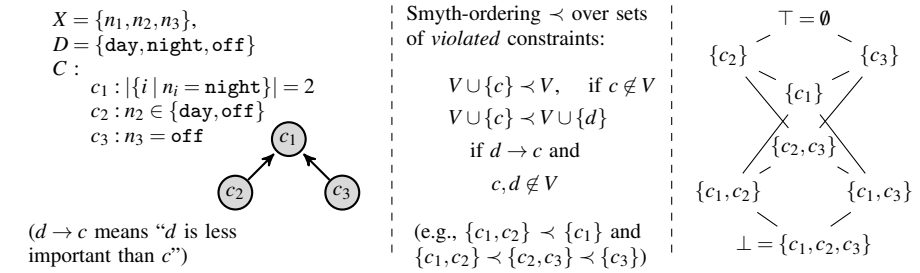


Figure 1 Left: A rostering problem involving three nurses n_i with (comparative) constraint preferences. Not all three constraints can be satisfied simultaneously, e.g., c_1 forces that either n_2 or n_3 take the night shift which conflicts with c_2 or c_3. There are solutions satisfying two out of three constraints. The graph depicts a partial ordering U of the constraints with c_1 being most important and c_2 being incomparable to c_3. Center: An ordering over sets of violated constraints defined inductively by the two above rules (called the *Smyth*-ordering). Right: The Hasse diagram of $\prec$ over the valuation space: No violation ($\emptyset$) is best, and, e.g., $\{c_2, c_3\}$ is better than $\{c_1, c_2\}$ since it violates the *less important* constraint c_3 instead of c_1.

companied by a combination operation (to combine the valuations of several soft constraints for an assignment) and dedicated top and bottom elements, i.e., an *algebraic structure*. Instead of working with well-known *specific* orderings, such as $(\mathbb{N}, \leq)$, calculations and orderings are studied from an *abstract* algebra perspective (see Figure 1 right for an example). The leading frameworks are *c(onstraint)-semirings* [13] and (totally ordered) *valuation structures* [58], i.e., ordered monoids. These abstractions serve to both find general complexity-theoretic results and devise search and propagation algorithms for a broad class of problems. Moreover, by means of product operators such as a direct product (for Pareto orderings) and a lexicographic product, complex valuation structures can be formed from elementary ones, allowing for modular specification and runtime combinations [55, 28]. Generalized variants of branch-and-bound, soft arc consistency [18], and non-serial dynamic programming techniques [10] such as bucket elimination or cluster tree elimination [22] use these structures. In addition, some global constraints (including `alldifferent` and `gcc`) with dedicated propagators have been generalized to a soft variant [33], usually by considering one (integer) cost variable that measures violation.

However, ready-to-use implementations for modern constraint platforms are rare – with Toulbar2 [3], a solver designed for (integer) cost function networks as a specific valuation structure, being the exception to the rule.[1] Using cost function networks is justified since many concrete formalisms can be reduced to them in polynomial time [58] – possibly at the expense of totalizing a partial order. Still, to use Toulbar2, the problem has to be encoded in a flat format, lacking the convenient high-level abstractions that constraint modeling languages such as Essence [27], OPL [32], or MiniZinc [45] provide. Consider, e.g., reusable predicates and (partial) functions [66], option types [43], or safe indexation with decision variables, to name a few.

For novice constraint modelers, crafting new soft constraint models is hard without language tools and the algorithmic potential of solvers remains restricted to expert users. To improve the situation for Toulbar2, a Python interface has recently been provided via the Numberjack platform [35]. On the other hand, to express soft constraints in conventional constraint models, constraint violations have to be reified in one or more variables to be, e.g., minimized in an ad-hoc fashion. In any case, users either have to use cost function networks or encode soft constraints in existing languages without any support. For practical purposes, previous efforts in designing abstract frameworks for a variety of soft constraint formalisms are thus, unfortunately, nullified.

Therefore, this paper reduces the gap between abstract soft constraint frameworks and practical solvers by exploiting and extending MiniZinc [45], a constraint modeling language and MiniSearch [49], a script language for customized MiniZinc searches, and contributes:

- **MiniBrass**[2] A modeling language for soft constraint problems based on algebraic structures that is compiled into MiniZinc/MiniSearch code to inherit its support for a broad variety of solvers. Our language comes with an *extensible* type system including common types (weighted constraints [60], cost function networks, fuzzy constraints [53], constraint preferences [56], probabilistic constraints [24] ...) in the literature.
- A formal foundation for the semantics of the types implemented in MiniBrass which includes the systematic derivation of partially-ordered valuation structures from partial

[1] Evidenced by the fact that at the Third International CSP Competition (CPAI'08), Toulbar2 was the only solver registered for the weighted CSP (WCSP) category – with none in the (last) '09 edition.

[2] MiniBrass pays tribute to the tradition of naming NICTA's G12 software after elements in the 12th group of the periodic table. *Brass* is an alloy containing *zinc* that is *softer* than zinc alone. Cold forming is only possible with brass alloys containing up to 37% zinc.

orders using category-theoretical arguments. In the course of the derivation, we survey the adequacy of abstract frameworks in the literature with respect to model expressiveness and algorithmic efficiency – with an emphasis on expressiveness. Our results show how to extend any partially-ordered valuation structure to a c-semiring, if needed.
- The *MiniBrass library* providing reusable order predicates and combination functions, implemented in MiniZinc with existing global constraints.
- An empirical evaluation using modified benchmark problems from the MiniZinc benchmark library that are supplemented with explicit soft constraints in different formalisms. We compare the solving performance of classical constraint solvers working on encoded soft constraint problems to that of a dedicated soft constraint solver (Toulbar2), the influence the used formalism has on solving time, and the efficiency of generic soft constraint search heuristics.

After giving a broad overview of related work in Section 2, we survey common definitions of soft constraint problems using algebraic structures in Section 3 and conclude that *partially ordered* valuation structures (PVS, [28]) are well-balanced in terms of generality and specificity. Therefore, we show how any partial order over solutions can be canonically transformed into a PVS with the "lowest overhead". This construction of the *free* PVS employs the language of category theory and has been presented, in brevity, in [40] without any rationale about the construction. Furthermore, we shed light on the relationship between PVS and c-semirings by exemplifying bucket elimination on the *free* c-semiring. Section 4 describes the design of the MiniBrass language including its type system, concepts, and toolchain. We provide use cases and examples of existing soft constraint formalisms in MiniBrass, including structured types such as the free PVS or constraint preferences, combinations of PVS, and morphisms. Section 5 concludes by evaluating MiniBrass on a set of (slightly modified) benchmark problems taken from the MiniZinc challenges [65]. A central question examines whether an encoding-based approach using a set-based ordering is competitive to integer minimization and dedicated soft constraint solving.

2 Related Work

Pioneering attempts to generalize hard constraints were discussed in *partial constraint satisfaction* [26]: A metric measures the distance of an assignment to the solution space of the original problem. Proposed distance choices included the number of required domain items to "patch" the assignment or the number of *violated constraints*. The latter is better known as *Max-CSP*. To distinguish constraints, various formalisms have been explored. In weighted constraint problems (WCSP), each constraint is assigned a fixed penalty that is incurred if it is violated. As mentioned before, Toulbar2 is a dedicated WCSP solver using search strategies (e.g., [2,63]) and soft constraint propagation and filtering [18]. Also, [6] offers WSimply, a specification language for weighted CSP with a transformation to SMT.

Similar to WCSP, constraints can also be placed qualitatively: either in layers of importance, such as in constraint hierarchies [15] or only comparatively, as done in the aforementioned constraint preferences (see Figure 1). [47] introduced the notion of meta-constraints to explicitly talk about constraints in other constraints, such as "B has to hold only if A is violated". Solvers provide reified variants for several cost values, MiniBrass relies on that technique. Although this seems as if we are restricted to relatively simple arithmetic constraints, recent efforts have been made to systematically provide more reified variants of global constraints [9].

Instead of placing weights on constraints themselves, fuzzy constraints [53] consider the constraints' relation of valid assignments as fuzzy sets with a membership function declaring how strongly an assignment satisfies the constraint, ranging from 0 to 1. Another proposal suggested to interpret soft constraints probabilistically, leading to *probabilistic constraints* [24]: For every soft constraint, we have a probability p_i that the constraint is *actually* present. An assignment θ is judged by the probability of it being an "actual solution". Assuming independence, we obtain this probability by multiplying $1 - p_i$ for all violated soft constraints (i.e., all violated constraints have to be absent in order for θ to still be a solution).

As mentioned in Section 1, *valued constraints* map an assignment to a totally ordered monoid (i.e., a valuation structure) [58]. Similarly, the *c-semiring* soft constraint frameworks labels each assignment with a value in a semiring. In addition to the multiplication operator, it foresees an additive operation acting as the supremum of the induced partial order (i.e., $a \leq b \leftrightarrow a + b = b$) for *comparing* solutions. Having a supremum operation available is particularly useful for variable elimination approaches, as we discuss in Section 3.5. However, many relevant partial orders do not admit a least upper bound such as, e.g., a Pareto-ordering of two orders or the Smyth-ordering presented in Figure 1.

To compare frameworks, [34] presented a encoding of a subset of constraint hierarchies as c-semirings. We characterized the missing type of constraint hierarchies by noting the presence of so-called "collapsing elements" (introduced by [28]) that are prohibitive for lexicographic products in [55]. However, this line of reasoning was purely theoretical and has not yet been implemented and evaluated empirically.

Concerning solvers for general semiring or valued constraints, we previously noted the lack thereof. Most papers offer closed ad-hoc implementations focusing on one particular type such as [52] or [12] for fuzzy CSP. By contrast, [42] provides a formulation of c-semiring-based soft constraint problems as "weighted semiring Max-SAT" that uses the semiring values and ordering as "weights". The encoded problems are solved with basic implementations of branch-and-bound and GSAT, outperforming the fuzzy solver CONFLEX (which is not available anymore). However, these algorithms do not rely on the supremum operator of a c-semiring and could be run as well with partial valuation structures (see Section 3.5). In addition, the approach remained rather prototypical (random instances with up to 120 variables and 20 constraints), only supported strict domination search for partially ordered search spaces (see Section 4.5), and did not offer a public API to their system – which brings us to modeling languages.

MiniBrass is built on top of the MiniZinc environment which itself is a subset of the Zinc language. There are variations and extensions such as stochastic MiniZinc [50] for problems involving uncertainties, MiningZinc for constraint-based data mining [30], and MiniSearch [49] for customizable search. MiniZinc is a high-level modeling language that is compiled to the flat file format FlatZinc that is understood by many constraint, MIP, or SAT solvers. MiniSearch provides facilities to access a search tree at the solution level, making queries such as "fetch the next solution; when found, constrain the next solution to have to improve" (in terms of, e.g., some partial order) – effectively resulting in a form of propagation-based branch-and-bound. For abstract soft constraint models, this is the right level of granularity – as opposed to a fine-tuned programmable search. Moreover, with MiniSearch, a search strategy has to be defined just once and can be used by any FlatZinc solver instead of implementing custom search for each solver. MiniSearch does so by generating multiple FlatZinc files. In addition, there is native search tree interaction for Gecode [59].

Probably closest to our approach, [5] proposed the higher-level language "w-MiniZinc" which would extend MiniZinc to weighted CSP. However, their approach ended up not be-

ing implemented (although they provide a similar concept with WSimply [6]) and did not involve other (abstract) types of soft constraints such as those subsumed by c-semirings. MiniBrass, by contrast, is designed to be easily extended with new types and puts a layer of abstraction on top of MiniZinc, being its target language of compilation. In addition, the syntactical features offered by w-MiniZinc are also available in MiniBrass (see Section 4.2.1) – along with more types and complex preference structures.

Other constraint modeling languages include Essence [27] or OPL [32]. While due to the existence of OPL script, OPL would be suited for a soft constraint modeling language as well, Essence does not foresee search combinators or programmable search. We could only work with repeated solver calls or numeric (integer) objectives. OPL, on the other hand, is tied to the CP/MIP solver IBM ILOG CPLEX whereas MiniZinc supports a broad variety of solvers – a property that turned out to be beneficial in our evaluation in Section 5.

Clearly, other areas study preferences with different emphases, ranging from game theory, databases [39], the social sciences [1], mechanism design [46] to multi-agent systems, in general [62]. Often, a preference relation is represented by numeric utilities that can be translates to weighted or fuzzy constraints. CP-nets [16] provide the most common *qualitative* preference language used in the above domains. Users specify total orders over the domain of a variable depending on an assignment to other variables. For instance, $x_1 = d_1, \ldots, x_n = d_n : y = w_1 \succ \cdots \succ y = w_k$ indicates that if variables x_i are assigned to d_i, then variable y should preferably be assigned w_i than w_{i+1}. By applying these rules transitively under a *ceteris paribus* assumption, generally a preorder over assignments is induced. In terms of solution ordering, it is well-known that soft constraints and CP-nets are formally incomparable [44]. Compared to *constraint* preferences, CP-nets require users to rank domain *values* whereas constraint preferences are settled on a coarser level: solutions satisfying an important constraint A are better than solutions satisfying a less important constraint B – everything else held equal. The former is obviously better suited in problems involving rather small domains whereas the latter aims at ordering a large number of solutions in equivalence classes of manageable size.

Regarding the specification and aggregation of preferences in multi-agent settings, (computational) social choice provides formal foundations by means of axiomatizing desirable properties and postulating appropriate voting rules [17,62]. Little attention has yet been devoted to the combination of social choice with soft constraint problems consisting of n preference structures [20] even though the prevalent heterogeneity calls for such approaches. As of now, MiniBrass only supports Pareto-style and lexicographic combinations but has voting-based aggregation as a future goal. By contrast, the overall objective in distributed constraint optimization problems (DCOP) is usually a sum of local cost functions which amounts to the special case of a weighted CSP [25] as opposed to more generic frameworks.

3 Formal Foundations: Soft Constraints and Algebraic Structures

We begin by reviewing our notation for conventional constraint satisfaction problems as well as soft constraint problems and then discuss how the algebraic structures underlying MiniBrass are obtained from partial orders. Although these sections are rather formal, the presented constructions and orders are implemented in the MiniBrass library (see Section 4).

3.1 Soft Constraint Satisfaction and Optimization on Partial Valuation Structures

As usual, a *constraint (satisfaction) problem* $CSP = (X, D, C)$ is described by a set of decision variables X, their associated family of domains $D = (D_x)_{x \in X}$ containing possible values, and a set of (hard) constraints C that restrict valid assignments. An assignment θ over scope X is a mapping from X to D, written as $\theta \in [X \to D]$, such that each variable x maps to a value in D_x. A (hard) constraint $c \in C$ is understood as a map $c : [X \to D] \to \mathbb{B}$ where we write $\theta \models c$ to express that θ satisfies c (i.e., $c(\theta) = \mathit{true}$) and $\theta \not\models c$ to express that θ violates c. Each constraint has a scope $sc(c) \subseteq X$, i.e., the variables that actually influence its truth value. An assignment θ is a solution if $\theta \models c$ holds for all $c \in C$. The restriction of an assignment θ to a scope $X' \subseteq X$ is explicitly written as $\theta{\downarrow}X'$.

We obtain *constraint optimization problems* (*COP*) by adding an objective function $f : [X \to D] \to P$ where $(P, \leq_P)$ is a partial order, that is, $\leq_P$ is a reflexive, antisymmetric, and transitive relation over P. Elements of P are interpreted as *solution degrees*, denoting quality. Without loss of generality, we interpret $m <_P n$ as solution degree m being strictly worse than n and restrict our attention to maximization problems. Hence, a solution degree m is *optimal* with respect to a constraint optimization problem *COP* if for all solutions θ it holds either that $f(\theta) \leq_P m$ or $f(\theta) \parallel_P m$, expressing incomparability w.r.t. $\leq_P$. It is *reachable* if there is a solution θ such that $f(\theta) = m$. Non-reachable optimal solution degrees appear, e.g., as upper bounds. A solution θ^* is optimal if $f(\theta^*)$ is optimal.

A *soft constraint satisfaction problem* (SCSP) is defined as a COP where i) the objective is decomposable into multiple objectives (i.e., soft constraints) defined on their respective scopes and ii) the codomain of the objective admits additional algebraic and ordered structure for modeling purposes, such as valuation structures [58] or c-semirings [13]. Minimal requirements are that solution degrees obtained from the soft constraints should be combined using a binary operation, called multiplication, that there should be a neutral element representing *complete satisfaction*, and that combination should be monotone with respect to multiplication to denote that additional violation can only harm the quality further. These properties are captured by *partially ordered* valuation structures (PVS).

Definition 1 (PVS) A PVS $(M, \cdot_M, \varepsilon_M, \leq_M)$ is a partially ordered commutative monoid with $\varepsilon_M \in M$ being both the neutral element w.r.t. $\cdot_M$ and the top element w.r.t. $\leq_M$. Further, $\cdot_M$ is monotone w.r.t. $\leq_M$.

A PVS M is *bounded* if there also exists a minimal element $\bot \in M$ to represent complete dissatisfaction. A valuation structure [58] is a bounded PVS where $\leq_M$ is a total ordering. If M and N are PVS, so are $M \times N$, the direct (Cartesian) product, and $M \ltimes N$, the lexicographic product – as long as some conditions on M hold, as was shown in [28,55]. Furthermore, to allow for structure-preserving mappings between PVS, we define a PVS-homomorphism from a PVS $(M, \cdot_M, \varepsilon_M, \leq_M)$ to a PVS $(N, \cdot_N, \varepsilon_N, \leq_N)$ to be given by a mapping $\varphi : M \to N$ such that $\varphi(\varepsilon_M) = \varepsilon_N$, $\varphi(m \cdot_M n) = \varphi(m) \cdot_N \varphi(n)$, and $m \leq_M n \Rightarrow \varphi(m) \leq_N \varphi(n)$ (order-preservation). For a c-semiring, we need an idempotent additive operation $\oplus$ that is used to induce the ordering $\leq$ by letting $m \leq n \leftrightarrow m \oplus n = n$. Due to generality, we first restrict our attention to PVS-based soft constraints and extend our discussions to c-semirings in Section 3.4. Thus, we define a soft constraint μ over a PVS M as a map $\mu : [X \to D] \to M$, we denote the set of soft constraints by S and write a *SCSP* as $(X, D, C, (M, \cdot_M, \varepsilon_M, \leq_M), S)$ which can be seen as a *COP* $(X, D, C, (M, \leq_M), f)$ where

$$f(\theta) = \Pi_M \{\mu(\theta) \mid \mu \in S\} \tag{1}$$

using $\cdot_M$ to aggregate solution degrees of all soft constraints evaluated on an assignment.

Example 1 Consider again the rostering problem in Figure 1 and let (X,D) be as depicted and use $\mathsf{U} = (\{c_1,c_2,c_3\},\leq_\mathsf{U})$ with $\leq_\mathsf{U} = \{(c_2,c_1),(c_3,c_1)\}^*$ as a the partial order denoting urgency of constraints. For $C = \{c_1,c_2,c_3\}$ as hard constraints, the solution space is empty. Instead, we can convert each hard constraint c_i into a soft constraint μ_i by choosing a suitable PVS M. For instance, we could use the PVS $(\mathbb{N},+,0,\geq)$ and interpret each valuation as a penalty incurred for a violated soft constraint. The sum of penalties ought to be minimized. With weights $\vec{w} = [2,1,1]$, we define $\mu_i(\theta) = w_i$ if $\theta \not\models c_i$ and $\mu_i(\theta) = 0$ otherwise. Letting $C = \emptyset$ and $S = \{\mu_1,\mu_2,\mu_3\}$, the solution $\theta = (n_1 \to \texttt{night}, n_2 \to \texttt{night}, n_3 \to \texttt{off})$ is optimal with $f(\theta) = \sum_{\mu_i \in S} \mu_i(\theta) = 1$. The solution degree 0, being top in M, is not reachable.

3.2 Looking for *Free* Partial Valuation Structures

In Example 1, the choice of $M = (\mathbb{N},+,0,\geq)$ seems rather obvious, given that the weighting $\vec{w} = [2,1,1]$ is consistent with the intuition that c_1 should be weighted higher than c_2 and c_3. However, interpreting $\vec{w}$ as a function $w : S \to \mathbb{N}$, we see that w clearly is only a monotone (not isomorphic) function. It totalizes U by making the incomparable elements c_2 and c_3 *equal*. Alternatively, $\vec{w} = [3,1,1]$ would be consistent, as would be $\vec{w} = [3,2,1]$, although our intuition tells us that making c_2 *more important* than c_3 certainly is a bad idea. We could try another PVS, say $M' = (2^S,\cup,\emptyset,\supseteq)$ to denote solution degrees as sets of violated soft constraints that are combined by union. Then, however, c_1, c_2, and c_3 would each contribute equally to a solution's quality, contrary to our model marking c_2 and c_3 as less urgent than c_1. Certainly, we want any mapping φ into a PVS N to preserve the given order: $\varphi(p) \leq_N \varphi(q)$ whenever $p \leq_\mathsf{U} q$; otherwise we invert ordering decisions.

Our point is, there is an infinite number of PVS that represent U to a certain degree – but the essential question is what are the *minimum requirements* in terms of comparability *any* PVS have to fulfill? Which PVS is, in this sense, the *best*, i.e., most general, one?

