Higher Order Semiring-based Constraints

Stefano Bistarelli[1], Ugo Montanari[1], Francesca Rossi[2]

[1] University of Pisa, Dipartimento di Informatica, Corso Italia 40, 56125 Pisa, Italy.
{bista,ugo}@Di.Unipi.It
[2] University of Padova, Dipartimento di Matematica Pura ed Applicata, Via Belzoni 7, 35131 Padova, Italy. frossi@Math.Unipd.It

Abstract Semiring-based constraint problems (SCSPs) [BMR95,BMR97] extend classical constraint problems (CSPs) by allowing preferences, costs, priorities, probabilities, and other soft features. They are based over the notion of a semiring, that is, a set plus two operations. In this paper we introduce a uniform, abstract presentation of (soft) constraint satisfaction concepts and constructions. Moreover, the soft constraint environment is enriched with the operation of function abstraction and application that can suitably be used to give modularity and compositionality to the framework. Finally, a small language is defined and used to represent and give semantics to general local propagation and dynamic programming algorithms.

1 Introduction

Classical constraint satisfaction problems (CSPs) [Mac92] are a convenient and expressive formalism for many real-life problems, like scheduling, resource allocation, vehicle routing, timetabling, and many others [Wal96]. However, many of these problems are often more faithfully represented as *soft constraint satisfaction problems* (SCSPs), which are just like classical CSPs except that each assignment of values to variables in the constraints is associated to an element taken from a partially ordered set. These elements can then be interpreted as levels of preference, or costs, or levels of certainty, or many other criteria.

There are many formalizations of soft constraint problems. In this paper we consider the one based on semirings [BMR97,BMR95], where the semiring specifies the partially ordered set and the appropriate operations to combine constraints together.

This framework shows how using a *semiring* to represent the different levels of preference between solutions can give us a way to represent in an single framework different features (optimization, fuzziness, probability, ...) and techniques (local consistency, dynamic programming, branch&bound, ...).

In this paper the basic structures of the framework are represented as functions and this gives modularity and compositionality. Many of the algorithms defined over a constraint system, like constraint propagation and dynamic programming techniques, can easily be expressed using expressions and commands of a suitable language.

The basic elements that we need to consider are the *domains* and the *semirings*; all the structure of an SCSP will be represented as a function between these elements. As soon as we define constraints and SCSPs as specific functions, we can use abstraction, application and composition to model SCSP solution, dynamic programming and local consitency preprocessing algorithms.

The representation of constraints as functions gives the possibility to represent a lot of useful features present in real problems. By performing a step-by-step application of the definitions of the constraints to the resolution function, we can model environments where the constraints are specified at run time by the user, or we can compute solutions for partially instantiated problems.

The paper is organized as follows. First, in Section 2 we give the necessary notions about soft CSPs, local consistency and dynamic programming techniques. Then, in Section 3 we define the domain and semiring structure and we illustrate how constraints and SCSPs are represented as specific functions; in Section 4 we define a small language useful to represent algorithms over SCSPs; in Section 5 we define the solution and the preprocessing algorithms as specific expressions and commands of the language. Finally, in Section 6 we summarize our work and give hints about future directions.

2 Soft constraints

In the literature there have been many formalizations of the concept of *soft constraints*. Here we refer to a specific one [BMR95,BMR97], which however can be shown to generalize and express many of the others [BMR97,BFM$^+$96]. In a few words, a soft constraint is just a constraint where each instantiation of its variables has an associated value from a partially ordered set. Combining constraints will then have to take into account such additional values, and thus the formalism has also to provide suitable operations for combination ($\times$) and comparison (+) of tuples of values and constraints. This is why this formalization is based on the concept of semiring, which is just a set plus two operations.

A semiring is a tuple $\langle A, +, \times, 0, 1\rangle$ such that:

- A is a set and $0, 1 \in A$;
- $+$ is commutative, associative and 0 is its unit element;
- $\times$ is associative, distributes over $+$, 1 is its unit element and 0 is its absorbing element.

A c-semiring is a semiring $\langle A, +, \times, 0, 1\rangle$ such that: $+$ is idempotent with 1 as its absorbing element and $\times$ is commutative.

Let us consider the relation $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$. Then it is possible to prove that (see [BMR97]):

- $\leq_S$ is a partial order;
- $+$ and $\times$ are monotone on $\leq_S$;
- 0 is its minimum and 1 its maximum;
- $\langle A, \leq_S\rangle$ is a complete lattice and, for all $a, b \in A$, $a + b = lub(a, b)$.