To find an answer, we consider the more general question of how to "convert" any partial order P into a partial valuation structure. As first presented in [40] and proved in Section 3.3, we can indeed lift any P to $PVS\langle P\rangle$ – i.e., construct a suitable combination operation and neutral element: We take as elements $\mathcal{M}_{\text{fin}}(P)$, the set of finite multisets composed from elements in P. For instance, $\wr\!\int$, $\wr c_1,c_2,c_2\!\int, \ldots \in \mathcal{M}_{\text{fin}}(\mathsf{U})$. Two multisets are combined using multiset-union with $\wr\!\int$ being the neutral element. Finally, a compatible ordering (with $\wr\!\int$ being top) is found inductively by applying the Smyth-ordering on sets (see Figure 1) to multisets (then written as $\preceq^P$) [3]:

Definition 2 (Smyth-ordering over Multisets) The Smyth-ordering on $\mathcal{M}_{\text{fin}}(P)$ is the binary relation $\preceq^P \subseteq \mathcal{M}_{\text{fin}}(P) \times \mathcal{M}_{\text{fin}}(P)$, given by the reflexive-transitive closure of

$$p <_P q \Rightarrow T \uplus \wr p\!\int \prec^P T \uplus \wr q\!\int$$
$$T \supset U \Rightarrow T \prec^P U$$

Intuitively, when we compare two multisets according to $\preceq^P$, we have to match every element q on the right side with a dominated element $p = h(q)$ on the left side such that $p \leq_P q$ and h is injective (see Lemma 1). There may be additional elements on the left. For any elements p,p' in a partial order P, we have $p \leq_P p' \Leftrightarrow \wr p\!\int \preceq^P \wr p'\!\int$. Note the monotonicity of the Smyth-ordering with respect to multiset union; if $T \preceq^P U$, then $T \uplus V \preceq^P U \uplus V$, since

[3] This relation is, in its set version, used to express non-determinism of programs in denotational semantics (set-valued to "collect different program results"), i.e., so-called power domains [4, Ch. 9].

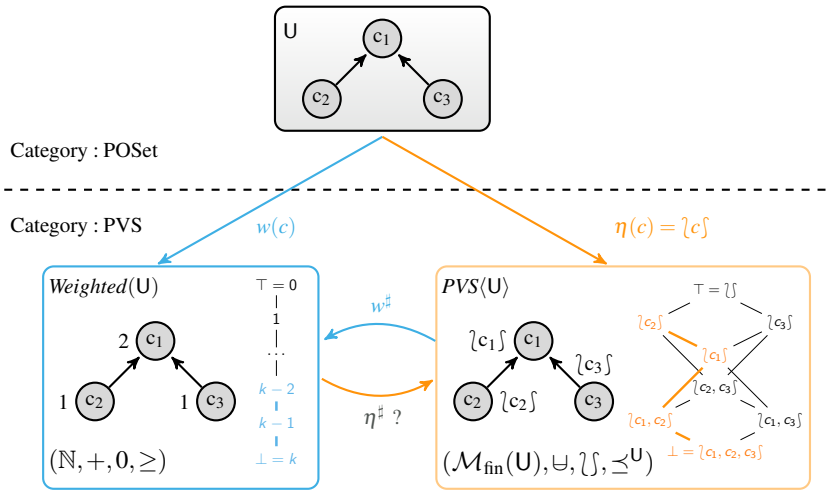


Figure 2 Encoding preferences given by the partial order U as two different PVS: *Weighted*(U) and *PVS*⟨U⟩. Highlighted paths show possible improvement steps during optimization. There can be no mapping $\eta^\sharp$ since distinct elements c_2 and c_3 are unified to 1 in *Weighted*(U) and would need to be represented by $\wr c_2 \int$ and $\wr c_3 \int$ in *PVS*⟨U⟩, respectively.

this holds for both defining clauses of the ordering. Antisymmetry is proved in Section 3.3. As an example, we have $\wr c_1, c_1, c_2 \int \prec^{\mathsf{U}} \wr c_1, c_2 \int$ or $\wr c_1, c_3 \int \prec^{\mathsf{U}} \wr c_2, c_3 \int$, if we read $c_2 \to c_1$ as $c_1 <_{\mathsf{U}} c_2$. In conclusion, $PVS\langle P\rangle = (\mathcal{M}_{\text{fin}}(P), \uplus, \wr\int, \preceq^P)$.

Since $PVS\langle P\rangle$ can be the codomain of any *SCSP*, soft constraints μ_i can arbitrarily map to $\mathcal{M}_{\text{fin}}(P)$, e.g., $\mu_i(\theta) = \wr c_1, c_1, c_2 \int$. We derive a particularly interesting instance instead (constraint preferences), if we convert a boolean soft constraint c_i into $\mu_i(\theta)$ which maps to $\wr c_i \int$ if θ satisfies c_i and $\wr\int$ otherwise. In the context of constraint preferences, the Smyth-ordering is called *single-predecessor-dominance* in Section 4 since – everything else being equal – a single predecessor can be dominated by a more important constraint due to the second clause of the ordering.

Figure 2 displays how we can encode a partial order P as either a weighted PVS or using $PVS\langle P\rangle$ by, e.g., representing c_1 as $w(c_1) = 1$ or as $\eta(c_1) = \wr c_1 \int$, respectively. Notice that a weighting $w : P \to \mathbb{N}$ can be "emulated" from $\mathcal{M}_{\text{fin}}(P)$ by defining its "lifted" version $w^\sharp : \mathcal{M}_{\text{fin}}(P) \to \mathbb{N}$ on the level of multisets: $w^\sharp(\wr p_1, \ldots, p_n \int) = \sum_{i=1}^n w(p_i)$ (arrow from right to left in the diagram). The converse, however, does not work: Once, e.g., c_2 and c_3 are both mapped to 1, we cannot "extract" information back about the origin to design a mapping $\eta^\sharp$ that can map from $\mathbb{N}$ into $\mathcal{M}_{\text{fin}}(\mathsf{U})$ since $\eta^\sharp(1)$ would have to simultaneously be equal to $\wr c_2 \int$ and $\wr c_3 \int$. This tells us (not surprisingly) that c_2 and c_3 *do not necessarily have to be treated as equal elements*, i.e., there exist other, more general, partial valuation structures encoding U that keep them as distinct elements.

It was not a coincidence that we found a lifted mapping $w^\sharp$ from $PVS\langle P\rangle$ to *Weighted*(P) that is equal to applying w directly from P. Instead, $PVS\langle P\rangle$ has the *universal mapping property* [40]: *Any* order-preserving function φ from P into the underlying partial order of a PVS can be decomposed into the form $\varphi^\sharp \circ \eta$ in a *unique* way. Thus, $PVS\langle P\rangle$ is also called the "free PVS over P". Practically, this means that we can always safely convert P into the free PVS before mapping to another PVS (e.g., if we need an integer objective for our implementation, see Section 4.3) without losing any information. Conversely, *Weighted*(P)

is not free as we cannot return to $PVS\langle P\rangle$ once P is mapped to $Weighted(P)$. Since free objects are unique up to isomorphism [54, p. 147], $PVS\langle P\rangle$ can be seen as *the most general* PVS over a partial order. We prove this fact in Lemma 2 in Section 3.3.

Our original question, "how to formulate an ordering over constraints as a PVS with the least overhead", thus boils down to the search for a *free construction*. Similar instances are the free monoid or the free group over a set. We can capture this task formally using the language of *category theory* (hinted in Figure 2) which studies, inter alia, algebraic structures along with their structure-preserving mappings. This perspective further enables us to treat the transformation from a partial order into a PVS and that from a PVS into a c-semiring *uniformly*, i.e., as defining a *left adjoint* to a forgetful functor. The subsequent sections hence draw on basic knowledge of category theory when they offer the derivation of the free PVS and the free c-semiring, respectively. In Appendix A, we introduce categorical concepts relevant to free constructions with the well-known free monoid over a set. Readers familiar with basic category theory may safely skip the appendix and readers familiar with term algebras may check the categorical presentation. As category theory has not been used extensively in constraint programming (except for [23]), we refer to excellent introductory material, e.g., [48, 8, 7].

As a very brief introduction to follow the proof obligations of the subsequent sections, we note that a *category* $\mathcal{C}$ refers to a collection of so-called *objects* and *morphisms* that generalize functions. For instance, the category Set has conventional sets as objects and functions as morphisms, whereas the category Mon has monoids as objects and monoid-homomorphisms as morphisms. A *functor* F between categories $\mathcal{C}$ and $\mathcal{D}$ is a mapping that sends every $\mathcal{C}$-object A to a $\mathcal{D}$-object $F(A)$ and every $\mathcal{C}$-morphism $\varphi : A \to B$ to a $\mathcal{D}$-morphism $F(\varphi) : F(A) \to F(B)$. For example, for every set A there is an associated monoid $(A^*, \cdot, \varepsilon)$ with words over A and concatenation. We can use this to define a functor $F : \text{Set} \to \text{Mon}$ by $F(A) = (A^*, \cdot, \varepsilon)$ and $F(f : A \to B) = f^\sharp : A^* \to B^*$ with $f^\sharp(a_1 \cdot \ldots \cdot a_n) = f(a_1) \cdot \ldots \cdot f(a_n)$. Conversely, there is the *underlying* functor $|-| : \text{Mon} \to \text{Set}$ with $|(A, \cdot, \varepsilon)| = A$ and $|\varphi : (A_1, \cdot_1, \varepsilon_1) \to (A_2, \cdot_2, \varepsilon_2)| = \varphi : A_1 \to A_2$ yielding the *underlying set* of a monoid.

This operator $|-|$ is a convention present in category-theoretical arguments. It allows to distinguish structures and sets and must not be confused with set cardinality. We follow this convention in the remainder of the paper and, e.g., will write a partial order as $P = (|P|, \leq_P)$.

Using the above notions, we can now formally state what a free object is:

Definition 3 (Free object) Given two categories $\mathcal{A}$ and $\mathcal{B}$ and a functor $G : \mathcal{B} \to \mathcal{A}$, the *free object* $F(A)$ in $\mathcal{B}$ over an object A of $\mathcal{A}$ is characterized by a *unit* morphism $\eta_A : A \to G(F(A))$ in $\mathcal{A}$ such that for every $\mathcal{A}$-morphism $\varphi : A \to G(B)$ with B an object of $\mathcal{B}$, there is a **unique** *lifting* $\mathcal{B}$-morphism $\varphi^\sharp : F(A) \to B$ satisfying $G(\varphi^\sharp) \circ \eta_A = \varphi$.

A free object $F(A)$ is unique up to isomorphism and the composition of two free constructions yields another free construction [54, Ch. 3]. Incidentally, the monoid $(A^*, \cdot, \varepsilon)$ is the free monoid over a set A (see Appendix A). A free object does not have to exist. We need to prove a particular free construction (e.g., free monoid or free PVS) by choosing the appropriate categories, functors, and, of course, the free object itself.

3.3 The Free Partial Valuation Structure over a Partial Order

Motivated by the goal of finding the most general PVS to encode constraint preferences, the search for the free PVS over a partial order P answers a more fundamental problem:

Which ordering decisions always have to hold if we extend any partial order with a combination operation (multiplication) and neutral top element?

More formally, this is the case if we have several soft constraints $\mu_1, \ldots, \mu_n$ that each grade an assignment θ in the same partial order P and we take a product $\mu_1(\theta) \cdot \ldots \cdot \mu_n(\theta)$. Which $\preceq$-relations *must* certainly hold if we compare $\mu_1(\theta) \cdot \ldots \cdot \mu_n(\theta)$ with $\mu_1(\theta') \cdot \ldots \cdot \mu_n(\theta')$? How shall we even represent these products?

A seemingly obvious choice would be to collect all soft constraints' valuations as a set, i.e., $\{p_1, \ldots, p_n\}$. Each $p \in P$ could then individually be represented by the unit morphism $\eta(p) = \{p\}$ and then combined using set union. Since $\emptyset$ should be top in a PVS, we aim to order the sets by size and according to P. That means, we want $X \preceq \emptyset$ for any set X and $\eta(p_1) = \{p_1\} \preceq \{p_2\} = \eta(p_2)$ if $p_1 \leq_P p_2$. Both cases are covered by the Smyth-ordering over sets (cf. Section 3.2). However, that approach does not yield a proper PVS if we consider that we can multiply elements $\{p_1\}$ with themselves: Assuming $p_1 \leq_P p_2$, also $\{p_1\} \preceq \{p_2\}$ holds. Combining with $\{p_1\}$ on both sides yields $\{p_1\} \preceq \{p_1, p_2\}$, by monotonicity. But, by the definition of the Smyth-ordering, we also have $\{p_1, p_2\} \preceq \{p_1\}$ and thus antisymmetry is violated.

It turns out that the idempotency of set union is the culprit, in particular the fact that $\eta(p_1) \cup \eta(p_1) = \eta(p_1)$. This fact is *not* required by PVS axioms. Instead, commutativity and associativity provide a hint about the underlying set of the free PVS: The free monoid over a set A uses A^*, finite lists over A, embedded by $\eta'(a) = [a]$ and combined with concatenation ::) since we only need associativity: $\eta'(a) :: (\eta'(b) :: \eta'(c)) = (\eta'(a) :: \eta'(b)) :: \eta'(c) = [a,b,c]$ (see Appendix A). For the free PVS, we additionally need to to equate $\eta(a) \cup \eta(b)$ with $\eta(b) \cup \eta(a)$, but again, not necessarily $\eta(a) \cup \eta(a)$ with $\eta(a)$. This is precisely what we achieve with $\mathcal{M}_{\text{fin}}(P)$, finite multisets over P and $\eta(a) = \llparenthesis a \rrparenthesis$. Taking plain sets over P would additionally assume idempotency and is thus too specific.[4]

Figure 3 instantiates Definition 3 for the task of proving that $PVS\langle P\rangle$ is indeed the free PVS over a partial order P. We start in the category PO of partial orders as objects and monotone functions as morphisms and map to PVS, the category of partial valuation structures as objects and PVS-homomorphisms as morphisms. To switch between partial orders and partial valuation structures we need appropriate functors. First, the (free) functor $PVS\langle P\rangle$:

$$PVS\langle P\rangle = (\mathcal{M}_{\text{fin}}(P), \uplus, \llparenthesis\rrparenthesis, \preceq^P) \,,$$
$$PVS\langle \varphi : P \to Q\rangle = \lambda \llparenthesis p_1, \ldots, p_n \rrparenthesis \,.\, \llparenthesis \varphi(p_1), \ldots, \varphi(p_n) \rrparenthesis \,.$$

In the other direction, the (forgetful) functor $PO : \text{PVS} \to \text{PO}$ is defined by

$$PO(M) = (|M|, \leq_M) \,,$$
$$PO(\varphi : M \to N) = \varphi \,.$$

Starting from a partial order P, commutativity and associativity motivate the underlying set $\mathcal{M}_{\text{fin}}(P)$. We can also justify each rule of the Smyth-ordering over multisets by applying Definition 3. First, as each $p \in |P|$ is found in $\mathcal{M}_{\text{fin}}(P)$ by $\eta_P(p) = \llparenthesis p \rrparenthesis$ and η_P is a monotone function, we have that $p_1 \leq_P p_2 \Rightarrow \llparenthesis p_1 \rrparenthesis \preceq^P \llparenthesis p_2 \rrparenthesis$. This ensures that P is preserved over their embedded counterparts. The other rule $T \supset U \Rightarrow T \preceq^P U$ stems from the fact that the neutral element is the top of the ordering in a PVS – which is the most prevalent choice in soft constraints [44]. This implies $m \cdot_M n \leq_M m$ since $n \leq_M \varepsilon_M \Rightarrow m \cdot n \leq_M m$, by monotonicity.

[4] Interestingly enough, the fact that partial valuation structures need not be idempotent in general (e.g., weighted constraints) disallows a straightforward extension of local consistency techniques to soft constraints [19].

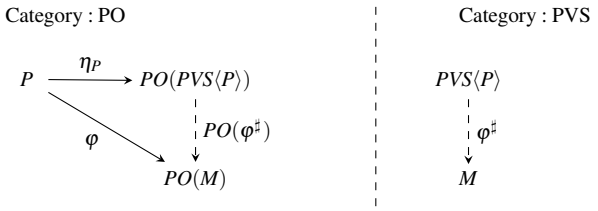


Figure 3 Diagram of the free PVS over a partial order. For an arbitrary PVS M that we map into from a partial order P using φ, we can lift this mapping to $\varphi^\sharp : PVS\langle P\rangle \to M$ such that $\varphi = PO(\varphi^\sharp) \circ \eta_P$. Consequently, $PVS\langle P\rangle$ only identifies and orders elements as absolutely required by PVS axioms – it is most general.

Consequently, for the free PVS, $\wr m,n\int \preceq^P \wr m\int$ needs to hold, as does $T \preceq^P \wr\int$, both of which are represented by the above rule. Dually, we would have $T \Subset U \Rightarrow T \preceq^P U$, had we defined the neutral element to be bottom of the ordering.

Next, we have to confirm that $PVS\langle P\rangle = \langle \mathcal{M}_{\text{fin}}(P), \uplus, \wr\int, \preceq^P\rangle$ is a partial valuation structure, to begin with. Associativity and commutativity of $\uplus$ and neutrality of $\wr\int$ with respect to $\mathcal{M}_{\text{fin}}(P)$ are obvious, we have already discussed reflexivity and transitivity of $\preceq^P$ as well as monotonicity of $\uplus$ with respect to $\preceq^P$ in Section 3.2. To show antisymmetry of $\preceq^P$, we prove a result that also turns out to be useful later on when we implement the Smyth-ordering as a MiniZinc predicate to be used in search. To do so, we introduce a bit of notation to "unfold" a multiset T into a set representation $\mathcal{S}(T)$, e.g., $\mathcal{S}(\wr x,x,y\int) = \{(1,x),(2,x),(1,y)\}$. Formally, for a multiset $T = \wr l_1 x_1, \ldots, l_n x_n\int \in \mathcal{M}_{\text{fin}}(X)$ with $l_1, \ldots, l_n > 0$ and $x_i \neq x_j$ if $i \neq j$, let $\mathcal{S}(T) = \bigcup_{1\leq i\leq n}\{(j,x_i) \mid 1 \leq j \leq l_i\}$.

Lemma 1 (Witness for $\preceq^P$) *$T \preceq^P U$ if, and only if, there is an injective map $h : \mathcal{S}(U) \to \mathcal{S}(T)$ (called a* witness *function) with $p \leq_P q$ if $h(j,q) = (k,p)$ for all $(j,q) \in \mathcal{S}(U)$.*

Proof Let first $T \preceq^P U$ hold. We restrict our attention w.l.o.g. to the case $T \neq U$ as otherwise the claim trivially holds. Then there is a sequence of multisets $T_1, \ldots, T_n \in \mathcal{M}_{\text{fin}}(P)$ with $n > 1$ such that $T_1 = T$, $T_n = U$, and for each $1 \leq i < n$, either $T_i \supsetneq T_{i+1}$ or $T_i = T_i' \uplus \wr p\int$ and $T_{i+1} = T_i' \uplus \wr q\int$ with $p \leq_P q$. As required in the claim, for each $1 \leq i < n$ there is a witness $h_i : \mathcal{S}(T_{i+1}) \to \mathcal{S}(T_i)$ as follows: If $T_i \supsetneq T_{i+1}$, then we choose $h_i = \text{id}_{\mathcal{S}(T_i)}$. If $T_i = T_i' \uplus \wr p\int$ and $T_{i+1} = T_i' \uplus \wr q\int$ with $p \leq_P q$, then we choose $h_i = \text{id}_{\mathcal{S}(T_i')} \cup \{(j,p) \mapsto (k,q)\}$ where $j = \max\{l \mid (l,p) \in \mathcal{S}(T_i')\} + 1$ and $k = \max\{l \mid (l,q) \in \mathcal{S}(T_i')\} + 1$. Then $h_1 \circ \ldots \circ h_{n-1} : \mathcal{S}(U) \to \mathcal{S}(T)$ is a witness function.

For the converse, we prove that if $h : \mathcal{S}(U) \to \mathcal{S}(T)$ is a witness function, then $T \preceq^P U$ by induction on the cardinality of $\mathcal{S}(U)$. Let $h : \mathcal{S}(U) \to \mathcal{S}(T)$ be given. If $|\mathcal{S}(U)| = 0$, then $T \preceq^P \wr\int = U$. Now let $|\mathcal{S}(U)| > 0$ and let $(j,q) \in \mathcal{S}(U)$ such that j is maximal. Then $h(j,q) = (k,p)$ with $p \leq_P q$. Let $T', U' \in \mathcal{M}_{\text{fin}}(P)$ be defined by $T = T' \uplus \wr p\int$ and $U = U' \uplus \wr q\int$. To construct a witness function between T' and U' and apply the induction hypothesis, we define $g : \mathcal{S}(T) \to \mathcal{S}(T')$ by $g(l,r) = (l,r)$ if $r \neq p$ or $l < k$, and $g(l,p) = (l-1,p)$ if $l > k$. Essentially, g closes possible "gaps" in the image of h. Then $\mathcal{S}(U') = \mathcal{S}(U) \setminus \{(j,q)\}$ and $h' : \mathcal{S}(U') \to \mathcal{S}(T')$ defined as $h' = g \circ h$ is a witness function between T' and U'. By induction hypothesis, hence, $T' \preceq^P U'$ and thus, by the monotonicity of $\preceq^P$ (see Section 3.2), $T = T' \uplus \wr p\int \preceq^P U' \uplus \wr p\int \preceq^P U' \uplus \wr q\int = U$. □

The witness function can be interpreted as assigning an "inferior" to every element on the right-hand side. To see the antisymmetry of the Smyth-ordering[5], assume for a con-

[5] Curiously enough, the Smyth-ordering on mere sets is *not* antisymmetric. It is just a preorder.

tradiction that there are T and U with both $T \preceq^P U$ and $U \preceq^P T$, but $T \neq U$ and choose one T with minimal cardinality satisfying this property. Then T has to be non-empty. Let $f : \mathcal{S}(U) \to \mathcal{S}(T)$ and $g : \mathcal{S}(T) \to \mathcal{S}(U)$ be witnessing maps for $T \preceq^P U$ and $U \preceq^P T$, respectively. Choose an element $(j,q) \in \mathcal{S}(U)$ such that q is minimal w.r.t. $\leq_P$ in U. Then there is an inferior $(k,p) = f(j,q)$ in $\mathcal{S}(T)$ with $p \leq_P q$. If $p \neq q$, as $U \preceq^P T$ holds as well, there would be yet another inferior $g(k,p) = (j',q') \in \mathcal{S}(U)$ such that $q' \leq_P p$ and thus $q' \leq_P p <_P q$, contradicting the minimality of q in U; thus $f(j,q) = (k,q)$. Assume, without loss of generality, that j and k are maximal. Remove the occurrence of p from T and U, obtaining T' and U', respectively. Then $T' \preceq^P U'$ and $U' \preceq^P T'$ hold as well, since the reduced-domain functions $f' : \mathcal{S}(T') \to \mathcal{S}(U')$ with $f'(l,p) = f(l,p)$ and, similarly, $g' : \mathcal{S}(U') \to \mathcal{S}(T')$, are witnessing maps. This contradicts the assumed minimality of T. Thus, $PVS\langle P\rangle$ fulfills all axioms of a partial valuation structure and we are ready to show that is indeed free.