Moreover, if $\times$ is idempotent, then $\langle A, \leq_S\rangle$ is a complete distributive lattice and $\times$ is its glb. Informally, the relation $\leq_S$ gives us a way to compare (some of the) tuples of values and constraints. In fact, when we have $a \leq_S b$, we will say that *b is better than a*.

A *constraint system* is a tuple $CS = \langle S, D, V \rangle$ where S is a c-semiring, D is a finite set (the domain of the variables) and V is an ordered set of variables. Given a semiring $S = \langle A, +, \times, 0, 1 \rangle$ and a constraint system $CS = \langle S, D, V \rangle$, a *constraint* is a pair $\langle def, con \rangle$ where $con \subseteq V$ and $def : D^{|con|} \to A$. Therefore, a constraint specifies a set of variables (the ones in con), and assigns to each tuple of values of these variables an element of the semiring.

A soft *constraint problem* is a pair $\langle C, con \rangle$ where $con \subseteq V$ and C is a set of constraints: con is the set of variables of interest for the constraint set C, which however may concern also variables not in con. Note that a classical CSP is an SCSP where the chosen c-semiring is:

$$S_{CSP} = \langle \{false, true\}, \vee, \wedge, false, true \rangle.$$

Fuzzy CSPs [DFP93,Rut94,Sch92] can instead be modeled in the SCSP framework by choosing the c-semiring:

$$S_{FCSP} = \langle [0, 1], max, min, 0, 1 \rangle.$$

Figure 1 shows a fuzzy CSP. Variables are inside circles, constraints are represented by undirected arcs, and semiring values are written to the right of the corresponding tuples. Here we assume that the domain D of the variables contains only elements a and b.

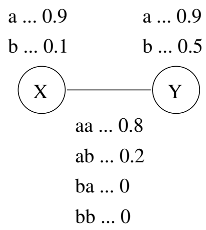

Figure1. A fuzzy CSP.

Combining and projecting soft constraints Given two constraints $c_1 = \langle def_1, con_1 \rangle$ and $c_2 = \langle def_2, con_2 \rangle$, their *combination* $c_1 \otimes c_2$ is the constraint $\langle def, con \rangle$ defined by $con = con_1 \cup con_2$ and $def(t) = def_1(t \downarrow^{con}_{con_1}) \times def(t \downarrow^{con}_{con_2})$, where $t \downarrow^{X}_{Y}$ denotes the tuple of values over the variables in Y, obtained by projecting tuple t from X to Y. In words, combining two constraints means building a new constraint involving all the variables of the original ones, and which associates to each tuple of domain values for such variables a semiring element which is obtained by multiplying the elements associated by the original constraints to the appropriate subtuples.

Given a constraint $c = \langle def, con \rangle$ and a subset I of V, the *projection* of c over I, written $c \Downarrow_I$ is the constraint $\langle def', con' \rangle$ where $con' = con \cap I$ and

$def'(t') = \sum_{t/t\downarrow^{con}_{I\cap con}=t'} def(t)$. Informally, projecting means eliminating some variables. This is done by associating to each tuple over the remaining variables a semiring element which is the sum of the elements associated by the original constraint to all the extensions of this tuple over the eliminated variables. Summarizing, combination is performed via the multiplicative operation of the semiring, and projection via the additive one.

Let us recall also the meaning of the *disjunction* operator between constraints; Note that this definition is slightly different from that in [BCGR00]. Given two constraints $c_1 = \langle def_1, con_1 \rangle$ and $c_2 = \langle def_2, con_2 \rangle$ over a constraint system CS, their *disjunction* $c_1 \oplus c_2$ is the constraint $c = \langle def, con \rangle$ with $con = con_1 \cap con_2$ and $def(t) = def_1(t \downarrow^{con_1}_{con}) + def_2(t \downarrow^{con_2}_{con})$.

Solutions The *solution* of an SCSP problem $P = \langle C, con \rangle$ is the constraint $Sol(P) = (\bigotimes C) \Downarrow_{con}$. That is, we combine all constraints, and then project over the variables in con that represents the set of *variables of interest*. In this way we get the constraint over con which is "induced" by the entire SCSP.

For example, each solution of the fuzzy CSP of Figure 1 consists of a pair of domain values (that is, a domain value for each of the two variables) and an associated semiring element. Such an element is obtained by looking at the smallest value for all the subtuples (as many as the constraints) forming the pair. For example, for tuple $\langle a, a \rangle$ (that is, $x = y = a$), we have to compute the minimum between 0.9 (which is the value for $x = a$), 0.8 (which is the value for $\langle x = a, y = a \rangle$) and 0.9 (which is the value for $y = a$). Hence, the resulting value for this tuple is 0.8.

Sometimes it may be useful to find the semiring value of the optimal solutions. This is called the *best level of consistency* of an SCSP problem P and it is defined by $blevel(P) = Sol(P) \Downarrow_{\emptyset}$. We also say that: P is α-consistent if $blevel(P) = \alpha$; P is consistent iff there exists $\alpha >_S 0$ such that P is α-consistent; P is inconsistent if it is not consistent.

Local consistency SCSP problems can be solved by extending and adapting the techniques usually used for classical CSPs. For example, to find the best solution we could employ a branch-and-bound search algorithm (instead of the classical backtracking), and also the successfully used propagation techniques, like arc-consistency, can be generalized to be used for SCSPs.

Instead of deleting values of tuples, in SCSPs obtaining some form of local consistency means changing the semiring values associated to some tuples or domain elements. In particular, the change always brings these values towards the worst value of the semiring, that is, the 0. In the following of this section, we recall the basic formal notions and properties of local consistency over SCSPs.

First, we need the notion of *local inconsistency*: We say that the SCSP problem $P = \langle C, con \rangle$ is *locally inconsistent* if there exist $C' \subseteq C$ such that $blevel(C') = 0$. Consider a set of constraints C and $C' \subseteq C$. If C is α-consistent then C' is β-consistent with $\alpha \leq_S \beta$. As a corollary: if a problem is locally inconsistent, then it is inconsistent.

Now, in order to define local consistency algorithms, we use the notion of *typed location*, which is just a set of variables, that will be used to identify a subproblem of a given SCSP. A *typed location* l is a set of variables. Given a problem $P = \langle C, con \rangle$, if there is a constraint $c = \langle \mathrm{def}, l \rangle$ in C, then the value $[l]_P$ of the location l in P is $\langle \mathrm{def}, l \rangle$; otherwise it is $\langle \mathbf{1}, l \rangle$. The value $[\{l_1, \cdots, l_n\}]_P$ of the set of locations $\{l_1, \cdots, l_n\}$ is the set $\{[l_1]_P, \cdots, [l_n]_P\}$.

An *assignment* is a pair $l := c$ where $c = \langle \mathrm{def}, l \rangle$. Given a problem $P = \langle C, \mathrm{con} \rangle$, the result of the assignment $l := c$ is defined as $[l := c](P) = \langle \{\langle \mathrm{def}', \mathrm{con}' \rangle \in C \mid \mathrm{con}' \neq l\} \cup c, \mathrm{con} \rangle$. In words, the assignment $l := c$ in P produces a new problem P' which is the same as P, except that it has an additional constraint, c, over the variables in l, and that the old constraints over l are removed.

A *local consistency rule* is written as $l \leftarrow L$, where l is a location, L a set of locations and $l \notin L$. The result of applying the rule $l \leftarrow L$ to the problem P is:

$$[l \leftarrow L](P) = [l := \mathrm{Sol}(\langle [L \cup \{l\}]_P, l \rangle)](P).$$

In words, the application of $l \leftarrow L$ to P adds to P the constraint $\mathrm{Sol}(\langle [L \cup \{l\}]_P$ over the variables in l. This constraint, by definition of Sol, is obtained by combining all constraints identified by $L \cup \{l\}$ and then projecting over l. The application of a sequence of rules $r; R$ is defined by $[r; R](P) = [R]([r](P))$. It is important to observe at this point that, given a problem P and a rule r, we have that $P \equiv [r](P)$, that is, the solution is not changed.

Dynamic Programming Dynamic programming [Bel57,BD62] can be used to solve a problem by solving some subproblems of it and then combining their solutions to obtain the solution of the whole problem (see for example [GMM81] for its use in graphs and [MR86] for its adoption in classical CSPs). In the SCSP framework, a suitably instantiated version of dynamic programming can be fruitfully used as well: at each step, a subset of constraints is chosen and solved, and its solution (which, we recall, is a constraint) replaces the whole subset of constraints.

Let we describe in a tree-like way the SCSP to be solved; a dynamic programming algorithm can then follow the tree structure in a bottom-up way.