Lemma 2 (Free PVS) *$PVS\langle P\rangle$ is the free PVS over the partial order P.*

Proof Let P be a partial order $(|P|, \leq_P)$ and $\varphi : P \to PO(M)$ be a PVS-homomorphism into the underlying partial order of any PVS M. To show the existence of a lifted variant of φ, we define $\varphi^\sharp : PVS\langle P\rangle \to M$ as a PVS-morphism by

$$\varphi^\sharp(\wr p_1, \ldots, p_n \wr) = \varphi(p_1) \cdot_M \ldots \cdot_M \varphi(p_n)$$

for all $\wr p_1, \ldots, p_n \wr \in \mathcal{M}_{\text{fin}}(P)$, where, if $n = 0$, $\varphi^\sharp(\wr \wr) = \varepsilon_M$. This is well-defined, i.e., $\varphi^\sharp$ is indeed a PVS-homomorphism, since $\varphi^\sharp(\wr p_1, \ldots, p_m \wr \uplus \wr q_1, \ldots, q_n \wr) = \varphi(p_1) \cdot_M \ldots \cdot_M \varphi(p_m) \cdot_M \varphi(q_1) \cdot_M \ldots \cdot_M \varphi(q_n) = \varphi^\sharp(\wr p_1, \ldots, p_m \wr) \cdot_M \varphi^\sharp(\wr q_1, \ldots, q_n \wr)$, $\varphi^\sharp(\wr \wr) = \varepsilon_M$, and, if $T \leq_{PVS\langle P\rangle} U$, then $\varphi^\sharp(T) \leq_M \varphi^\sharp(U)$: We consider the generating cases for one step of the Smyth-ordering as it is straightforward to consider the extension to sequences $T_1, \ldots, T_n$ as done in the proof of the witness function. Assume $T \leq_{PVS\langle P\rangle} U$. Either $T = \wr p \wr \uplus T'$ and $U = \wr q \wr \uplus T'$ with $p \leq_P q$. Then $\varphi^\sharp(T) = \varphi(p) \cdot_M \varphi^\sharp(T')$ and $\varphi^\sharp(U) = \varphi(q) \cdot_M \varphi^\sharp(T')$. And since $\varphi(p) \leq_M \varphi(q)$ due to φ being a PO-morphism, we have $\varphi^\sharp(T) \leq_M \varphi^\sharp(U)$, by monotonicity of $\cdot_M$. Or, it is the case that $T \supseteq U$. Then $T = U \uplus T'$ (T' may be empty) and thus $\varphi^\sharp(T) = \varphi^\sharp(T') \cdot_M \varphi^\sharp(U) \leq_M \varphi^\sharp(U)$, by the PVS axiom $m \cdot n \leq_M m$. Consequently $\varphi^\sharp$ is a PVS-homomorphism.

Moreover, $\varphi = PO(\varphi^\sharp) \circ \eta_P$ with $\eta_P(p) = \wr p \wr$ for all $p \in |P|$, since $PO(\varphi^\sharp)(\eta_P(p)) = \varphi^\sharp(\wr p \wr) = \varphi(p)$; hence the diagram in Figure 3 commutes.

Finally, $\varphi^\sharp$ is unique with this property: Assume there would be another PVS-homomorphism $\psi : PVS\langle P\rangle \to M$ that satisfies $PO(\psi) \circ \eta_P = \varphi$. Due to this requirement, we have $\psi(\wr p \wr) = \varphi(p)$ for every $p \in |P|$. Thus, for ψ, we have $\psi(\wr \wr) = \varepsilon_M = \varphi^\sharp(\wr \wr)$ and

$$\begin{aligned}\psi(\wr p_1, \ldots, p_n \wr) &= \psi(\wr p_1 \wr) \cdot_M \ldots \cdot_M \psi(\wr p_n \wr) \\ &= \varphi(p_1) \cdot_M \ldots \cdot_M \varphi(p_n) = \varphi^\sharp(\wr p_1 \wr) \cdot_M \ldots \cdot_M \varphi^\sharp(\wr p_n \wr) \\ &= \varphi^\sharp(\wr p_1, \ldots, p_n \wr)\end{aligned}$$

since ψ is a PVS-homomorphism and by the previous remark. Hence $\varphi^\sharp = \psi$, as claimed, and $PVS\langle P\rangle$ is indeed the free partial valuation structure over a partial order. □

This concludes our theoretical considerations of the free partial valuation structure. An example of a *SCSP* employing the free PVS is depicted in Figure 5 in Section 3.5. It actually uses soft constraints that directly map to the free PVS (i.e., $\mathcal{M}_{\text{fin}}(P)$ rather than P). For constraint preferences, we only distinguish between $\wr c_1 \wr$ and $\wr \wr$. Both the free PVS and a specialized type for constraint preferences are available in MiniBrass (cf. Section 4.2.2).

3.4 The Free C-Semiring over a Partial Valuation Structure

As mentioned before, (partial) valuation structures are not the only abstract algebraic framework for soft constraints in the literature. C-semirings constitute a particularly popular choice. They are purely algebraic by requiring a second "additive" operation instead of a partial ordering to form an (upper semi-)lattice. This idempotent, commutative, and associative operation is then used to induce a partial ordering. Moreover, any c-semiring is bounded above and below by two designated constants. We will proceed to show that every c-semiring gives rise to a bounded PVS, and, conversely, every PVS can be extended to a c-semiring by means of another free construction – although not every PVS *is* a c-semiring since the additive operation in fact returns a supremum which need not exist in a PVS.

This section therefore extends previous work that examined the similarities between c-semirings and (totally ordered bounded) valuation structures [14]. The authors identified a valuation structure with every *totally ordered* c-semiring only. For branch-and-bound and similar search algorithms, a partial ordering indeed suffices (see Section 4 or [36,44]). The main algorithmic advantage of having a second algebraic operation instead of the partial ordering lies in the thereby guaranteed existence of a supremum. This least upper bound can be used for non-serial dynamic programming, i.e., variable elimination. These algorithms may, however, return an unreachable optimal solution degree (e.g., the supremum of *all* reachable optima). From a practical perspective, this free construction of a c-semiring from a PVS alleviates the need to model in c-semirings in the first place. If a fruitful algorithmic technique for c-semirings (relying on the addition) is discovered, it can also be applied to a PVS when raised to the free c-semiring. We sketch such an application in Section 3.5 but first actually derive the free c-semiring over a PVS.

Formally, a *c-semiring* [13] $A = (|A|, \oplus_A, \otimes_A, \mathbf{0}_A, \mathbf{1}_A)$ is given by an (underlying) set $|A|$, two binary operations $\oplus_A, \otimes_A : |A| \times |A| \to |A|$, and two constants $\mathbf{0}_A, \mathbf{1}_A \in |A|$ such that the following axioms are satisfied for all $x, y, z \in |A|$:

- $\oplus_A$ is associative and commutative and has $\mathbf{1}_A$ as annihilator and $\mathbf{0}_A$ as neutral element
- $\otimes_A$ is associative and commutative, has $\mathbf{0}_A$ as annihilator and $\mathbf{1}_A$ as neutral element
- $\otimes_A$ distributes over $\oplus_A$

To preserve this structure, a *c-semiring homomorphism* $\varphi : A \to B$ from a c-semiring A to a c-semiring B is given by a map $\varphi : |A| \to |B|$ such that for all $a_1, a_2 \in |A|$:

1. $\varphi(a_1 \oplus_A a_2) = \varphi(a_1) \oplus_T \varphi(a_2), \qquad \varphi(a_1 \otimes_A a_2) = \varphi(a_1) \otimes_T \varphi(a_2)$
2. $\varphi(\mathbf{0}_A) = \mathbf{0}_T, \qquad \varphi(\mathbf{1}_A) = \mathbf{1}_T$

Consequently, the category cSRng of c-semirings has the c-semirings as objects and the c-semiring homomorphisms as morphisms. Note that in a c-semiring A the operation $\oplus_A$ is idempotent:

$$a \oplus_A a = (a \otimes_A \mathbf{1}_A) \oplus_A (a \otimes_A \mathbf{1}_A) = a \otimes_A (\mathbf{1}_A \oplus_A \mathbf{1}_A) = a \otimes_A \mathbf{1}_A = a \,.$$

Hence, $\oplus_A$ can be used to induce a partial ordering $\leq_A$ by interpreting it as the least upper bound: $a \leq_A b \leftrightarrow a \oplus_A b = b$. Clearly, $\leq_A$ is reflexive due to the idempotency, transitive due to associativity, and antisymmetric due to commutativity of $\oplus_A$. With this definition, for all $a, b, c \in |A|$ it holds that

1. $\mathbf{0}_A \leq_A a \leq_A \mathbf{1}_A$;
2. $a \leq_A a \oplus_A b$ and $b \leq_A a \oplus_A b$;
3. if $a \leq_A c$ and $b \leq_A c$, then $a \oplus_A b \leq_A c$.

In particular, $a \oplus_A b$ is the supremum of a and b with respect to $\leq_A$. Also $\oplus_A$ is monotone w.r.t. $\leq_A$ in both arguments, i.e., $a \leq_A a'$ and $b \leq_A b'$ implies $a \oplus_A b \leq_A a' \oplus_A b'$. Additionally, the *combination* operation $\otimes_A$ is monotone w.r.t. the induced ordering $\leq_A$, since if $a \leq_A a'$ (i.e., $a \oplus_A a' = a'$) then $(a \otimes_A b) \oplus_A (a' \otimes_A b) = (a \oplus_A a') \otimes_A b = a' \otimes_A b$, i.e., $a \otimes_A b \leq_A a' \otimes b$, from which it follows that $a \leq_A a'$ and $b \leq_A b'$ implies $a \otimes_A b \leq_A a' \otimes b'$. Furthermore, for all $a, b \in |A|$, it holds that $a \otimes_A b \leq_A a$ and $a \otimes_A b \leq_A b$, since $(a \otimes_A b) \oplus_A a = (a \otimes_A b) \oplus_A (a \otimes_A \mathbf{1}_A) = a \otimes_A (b \oplus_A \mathbf{1}_A) = a \otimes_A \mathbf{1}_A = a$.

As a consequence, we can easily convert any c-semiring into a PVS by defining the functor $PVS : \text{cSRng} \to \text{PVS}$:

$$PVS(A) = (|A|, \otimes_A, \mathbf{1}_A, \leq_A) ,$$
$$PVS(\varphi : A \to B) = \varphi .$$

Note that $PVS(A)$ is a bounded PVS with $\bot_{PVS(A)} = \mathbf{0}_A$. This leaves us with the first part of a free construction between categories PVS and cSRng (cf. Definition 3). The opposite direction, constructing a c-semiring starting from a PVS, is not as obvious since the partial order of a PVS need not show suprema that are required to exist for the $\oplus$ operator (they clearly exist in total orders, making the conversion from totally ordered c-semirings to valuation structures more straightforward [14]). For instance, in Figure 1, we saw that both $\wr c_1 \int$ and $\wr c_2, c_3 \int$ are upper bounds of $\wr c_1, c_2 \int$ and $\wr c_1, c_3 \int$ but they are incomparable.

When allowing partiality, we can always find an "artificial" supremum by collecting all (incomparable) valuations in a set and ordering these sets appropriately. Consider an arbitrary PVS $M = (|M|, \cdot_M, \varepsilon_M, \leq_M)$. We write $\mathcal{I}_{\text{fin}}(M)$ to denote the set of finite sets composed of *incomparable* elements from $|M|$ (i.e., if $X \in \mathcal{I}_{\text{fin}}(M)$ then for any $x \neq y \in X$ we have $x \parallel_M y$) and $\text{Max}^{\leq_M}(X)$ to denote the maximal elements of X with respect to $\leq_M$. For instance, if $|M| = \{1, 2, \text{III}, \text{IV}\}$ and $1 <_M 2$, $\text{III} <_M \text{IV}$, the sets $\{2, \text{IV}\}$ or $\{1, \text{III}\}$ are in $\mathcal{I}_{\text{fin}}(M)$ but $\{1, \text{III}, \text{IV}\}$ is not and $\text{Max}^{\leq_M}(|M|) = \{2, \text{IV}\}$. We define the binary operations $\tilde{\cup}_M$ and $\tilde{\cdot}_M$ over $\mathcal{I}_{\text{fin}}(M)$ by

$$I \,\tilde{\cup}_M J = \text{Max}^{\leq_M}(I \cup J) ,$$
$$I \,\tilde{\cdot}_M J = \text{Max}^{\leq_M}\{m \cdot_M n \mid m \in I,\ n \in J\} .$$

Clearly, $\tilde{\cup}_M$ inherits commutativity from $\cup$, and is idempotent since $\text{Max}^{\leq_M}(X) = X$ for any set X consisting of already incomparable elements. It is easy to check that it is also associative. Further, $\{\varepsilon_M\}$ is an annihilator for $\tilde{\cup}_M$ since ε_M is the greatest element of $|M|$ with respect to $\leq_M$, and $\emptyset$ is its neutral element.

Similarly, $\tilde{\cdot}_M$ is obviously commutative since $\cdot_M$ is commutative. Dually to $\tilde{\cup}_M$, it has $\{\varepsilon_M\}$ as neutral element (since ε_M is neutral in M) and $\emptyset$ as annihilator. For the associativity of $\tilde{\cdot}_M$, we have

$$\begin{aligned}
&I \,\tilde{\cdot}_M (J \,\tilde{\cdot}_M K) = \\
&\quad \text{Max}^{\leq_M}\{m_I \cdot_M m_{JK} \mid m_I \in I,\ m_{JK} \in \text{Max}^{\leq_M}\{m_J \cdot_M m_K \mid m_J \in J,\ m_K \in K\}\} = \\
&\quad \text{Max}^{\leq_M}\{m_I \cdot_M m_J \cdot_M m_K \mid m_I \in I,\ m_J \in J,\ m_K \in K\}\} = \\
&\quad \text{Max}^{\leq_M}\{m_{IJ} \cdot_M m_K \mid m_{IJ} \in \text{Max}^{\leq_M}\{m_I \cdot_M m_J \mid m_I \in I,\ m_J \in J\},\ m_K \in K\} = \\
&\quad (I \,\tilde{\cdot}_M J) \,\tilde{\cdot}_M K ,
\end{aligned}$$

since $\text{Max}^{\leq_M}\{m \cdot_M n \mid m \in I,\ n \in \text{Max}^{\leq_M}(X)\} = \text{Max}^{\leq_M}\{m \cdot_M n \mid m \in I,\ n \in X\}$ for all finite sets $X \subseteq |M|$. Finally, $\tilde{\cdot}_M$ distributes over $\tilde{\cup}_M$:

$$I \,\tilde{\cdot}_M (J \,\tilde{\cup}_M K) =$$

$$
\begin{aligned}
&\mathrm{Max}^{\leq_M}\{m_I \cdot_M m_{JK} \mid m_I \in I,\ m_{JK} \in \mathrm{Max}^{\leq_M}(J \cup K)\} = \\
&\mathrm{Max}^{\leq_M}\{m_I \cdot_M m_{JK} \mid m_I \in I,\ m_{JK} \in J \cup K\} = \\
&\mathrm{Max}^{\leq_M}(\{m_I \cdot_M m_J \mid m_I \in I,\ m_J \in J\} \cup \{m_I \cdot_M m_K \mid m_I \in I,\ m_K \in K\}) = \\
&\mathrm{Max}^{\leq_M}(\mathrm{Max}^{\leq_M}\{m_I \cdot_M m_J \mid m_I \in I,\ m_J \in J\} \cup \\
&\qquad \mathrm{Max}^{\leq_M}\{m_I \cdot_M m_K \mid m_I \in I,\ m_K \in K\}) = \\
&(I \tilde{\cup}_M J) \,\tilde{\cup}_M\, (I \tilde{\cup}_M K)\ ,
\end{aligned}
$$

since $\mathrm{Max}^{\leq_M}(I \cup \mathrm{Max}^{\leq_M}(X)) = \mathrm{Max}^{\leq_M}(I \cup X)$ for all finite $X \subseteq |M|$. Thus, we conclude

Lemma 3 *$(\mathcal{I}_{\mathrm{fin}}(M), \tilde{\cup}_M, \tilde{\cdot}_M, \emptyset, \{\varepsilon_M\})$ is a c-semiring.* □

This structure will serve to define the object part of a free functor from PVS to cSRng.

At this point, it is worth noting that a similar construction of c-semiring addition and multiplication operations has been introduced by Rollón [51], although starting from a given c-semiring instead of a PVS. She proves that when A is a c-semiring, its so-called *frontier algebra* $\mathcal{A} = (\mathcal{I}(A) \setminus \{\emptyset\}, \tilde{\oplus}_A, \tilde{\otimes}_A, \{\mathbf{0}_A\}, \{\mathbf{1}_A\})$ again is a c-semiring, where $\mathcal{I}(A)$ are (possibly *infinite*) subsets of $|A|$ containing only pairwise incomparable elements w.r.t. $\leq_A$, and,

$$
\begin{aligned}
I \,\tilde{\oplus}_A\, J &= \mathrm{Max}^{\leq_A}(I \cup J)\ , \\
I \,\tilde{\otimes}_A\, J &= \mathrm{Max}^{\leq_A}\{i \otimes_A j \mid i \in I,\ j \in J\}
\end{aligned}
$$

for all $I, J \in \mathcal{I}(A) \setminus \{\emptyset\}$. The underlying set of the frontier algebra thus contains sets of arbitrary cardinality, not only finite sets as in our approach of the free construction. In fact, such infinite sets would correspond to "junk elements" (cf. Appendix A), i.e., they would be unnecessary to have in the carrier set of the free c-semiring since we only have the finitary combination and supremum operation.

In [51], the condition that only *non-empty* sets have to be considered is missing. The empty set has to be excluded, however, since otherwise $\emptyset \,\tilde{\otimes}_A\, \{\mathbf{0}_A\} = \emptyset$, although $\{\mathbf{0}_A\}$ has to be the annihilator for $\tilde{\otimes}_A$, and $\emptyset \,\tilde{\oplus}_A\, \{\mathbf{0}_A\} = \{\mathbf{0}_A\}$, i.e., $\emptyset \leq_{\mathcal{A}} \{\mathbf{0}_A\}$ contradicting that $\{\mathbf{0}_A\}$ has to be the smallest element w.r.t. $\leq_{\mathcal{A}}$. By contrast, in our approach, we have to consider $\emptyset$ as well in order to obtain a "fresh" bottom element of the free c-semiring over an arbitrary PVS. If we only applied the construction of a free c-semiring to the sub-category of bounded PVS, we also could exclude $\emptyset$ and would obtain $\{\bot_M\}$ as bottom element of the free c-semiring over the bounded PVS M. However, the free PVS over a partial order – our original mission – clearly is not bounded.

To verify that we can design the morphism part of a free functor, it is useful to convince ourselves that the application of the maximum operator in a target structure subsumes the maximum operator in a source structure.

Lemma 4 (Subsumption of Maximum) *Let $\varphi : M \to N$ be a PVS homomorphism. For finite sets $X \subseteq |M|$, we have* $\mathrm{Max}^{\leq_N}(\varphi(\mathrm{Max}^{\leq_M}(X))) = \mathrm{Max}^{\leq_N}(\varphi(X))$.

Proof First, $\mathrm{Max}^{\leq_N}(\varphi(\mathrm{Max}^{\leq_M}(X))) \subseteq \mathrm{Max}^{\leq_N}(\varphi(X))$, since $\mathrm{Max}^{\leq_M}(X) \subseteq X$ holds which in turn implies $\varphi(\mathrm{Max}^{\leq_M}(X)) \subseteq \varphi(X)$.
To conversely show $\mathrm{Max}^{\leq_N}(\varphi(X)) \subseteq \mathrm{Max}^{\leq_N}(\varphi(\mathrm{Max}^{\leq_M}(X)))$, it suffices to show that for each $n \in \varphi(X)$ there is a (weakly dominating) $n' \in \varphi(\mathrm{Max}^{\leq_M}(X))$ such that $n \leq_N n'$: If $n \in \varphi(X)$ then $n = \varphi(m)$ for some $m \in X$. Either m is maximal, in which case n is obviously in $\varphi(\mathrm{Max}^{\leq_M}(X))$ as well. Otherwise, there is an $m' \in \mathrm{Max}^{\leq_M}(X)$ with $m \leq_M m'$, hence $n = \varphi(m) \leq_N \varphi(m')$, and $\varphi(m') \in \varphi(\mathrm{Max}^{\leq_M}(X))$. □

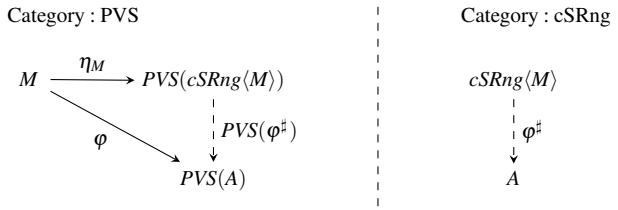


Figure 4 Diagram of the free c-semiring over a PVS. As with previous free constructions, $cSRng\langle M\rangle$ only identifies and orders elements as absolutely required by c-semiring axioms – it is again most general.