Given an SCSP $P = \langle C, \mathrm{con} \rangle$, a parsing tree of P is a sequence of local consistency rules $S = r_1; \ldots; r_n$, where $r_i = (l_i \leftarrow L_i)$. Let

- $L = \{l_i, i = 1, \ldots, n\} \cup \bigcup_{i=1,\ldots,n} L_i$ (that is, L is the set of all locations occurring in all rules in S);
- $l_i \prec l$ iff $l \in L_i$ (that is, rule r_i "generates" location l);
- $l_i \prec v$ iff $v \notin l_i$ and $v \in l'$ with $l' \in L_i$ (that is, rule r_i "generates" variable v);
- for any x location or variable, $\mathrm{prec}(x) = \{l \mid l \prec x\}$.

Then we require that

- *(tree-like)* for any x variable or location, if $x = l_n$ or $x \in l_n$, then $\mathrm{prec}(x) = \emptyset$, else $\mathrm{prec}(x)$ is a singleton;
- *(cover)* $\mathrm{con} = l_n$, and $\langle \mathrm{def}, \mathrm{con}' \rangle \in C$ implies $\mathrm{con}' \in L$;
- *(bottom-up parsing)* $l_i \prec l_j$ implies $j < i$.

In words, the above definition provides a sequence of rules which cover the whole problem and are connected among them as a tree. Moreover, the sequence gives a bottom-up visit of the tree. It is now easy to define a dynamic programming algorithm where, at each step, one rule is processed, according to the given sequence.

Given an SCSP P and a parsing tree $S = r_1; \ldots ; r_n$ of P, a dynamic programming algorithm compute $[S](P)$. It is important to notice that given an SCSP $P = \langle C, l\rangle$ and a parsing tree S for P, we have that $Sol(P) = [l]_{[S](P)}$.

3 Domains and semirings

All the SCSP's elements (constraints and problems) are defined here as functions over the set of domains D and semirings S. A domain can be a set D_0 (representing all the values that can be assigned to variables) or can be constructed using the cartesian product. Note that the domain of the variables may be different, so we have a collection $D_1, \ldots, D_n$ of variable domains.

Definition 1 (domains). *The class of domains is denoted* D *and it is defined as*

$$D ::= D_1 \mid \cdots \mid D_n \mid D \times D$$

Intuitively, a value $d \in D$ represent a domain value for a variable in a given SCSP problem, while a value $\langle d_1, d_2\rangle \in D_1 \times D_2$ and a value $\langle d_1, \ldots, d_k\rangle \in D_1 \times \cdots \times D_k$ respectively represents a pair of domain values for a binary constraint and a k-tuple of domain values for a k-ary constraint.

The semiring structure will be represented as S; note that S represent all the primitive instances (for example the structures used for *fuzzy-CSPs*, *probabilistic-CSPs*, *weighted-CSPs*, etc.) and also the semiring obtained by using some special operators like the cartesian product (described in [BMR95,BMR97]) or the *power-set of Hoare* (described in [BMR00]). In this case there is a collection of basic semiring $S_1, \ldots, S_n$ (see [BMR97] for a deeper explanation) plus the semirings obtained by using the power-set operation (that here we consider using a specific primitive semiring , says S_j) and their cartesian product.

Definition 2 (semiring). *The class of semirings is denoted by* S *and is defined as*

$$S ::= S_1 \mid \cdots \mid S_n \mid S \times S$$

By using the properties of $\times$, $+$ and of the cartesian product, we can write down some basic relationships between semiring elements; In Table 1 symbol $\star$ denotes any of the semiring operator $\times$ and $+$. The table shows how the semiring operations of the cartesian product is defined using the corresponding operations in the primitive semiring.

By using functions from D to S we can now represent (soft) constraints.

Definition 3 (constraints). *The set of constraints (over a semiring* S*) is denoted by* C *and it is defined as*

$$C ::= D \to S$$

Note that the syntactic category D represents the set of domains of all the constraints of the problem. For unary constraints (that is, variable domain), there will be functions $D_i \to S$ for some i, and for the non unary constraint over $k \geq 1$ variables there will be a function $D_1 \times \cdots \times D_k \to S$.

We can now describe entire constraint problems:

Table1. Relationships between semiring elements.

$$\frac{a_1 \in S_1, a_2 \in S_2}{\langle a_1, a_2 \rangle \in S_1 \times S_2}$$

$$\frac{\langle a_1, a_2 \rangle, \langle b_1, b_2 \rangle \in S_1 \times S_2}{\langle a_1, a_2 \rangle \star_{S_1 \times S_2} \langle b_1, b_2 \rangle = \langle a_1 \star_{S_1} b_1, a_2 \star_{S_2} b_2 \rangle \in S_1 \times S_2}$$

Definition 4 (constraint problem). *Fixed a semiring* S *and the number* n *of constraints, a constraint problem* P *is defined as*

$$P ::= C^n \to C.$$

where C^n *is obtained as the cartesian product of* n *constraints, that is* $C^n = \underbrace{C \times \cdots \times C}_{n \textit{ times}}$

Example 1. Consider the fuzzy CSP represented in Figure 2. In this case the semiring for the problem is

$$S_{FCSP} = \langle [0, 1], max, min, 0, 1 \rangle.$$

The domain is the same for all the variables that is $D = \{a, b\}$. The constraints c_x, c_y, c_z are functions $\{a, b\} \to S_{FSCP}$,and constraints c_1 and c_2 are functions $\{a, b\}^2 \to S_{FSCP}$.

The problem represented in Figure 2 is instead a function $c_x \times c_y \times c_z \times c_1 \times c_2 \to (D^2 \to S_{FSCSP})$.

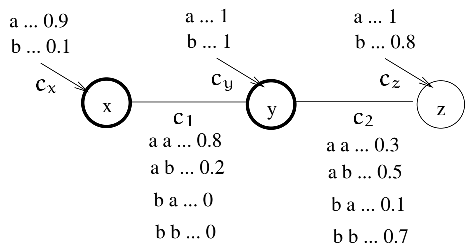

Figure2. A 3-variable fuzzy CSP.

3.1 Costraint problems and solutions

Since both constraints and constraint problems are defined as functions, we are now able to use function composition, abstraction and application to define in a general and easy way many solution or preprocessing algorithms. The definition of a constraint problem P can be split in two parts: the first one is called *constraint graph* and describes its *topology*, and the second one is called *solution* and represent its *value*. Note that problems with different topology can have the same value, and that many times the same topology corresponds to different values.

To define the constraint graph in a concise way we need to introduce some special operators and properties:

Definition 5 (needed operators and properties).

- *The* restriction *operator ν will be useful to capture the variables of interest of a problem:*

$$\frac{f \in D \to S}{\nu f = \sum^S_{x \in D} f(x)}$$

We will write $\nu x.M$ instead of $\nu(\lambda x.M)$ and $\nu\langle x, y\rangle$ with the meaning of restrict over pairs (or tuples) of domains instead that single domains.
- *some of its important properties*

$$\nu x.(t_1 \star t_2) = (\nu x.t_1) \star t_2 \textit{ se } x \notin FV(t_2) \qquad (\nu\text{-distributivity})$$
$$\nu x.(t_1 +_S t_2) = (\nu x.t_1) +_S (\nu x.t_2) \qquad (\nu\text{-associativity})$$

Let us define now the *constraint graph* and the *problem solution*.

Definition 6 (Constraint graph). *Given an SCSP* $P = \langle C, con\rangle$ *over a constraint system* $CS = \langle S, D, V\rangle$ *with* $C = \{c_1, \dots, c_n\}$*,* $c_i = \langle def_i, con_i\rangle$*,* $def_i : D^{|con_i|} \to S$ *and* $V = \{x_1, \dots, x_k\}$*, a* constraint graph *is a a function defined as follows:*

$$P = \lambda\langle c_1, \dots, c_n\rangle.\lambda\langle con\rangle.\nu\langle V - con\rangle. \prod_{\substack{S\\ i=1\dots n}} c_i(con_i)$$

The meaning of this function is to take the constraints $\{c_1, \dots, c_n\}$ over the variables in V and to perform their combination $\prod_{S\, i=1\dots n} c_i(con_i)$. Since we are interested only to a subset of the variable $con \in V$, we need to use the ν operator to sum up over the variable in $V - con$. Note that con and $V - con$ are sets, but since we consider the set of variables ordered in a fixed way, they can be seen as tuple of variables. This is not completely correct because con and $V - con$ are set of variables and not tuples. Anyway, since we fix a lexicographic order over the variables in V it is easy to pass from sets to tuples.

The *value* of an SCSP is obtained applying the constraint graph to the definition of its constraints. The result of this application represents the *problem solution*.