Finally, we define the functor $cSRng\langle -\rangle : \text{PVS} \to \text{cSRng}$ as

$$cSRng\langle M\rangle = (\mathcal{I}_{\text{fin}}(M), \breve{\cup}_M, \tilde{\cdot}_M, \emptyset, \{\varepsilon_M\}) ,$$
$$cSRng\langle \varphi : M \to N\rangle = \lambda\{m_1,\dots,m_k\} \in \mathcal{I}_{\text{fin}}(M) \,.\, \text{Max}^{\leq_N}\{\varphi(m_1),\dots,\varphi(m_k)\} .$$

We need to check (using Lemma 4) that $cSRng\langle \varphi : M \to N\rangle$ is indeed a c-semiring homomorphism from $cSRng\langle M\rangle$ to $cSRng\langle N\rangle$ for the functor to be well-defined:

$$cSRng\langle\varphi\rangle(\emptyset) = \emptyset , \quad cSRng\langle\varphi\rangle(\{\varepsilon_M\}) = \{\varphi(\varepsilon_M)\} = \{\varepsilon_N\} ,$$
$$cSRng\langle\varphi\rangle(I_1 \breve{\cup}_M I_2) = cSRng\langle\varphi\rangle(\text{Max}^{\leq_M}(I_1 \cup I_2)) =$$
$$\text{Max}^{\leq_N}(\varphi(\text{Max}^{\leq_M}(I_1 \cup I_2))) = \text{Max}^{\leq_N}(\varphi(I_1 \cup I_2)) = \text{Max}^{\leq_N}(\varphi(I_1) \cup \varphi(I_2)) =$$
$$cSRng\langle\varphi\rangle(I_1) \breve{\cup}_N cSRng\langle\varphi\rangle(I_2) ,$$
$$cSRng\langle\varphi\rangle(I_1 \tilde{\cdot}_M I_2) = cSRng\langle\varphi\rangle(\text{Max}^{\leq_M}\{m_1 \cdot_M m_2 \mid m_1 \in I_1,\ m_2 \in I_2\})) =$$
$$\text{Max}^{\leq_N}(\varphi(\text{Max}^{\leq_M}\{m_1 \cdot_M m_2 \mid m_1 \in I_1,\ m_2 \in I_2\})) =$$
$$\text{Max}^{\leq_N}\{\varphi(m_1 \cdot_M m_2) \mid m_1 \in I_1,\ m_2 \in I_2\} =$$
$$\text{Max}^{\leq_N}\{\varphi(m_1) \cdot_N \varphi(m_2) \mid m_1 \in I_1,\ m_2 \in I_2\} =$$
$$\text{Max}^{\leq_N}\{n_1 \cdot_N n_2 \mid n_1 \in \varphi(I_1),\ n_2 \in \varphi(I_2)\} =$$
$$cSRng\langle\varphi\rangle(I_1) \tilde{\cdot}_N cSRng\langle\varphi\rangle(I_2) .$$

With these functors from PVS to cSRng and vice versa defined, we are ready to apply Definition 3 to the problem of finding the free c-semiring over a PVS, as depicted in Figure 4. As unit morphism, we define $\eta_M : M \to PVS(cSRng\langle M\rangle)$ for every PVS M by $\eta_M(m) = \{m\}$. Now let M be some PVS, A a c-semiring, and $\varphi : M \to PVS(A)$ be a PVS-homomorphism. Again, we search a lifting $\varphi^\sharp$ that "emulates" (and extends) the PVS-homomorphism φ at the c-semiring level, i.e., makes the diagram in Figure 4 commute by asserting that $PVS(\varphi^\sharp) \circ \eta_M = \varphi$. We define $\varphi^\sharp : cSRng\langle M\rangle \to A$ as a function $\varphi^\sharp : \mathcal{I}_{\text{fin}}(M) \to |A|$ and need to show that it is a c-semiring homomorphism:

$$\varphi^\sharp(\{m_1,\dots,m_n\}) = \varphi(m_1) \oplus_A \cdots \oplus_A \varphi(m_n)$$

for all $\{m_1,\dots,m_n\} \in \mathcal{I}_{\text{fin}}(M)$, where, if $n = 0$, $\emptyset$ is mapped to $\mathbf{0}_A$; $\varphi^\sharp$ is indeed a c-semiring homomorphism, since for the constants, $\varphi^\sharp(\mathbf{0}_{cSRng\langle M\rangle}) = \varphi^\sharp(\emptyset) = \mathbf{0}_A$ and $\varphi^\sharp(\mathbf{1}_{cSRng\langle M\rangle}) = \varphi^\sharp(\{\varepsilon_M\}) = \varphi(\varepsilon_M) = \varepsilon_{PVS(A)} = \mathbf{1}_A$. To show that $\varphi^\sharp$ preserves the operations $\tilde{\cdot}_M$ and $\breve{\cup}_M$, we first note that for each finite set $\{m_1,\dots,m_n\} \subseteq |M|$ (not necessarily composed of incomparable elements) it holds that $\varphi^\sharp(\text{Max}^{\leq_M}\{m_1,\dots,m_n\}) = \varphi(m_1) \oplus_A \dots \oplus_A \varphi(m_n)$: if some dominating $m_i \leq_M m_j$ exists, then $\varphi(m_i) \leq_{PVS(A)} \varphi(m_j)$ (since φ is a PVS-homomorphism),

hence, $\varphi(m_i) \oplus_A \varphi(m_j) = \varphi(m_j)$. We can thus "remove" each occurrence of the dominated m_i in $\varphi(m_1) \oplus_A \ldots \oplus_A \varphi(m_n)$ since its dominator m_j is included in that term. Therefore,

$$\begin{aligned}
&\varphi^\sharp(\{m_1,\ldots,m_k\} \tilde{\cup}_M \{m_{k+1},\ldots,m_n\}) = \varphi^\sharp(\mathrm{Max}^{\leq_M}\{m_1,\ldots,m_n\}) = \\
&\quad \varphi(m_1) \oplus_A \ldots \oplus_A \varphi(m_n) = \\
&\quad (\varphi(m_1) \oplus_A \ldots \oplus_A \varphi(m_k)) \oplus_A (\varphi(m_{k+1}) \oplus_A \ldots \oplus_A \varphi(m_n)) = \\
&\quad \varphi^\sharp(\{m_1,\ldots,m_k\}) \oplus_A \varphi^\sharp(\{m_{k+1},\ldots,m_n\}) \,.
\end{aligned}$$

Similarly, for two sets $I, J \in \mathcal{I}_{\mathrm{fin}}(M)$

$$\begin{aligned}
&\varphi^\sharp(I \tilde{\cdot}_M J) = \varphi^\sharp(\mathrm{Max}^{\leq_M}\{m_1 \cdot_M m_2 \mid m_1 \in I,\ m_2 \in J\})) = \\
&\quad \bigoplus_A \{\varphi(m_1 \cdot_M m_2) \mid m_1 \in I,\ m_2 \in J\} \overset{\text{PVS hom.}}{=} \\
&\quad \bigoplus_A \{\varphi(m_1) \cdot_{PVS(A)} \varphi(m_2) \mid m_1 \in I,\ m_2 \in J\} = \\
&\quad \bigoplus_A \{\varphi(m_1) \otimes_A \varphi(m_2) \mid m_1 \in I,\ m_2 \in J\} \overset{\text{distr.}}{=} \\
&\quad \bigoplus_A \{\varphi(m_1) \mid m_1 \in I\} \otimes_A \bigoplus_A \{\varphi(m_2) \mid m_2 \in J\} = \varphi^\sharp(I) \otimes_A \varphi^\sharp(J) \,.
\end{aligned}$$

Thus, $\varphi^\sharp$ is a c-semiring homomorphism and additionally, $PVS(\varphi^\sharp)(\eta_M(m)) = \varphi(m)$, i.e., the diagram in Figure 4 commutes, and $\varphi^\sharp$ is unique with this property (the proof is analogous to that of Lemma 2). We may thus conclude:

Lemma 5 *$cSRng\langle M\rangle$ is the free c-semiring over the partial valuation structure M.* □

From the fact that the composition of two free constructions is a free construction itself [54, Ch. 3], we further know:

Corollary 1 *$cSRng\langle PVS\langle P\rangle\rangle$ is the free c-semiring over the partial order P.* □

Therefore, we abbreviate $cSRng\langle PVS\langle P\rangle\rangle$ as $cSRng\langle P\rangle$ and obtain a *generic* way to embed *any* partial order P into a c-semiring in a canonical way. More explicitly, this c-semiring has finite sets of incomparable (w.r.t. the Smyth-ordering) multisets composed of elements from $|P|$ as its elements. If, e.g., $P = (\{\mathrm{I}, \mathrm{II}, 1, 2\})$, then $\{\lbag \mathrm{I}, 2 \rbag\}$ or $\{\lbag \mathrm{I}, \mathrm{I} \rbag, \lbag 1, \mathrm{I} \rbag\}$ are in $\mathcal{I}_{\mathrm{fin}}(\mathcal{M}_{\mathrm{fin}}(P))$ but $\{\lbag \mathrm{I}, \mathrm{I} \rbag, \lbag \mathrm{II} \rbag\}$ is not since $\lbag \mathrm{I}, \mathrm{I} \rbag \preceq^P \lbag \mathrm{II} \rbag$ (cf. Figure 5 for a similar ordering). In the following section, we revisit this free c-semiring to illustrate the application of $\oplus$ and the required distributivity with respect to possible solving algorithms.

3.5 Adequacy of Algebraic Structures for Soft Constraints

The original goal of algebraic abstractions of specific soft constraint formalisms was to provide a common theoretical ground for questions of computational complexity and, perhaps more intensely studied, efficient solving algorithms. The latter include search strategies, dynamic programming techniques, and constraint propagation.

In terms of model expressiveness, we seek a fairly general structure that captures a broad variety of formalisms. In terms of algorithmic efficiency, however, we are inclined to sacrifice generality for additional structure that makes search and propagation more effective. Most algorithmic efforts can roughly be divided into:

– Classical search algorithms such as branch-and-bound, limited discrepancy search, or large neighborhood search [61] with accompanying search heuristics and efficient bounding techniques such as russian doll search or mini bucket elimination [44].

- Soft local consistency and soft global constraints to enhance a search scheme [19, 18]
- Dynamic programming algorithms (variable elimination, bucket elimination, cluster tree elimination) [10, 11, 21]

Originally, valued constraints and c-semiring-based soft constraints generalized weighted constraints and fuzzy constraints, respectively. While c-semirings additionally allowed for partiality to better represent incomparable decisions, valued constraints put a total ordering first instead of an operator for the supremum.[6] Totality is beneficial for solving as it reduces search to more well-known scalar optimization tasks with a unique optimal solution degree and allow for more efficient pruning. Soft local consistency techniques with non-idempotent combination operators further require so-called "fair" valuation structures that admit a difference operator $a \ominus b$ – which is not mandatory for a PVS.

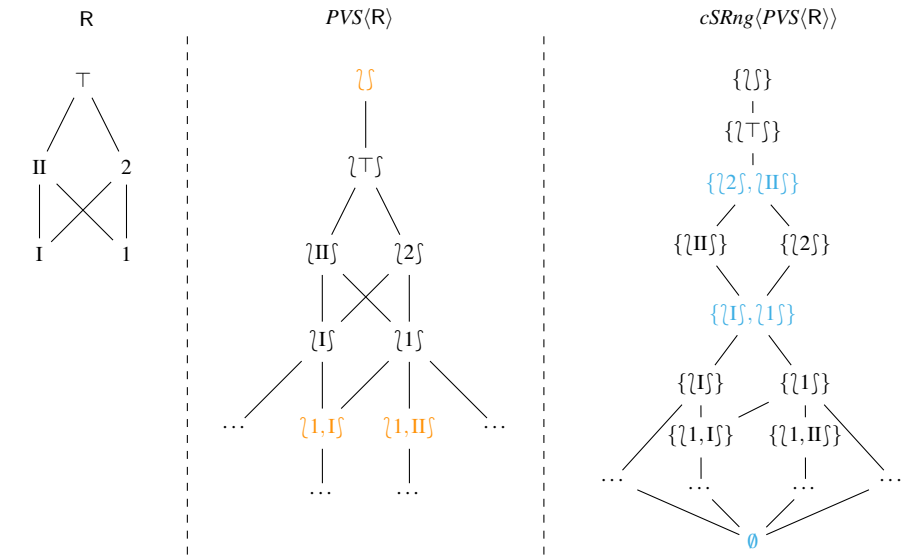
Similarly, the supremum $\oplus$ presupposed by a c-semiring is put to use in non-serial dynamic programming such as bucket elimination whenever we perform a "projection" operation. Projection means finding the best extension (with a greater variable scope) of a given assignment. If we are dealing with PVS without a supremum (such as the free PVS), these algorithms are not directly applicable. However, as a remedy, we can still use this family of algorithms if we put in place the free c-semiring instead. Example 2 demonstrates this procedure for bucket elimination. This algorithm proceeds by picking a variable elimination order, leading to "buckets" for each variable x which collect all soft constraints μ that have x as next (not yet eliminated) variable in their scope. Intermediate soft constraints ν are generated by taking the union of all variables in a bucket, then calculating the intermediate results (i.e., the combination over all soft constraint valuations in the bucket) for each assignment in the Cartesian product of the domains, and projecting out x (see [21]). One can check that each elimination step is an application of the distributivity law (see Lemma 3). All known limitations regarding time and space which prohibit widespread usage in practice, of course, remain [44].

Example 2 Consider a decision that is made based on some abstract "rating system" R (as can be seen in Figure 5) that is inspired by, e.g., two executives that make an independent choice, denoted by $\{1,2\}$ and $\{\mathrm{I},\mathrm{II}\}$, respectively, where a higher number means a *better* evaluation. Any "two", however, is better than any "one". There is an explicit top element $\top$ representing maximal satisfaction. There is no unique least upper bound for 1 and I, though. We assume that soft constraints are specified by a map from variable assignments to elements of $|\mathsf{R}|$, as presented in the figure. To consider combinations of individual soft constraint valuations, i.e., to have a proper *SCSP*, we use the free partial valuation structure $PVS\langle\mathsf{R}\rangle$ to obtain a multiplication. We represent every element r other than $\top$ as $\eta^{\mathrm{PVS}}_{\mathsf{R}}(r) = \wr r \int$ and let $\top$ map to the neutral element $\wr\int$. Note how the resulting partial order $PO(PVS\langle\mathsf{R}\rangle)$ over $\mathcal{M}_{\mathrm{fin}}(|\mathsf{R}|)$ is not suprema-closed (center in Figure 5). To still be able to apply bucket elimination, we embed $PVS\langle\mathsf{R}\rangle$ into its associated free c-semiring $cSRng\langle PVS\langle\mathsf{R}\rangle\rangle = cSRng\langle\mathsf{R}\rangle$. Consequently, we embed any soft constraint μ mapping to $|\mathsf{R}|$ into $cSRng\langle\mathsf{R}\rangle$ as follows:

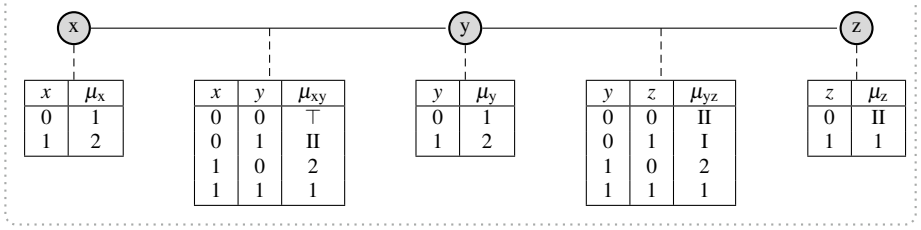
$$\mu^{\mathsf{e}}(\theta) = \eta^{\mathrm{cSRng}}_{PVS\langle\mathsf{R}\rangle}(\eta^{\mathrm{PVS}}_{\mathsf{R}}(\mu(\theta))) = \begin{cases} \{\wr\int\} & \text{if } \mu(\theta) = \top \\ \{\wr\mu(\theta)\int\} & \text{if } \mu(\theta) \neq \top \end{cases} \tag{2}$$

For instance, $\mu^{\mathsf{e}}_{\mathrm{x}}(\{\mathrm{x} \mapsto 1\}) = \{\wr 2\int\}$ and $\mu^{\mathsf{e}}_{\mathrm{xy}}(\{\mathrm{x} \mapsto 0, \mathrm{y} \mapsto 0\}) = \{\wr\int\}$. Finally, we invoke bucket elimination to obtain the optimal solution degrees of (see [21] for a similar illustration). The algorithm terminates with the optimal solution degree $\{\wr 1,1,\mathrm{II},\mathrm{II}\int, \wr \mathrm{I},2,2,\mathrm{II},\mathrm{II}\int,$

[6] Obviously, in a total ordering, the supremum is just min/max.



SCSP



x	μ_x
0	1
1	2

x	y	μ_{xy}
0	0	$\top$
0	1	II
1	0	2
1	1	1

y	μ_y
0	1
1	2

y	z	μ_{yz}
0	0	II
0	1	I
1	0	2
1	1	1

z	μ_z
0	II
1	1

Bucket z:
In: μ_z, μ_{yz}
Out: ν_y

y	z	$\nu_{yz} = \mu_{yz} \otimes \mu_z$
0	0	$\{\ʅII,IIʃ\}$
0	1	$\{\ʅ1,Iʃ\}$
1	0	$\{\ʅ2,IIʃ\}$
1	1	$\{\ʅ1,1ʃ\}$

y	$\nu_y = \bigoplus_{z' \in \{0,1\}} \nu_{yz}(y,z')$
0	$\{\ʅII,IIʃ\}$
1	$\{\ʅ2,IIʃ\}$

Bucket y:
In: μ_y, μ_{xy}
In: ν_y
Out: ν_x

x	y	$\nu_{xy} = \nu_y \otimes (\mu_{xy} \otimes \mu_y)$
0	0	$\{\ʅ1,II,IIʃ\}$
0	1	$\{\ʅ1,2,II,IIʃ\}$
1	0	$\{1,2,II,IIʃ\}$
1	1	$\{\ʅ1,2,2,IIʃ\}$

x	$\nu_x = \bigoplus_{y' \in \{0,1\}} \nu_{xy}(x,y')$
0	$\{\ʅ1,II,IIʃ\}$
1	$\{\ʅ1,2,II,IIʃ,$ $\ʅ1,2,2,IIʃ\}$

Bucket x:
In: μ_x, ν_x
Out: ν

x	$\nu = \mu_x \otimes \nu_x$
0	$\{\ʅ1,1,II,IIʃ\}$
1	$\{\ʅ1,2,2,II,IIʃ,\ʅ1,2,2,2,IIʃ\}$

$\text{osd} = \nu(0) \oplus \nu(1)$
$\{\ʅ1,1,II,IIʃ,$ $\ʅ1,2,2,II,IIʃ,$ $\ʅ1,2,2,2,IIʃ\}$

Figure 5 The upper part illustrates the rating system (R), its free partial valuation structure $PVS\langle R\rangle$ and the free c-semiring $cSRng\langle R\rangle = cSRng\langle PVS\langle R\rangle\rangle$. Highlighted elements are introduced by axioms of the respective algebraic structure. The center part presents a *SCSP* with variables $\{x,y,z\}$, domain $\{0,1\}$ and five (unary or binary) soft constraints that map assignments to $|R|$. The lower part finds the optimal solution degrees of *SCSP* by applying bucket elimination on $cSRng\langle R\rangle$ with the elimination order $\langle x,y,z\rangle$. Note that valuations of soft constraints μ are embedded into $cSRng\langle R\rangle$ according to Equation (2).

$\wr 1,2,2,2,\mathrm{II}\int\}$ that is clearly not reachable by any individual assignment. However, each of the three components (i.e., multiset over $|\mathsf{R}|$) corresponds to one assignment. By employing appropriate bookkeeping during the elimination process, we find that $\theta_1 = \{\mathrm{x} \mapsto 0, \mathrm{y} \mapsto 0, \mathrm{z} \mapsto 0\}$, $\theta_2 = \{\mathrm{x} \mapsto 0, \mathrm{y} \mapsto 1, \mathrm{z} \mapsto 0\}$, and $\theta_3 = \{\mathrm{x} \mapsto 1, \mathrm{y} \mapsto 1, \mathrm{z} \mapsto 0\}$ map to the respective optimal solution degrees and are thus optimal solutions. The free c-semiring provides enough information for said bookkeeping – another c-semiring returning a supremum of all solution degrees need not do this, in general.

Note that in fact we get a set of all PVS-optima as the unique optimal solution degree in the free c-semiring. Clearly, however, enforcing totality or a supremum for the only sake of better algorithms might counteract a modeler's intentions. Some (in reality incomparable) solutions are dominated by others. If we do not rely on explicit soft constraint operations but rather formulate it as a conventional constraint optimization problem that is solved by search and propagation (as in branch-and-bound or large neighborhood search), *the structure a PVS offers suffices* – which makes them the appropriate data structure for designing MiniBrass.

4 Implementation

Our considerations up to now have been mostly abstract and mathematical in terms of proper constructions of PVS and their relationship to c-semirings. We now turn to the design of MiniBrass[7] and how it includes existing formalisms in the literature. Indeed, the argumentation in Section 3.5 motivates that PVS are the adequate algebraic structure to *encode* soft constraint formalisms such that the theoretical constructions substantiate the MiniBrass language. MiniZinc, on the other hand, offers a well-balanced compromise between expressive power (high-level concepts such as functions and predicates) and broad support by a variety of solvers including propagation engines such as Gecode or JaCoP but also other paradigms such as MIP or SAT solvers. To smoothen the transition between conventional constraint models and soft constraint models, MiniBrass follows many MiniZinc conventions such as having a "solve" item, independence of order of statements and the notation, in general.

MiniBrass is a file-based soft constraint modeling language that revolves around the concept of partial valuation structures. A model (resp., instance) is divided into a (hard) *constraint model* (see Listing 1) written in conventional MiniZinc, consisting of variable definitions and classical constraints, and a *preference model* (see Listing 2) which contains PVS type declarations along with instantiations, soft constraint definitions based on the variables in the constraint model, and combinations (Pareto and lexicographic) of instances. MiniBrass separates essential constraints of a problem and its objective because:

- Existing soft constraint formalisms in the literature (weighted, fuzzy, constraint preferences, ...) are available for a preference model using the respective PVS types.
- Preferences can be elicited and specified using PVS type $\mathcal{A}$ (perhaps having a non-trivial (multi)set-based order such as the free PVS) which is then transformed to another PVS type $\mathcal{B}$ that is better supported by existing solvers (cost function networks/integer objectives) using morphisms (see Section 4.3).
- By exploiting modularity, users can combine several preference structures (perhaps stemming from different agents) at runtime (Pareto or lexicographic).
- Multiple preference models (e.g. user perspectives) for the same hard constraint model can exist and be selected depending on other context factors.

[7] `https://github.com/isse-augsburg/minibrass`

```
include "hello-world_o.mzn"; % output of minibrass
include "soft_constraints/pvs_gen_search.mzn"; % for generic branch and bound

% the basic, "classic" CSP
set of int: NURSES = 1..3;
int: day = 1; int: night = 2; int:off = 3;
set of int: SHIFTS = {day,night,off};
array[NURSES] of var SHIFTS: n;

% Encodes a multiset embedding of a boolean expression e over 1..maxP
% that returns {{ }} iff e is true and {{ index }} else.
function array[int] of var int: embed(var bool: expression,
                                      par int: index, par int: maxP) = let {
   set of int: P = 1..maxP;
   array[P] of var 0..1: returnedMSet;
   constraint expression -> returnedMSet == [0 | q in P];
   constraint not(expression) -> (forall(q in P) (
                                  (returnedMSet[q] = 0 /\ index != q) \/
                                  (returnedMSet[q] = 1 /\ index = q) ) );
 } in returnedMSet;

solve
:: pvsSearchHeuristic % compiler writes this into classic_o.mzn
search pvs_BAB(); % calls to a generic PVS-based branch-and-bound

output ["n = \(n)\nValuations:  topLevelObjective = \(topLevelObjective)\n"];
```

Listing 1 `hello-world.mzn`: The conventional constraint model contains all variable definitions and hard constraints. It includes the compiled MiniBrass output (`hello-world_o.mzn`) which contains generated variables, linking constraints, search procedures relevant to MiniSearch (`pvs_BAB`, defined in `pvs_gen_search.mzn`), and the (optional) search annotation `pvsSearchHeuristic`.

Conceptually, the main idea of how to encode a soft constraint problem as a conventional constraint optimization problem has been outlined in [44] after being first described in [47]: For a soft constraint problem $SCSP = (X, D, C, M, S)$ with PVS $M = (|M|, \cdot_M, \varepsilon_M, \leq_M)$, we define the classical constraint model (X, D, C) as usual and for every soft constraint $\mu_i \in S$, we have a constraint along the lines of "$\texttt{valuation}[\texttt{i}] = \mu_i(X)$" where `valuation` is an array of variables of type $|M|$ and $\mu_i(X)$ stands for the soft constraint (MiniZinc) expression based on variable assignments to X. Additionally, there is an $|M|$-variable `overall` holding the overall valuation which is constrained such that "$\texttt{overall} = \texttt{valuation}[1] \cdot_M \ldots \cdot_M \texttt{valuation}[\texttt{nScs}]$" where `nScs` refers to the number of soft constraints. The partial ordering $\leq_M$ is used to generate constraints on future solutions such as "$\texttt{overall} <_M \texttt{overall}'$" to ask for the next solution $\texttt{overall}'$ to be *better* than the current one `overall`. Branch-and-bound (see Section 4.5) is based on this predicate.

Listings 1 and 2 exemplify these ideas using the introductory toy rostering problem covered in Figure 1. The problem is specified with constraint preferences which are mapped to the free PVS as discussed in Section 3.3.[8] Each soft constraint μ_i maps to $\wr i \int$ if violated and $\wr \int$ otherwise (implemented by the function `embed` in Listing 1 since conditional statements in MiniZinc cannot have array-valued return types). The corresponding PVS-type `FreePVS` implements multisets of elements from an underlying partial order as solution degrees, multiset union as the combination operation, and the Smyth-ordering as MiniZinc functions and predicates in the file `free-pvs-type.mzn`. Said partial order P is passed by parameters `maxP` (denoting the highest index) and `orderingP` (the ordering relation as list of edges) during instantiation.

[8] Listing 2 presents the type definition as well but the type is actually implemented in `defs.mbr` such that it can be readily used in any MiniBrass instance.

```
type FreePVS = PVSType<mset[maxOccurrences] of 1..maxP> =
  params {
    array[int, 1..2] of 1..nScs: orderingP;
    int: maxP;
    int: maxPerSc;
    int: maxOccurrences :: default('mbr.nScs * mbr.maxPerSc');
  } in
  instantiates with "soft_constraints/mbr_types/free-pvs-type.mzn" {
    times -> multiset_union;
    is_worse -> isSmythWorse;
    top -> {};
  };

PVS: fp = new FreePVS("fp") {
   soft-constraint c1: 'embed(sum(i in NURSES)(bool2int(n[i] = night)) = 2, 1, 3)';
   soft-constraint c2: 'embed(n[2] in {day,off}, 2, 3)';
   soft-constraint c3: 'embed(n[3] = off, 3, 3)';

   orderingP : '[| 2, 1 | 3, 1 |]';
   maxP: '3' ;
   maxPerSc : '1';
};

solve fp;
```

Listing 2 `hello-world.mbr`: A preference model of the problem in Figure 1 with PVS type for a free PVS and one PVS-instance that also serves as the "solve"-item analogous to MiniZinc. An optimal solution is, e.g., `n = [day, day, off]` with solution degree $\wr p_1 \int$. Note how an edge (p_2, p_1) hereby indicates that $p_2 \leq_P p_1$ and thus $\wr p_2 \int \preceq^P \wr p_1 \int$.

Multisets are not natively supported by MiniZinc and need thus to be encoded for solvers. In the free PVS, the set $\mathcal{M}_{\text{fin}}(P)$ is clearly infinite as we can reach any finite multiset over P by applying combination (i.e., multiset union) often enough. Since solvers operate on *finitely* many decision variables with finite domains, we never have to use the full range of $\mathcal{M}_{\text{fin}}(P)$, though, and always operate on a finite subset of it. Put differently, the maximal multiplicity of any element of P is necessarily restricted. To put a meaningful upper bound on the multiplicities, we note that in a *SCSP*, the overall valuation is given by $\prod_M \{\mu_i(\theta) \mid \mu_i \in S\}$. If we can determine the maximal occurrence any P-element has in any individual $\mu_i(\theta)$, say k, we simply use $\texttt{nScs} \cdot k$ as the maximal occurrence for the overall valuation. Taking constraint preferences as instance, this value is in fact easy to determine as any soft constraint can obviously only be violated once – we exploit this further in Section 4.2.2. With these restrictions, a multiset composed of P-elements with maximal multiplicity `maxOccurrences` is just encoded as an `array[1..maxP] of var 0..maxOccurrences`. We discuss the model elements further in the following sections.

The toolchain needed for MiniBrass adds an additional preceding step to the conventional MiniZinc/MiniSearch workflow. Figure 6 illustrates the involved processes: First, the MiniBrass preference model (e.g., Listing 2) is compiled into MiniZinc or MiniSearch code using `mbr2mzn`. During this process, auxiliary variables taking soft constraint valuations and their aggregations, improvement and not-worsening constraints for branch-and-bound, as well as variable orderings for search heuristics and complex order predicates (in case of Pareto or lexicographic combinations) are generated. Finally, the classical constraint model (e.g., Listing 1) includes the compiled MiniBrass output and is solved by MiniZinc (`mzn2fzn`) or MiniSearch (`minisearch`).

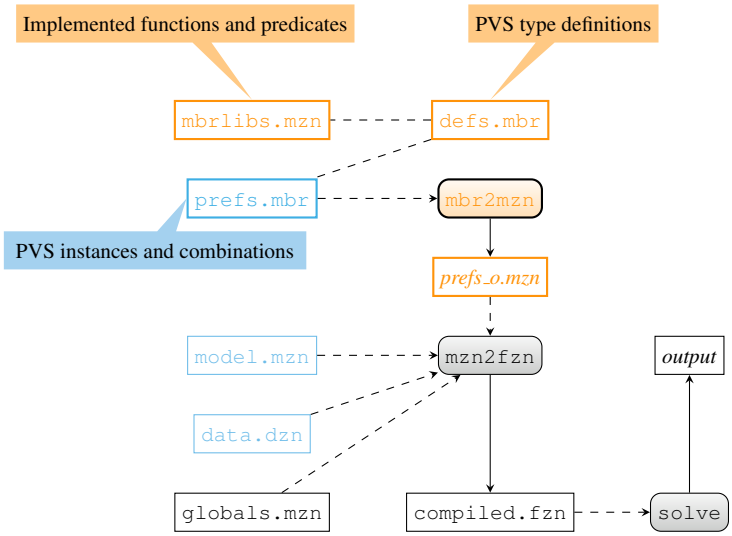


Figure 6 Toolchain of MiniBrass and its integration with MiniZinc. Blue elements indicate artifacts that have to be created for every new problem instance whereas orange ones refer to reusable (MiniBrass) library items that do not need to be modified by end users.

4.1 PVS Types and Instantiations

Every PVS type definition has to specify the possible solution degrees, ordering predicate and combination operation in MiniZinc. The solution degrees (semantically the underlying set of the PVS) are referred to as the *element type E* of a PVS type. This can be any MiniZinc base type such as `int`, `float`, `bool`, or subtypes thereof as well as sets or arrays of a base type. For instance, the aforementioned free PVS uses `mset` which is a *MiniBrass* keyword abbreviating an array of integers representing a multiset. Weighted constraints or cost function networks map to `int` or some integer range `0..k`, and fuzzy constraints employ the float range `0.0 .. 1.0`.

The element type E corresponds to (a subset) of the carrier set $|M|$ of a resulting PVS instance $M = (|M|, \cdot_M, \varepsilon_M, \leq_M)$ and prescribes the signature of the ordering $\leq_M$, i.e., $\leq_M \subseteq E \times E$. Canonically, every soft constraint μ_i maps to E and the combination operation has to be a function $\cdot_M : E \times E \to E$, as was shown in Listing 2. However, there are cases where the essential model information is different from an E-element – compare Figure 5 or our previous toy example where soft constraints map to E but are wrapped by an embedding function. This becomes more evident when we consider PVS that are based on "violated soft constraints" such as, e.g., weighted constraints. Each soft constraint $\mu_c : [X \to D] \to |M|$ (parametrized by some conventional constraint c and associated weight w_c) then has the form $\mu_c(\theta) = 0$ if $\theta \models c$ and w_c otherwise: The *essential* information here is whether $\theta \models c$. Wrapping every boolean expression (as done in Listing 2 with the `embed` function) obviously leads to cluttered and less readable constraint and preference models, also evidenced by even having the `embed` definition in Listing 1. To avoid this clutter on the syntactic level, PVS types can be augmented with a *soft constraint type S* that defines the type of each soft

constraint expression (here `bool`). We semantically define a mapping $g_i : S \to E$ that maps each S-expression of soft constraint μ_i to an E-element, all of which are combined with $\cdot_M$. However, there are cases when we do not want or need to have `nScs` E-expressions but rather emulate the above combination using an embedding $S^n \to E$ – especially if there are beneficial global constraints involved (see Listing 3). In both cases, we end up with an overall valuation of type E that is ordered using $\leq_M$ during search. As a consequence, we have the combination operation map several S-expressions to one element in E (where $S = E$ is of course valid) and have it mimic the $\cdot_M$ operation on E.

Moreover, PVS types may declare parameters that need to be specified during the instantiation of concrete PVS in a preference model. Examples thereof are the maximally allowed weight k in weighted constraints or the preference graph in constraint preferences. The MiniZinc implementation of the combination function and ordering predicate has to be supplied in a separate MiniZinc file that will be included by the compiled MiniZinc output. All presented types are included in the MiniBrass standard library but new definitions can likewise be added with regard to extensibility.

To sum up, for a PVS type parametrized by soft constraint type S and element type E, the ordering predicate, the combination, and the top element have to implement the following interfaces:

```
predicate is_worse(var E: x, var E: y, par int: nScs, [PARAMETERS]);
function var E: times(array[int] of var S: v, par int: nScs, [PARAMETERS]);
par E: top;
```

Note that the ordering of the parameters must match the order in the PVS type declaration. In a similar way, some PVS types offer generic search heuristics that can be provided (see the keyword `pvsSearchHeuristic` below). The interface is expected to be:

```
function ann: searchHeuristic(array[int] of var S: values, var E: overall,
                              par int: nScs, [PARAMETERS]);
```

It should generally be noted that `is_worse` always corresponds to a predicate denoting *strict* worsening (this is the most common type of predicate used in branch-and-bound). The `top` element is beneficial for bounding search and having default soft constraints.

Besides the type declarations, there are a few "technical" MiniBrass keywords that are sure to be found in the compiled MiniZinc output (e.g., `prefs_o.mzn` in Figure 6) and can thus be accessed from the constraint model (e.g., `model.mzn` in Figure 6):

- `topLevelObjective`: contains a `var E`-expression for an *atomic* top level PVS (the instance specified in the `solve` item), with element type E; not applicable if the top level PVS is complex (e.g., a lexicographic product). It appears in the output in Listing 1 and could also be a MiniZinc objective: `solve minimize topLevelObjective` (only if E is scalar).
- `pvsSearchHeuristic`: contains an annotation object for the top level PVS that holds a particular variable order (of the generated variables) that depends on the PVS-type(s) involved (see Section 4.2.1). For complex PVS, multiple heuristics are concatenated sequentially.
- `postGetBetter`: contains a MiniSearch procedure that is used to post a constraint requiring the next solution to be better than the current one. The generic branch-and-bound procedure `pvs_BAB` used in Listing 1 (which is defined in `pvs_gen_search.mzn` and explained in Section 4.5) relies on `postGetBetter` being written by the compiler.
- `postNotGetWorse`: dually, this MiniSearch procedure only requires the next solution not to be dominated (important to find all optima of an instance).

Once PVS types are declared, we can use them for the instantiation of concrete PVS objects. A PVS object stores a specific set of parameters and includes the actual soft constraints mapping to E (or S) as MiniZinc expressions – thereby connecting the constraint and preference model. In addition, the operators `pareto` and `lex` can be used to compose complex preference structures from elementary ones.

4.2 Examples of Soft Constraint Formalisms as PVS Types

For illustration purposes, we survey the most common soft constraint formalisms (see Section 2) presented as PVS types. Throughout the examples, we assume a simplistic classical constraint model without any actual hard constraints except for the domain restrictions:

```
set of int: DOM = 1..3;
var DOM: x; var DOM: y; var DOM: z;
```

4.2.1 Integer-Valued: Weighted CSP or Cost Function Networks

The PVS types for weighted constraints and cost function networks are naturally very similar. The latter are defined as integer-valued soft constraints that map any assignment to some value in the range $[0 \dots k]$ for some parameter k denoting maximal violation. Consequently, there is no distinct soft constraint type but just the element type `0..k`.

```
type CostFunctionNetwork = PVSType<0..k> =
  params {
    int: k :: default('1000');
  } in
  instantiates with "soft_constraints/mbr_types/cfn_type.mzn" {
    times -> k_bounded_sum;
    is_worse -> is_worse_weighted;
    top -> 0;
 };
```

Combination means adding individual costs (capping at k) and the ordering relation is the integer greater-than ordering (consistently with literature, cost minimization is default).

```
% Inside soft_constraints/mbr_types/cfn_type.mzn
predicate is_worse_weighted(var int: x, var int: y, int: nScs, int: k) =
  x > y;

function var int: k_bounded_sum(array[int] of var int: b, int: nScs, int: k) =
  if sum(b) > k then k else sum(b) endif;
```

Moreover, to facilitate better access to native cost function implementations in Toulbar2, the MiniBrass library also offers a global constraint (along with a default decomposition for other solvers) that is handled by Numberjack and properly given to Toulbar2. For instance,

```
predicate cost_function_binary(var int: x, var int: y,
                               array[int] of int: costs, var int: costVariable)
```

In a similar spirit, soft global constraints are implemented in MiniBrass. Since the examples shown in the literature map to a numeric variable, they naturally fit into the framework of cost function networks. For instance, a soft variant of `alldifferent` counts the variables taking the same value as a measure of violation:

```
function var int: soft_all_different(array[int] of var int: x) :: promise_total =
  let { set of int: seenValues = dom_array (x); }
  in (sum(s in seenValues) (max(count(x, s) - 1, 0)));
```

```
% [...] Used in a MiniBrass preference model
include "soft_constraints/soft_alldifferent.mzn";
% [...]
array[STUDENT] of var PROJECT: x;

soft-constraint c1: 'soft_alldifferent(x)';
```

There are native implementations for `soft_alldifferent` in JaCoP [41] which can make use of dedicated propagation instead of the provided decomposition – precisely like in conventional global constraints.

By contrast, weighted constraints focus on a binary judgment – a soft constraint is violated or not, and if so, punished with a weight. This is reflected by using the soft constraint type `bool` that is mapped to the element type `0..k`.

```
type WeightedCsp = PVSType<bool, 0..k> =
  params {
    int: k :: default('1000');
    array[1..nScs] of int: weights :: default('1');
  } in
  instantiates with "soft_constraints/mbr_types/weighted_type.mzn" {
    times -> weighted_k_bounded_sum;
    is_worse -> is_worse_weighted;
    top -> 0;
 }
 offers {
    heuristics -> getSearchHeuristicWeighted;
 };
```

Weighted constraints represent the first example of a generic search heuristic annotation that is foreseen by the PVS type. The MiniZinc function `getSearchHeuristicWeighted` (included in the file `weighted_type.mzn`) provides a particular variable ordering: the variables containing the highest-weighted possible violation first (called *most important first* in [57]). That way, search can start by setting the generated satisfaction variables of all soft constraints to true and let propagation take over to possibly find high-quality solutions early.

```
function ann: getSearchHeuristicWeighted(array[int] of var bool: degs,
                                         var int: overall,
                                         par int: nSoftConstraints,
                                         int: k, array[int] of int: weights) =
let {
  set of int: sCs = 1..nSoftConstraints;
  % find the sorted permutation of soft constraint instances
  array[sCs] of sCs: sortPerm = arg_sort(weights);
  % invert, since arg_sort use <= and we need decreasing order
  array[sCs] of sCs: mostImpFirst = [ sortPerm[nSoftConstraints-s+1] | s in sCs];
  array[sCs] of var bool: mifSatisfied = [ degs[mostImpFirst[s]] | s in sCs];
} in
int_search(mifSatisfied, input_order, indomain_max, complete);
```

Section 5.3 provides some insight in the effectiveness of the above search heuristic.

There is a subtle double-usage of the PVS type `WeightedCsp`. As we set the weights' default value to 1, we have a Max-CSP instance if no weights are supplied. We can however add weights (more generally parameter values that are connected to every soft constraint) by annotating a soft constraint during the instantiation of a weighted PVS.

```
PVS: wcsp = new WeightedCsp("wcsp") {
  soft-constraint c1: 'x + 1 = y' :: weights('2');
  soft-constraint c2: 'z = y + 2' :: weights('1');
  soft-constraint c3: 'x + y <= 3' :: weights('1');
  k : '20';
};
solve wcsp;
```

4.2.2 Comparative: Free PVS and Constraint Preferences

Next, we revisit purely comparative preference structures that operate directly on partial orders such as the free PVS previously seen in Listing 2. As mentioned above, its element type is `mset[maxOccurrences] of 1..maxP` which is syntactic sugar for an `array[1..maxP] of var 0..maxOccurrences` that represents the overall solution degree. This type is the most general available in MiniBrass and can be used to encode any partial order such as the rating system R in Figure 5 as a PVS.