Definition 7 (Problem solution). *Given a SCSP* $P = \langle C, con\rangle$ *over a constraint system* $CS = \langle S, D, V\rangle$ *with* $C = \{c_1, \dots, c_n\}$*,* $c_i = \langle def_i, con_i\rangle$*,* $def_i : D^{|con_i|} \to S$*,* $V = \{x_1, \dots, x_k\}$ *and the corresponding constraint graph*

$$P = \lambda\langle c_1, \dots, c_n\rangle.\lambda\langle con\rangle.\nu\langle V - con\rangle. \prod_{\substack{S\\ i=1\dots n}} c_i(con_i),$$

the function solution is defined as

$$\mathsf{Sol}(P) = P(\mathsf{DEF}_P) = \lambda\langle con\rangle.\nu\langle V - con\rangle. \prod_{i=1\ldots n}{}_S\, def_i(con_i)$$

where DEF_P *represents the set of all the definitions of the constraints in* P.

Example 2. Consider again the fuzzy CSP represented in Figure 2. The constraint graph is represented by the function

$$\lambda\langle c_1, c_2, c_x, c_y, c_z\rangle.\lambda\langle x, y\rangle.\nu z.c_1(x,y) \times_S c_2(y,z) \times_S c_x(x) \times_S c_y(y) \times_S c_z(z).$$

The structure of this term gives an explicit way to represent an SCSP with 5 constraints and 3 variable, and shows how constraints c_1 and c_2 share one of the variables. Note that in this term, the variables x, y and z, and the constraints c_1,c_2,c_x,c_y,c_z are bound by λ-abstraction, so any term obtained from this one via α-renaming represents the same SCSP (i.e., α-renamed terms represent all the SCSPs with this shape). Note also that the $\times_S$ operator in the body of the function is not instantiated to the corresponding operator of the (fuzzy in this case) semiring. This means that only the shape of the problem is represented and not its value. In other words, the semantical meaning of the $\times$ operator (that is the min) is not used.

To compute the solution of the problem, we need first to apply the definition of the constraints to the constraint graph, that is to compute $P(\mathsf{DEF}_P)$. In this way we obtain

$$\lambda\langle x, y\rangle.\nu z.def_1(x,y) \times_S def_2(y,z) \times_S def_x(x) \times_S def_y(y) \times_S def_z(z).$$

Now, if we want to compute the value of this SCSP we have to instantiate the variable and to compute the value of the solution function. As an example of solution, consider the pair $x = a$ and $y = b$. We obtain the function

$$\nu z.0.2 \times_S def_2(b,z) \times_S 0.9 \times_S 1 \times_S def_z(z) =$$

by istantiating the $\times_S$ operator to min

$$\nu z.min(0.2, def_2(b,z), 0.9, 1, def_z(z)).$$

By definition of ν we obtain

$max_{z\in\{a,b\}}min(0.2, def_2(b,z), 0.9, 1, def_z(z)) =$
$max(min(0.2, def_2(b,a), 0.9, 1, def_z(a)), min(0.2, def_2(b,b), 0.9, 1, def_z(b))) =$
$max(min(0.2, 0.1, 0.9, 1, 1), min(0.2, 0.7, 0.9, 1, 0.8)) =$
$max(0.1, 0.2) = 0.2$

4 A small language to program with soft constraints

We will define now a small language that will be useful to represent and solve SCSPs, using local consistency and dynamic programming algorithms. To define such algorithms, we need to explicitely define the *typed locations*, which we remind are just a set of variables, that will be used to identify a subproblem of a given SCSP.

Definition 8 (Typed location). *A* typed location l *is a set of variables. Given a constraint system $\langle S, D, V\rangle$, the set of locations $\mathcal{L}$ is defined as $\mathcal{L} = \wp(V)$.*

Locations represent variables of type *constraint* that is, each location can contain a constraint (i.e. a function $D \to S$). Let us now define the *(Constraint) Store*.

Definition 9 (Constraint Store). *A constraint store is a function σ such that $\sigma l : D^{|l|} \to S$, where we have $\sigma l\langle x_1, \ldots, x_{|l|}\rangle = 1$ for all except a finite number of locations.*

Given a constraint Problem $P = \langle C, con\rangle$, we can built the corresponding constraint store σ_C defined as follows:

$$\sigma_C l = \begin{cases} def_i \text{ if } c_i = \langle def_i, l\rangle \in C \\ \lambda\langle v_1, \ldots, v_{|l|}\rangle.1 \text{ otherwise.} \end{cases}$$

In the following we will write σ instead of σ_C if the corresponding problem $P = \langle C, con\rangle$ is easy to be recognized.