Each individual soft constraint has to map to a multiset with bounded multiplicity, as indicated by the parameters. While the combination (`multiset_union`) is straightforward by just taking the sums of individual soft constraints' element multiplicities, implementing a MiniZinc predicate for the Smyth-ordering (cf. Section 3.2) is a bit more involved. In essence, to establish $T \prec^P U$ the key idea is to apply Lemma 1 and have the witness $h : \mathcal{S}(U) \to \mathcal{S}(T)$ be *decided* by the solver using local decision variables of the predicate. Recall that $\mathcal{S}(U)$ refers to the set of pairs representation of a multiset. Thus h is defined on *pairs* and has to obey the constraints $p \leq_P q$ whenever $h(j,q) = (k,p)$. The injectivity of h is best represented by an `alldifferent`-constraint but there is none for pairs. We can mitigate this by constructing a one-dimensional witness and apply the bijective Cantor pairing function $\pi : \mathbb{N}^2 \to \mathbb{N}$ defined by $\pi(k_1,k_2) := \frac{1}{2}(k_1+k_2)(k_1+k_2+1)+k_2$. The resulting one-dimensional array can be constrained to be all different (i.e., injective), as usual.

```
predicate isSmythWorse ( array[int] of var int: T,
              array[int] of var int: U, int: nScs,
              array[int, 1..2] of int: edges,
              int: maxP, int: maxPerSc, int: maxOccurrences
          ) = let {
  set of int: P = 1..maxP;
  par int: maxOcc = maxPerSc*maxOccurrences;
  set of int: OCCS = 0..maxOcc;
  set of int: PosOCCS = OCCS diff {0};
  set of int: P0 = {0} union P; % 0 representing no assignment

  int: le = min(index_set_1of2(edges));
  int: ue = max(index_set_1of2(edges));

  array[P] of set of P: lessThanOrEquals =
    [ {q} union {p | p in P where exists(e in le..ue)
         (edges[e,1] = p /\ edges[e,2] = q)} |  q in P];

  % We have to split the witness function h : S(U) \to S(T) into
  % two arrays of decision variables.
  array[OCCS,P] of var P0: witnessElem;
  array[OCCS,P] of var OCCS: witnessOcc;

  % First, we make sure all (j,q) tuples for occurrences j greater than the
  % actual number of q elements in U map to non-existence.
  constraint forall(q in P, j in OCCS where j > U[q]) (
    witnessElem[j,q] = 0 /\ witnessOcc[j,q] = 0
  );

  % Now, for all (j,q) tuples in S(U), they have to map
  % to a (k,p) tuple in S(T) such that p <= q.
  constraint forall(q in P, j in PosOCCS where j <= U[q])  (
    % p must not be 0      and p must be leq than q
    witnessElem[j,q] != 0 /\ witnessElem[j,q] in lessThanOrEquals[q] /\
    % k must be between 1 and the actual number of p-occurrences in T
    witnessOcc[j,q] >= 1 /\ witnessOcc[j,q] <= T[witnessElem[j,q]]
  );

  % Lastly, we have to assert injectivity of our witness.
  % Since there is no alldifferent propagator on pairs, we use
  % the Cantor pairing function to map S(U) to the integers and
  % constrain the Cantorized witness to be alldifferent.
  array[OCCS,P] of var 0 .. maxP + (maxOcc) * (maxOcc+maxP+1) div 2:
```

```
    cantoredWitness;
  constraint forall(i in OCCS, p in P) (
   cantoredWitness[i,p] = witnessOcc[i,p] + (witnessElem[i,p]+witnessOcc[i,p])
                        *(witnessElem[i,p]+witnessOcc[i,p]+1) div 2);

  constraint alldifferent_except_0([cantoredWitness[i,p] | i in OCCS, p in P]);
  % A bit of symmetry breaking on the exchangeable occurrences
  constraint value_precede_chain(OCCS, [witnessOcc[i,p] | i in OCCS, p in P]);

  % Make sure we have inequality
  constraint exists(i in P) (T[i] != U[i]);
} in (  true  );
```

At this point, we want to emphasize that end-users (i.e., modelers) do not need to fully understand the implementation of the Smyth-ordering in MiniZinc but only its (rather intuitive) inductive definition to apply it in their models. The above definition is fully encapsulated by the `freePVS`-type. We have to note, however, that this predicate relies on *local free variables* (e.g., `witnessOcc`) which prohibit its usage in a negative or mixed context [66] such as "it must *not* be the case that the next solution is Smyth-worse than the current" (cf. non-domination search in Section 4.5).

By construction, `freePVS` is well-suited to transform any partial order such as, e.g., the rating system R into a PVS. However, for problems involving constraint preferences (such as Listing 2), `freePVS` might seem too rich in generality. We only observe the multisets $\wr i \int$ or $\wr \int$ for distinct soft constraints μ_i. Similar to weighted constraints, we could thus make use of the soft constraint type `bool` that relieves us from the `embed` function in Listing 2 (`times` uses $\wr \int$ if a soft constraint μ_i holds and `msetVal[i]` otherwise):

```
PVS: freePrefs = new FreeConstraintPreferences("freePrefs") {
  soft-constraint c1: 'x + 1 = y'   :: msetVal('[1,0,0]');
  soft-constraint c2: 'z = y + 2'   :: msetVal('[0,1,0]');
  soft-constraint c3: 'x + y <= 3'  :: msetVal('[0,0,1]');
  [...]
};
```

However, we can further improve the encoding of the free PVS for constraint preferences by noting that no element i can occur more than once in any reachable overall valuation. Each such m in $\mathcal{M}_{\text{fin}}(P)$ can then just be *represented* by a *set* of integers – a type that is natively supported with appropriate global constraints by MiniZinc and several constraint solvers. Hence, we define a dedicated PVS type `ConstraintPreferences` with `bool` as soft constraint type and `set of int` as element type that, in fact, only operates on a certain subset of the free PVS.

More precisely, we use `set of 1..nScs`, and identify each soft constraint with a number. By channeling the boolean expressions evaluating to `false` to a set for the combination, we obtain the set of all violated soft constraints (see Listing 3). The Smyth-ordering on sets (cf. Section 3.2) is also implemented as a MiniZinc predicate and is activated by having the `useSPD` parameter set to true. Alternatively, one may use the so-called *transitive-predecessor-ordering* (TPD) [40] that defines a more important constraint to dominate a whole set of less important ones:

$$\forall p_i \in \{p_1, \ldots, p_n\} : p_i \leq_P q \Rightarrow T \uplus \wr p_1, \ldots, p_n \int \prec^P_{\text{TPD}} T \uplus \wr q \int$$
$$T \supset U \Rightarrow T \prec^P_{\text{TPD}} U$$

Clearly, $T \preceq^P U \rightarrow T \preceq^P_{\text{TPD}} U$, i.e., TPD adds ordering relations to SPD. But perhaps more usefully, the above predicate is easier to decide, i.e., no free local variables are involved – hence, TPD is suited for mixed and negative contexts such as non-domination search.

```
include "link_set_to_booleans.mzn";
function var set of int: link_invert_booleans(array[int] of var bool: b,
  par int: nScs, array[int, 1..2] of par int: crEdges, par bool: useSPD) =
let {
  var set of index_set(b): violatedSet;
  constraint link_set_to_booleans(violatedSet, [ not b[i]  | i in index_set(b) ]);
} in violatedSet;
```

Listing 3 Using the MiniZinc global `link_set_to_booleans` to connect reified soft constraints with an overall solution degree denoting the set of violated soft constraints.

Of course, an instance of `ConstraintPreferences` also needs the actual DAG (the `crEdges` parameter) over soft constraints. For convenience, we include here that the transitive closure is automatically calculated by MiniBrass during compilation (turning the DAG into an ordering) – as an example of a parameter wrapping method. Such methods could either be MiniZinc functions for data transformation or Java methods. Ensuring correct user input (i.e., acyclicity or other validations by means of MiniZinc assertions or Java exceptions) can be done here as well.

```
type ConstraintPreferences = PVSType<bool, set of 1..nScs> represents FreePVS =
  params {
    array[int, 1..2] of 1..nScs: crEdges ::
      wrappedBy('java', 'isse.mbr.extensions.preprocessing.TransitiveClosure');
    bool: useSPD :: default('true');
  } in
  instantiates with "soft_constraints/mbr_types/cr_type.mzn" {
    times -> link_invert_booleans;
    is_worse -> is_worse_cr;
    top -> {};
  }
  offers {
    heuristics -> getSearchHeuristicCR;
  };
```

Furthermore, the fact that the type `ConstraintPreferences` represents valuations in the underlying `FreePVS` can optionally be made explicit in the code using the keyword `represents`. Although currently this does not affect code generation, we consider it valuable for documentation purposes and perhaps the ability to generate redundant constraints using the represented PVS type.

The restriction to `set of int` also eases the implementation of the Smyth-ordering as we do not have functions over pairs – as opposed to the multiset case. As a corollary to Lemma 1, on two *sets* T and U, $T \preceq^P U$ holds if and only if there exists an *injective* witness function $f : U \to T$ such that $f(p) \leq_P p$ for all $p \in U$. Similar to the multiset case, we enforce (and propagate) the injectivity of f with `alldifferent` and make sure the witness property is fulfilled.

4.2.3 Real-Valued: Fuzzy CSP and Probabilistic CSP

A third class of soft constraint formalism is best characterized by the element type being the reals over $[0.0, 1.0]$. Starting with fuzzy constraints, each soft constraint maps to $[0.0, 1.0]$ with the combination being defined as the minimum operator – in this case, soft constraint type and element type coincide.

```
type FuzzyConstraints = PVSType<0.0 .. 1.0> =
  instantiates with "soft_constraints/mbr_types/fuzzy_type.mzn" {
    times -> min;
    is_worse -> is_worse_fuzzy;
```

```
    top -> 1.0;
};
```

The resulting soft constraints of element type `0.0 .. 1.0` could directly be defined as MiniZinc functions but we added some support as a global constraint which is included in `fuzzy_type.mzn`. For instance, consider a soft constraint μ_1 defined over two boolean variables `mainCourse` and `wine`: $\mu_1 = \{(0,0) \to 1.0, (0,1) \to 0.8, (1,0) \to 0.3, (1,1) \to 0.7\}$. In MiniBrass, this can be written as follows:[9]

```
PVS: fz1 = new FuzzyConstraints("fz1") {
   soft-constraint mu1: 'fbinary_fuzzy([1.0, 0.8, 0.3, 0.7], mainCourse, wine)';
   soft-constraint mu2: 'fbinary_fuzzy([1.0, 0.8, 0.8, 1.0], mainCourse, lunch)';
};
solve fz1;
```

On the other hand, probabilistic constraints bear similarities to both weighted and fuzzy constraints. We use `bool` as soft constraint type to denote violated constraints and `0.0 .. 1.0` for probabilities as element type. Formally, the objective is $\prod_{\mu_i:\theta \not\models \mu_i} 1 - p_i$. The "constraint presence" probabilities p_i are, analogously to weights, supplied as parameters.

```
type ProbabilisticConstraints = PVSType<bool, 0.0 .. 1.0> =
  params {
    array[1..nScs] of float: probs :: default('1.0');
  } in
  instantiates with "soft_constraints/mbr_types/probabilistic_type.mzn" {
    times -> prod;
    is_worse -> is_worse_prob;
    top -> 1.0;
};
[...]
% usage example
soft-constraint c2: 's1 + s2 >= 10' :: probs('0.7');
```

Both fuzzy and probabilistic constraints aim at *maximization* of the solution degree such that `is_worse_prob(x,y)` corresponds to `x < y`.

4.3 Morphisms to Switch PVS

There are at least two reasons why users specify their *SCSP* using one PVS type but *solve* the problem using another: If the original PVS shows many incomparable optimal solutions, we might want to totalize the ordering – if only for testing and debugging. But more frequently, solvers do not support the data structures required to represent a PVS type even though they have to be used for performance reasons or due to the target software environment. For instance, set-based types for constraint preferences or real-valued domains with suitable global constraints for fuzzy constraints are not universally implemented. A modeler would only accept transforming the *SCSP in a structure-preserving way*: at least, existing strict "is better than" decisions in the original ordering are not to be contradicted; at most, incomparable assignments may become comparable – precisely what PVS-homomorphisms offer.

We saw an example in Figure 2 where we compared $PVS\langle P\rangle$ and $Weighted(P)$, i.e., we can calculate a weight for each constraint and transform a constraint preferences problem into a weighted CSP instance. In MiniBrass, we first define a morphism

[9] Note that the encoding employs a table constraint for floats which is not supported well by many solvers. Therefore a workaround using integers is also provided in the MiniBrass library that can be seen as another example for a PVS type representing a different one.

```
% defined in the MiniBrass library
morph ConstraintPreferences -> WeightedCsp: ToWeighted =
  params {
    k = 'mbr.nScs * max(i in 1..mbr.nScs) (mbr.weights[i]) ';
    weights = calculate_cr_weights;
  } in id;
```

using a function that is applied to each original soft constraint expression (here just the identity `id`) and then transforming a specific PVS instance:

```
PVS: cr1 = new ConstraintPreferences("cr1") {
  soft-constraint c1: 'x + 1 = y';
  soft-constraint c2: 'z = y + 2';
  soft-constraint c3: 'x + y <= 3';

  crEdges : '[| mbr.c2, mbr.c1 | mbr.c3, mbr.c1 |]';
  useSPD: 'false' ;
};
solve ToWeighted(cr1);
```

By devising similar morphisms for other PVS types, we can integrate the previously mentioned fact that many soft constraint formalisms can be (monotonically) encoded as cost function networks in polynomial time [58], the type for which Toulbar2 offers efficient dedicated algorithms. For instance, a probabilistic PVS having a multiplicative maximization objective $f(\theta) = \prod_{\mu_i:\theta\not\models\mu_i} 1 - p_i$ can be transformed into an additive minimization problem by taking the negative logarithm of f: $-\log f(\theta) = \sum_{\mu_i:\theta\not\models\mu_i} -\log p_i$ where we can precalculate the $-\log p_i$ terms as weights:

```
% a morphism converting a probabilistic CSP to weighted CSP using log
morph ProbabilisticConstraints -> WeightedCsp: ProbToWeighted =
  params generatedBy('isse.mbr.extensions.weighting.ProbWeighting') {
    k = 'mbr.nScs * max(i in 1..mbr.nScs) (mbr.weights[i])';
    weights = generated;
  } in id;
```

The above-mentioned calculation here takes place in the class `ProbWeighting`, indicated by the `generated` keyword. While this morphism definition is mathematically proper, for an implementation as weighted CSP, we have to round the terms to the nearest integer.

4.4 Products of PVS

An important advantage of algebraic soft constraint specifications is their modular nature – regardless of the actual underlying PVS type. Partial valuation structures are always closed under direct products and have lexicographic products under certain conditions [28,55]. Formally, for two PVS M and N, we can construct the direct product as follows:

$$(m,n) \leq_{M\times N} (m',n') \leftrightarrow m \leq_M m' \wedge n \leq_N n'$$

which is denoted by `pareto` in MiniBrass as it corresponds to a Pareto-ordering over the underlying orderings of M and N. Similarly, the lexicographic product is defined as

$$(m,n) \leq_{M\times N} (m',n') \leftrightarrow (m <_M m') \vee (m = m' \wedge n \leq_N n')$$

and denoted by `lex` in MiniBrass. It allows us to express hierarchical relationships between PVS. We can combine these two operators and morphisms to form complex PVS. Consider these exemplary use cases:

```
solve cfn1 pareto cfn2;
solve cfn1 lex cfn2;
solve ToWeighted(cfn1) pareto (cfn2 lex cfn3);
```

4.5 PVS-based Search

With the tools at hand, we are able to define PVS-types, instantiate them, and combine and morph them to more complex structures. The overall goal is the PVS passed in the `solve`-item. To find optimal solutions, MiniBrass relies on classical systematic constraint solving and optimization using propagation and search, as outlined in the beginning of Section 4. The necessary facilities are provided by MiniZinc/MiniSearch and the underlying solvers. If the element type is numeric, the problem can be solved in *MiniZinc* by minimizing (or maximizing) `topLevelObjective`. However, the full strength of abstract soft constraint formalisms precisely is the presence of partial and product orders. *MiniSearch* provides blueprints for various of the classical searches that can be customized that way.

The first search strategy corresponds to classical branch-and-bound (BAB) search in propagation engines. For every found solution, a constraint is imposed that the next solution has to be strictly better.

```
% Only declare minisearch function; implementation generated during MiniBrass compilation
function ann: postGetBetter();

function ann: pvs_BAB() =
  repeat(if next()
         then print("Intermediate solution:") /\ print() /\ commit() /\ postGetBetter()
         else break
         endif);
```

While this procedure yields optimal solutions, it is not ideal for partially ordered objectives since another optimum does not have to be *better* than the current solution. Instead, it *must not be dominated* by *any* solution seen so far [36]. When solving for a PVS M, we have a set of lower bounds (the valuations of previous solutions) $L = \{l_1, \ldots, l_m\} \subseteq |M|$ and require that it must not be that $\exists l \in L : \texttt{obj} \leq_M l$ where `obj` denotes the generated MiniZinc variable(s) holding the overall objective. The next solution must be strictly better than any one of the maxima of L or incomparable to all of them.

```
function ann: postNotGetWorse();

function ann: pvs_BAB_NonDom() =
  repeat(if next()
         then print("Intermediate solution:") /\ print() /\ commit() /\ postNotGetWorse()
         else break
         endif);
```

There is a caveat to this solution. With the `is_worse` predicates that PVS types offer, we can generate a MiniSearch procedure "`postNotGetWorse`" during compilation as well. However, we have to *negate* this predicate, i.e., change its boolean context. This leads to problems if the predicate shows free local variables [66]. We have seen this in Section 4.2.2 for the witness function necessary to decide the Smyth-ordering which is not compatible with `postNotGetWorse`. For constraint preferences, we have to resort to the TPD-ordering instead. Since we expect future non-trivial PVS-types to rely on local variables, we need modelers to be aware of this restriction.

Example 3 Consider the following simplified example to illustrate the difference:

```
% In the classical constraint model:
var 1..3: x;
solve :: int_search([x], input_order, indomain_max, complete)
search pvs_BAB_NonDom();

% In the preference model
PVS: cr1 = new ConstraintPreferences("cr1") {
  soft-constraint c1: 'x in {2,3}';
```

```
  soft-constraint c2: 'x in {1,3}';
  soft-constraint c3: 'x in {1,2}';

  crEdges : '[| mbr.c2, mbr.c1 | mbr.c3, mbr.c1 |]';
  useSPD: 'false';
};

solve cr1;
```

We explore x in a decreasing order. Each assignment to x violates precisely one soft constraint. This results in the sequence $\langle\{3\},\{2\},\{1\}\rangle$ of solution degrees. $\{3\}$ and $\{2\}$ both dominate $\{1\}$ but are incomparable using TPD-ordering (and Smyth, too). The reachable optima of this problem are clearly $\{\{2\},\{3\}\}$ but `pvs_BAB` would stop after $\{2\}$ since $\{3\}$ is not better. By contrast, `pvs_BAB_NonDom` returns both optimal solution degrees.

MiniSearch actually offers much more flexibility in crafting problem-specific searches than just branch-and-bound. For instance, designing large-neighborhood-search for PVS-based models can be done using their concepts of scopes, as described in [49].

```
% Adapted from lns_max an objective value
function ann: pvs_LNS(array[int] of var int: x,
                      int: iterations, float: d, int: exploreTime) =
  repeat (i in 1..iterations) (
    print("Starting iteration ... \(i)\n") /\
    scope(post(neighbourhoodCts(x,d)) /\
          time_limit(exploreTime, pvs_BAB()) /\ commit() /\
          print("Intermediate solution\n") /\ print()) /\ postGetBetter()
  );
```

In a similar way, we can anticipate many variants of search algorithms with `postGetBetter` or `postNotGetWorse`. By means of the separation of concerns between constraint and preference model, the preference model in MiniBrass can be tested with various searches.

5 Evaluation

To evaluate MiniBrass in a way that complements other experimental work (such as the recent substantial evaluation of Toulbar2 in the context of graphical models [35]), we decided to model soft constraint problems using the PVS type *constraint preferences* (see Section 4.2.2). We used MiniZinc benchmark problems[10] as the underlying constraint models. These are taken from several editions of the MiniZinc challenge [65]. Optimizing according to constraint preferences requires set-based variables and compatibility with MiniSearch. By applying morphisms as described in Section 4.3, we obtain *weighted CSP* versions that are compatible with a wider range of solvers.

Alternatively, we could have resorted to the existing cost function networks benchmark library that also offers MiniZinc models for a tabularized encoding.[11] However, conventional constraint solvers have already been shown to be dominated on these problems by Toulbar2. Moreover, these problems only address one particular PVS. Optimizing according to, e.g., the Smyth-ordering in the context of soft constraint problems has not been addressed before. In particular, we introduce some partiality in the models which generally makes the task of finding optima more demanding due to reduced pruning.

[10] `https://github.com/MiniZinc/minizinc-benchmarks`

[11] `https://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/BenchmarkS` and `https://github.com/MiniZinc/minizinc-benchmarks/blob/master/proteindesign12/wcsp.mzn`

The problems were selected according to features that justify an encoding approach (i.e., efficient conventional propagation), feasibility of decompositions for many solvers, and "meaningful soft constraint addition". Certainly, there are cost function network problems (e.g., in bioinformatics) that are out of reach for conventional solvers, as [35] demonstrated. However, we argue that there are many practical cases with relatively few soft constraints and many conventional constraints. Among those, we investigate the following problems:

Soft N-Queens is a toy *SCSP* that adds three artificial soft constraints with a preference relation over them to a classical N-queens problem (such as, e.g., having a queen in the center of the grid), not to be mistaken with the M-queens optimization problem.

Photo Placement asks to place people close to their friends – in its original version it was already designed to handle preferences but we (morally questionably) allowed for some friends to be *more important* to stand close to than others.

Talent Scheduling aims at scheduling movie scenes including various actors cost-effectively. We augmented the conventional problem with preferences to avoid being simultaneously on set with a rival actor and early/late times for specific scenes.

On-call Rostering requires to assign staff members to days in a rostering period, respecting work constraints and unavailabilities. The original formulation already contained preferences for not being on-call for more than two days in a row or not being on-call for a weekend and a consecutive day. We modeled these existing preferences in MiniBrass and added additional ones regarding preferred co-workers.

Multi-Skilled Project Scheduling (MSPSP) is a variant of resource-constrained project scheduling and asks to assign a set of tasks to workers such that the required set of skills for a task is provided by its assigned worker. To add soft constraints, we again allowed workers to state with whom they would like to work and which tasks they would like to work on or which ones they would rather avoid.

More detailed information about the changed and added aspects can be found online.[12] Each problem is tested with three to six instances, totaling 28 evaluation instances. Since most of these problems already were formulated as constraint optimization problems to begin with, we had to deal with two objectives: the original one and the soft constraint objective. First, we converted the problems to constraint satisfaction problems by imposing the original objective value to lie within a 10%–15% boundary around the (previously determined) optimal value, and eventually used the soft constraint objective. Clearly, this process is not feasible for real optimization problems but is rather aimed at producing challenging soft constraint benchmark problems.

We solved the resulting models (including parameters that will be subject to the respective experiment questions) using branch-and-bound[13] (cf. Section 4.5) with the classical constraint solvers Gecode 5.0.0 [59], JaCoP 4.4.0 [41], Google OR-Tools 5.1.0 (CP solver) [29], Choco 4.0.3 [37], and G12 1.6.0 [64], as well as with the only competitive cost function network solver Toulbar2 0.9.8 [3], accessed via Numberjack 1.1.0 [31]. Each presented experiment was run on a machine having 4 Intel Xeon CPU 3.20 GHz cores and 14.7 GB RAM on a 64 bit Ubuntu 15.04 with a timeout set to 10 minutes per instance.

Our primary goal is to demonstrate the feasibility of implementing soft constraint formalisms more generally than a numeric objective at low runtime overheads – a capability that is not shared by any state-of-the-art soft constraint solver. Besides, even in the realm of cost

[12] `https://github.com/isse-augsburg/minibrass/tree/master/evaluation/problems`

[13] We experimented with large neighborhood search as well but did not find it to be effective enough for the selected evaluation problems.

function networks and weighted constraints, it can pay off to use an encoding approach with a conventional constraint solver as opposed to a dedicated soft constraint solver. Furthermore, we examine the effects of our proposed search heuristics for weighted constraints.

5.1 Comparing Performance: Encoded Weighted CSP versus Native Toulbar2

If we want to obtain a comparative view on the performance of MiniBrass models, we have to use cost function networks. For one thing, they are the native formalism Toulbar2 supports, for another thing the task boils down to minimizing a numeric value in conventional models which is directly supported by MiniZinc. On the one hand, Toulbar2 can be seen as the only true state-of-the-art alternative to MiniBrass (given that `WSimply` [6] has no MiniZinc or Numberjack interface and only runs on a 32 bit Linux distribution) – on the other hand it serves as a well-supported backend. Therefore, this evaluation cannot be truly seen in a competitive light as MiniBrass is a modeling language. Here, the central question is:

How fast and effectively (in terms of finding optima) can WCSP instances be solved by encoding them as COPs versus using a dedicated solver?

Table 1 presents the results for this first question with times and objectives being averaged over all instances for the respective problem, ranked by runtimes.[14] Values in parentheses denote *averaged relative values* with respect to the minimum (ratio for time or excess penalty violation, resp.) for each instance – as opposed to *relative average values*. Therefore, the relative overhead does not necessarily correlate with the absolute values (e.g., Toulbar2 versus Gecode on Talent Scheduling). The number of wins indicates how many instances a solver won (i.e., being fastest) within the respective problem. If an instance was not solved at all within the specified time limit, the maximally possible violation for it was assumed.

We observed a fairly even distribution of solvers performing well with OR-Tools being among the top three on all problems, showing the most reliable contribution of all conventional constraint solvers. In addition to the table, we noted that over all problems, OR-Tools had the lowest average runtime (97.39 secs) and the lowest average objective value (6.18), whereas Gecode achieved the most wins (12). Interestingly, Toulbar2 managed to achieve the best (or second best) average runtimes for three problems, excelling in On-call Rostering. However, the memory-intensive decompositions required for MSPSP and Talent Scheduling had Toulbar2 fail during model creation without returning a solution. To conclude, even though Toulbar2 is a strong choice when dealing with cost function networks, there are cases where only an encoding approach succeeded at all (MSPSP) – or substantially faster (Talent Scheduling). With problems modeled in MiniBrass, both options remain.

5.2 Comparing Models: Smyth-Optimization versus Weighted-Optimization

Upon learning that weighted instances can be solved efficiently by conventional constraint solvers, we extend our considerations to optimization according to the Smyth-ordering. MiniBrass was explicitly designed to offer more abstract orderings than numeric objectives – in particular, Smyth as the ordering of the free PVS. We want to quantify how expensive the partiality of an original model is with respect to the totalization obtained by

[14] The fact that Choco shows a higher average objective on Photo Placement albeit claiming to have proved optimality results from a bug in the solver induced by the problem-specific search heuristics.

Table 1 Comparison of solvers' performance on the weighted CSP representations. Values in parentheses denote *averaged relative values* with respect to the minimum (ratio for time or excess penalty violation).

Solver	Time (secs)		# Wins	Objective		% Solved	% Optimal
MSPSP (8 instances)							
Gecode	0.32	(1.00)	8	2.50	(0.00)	100.00	100.00
G12	0.32	(1.01)	0	2.50	(0.00)	100.00	100.00
OR-Tools	0.33	(1.05)	0	2.50	(0.00)	100.00	100.00
JaCoP	0.52	(1.73)	0	2.50	(0.00)	100.00	100.00
Choco	0.70	(2.46)	0	2.50	(0.00)	100.00	100.00
Toulbar2	312.56	(1052.07)	0	29.13	(26.63)	0.00	0.00
On-call Rostering (7 instances)							
Toulbar2	40.73	(1.44)	3	1.57	(0.00)	100.00	100.00
OR-Tools	275.23	(5.55)	2	3.71	(2.14)	100.00	57.14
Gecode	275.23	(5.54)	1	4.57	(3.00)	100.00	57.14
G12	276.36	(5.63)	1	5.57	(4.00)	100.00	57.14
JaCoP	276.63	(5.86)	0	5.14	(3.57)	100.00	57.14
Choco	276.72	(6.26)	0	5.14	(3.57)	100.00	57.14
Photo Placement (3 instances)							
Toulbar2	0.80	(1.11)	0	13.33	(0.00)	100.00	100.00
Choco	0.83	(1.21)	2	25.00	(11.67)	100.00	100.00
OR-Tools	1.49	(1.71)	1	13.33	(0.00)	100.00	100.00
JaCoP	3.18	(3.61)	0	13.33	(0.00)	100.00	100.00
Gecode	22.24	(21.62)	0	13.33	(0.00)	100.00	100.00
G12	27.40	(29.62)	0	13.33	(0.00)	100.00	100.00
Soft N-Queens (3 instances)							
OR-Tools	0.03	(1.00)	3	0.33	(0.00)	100.00	100.00
Toulbar2	0.30	(10.43)	0	0.33	(0.00)	100.00	100.00
Choco	0.35	(12.54)	0	0.33	(0.00)	100.00	100.00
JaCoP	57.22	(1707.98)	0	0.33	(0.00)	100.00	100.00
Gecode	210.02	(6266.00)	0	1.67	(1.33)	100.00	66.67
G12	210.02	(6266.14)	0	1.67	(1.33)	100.00	66.67
Talent Scheduling (7 instances)							
OR-Tools	113.29	(1.01)	3	12.29	(0.00)	100.00	85.71
JaCoP	117.71	(1.84)	0	12.29	(0.00)	100.00	85.71
Choco	129.12	(3.27)	1	12.29	(0.00)	100.00	85.71
Toulbar2	158.27	(60.70)	0	28.43	(16.14)	28.57	28.57
Gecode	183.29	(4.70)	3	12.29	(0.00)	100.00	85.71
G12	194.91	(2.87)	0	12.29	(0.00)	100.00	85.71

weighting constraints. To solve these models, only Gecode and JaCoP are applicable, as they are both compatible with MiniSearch and support set-based variables to the necessary extent. For these solvers, we compare the running times and objective values[15] for the original Smyth-based model and the (morphed) weighted CSP. Gecode is provided in a native version directly accessed by MiniSearch (see Section 2) and a FlatZinc-based execution – with the latter being more recent than the native one. JaCoP is only available using FlatZinc. Where applicable, i.e., if Toulbar2 solved the instances, we additionally provide its reference values (Toulbar2 is restricted to the weighted version). Here, the central question is:

Is optimizing according to the Smyth-ordering much more expensive than solving a weighted counterpart obtained by a morphism?

Table 2 presents our results answering this question. Note that, for this evaluation, the Smyth-based models have been solved with strict domination BaB since this is the only way

[15] Note that the "objective values" for the Smyth-model are provided only for comparative reasons. Optimization was done purely according to the Smyth-ordering on the set of violated soft constraints.

Table 2 Comparing the solvers' performance on a Smyth-based model and the weighted CSP representations. Times and objectives are averaged over all instances for a given problem and can be compared. We only considered *solved* instances in this evaluation. Bold-face highlighting indicates the faster model per solver. Runtimes of Toulbar2 on the weighted instances are given "out of competition" where applicable – i.e., if the decomposition succeeded on all competing instances. Times are given in seconds.

Solver	Time Smyth	Time Weighted	Time Toulbar2	Obj. Smyth	Obj. Weighted
MSPSP (6 instances)					
Gecode	12.74	**0.34**	-	5.50	2.67
Native Gecode	7.82	**0.26**	-	5.80	2.80
JaCoP	4.18	**0.45**	-	6.00	2.00
On-call Rostering (5 instances)					
Gecode	220.46	**133.32**	*14.52*	7.20	3.20
Native Gecode	192.50	**133.32**	*14.52*	25.20	3.20
JaCoP	194.06	**135.28**	*14.52*	26.80	3.20
Photo Placement (3 instances)					
Gecode	6.69	**1.03**	*0.68*	13.00	13.00
Native Gecode	**9.96**	22.22	*0.80*	13.33	13.33
JaCoP	15.73	**3.18**	*0.80*	13.33	13.33
Soft N-Queens (3 instances)					
Gecode	**3.45**	210.02	*0.30*	2.00	1.67
Native Gecode	**3.49**	210.02	*0.30*	1.33	1.67
JaCoP	**3.94**	57.22	*0.30*	1.00	0.33
Talent Scheduling (6 instances)					
Gecode	**7.78**	158.94	-	14.25	12.50
Native Gecode	**13.50**	141.09	-	14.67	12.33
JaCoP	**15.63**	120.42	-	14.17	12.33

the totalized weighted version can operate. We expected the weighted problems to be much easier to solve since there is possibly stronger pruning and propagation involved. To our surprise we noticed that, whereas for most instances (87.8%), the weighted counterpart was indeed easier to solve, there were instances where the problem formulated with constraint preferences took substantially less time – as in Talent Scheduling and Soft N-Queens. A possible explanation is that optimality can be easier proved using propagation of the witness function of the Smyth-ordering. Put differently, there could be better solutions in terms of weights but not Smyth, therefore search can be pruned earlier. We may also notice that, on these instances, Toulbar2 can provide much better performance than the constraint solvers on the weighted counterparts – when applicable. This is mostly due to the fact that Choco and OR-Tools are left out (as opposed to Section 5.1) since they currently do not support set variables. In terms of objective values, even though optimality is proven in most cases, the Smyth and weighted versions yield different values which is not surprising as, again, a "weight-better" solution need not be "Smyth-better". Thus, there are generally lower values to be expected using the weighted version. The attentive reader will notice that the average objective for the Smyth-model is in fact *lower* than for the weighted model in Soft N-queens solved by the native Gecode solver. In fact, the solver timed out on one weighted instance at the sub-optimal objective value 4 whereas the Smyth-based variant happened to yield a Smyth-optimal solution that is also weight-optimal with objective value 2.

With strict BaB only, we only get one optimal solution – at best. The advantage of using partial orders clearly is having multiple incomparable optima at modeling and not having to totalize the ordering by weighing. However, searching for a whole set of optima (as done in *non-domination* BaB) obviously leads to longer runtimes than stopping at the first found

Table 3 Comparison of runtimes between searching for all optima instead of a strict domination improvement. We only considered *solved* instances in this evaluation. Bold-face highlighting indicates the faster search type. Values are averaged over instances and solvers. Times are given in seconds.

Problem	Time Non-Dominated BaB	Time Strict BaB	Absolute Overhead	Relative Overhead
MSPSP	**7.31**	8.89	−1.58	1.50
On-call Rostering	329.44	**199.21**	130.23	1.82
Photo Placement	55.09	**7.51**	47.58	9.72
Soft N-Queens	**2.24**	3.65	−1.41	1.91
Talent Scheduling	33.44	**12.24**	21.21	2.30
Overall	102.00	**57.20**	44.80	2.97

optimum (as done in *strict* BaB). We investigate the differences in Table 3 (recall that TPD has to be used for non-domination search, see Section 4.2.2). On the examined benchmark problems, we observe a factor of around two to three in terms of runtime that has to be expected when using non-domination. However, in some cases the difference between non-domination and strict BaB was negligible (i.e., MSPSP and Soft N-Queens) – mostly due to the set of optima actually being small where strict and non-domination BaB converge to similar search trees.

5.3 Comparing Search Heuristics: Most Important First versus Default

Lastly, with abstract higher level preference models, we can use generic search heuristics that align with the optimization goals – dependent on the PVS type in use. Here, our simple strategy (shown in Section 4.2.1) is to try and assign `true` to the boolean variables reifying[16] the satisfaction of soft constraints in the order of decreasing weight (i.e., importance). We refer to this heuristic as *most important first* (MIF). Some of the benchmark problems already shipped with a problem-specific variable ordering heuristic. In such cases, activated MIF prepends the reified satisfaction variables to the existing heuristic. We compare the effects of MIF on various types of searches (strict, non-domination, weighted), problems, and solvers. The central question is:

Can a generic heuristic (MIF) for soft constraint problems speed up the search for optima?

Over all 168 runs across solvers, problem instances, and search types, the MIF heuristic led to a faster runtime in 73 cases (43 %) with the average runtime reduced by 6.22 seconds. Yet, MIF does not uniformly lead to better runtimes but is more effective for some solvers than others. Similarly some problems are affected more positively. Table 4 presents results for this question in a more fine-grained fashion, grouping the evaluated data by problems and solvers, respectively. We find that MIF seems to negatively influence the performance compared to the built-in default search strategies in particular for OR-Tools, JaCoP, and Toulbar2 but can lead to tremendous improvements for Choco (cf. Figure 7(a)). But even if MIF led to faster runtimes on average (e.g., −73.14 seconds for Choco), the *relative* improvement was not as effective – resulting in averaged relative values > 1. For one thing, the relative metric is agnostic of the absolute times, which makes, e.g., a runtime slowdown from 0.53 to 0.57 seconds count about as much as a speedup from 539 to 521 seconds. For another thing, we see that even though the speedup is substantial in several cases (leading to

[16] Certainly, some global constraints cannot be reified yet or only support half-reification but we expect them to increasingly do so [9].

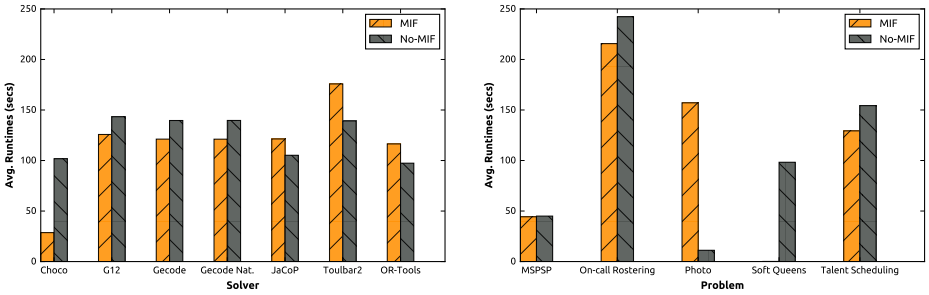
Table 4 Runtime difference between models with MIF activated and deactivated. Negative values indicate that MIF led to faster solving times. Winning ratios are given with respect to the number of instances. Relative runtime differences are again *averaged relative values*. Solved configurations include Smyth-based and weighted models. For the actual runtimes, see Figure 7. Times are given in seconds.

Grouped by solvers

	Choco	G12	Gecode	Native Gecode	JaCoP	Toulbar2	OR-Tools
Instances	28	28	28	28	28	28	28
Runtime difference	**−73.14**	**−17.57**	**−18.42**	**−18.53**	16.15	36.63	19.05
Rel. runtime difference	1.07	1.59	2.52	2.49	7.24	1.18	11.02
Ratio MIF wins	**0.64**	0.32	0.29	0.18	0.46	**0.57**	0.32

Grouped by problems

	MSPSP	On-call Rostering	Photo Placement	Soft N-Queens	Talent Scheduling
Instances	56	49	21	21	49
Runtime difference	**−0.68**	**−26.63**	145.93	**−98.15**	**−24.96**
Rel. runtime difference	1.09	0.92	26.38	0.76	1.70
Ratio MIF wins	0.36	**0.51**	0.05	**0.52**	0.43



(a) Runtimes grouped by solver for MIF on/off.

(b) Runtimes grouped by problem for MIF on/off.

Figure 7 Average runtimes for instances solved with and without MIF activated for several models (Smyth or weighted). Figures correspond to the data showing differences in Table 4.

an overall smaller runtime) there are still many instances that did not experience a speedup or were even slowed down (cf. the ratio of MIF-wins).

Similarly, when grouping by problem, On-call Rostering and Talent Scheduling benefited the most from activating MIF (cf. Figure 7(b)), both showing a speedup on average (relative differences). We suspect that for the other problems, either the built-in heuristics were effective enough or MIF led to thrashing behavior if the best solutions still violated many soft constraints. Then, MIF initiates many "pointless" searches by setting all soft constraints to be satisfied and propagation fails to prove infeasibility fast enough. With problems such as Photo Placement or MSPSP, we admittedly better use the default search strategy. MIF clearly is no silver bullet. However, since activating a search heuristic in MiniBrass only amounts to placing `pvsSearchHeuristic` in front of the search procedure to be used (cf. Listing 1), this could be an easy first step in tweaking the performance of PVS-based models.

6 Conclusion and Future Work

We presented and evaluated MiniBrass, a soft constraint modeling language building on MiniZinc that closes the gap between algebraic soft constraint frameworks and state-of-the-art solvers. We motivated why the concept of free constructions is an appropriate tool to facilitate the transition from partial orders to PVS and from PVS to c-semirings with the least overhead and provided proofs for these constructions. MiniBrass is capable of expressing a broad variety of soft constraint formalisms in the literature that are subsumed by partial valuation structures. Moreover, it allows designing complex preference structures using product operators and morphisms separately from conventional constraints. Finally, we evaluated MiniBrass on a set of "softened" benchmark problems and found that on these problems an encoding-approach is competitive with dedicated soft constraint solving, optimizing with the Smyth-ordering is only slightly more expensive than weighted problems, and the most-important-first heuristic can lead to significant runtime savings.

In the future, we plan to extend MiniBrass to distributed settings where we use other preference aggregation strategies than `pareto` or `lex` to combine several agents' PVS specifications. Moreover, we develop a graphical interface to MiniBrass to facilitate modeling. We also plan to extend the number of available backends such as, e.g., WSimply, to further broaden the applicability of the algebraic soft constraint modeling language MiniBrass.