Let us now define the syntactic structure of our language: the *expressions* $\mathtt{Expr}$ and the *commands* $\mathtt{Com}$.

Definition 10 (Expressions). *The expressions we can build with the language are:*

$$\mathtt{Expr} ::= c \mid \mathtt{Sol}(l) \mid \mathtt{Sol}(l, L) \mid \phi(l, L)$$

The element c represent a constraint (that is, a basic piece of information); $\mathtt{Sol}(l)$ represents the solution of a constraint problem whose variables of interest are the variables in l. $\mathtt{Sol}(l, L)$ represents instead the solution of a subproblem (whose constraints are those connecting variables in $L \cup \{l\}$) and will be useful to define local consistency and dynamic programming algorithms. Finally, $\phi(l, L)$ is a generic monotone and extensive function that will be useful to represent generic soft local consistency algorithms.

Before giving a semantics to the expressions, let us introduce the syntactic category of commands.

Definition 11 (Commands). *The commands we can build with the language are defined by the following grammar:*

$$\mathtt{Com} ::= l := \mathtt{Expr} \mid \mathtt{Com}; \mathtt{Com}$$

Note that the assignement command "$l := \mathtt{Expr}$" can be performed only if the type of the expression $\mathtt{Expr}$ is compatible with the location l.

The assignment command is useful to change the value of a location presents in the constraint store, and the ";" operator is useful to perform several assignments sequentially.

To formally compute the *semantics* of the expressions and of the commands, we need to define two semantic functions: $\mathcal{E}$ for the expressions and $\mathcal{C}$ for the commands.

Function $\mathcal{E}$ evaluates an expression on a constraint store and gives as a result a constraint (that is, an element $c \in C$ that assigns to each tuple of elements in D a value in S).

Definition 12 (Function $\mathcal{E}$). *The semantic function*

$$\mathcal{E} : \mathrm{Expr} \to \sigma \to C$$

is defined by structural recursion as follows:

$$\mathcal{E}[\![\, \mathrm{Sol}(l) \,]\!]\sigma = \lambda\langle l\rangle.\nu\langle V - l\rangle. \prod_{l' \in \mathcal{L}} {}_S \, \sigma l'\langle l'\rangle$$

$$\mathcal{E}[\![\, \mathrm{Sol}(l, L) \,]\!]\sigma = \lambda\langle l\rangle.\nu\langle L\rangle. \prod_{l' \in (L \cup \{l\})} {}_S \, \sigma l'\langle l'\rangle$$

$$\mathcal{E}[\![\, c \,]\!]\sigma = \hat{c}$$

$$\mathcal{E}[\![\, \phi(l, L) \,]\!]\sigma = \hat{\phi}(l, L)$$

The element $\hat{c}$ and $\hat{\phi}(l, L)$ are basic element representing respectively a constraint definition (that is its def *function) and a generic monotone and extensive function $\hat{\phi}(l, L) : (\prod_{l' \in (L \cup \{l\})} D^{|l'|} \to S) \to (D^{|l|} \to S)$. The function* $\mathrm{Sol}(l, L)$ *is a specific $\hat{\phi}(l, L)$, and the function* $\mathrm{sol}(l)$ *is obtained by performing* $\mathrm{Sol}(l, \mathcal{L})$ *where $\mathcal{L}$ is the set of all the locations of the problem.*

Since the language we are going to define use assignments, we need to define the meaning of store modification:

Definition 13 (Store modifications). *Given a constraint store σ, and a constraint $c : D^{|l|} \to S$, the store modification operator is defined as follows:*

$$\sigma[c/l]l' = \begin{cases} c \textit{ if } l' = l \\ \sigma l' \textit{ otherwise.} \end{cases}$$

Let us now evaluate the commands using the function $\mathcal{C}$.

Definition 14 (The Function $\mathcal{C}$). *The semantic function*

$$\mathcal{C} : \mathrm{Com} \to \sigma \to \sigma$$

is defined by structural recursion as follows:

$$\mathcal{C}[\![\, l := e \,]\!]\sigma = \sigma[\mathcal{E}[\![\, e \,]\!]\sigma/l]$$

$$\mathcal{C}[\![\, com_1; com_2 \,]\!]\sigma = \mathcal{C}[\![\, com_2 \,]\!](\mathcal{C}[\![\, com_1 \,]\!]\sigma)$$

5 Solving SCSPs

In this section we will show how the dynamic programming and local consistency algorithms can be represented as specific expressions of the language defined in Section 4.