References

1. Allen, T.E., Chen, M., Goldsmith, J., Mattei, N., Popova, A., Regenwetter, M., Rossi, F., Zwilling, C.: Beyond theory and data in preference modeling: Bringing humans into the loop. In: T. Walsh (ed.) Proc. 4th Intl. Conf. Algorithmic Decision Theory (ADT'15), *Lect. Notes Comp. Sci.*, vol. 9346, pp. 3–18. Springer (2015)
2. Allouche, D., de Givry, S., Katsirelos, G., Schiex, T., Zytnicki, M.: Anytime hybrid best-first search with tree decomposition for weighted CSP. In: G. Pesant (ed.) Proc. 21st Intl. Conf. Principles and Practice of Constraint Programming (CP'15), *Lect. Notes Comp. Sci.*, vol. 9255, pp. 12–29. Springer (2015)
3. Allouche, D., de Givry, S., Schiex, T.: Toulbar2, an open-source exact cost function network solver. Tech. rep., INRIA (2010)
4. Amadio, R.M., Curien, P.L.: Domains and Lambda-Calculi. Cambridge Tracts in Theoretical Computer Science 46. Cambridge University Press (1998)
5. Ansótegui, C., Bofill, M., Palahí, M., Suy, J., Villaret, M.: W-minizinc: A proposal for modeling weighted CSPs with MiniZinc. In: Proc. 1st Intl. Ws. MiniZinc (MZN'11) (2011)
6. Ansótegui, C., Bofill, M., Palahí, M., Suy, J., Villaret, M.: Solving weighted CSPs with meta-constraints by reformulation into satisfiability modulo theories. Constraints **18**(2), 236–268 (2013)
7. Awodey, S.: Category Theory. Oxford University Press (2010)
8. Barr, M., Wells, C.: Category Theory for Computing Science. Prentice Hall (1990)
9. Beldiceanu, N., Carlsson, M., Flener, P., Pearson, J.: On the reification of global constraints. Constraints **18**(1), 1–6 (2013)
10. Bertele, U., Brioschi, F.: On non-serial dynamic programming. J. Combinatorial Theory, Series A **14**(2), 137–148 (1973)
11. Bistarelli, S.: Semirings for Soft Constraint Solving and Programming, *Lect. Notes Comp. Sci.*, vol. 2962. Springer (2004)
12. Bistarelli, S., Fung, S.K.L., Lee, J.H.M., Leung, H.: A local search framework for semiring-based constraint satisfaction problems. In: Proc. Ws. Soft Constraints (Soft'03) (2003)
13. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. J. ACM **44**(2), 201–236 (1997)
14. Bistarelli, S., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G., Fargier, H.: Semiring-based CSPs and valued CSPs: Frameworks, properties, and comparison. Constraints **4**(3), 199–240 (1999)
15. Borning, A., Freeman-Benson, B., Wilson, M.: Constraint hierarchies. LISP Symb. Comp. **5**, 223–270 (1992)
16. Boutilier, C., Brafman, R.I., Domshlak, C., Hoos, H.H., Poole, D.: CP-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. J. Artif. Intell. Res. **21**, 135–191 (2004)

17. Brandt, F., Conitzer, V., Endriss, U.: Computational social choice. In: G. Weiß (ed.) Multiagent Systems, 2nd edn., chap. 6, pp. 213–283. MIT Press (2013)
18. Cooper, M.C., de Givry, S., Sánchez, M., Schiex, T., Zytnicki, M., Werner, T.: Soft arc consistency revisited. Artif. Intell. **174**(7), 449–478 (2010)
19. Cooper, M.C., Schiex, T.: Arc consistency for soft constraints. Artif. Intell. **154**(1), 199–227 (2004)
20. Dalla Pozza, G., Pini, M.S., Rossi, F., Venable, K.B.: Multi-agent soft constraint aggregation via sequential voting. In: T. Walsh (ed.) Proc. 22nd Intl. Joint Conf. Artificial Intelligence (IJCAI'11), pp. 172–177. IJCAI/AAAI (2011)
21. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artif. Intell. **113**(1), 41–85 (1999)
22. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
23. Diaconescu, R.: Category-based semantics for equational and constraint logic programming. Ph.D. thesis, Oxford University, Oxford (1994)
24. Fargier, H., Lang, J.: Uncertainty in constraint satisfaction problems: A probabilistic approach. In: M. Clarke, R. Kruse, S. Moral (eds.) Proc. Europ. Conf. Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU'93), *Lect. Notes Comp. Sci.*, vol. 747, pp. 97–104. Springer (1993)
25. Fioretto, F., Pontelli, E., Yeoh, W.: Distributed constraint optimization problems and applications: A survey (2016). CoRR abs/1602.06347
26. Freuder, E.C., Wallace, R.J.: Partial constraint satisfaction. Artif. Intell. **58**(1–3), 21–70 (1992)
27. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13**(3), 268–306 (2008)
28. Gadducci, F., Hölzl, M., Monreale, G.V., Wirsing, M.: Soft constraints for lexicographic orders. In: F. Castro, A. Gelbukh, M. González (eds.) Proc. 12th Mexican Int. Conf. Artificial Intelligence (MICAI'2013), *Lect. Notes Comp. Sci.*, vol. 8265, pp. 68–79. Springer (2013)
29. Google optimization tools. URL `https://developers.google.com/optimization`. [Online, accessed 2017/06/29]
30. Guns, T., Dries, A., Nijssen, S., Tack, G., De Raedt, L.: MiningZinc: A declarative framework for constraint-based mining. Artif. Intell. **244**, 6–29 (2017)
31. Hebrard, E., O'Mahony, E., O'Sullivan, B.: Constraint programming and combinatorial optimisation in Numberjack. In: A. Lodi, M. Milano, P. Toth (eds.) Proc. 7th Intl. Conf. Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'10), *Lect. Notes Comp. Sci.*, vol. 6140, pp. 181–185. Springer (2010)
32. van Hentenryck, P.: The OPL optimization programming language. MIT Press (1999)
33. van Hoeve, W.J.: Over-constrained problems. In: M. Milano, P. van Hentenryck (eds.) Hybrid Optimization, *Optimization and its Applications*, vol. 45, pp. 191–225. Springer (2011)
34. Hosobe, H.: Constraint hierarchies as semiring-based CSPs. In: Proc. 21st Intl. Conf. Tools with Artificial Intelligence (ICTAI'2009), pp. 176–183 (2009)
35. Hurley, B., O'Sullivan, B., Allouche, D., Katsirelos, G., Schiex, T., Zytnicki, M., de Givry, S.: Multi-language evaluation of exact solvers in graphical model discrete optimization. Constraints **21**(3), 413–434 (2016)
36. Junker, U.: Outer branching: How to optimize under partial orders? In: V. Barichard, M. Ehrgott, X. Gandibleux, V. T'Kindt (eds.) Proc. 7th Intl. Conf. Multiobjective Programming and Goal Programming (MOPGP'06), *Lect. Notes Econom. Math. Syst.*, vol. 618, pp. 99–109. Springer (2009)
37. Jussien, N., Rochart, G., Lorca, X.: Choco: An open-source Java constraint programming library. In: Proc. Ws. Open-source Software for Integer and Constraint Programming (OSSICP'08), pp. 1–10 (2008)
38. Kaci, S.: Working with Preferences: Less is More. Springer (2011)
39. Kießling, W., Köstler, G.: Preference SQL: Design, implementation, experiences. In: Proc. 28th Intl. Conf. Very Large Data Bases (VLDB'02), pp. 990–1001. Morgan Kaufmann (2002)
40. Knapp, A., Schiendorfer, A., Reif, W.: Quality over quantity in soft constraints. In: Proc. 26th Intl. Conf. Tools with Artificial Intelligence (ICTAI'2014), pp. 453–460 (2014)
41. Kuchcinski, K., Szymanek, R.: JaCoP — Java constraint programming solver. In: Proc. Ws. CP Solvers: Modeling, Applications, Integration, and Standardization (2013)
42. Leenen, L., Anbulagan, Meyer, T., Ghose, A.K.: Modeling and solving semiring constraint satisfaction problems by transformation to weighted semiring Max-SAT. In: M.A. Orgun, J. Thornton (eds.) Proc. 20th Australian Joint Conf. Artificial Intelligence (ACAI'07), *Lect. Notes Comp. Sci.*, vol. 4830, pp. 202–212. Springer (2007)
43. Mears, C., Schutt, A., Stuckey, P.J., Tack, G., Marriott, K., Wallace, M.: Modelling with option types in MiniZinc. In: H. Simonis (ed.) Proc. 11th Intl. Conf. Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR'14), *Lect. Notes Comp. Sci.*, vol. 8451, pp. 88–103. Springer (2014)

44. Meseguer, P., Rossi, F., Schiex, T.: Soft Constraints. In: F. Rossi, P. van Beek, T. Walsh (eds.) Handbook of Constraint Programming, chap. 9. Elsevier (2006)
45. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: C. Bessière (ed.) Proc. 13th Intl. Conf. Principles and Practice of Constraint Programming (CP'07), *Lect. Notes Comp. Sci.*, vol. 4741, pp. 529–543. Springer (2007)
46. Nisan, N., Ronen, A.: Algorithmic mechanism design. In: J.S. Vitter, L.L. Larmore, F.T. Leighton (eds.) Proc. 31st Ann. ACM Symp. Theory of Computing (STACS'99), pp. 129–140. ACM (1999)
47. Petit, T., Régin, J.C., Bessière, C.: Meta-constraints on violations for over constrained problems. In: Proc. 12th Intl. Conf. Tools with Artificial Intelligence (ICTAI'00), pp. 358–365 (2000)
48. Pierce, B.C.: Basic Category Theory for Computer Scientists. MIT Press (1991)
49. Rendl, A., Guns, T., Stuckey, P.J., Tack, G.: MiniSearch: A solver-independent meta-search language for MiniZinc. In: G. Pesant (ed.) Proc. 21st Intl. Conf. Constraint Programming (CP'2015), *Lect. Notes Comp. Sci.*, vol. 9255, pp. 376–392 (2015)
50. Rendl, A., Tack, G., Stuckey, P.J.: Stochastic MiniZinc. In: B. O'Sullivan (ed.) Proc. 20th Intl. Conf. Principles and Practice of Constraint Programming (CP'14), *Lect. Notes Comp. Sci.*, vol. 8656, pp. 636–645. Springer (2014)
51. Rollón, E.: Multi-objective optimization in graphical models. Ph.D. thesis, Universitat Politècnica de Catalunya, Barcelona (2008)
52. Rossi, F., Pilan, I.: Abstracting soft constraints: Some experimental results on fuzzy CSPs. In: K.R. Apt, F. Fages, F. Rossi, P. Szeredi, J. Váncza (eds.) Sel. Papers Joint ERCIM/CologNET Intl. Ws. Constraint Solving and Constraint Logic Programming (CSCLP'03), *Lect. Notes Comp. Sci.*, vol. 3010, pp. 107–123. Springer (2003)
53. Ruttkay, Z.: Fuzzy constraint satisfaction. In: Proc. 3rd IEEE Intl. Fuzzy Systems Conf., pp. 1263–1268. IEEE (1994)
54. Sannella, D., Tarlecki, A.: Foundations of Algebraic Specification and Formal Software Development. EATCS Monographs in Theoretical Computer Science. Springer (2012)
55. Schiendorfer, A., Knapp, A., Steghöfer, J.P., Anders, G., Siefert, F., Reif, W.: Partial valuation structures for qualitative soft constraints. In: R.D. Nicola, R. Hennicker (eds.) Software, Services and Systems — Essays Dedicated to Martin Wirsing on the Occasion of His Emeritation, *Lect. Notes Comp. Sci.*, vol. 8950, pp. 115–133. Springer (2015)
56. Schiendorfer, A., Steghöfer, J.P., Knapp, A., Nafz, F., Reif, W.: Constraint relationships for soft constraints. In: M. Bramer, M. Petridis (eds.) Proc. 33rd SGAI Intl. Conf. Innovative Techniques and Applications of Artificial Intelligence (AI'13), pp. 241–255. Springer (2013)
57. Schiendorfer, A., Steghöfer, J.P., Reif, W.: Synthesis and abstraction of constraint models for hierarchical resource allocation problems. In: Proc. 6th Intl. Conf. Agents and Artificial Intelligence (ICAART'14), Vol. 2, pp. 15–27. SciTePress (2014)
58. Schiex, T., Fargier, H., Verfaillie, G.: Valued constraint satisfaction problems: Hard and easy problems. In: Proc. 14th Intl. Joint Conf. Artificial Intelligence (IJCAI'95), Vol. 1, pp. 631–639. Morgan Kaufmann (1995)
59. Schulte, C., Lagerkvist, M.Z., Tack, G.: Gecode: Generic constraint development environment. In: INFORMS Ann. Meeting (2006)
60. Shapiro, L.G., Haralick, R.M.: Structural descriptions and inexact matching. IEEE Trans. Pattern Analysis Mach. Intell. **3**(5), 504–519 (1981)
61. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: M.J. Maher, J.F. Puget (eds.) Proc. 4th Intl. Conf. Principles and Practice of Constraint Programming (CP'98), *Lect. Notes Comp. Sci.*, vol. 1520, pp. 417–431. Springer (1998)
62. Shoham, Y., Leyton-Brown, K.: Multiagent Systems: Algorithmic, Game-theoretic, and Logical Foundations. Cambridge University Press (2008)
63. Sánchez, M., Allouche, D., de Givry, S., Schiex, T.: Russian doll search with tree decomposition. In: C. Boutilier (ed.) Proc. 21st Intl. Joint Conf. Artificial Intelligence (IJCAI'09), pp. 603–608 (2009)
64. Stuckey, P.J., de la Banda, M.G., Maher, M., Marriott, K., Slaney, J., Somogyi, Z., Wallace, M., Walsh, T.: The G12 project: Mapping solver independent models to efficient solutions. In: P. van Beek (ed.) Proc. 11th Intl. Conf. Principles and Practice of Constraint Programming (CP'05), *Lect. Notes Comp. Sci.*, vol. 3709, pp. 13–16. Springer (2005)
65. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The MiniZinc challenge 2008–2013. AI Mag. **35**(2), 55–60 (2014)
66. Stuckey, P.J., Tack, G.: MiniZinc with functions. In: C.P. Gomes, M. Sellmann (eds.) Proc. 10th Intl. Conf. Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR'13), *Lect. Notes Comp. Sci.*, vol. 7874, pp. 268–283. Springer (2013)

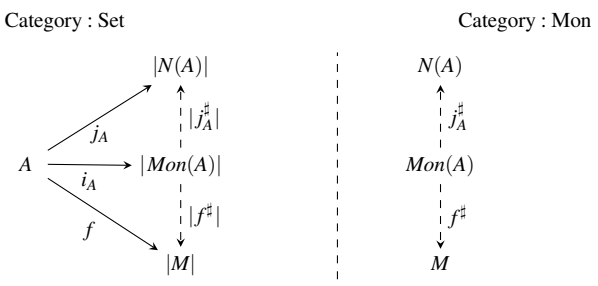


Figure 8 A diagram of the free monoid over a set from Example 4. $Mon(A)$ and $N(A)$ refer to $(A^*, ::, \varepsilon)$ and $(2^A, \cup, \emptyset)$, respectively and M just refers to *any* monoid. The embeddings $j_A(a) = \{a\}$ and $i_A(a) = [a]$ are defined analogously for any set A. A dashed arrow indicates that, e.g., there is a *unique* monoid homomorphism $f^\sharp$ that makes the diagram commute, i.e., $f = |f^\sharp| \circ i_A$.

A Free Objects in Category Theory: The Free Monoid over a Set

Mathematical categories are composed of *objects* (e.g,. algebraic structures) and *morphisms* (e.g., structure-preserving mappings) between them. Each morphism f admits a domain A and codomain B, both being objects, and is written as $f : A \to B$. For all morphisms $f : A \to B$ and $g : B \to C$ there has to be a composite arrow $(g \circ f) : A \to C$. Morphism composition $\circ$ needs to be associative and for each object A, there has to be an identity morphism $\mathrm{id}_A : A \to A$ acting as "neutral element" with respect to composition, i.e., $\mathrm{id}_B \circ f = f \circ \mathrm{id}_A = f$. The most straightforward example is given by the category Set, where objects are sets, morphisms are functions, composition is function composition, and the identity morphisms are just the identity functions. A slightly more elaborate example is given by PO, the category of partially-ordered sets, that has partial orders as objects and partial order homomorphisms (i.e., monotone functions) as morphisms. Note that this definition is proper since monotone functions are closed under function composition, i.e., if $\varphi : |P| \to |Q|$ and $\psi : |Q| \to |R|$ are monotone functions, so is $\psi \circ \varphi$.[17]

For our purposes, the true strength of category theory is visible when we consider transformations between different algebraic structures, e.g. between partial orders and PVS or between PVS and c-semirings. Such a mapping F between two categories $\mathcal{C}$ and $\mathcal{D}$ is called a *functor*. More precisely, F sends every $\mathcal{C}$-object A to a $\mathcal{D}$-object $F(A)$ and every $\mathcal{C}$-morphism $f : A \to B$ to a $\mathcal{D}$-morphism $F(f) : F(A) \to F(B)$ (respecting identity, i.e., $F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$). We have already seen an example, $|P|$ that returns the underlying set of a partial order on objects and the underlying function of a monotone function (here, just itself) on morphisms. To see a more interesting example of functors that will provide intuition for our endeavors in Sections 3.3 and 3.4, consider the task of constructing a plain monoid (a set and one associative binary operation $\cdot$ with neutral element ε) composed of elements taken from a set A. Our presentation closely follows [7, p. 20].

Example 4 (A monoid over a set) Let $A = \{a_1, a_2, \ldots\}$ be any set, called generators. We want to build a monoid $Mon(A) = (X, \cdot, \varepsilon)$ composed of the elements in A. A is an object in the category Set, $Mon(A)$ is an object in the category Mon. Assume that a function $i_A : A \to X$ maps every $a \in A$ to a different "new" element in our new underlying set X. For simplicity, we represent every $a \in A$ by itself. Next, we add a dedicated neutral element ε and define $\varepsilon \cdot x = x \cdot \varepsilon = x$ for every $x \in X$. Now, for every pair of generators a and b, we add a fresh element (denote it as $a \cdot b$ which is distinct from any other element $a' \in A$) and do so recursively for products of products etc. We only have to make sure to *equate the elements that have to be equal* by associativity, e.g., $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. This can be easily achieved if we represent every element "without parentheses", leading to X being the set of words over A (i.e., A^* with ε denoting the empty word) and $\cdot$ being the concatenation (written as ::). Now a functor $Mon : \mathrm{Set} \to \mathrm{Mon}$ takes every set A to $(A^*, ::, \varepsilon)$ and every function (morphism in the category Set) $f : A \to B$ to a monoid homomorphism $Mon(f) : Mon(A) \to Mon(B)$ which is defined as follows: $Mon(f)(\varepsilon) = \varepsilon$, $Mon(f)([a_1, \ldots, a_n]) = [f(a_1), \ldots, f(a_n)]$. Elements of A are represented in A^* by $i_A(a) = [a]$. Note that for any singleton list $[a] \in A^*$ iff $a \in A$.

[17] All categories relevant to the discussion of partial orders, partial valuation structures, and c-semirings are examples of so-called *concrete categories* with objects being sets with additional algebraic or ordered structure and morphisms being set-theoretic (structure-preserving) functions – in general, this need not be.

With respect to Example 4, in the quest for constructing a monoid from a set, we could have also tried another functor $N(A)$ that maps A to $(2^A, \cup, \emptyset)$ and represent $a \in A$ as $j_A(a) = \{a\}$. Clearly, $N(A)$ also satisfies the monoid axioms of associativity and $\emptyset$ being neutral. However, it is *too specific*: it assumes commutativity since $j_A(a) \cdot j_A(b) = \{a\} \cup \{b\} = \{b\} \cup \{a\} = j_A(b) \cdot j_A(a)$. But we have already seen another monoid, $Mon(A)$, where $i_A(a) \cdot i_A(b) = [ab] \neq [ba] = i_A(b) \cdot i_A(a)$. Hence, commutativity need not be a requirement for a monoid. Mapping A to $N(A)$ would consequently unify elements that need not be equal. Once that mapping is done, it should be impossible to map "back" to a more general structure where the unified elements are distinguishable. Put differently, there do exist functions f from A to a monoid M' that we cannot factorize as $f^\sharp \circ j_A = f$ for some $f^\sharp$.

Indeed, this is the case. Assume for a particular set $\mathsf{A} = \{\mathsf{a}, \mathsf{b}\}$ that we have some function f into $|Mon(\mathsf{A})|$, for instance $f(\mathsf{a}) = [\mathsf{aba}]$ and $f(\mathsf{b}) = [\mathsf{bab}]$. Now assume that we mapped A to $N(\mathsf{A})$ via j_A, having a and b now represented as $\{\mathsf{a}\}$ and $\{\mathsf{b}\}$, respectively. Is there a way we can "still" reconstruct the function f, starting from $N(A)$ and calling it $f^\sharp$? To fulfill $f = f^\sharp \circ j_\mathsf{A}$, we know that $f^\sharp(\{\mathsf{a}\}) = [\mathsf{aba}]$ and $f^\sharp(\{\mathsf{b}\}) = [\mathsf{bab}]$ must hold. But what about $f^\sharp(\{\mathsf{a}, \mathsf{b}\})$? To satisfy monoid homomorphism laws, $f^\sharp(\{\mathsf{a}, \mathsf{b}\})$ must equal $f^\sharp(\{\mathsf{a}\}) :: f^\sharp(\{\mathsf{b}\}) = [\mathsf{ababab}]$. But since $\{\mathsf{a}, \mathsf{b}\} = \{\mathsf{b}, \mathsf{a}\}$, it must also hold that $f^\sharp(\{\mathsf{a}, \mathsf{b}\}) = f^\sharp(\{\mathsf{b}, \mathsf{a}\}) = f^\sharp(\{\mathsf{b}\}) :: f^\sharp(\{\mathsf{a}\}) = [\mathsf{bababa}]$. Thus, no such function $f^\sharp$ can exist – $N(\mathsf{A})$ is too specific.

Exchanging the rôles of $N(\mathsf{A})$ and $Mon(\mathsf{A})$ does not lead to the same problem. For *any* function f from a set A to the underlying set of any other monoid M, there indeed exists *precisely one* monoid homomorphism $f^\sharp$ that emulates f such that $f = f^\sharp \circ i_A$, i.e., $\forall a \in A : f(a) = f^\sharp(i_A(a))$ (see Figure 8 or [7, p. 21] for a proof). This fact characterizes that $Mon(A)$ is called the *free monoid* over A, being the most general monoid a set can be mapped to. Note that the existence of $f^\sharp$ corresponds to a "no confusion" argument since no elements are equated that should not be whereas the uniqueness of $f^\sharp$ relates to a "no junk" argument: If, for instance, we used $Mon'(A) = ((A \cup w)^*, ::, \varepsilon)$ with $w \notin A$, then we are free to chose the value of $f^\sharp(\wr w \int)$ (a "junk element") as it is not constrained by the requirement $f = f^\sharp \circ i_A$ – in contrast to all elements in A. Generalizing from this example, category theory allows to state this relationship between algebraic structures formally (see Definition 3 in Section 3.2).