5.1 Dynamic programming techniques

Using dynamic programming techniques, the problem is not solved in one step, but it is partitioned in subproblems which form a *parsing tree* S, and then it is solved by solving such subproblems in a bottum-up visit of S.

A subproblem in a parsing tree S is represented by a rule $r_i = (l_i \leftarrow L_i)$ where l_i represents the location of interest of the problem, and L_i represent all the variables and constraints involved in the subproblem. The a parsing tree $S = r_1; \dots ; r_n$, as defined in [BMR97], represents a correct way to decompose the global problem, and gives also a computation order of the solutions of the subproblems. In our syntax the solution of a subproblem defined by a rule $l \leftarrow L$ is defined as the term $\mathsf{Sol}(l, L)$.

Theorem 1 (Dynamic programming solving). *Consider a problem* $P = \langle C, \mathsf{con} \rangle$ *over the constraint system* $CS = \langle S, D, V \rangle$*, the corresponding constraint store* σ *and one of its parsing tree* $S = r_1; \dots ; r_n$*, where* $r_i = (l_i \leftarrow L_i)$*, we have:*

$$\mathcal{E}[\![\mathsf{Sol}(\mathsf{con})]\!]\sigma = \mathcal{E}[\![\mathsf{Sol}(l_n, L_n)]\!](\mathcal{C}[\![l_2 := \mathsf{Sol}(l_2, L_2); \dots ; l_{n-1} := \mathsf{Sol}(l_{n-1}, L_{n-1})]\!]\sigma)$$

Proof. The proof of the theorem follows the lines of the corresponding one depicted in [BMR97], where the store σ take the place of P and the semantics of subproblem solution $\mathcal{C}[\![l := \mathsf{sol}(l, L)]\!]\sigma$ take the place of the rule application $[l \leftarrow L]P$.

Example 3. Let we review some step of the dynamic programming technique applied to the running example of the paper represented in Figure 2.

Suppose to have the strategy $S = r_1, r_2$ with $r_1 = \{y\} \leftarrow \{\langle y, z \rangle, z\}$ and $r_2 = \{\langle x, y \rangle\} \leftarrow \{x, y\}$. So we need to show that

$$\mathcal{E}[\![\mathsf{Sol}(\langle x, y \rangle)]\!]\sigma = \mathcal{E}[\![\mathsf{Sol}(\langle x, y \rangle, \{x, y\})]\!](\mathcal{C}[\![\{y\} := \mathsf{Sol}(\{y\}, \{\langle y, z \rangle, z\})]\!]\sigma).$$

By solving the subproblem represented by the first rul, we obtain the constraint problem represented in Figure 3. Let we call $\sigma_{C'}$ the constraint store represented in the figure.

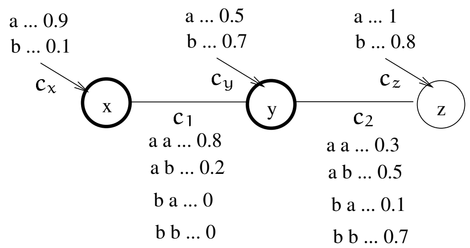

Figure3. The fuzzy CSP after the application of rule r_1.

By definition of the ";" operator, we need now to solve the problem

$$\mathcal{E}[\![\mathsf{Sol}(\langle x, y \rangle, \{x, y\})]\!]\sigma'_C.$$

It is now easy to check that the solution computed in this way is equivalent to the solution computed by giving semantics to $\mathcal{E}[\![\mathsf{Sol}(\langle x, y \rangle)]\!]\sigma$.

5.2 Local consistency techniques

In this section we show how several soft local consistency algorithms can be captured and described in an uniform way in the framework described in this paper. In [BMR95,BMR97] the authors describe local consistency rules by functions computing the solution of some subproblem. In the background part of this paper some of the results of this approach are given and will be used as a starting point to define a more general definition of local consistency algorithms. In [Apt99a,Apt99b] the author shows how constraint propagation algorithms for classical CSPs can be described in a uniform way by using a generic algorithm to compute the fix-point of a set of monotone and extensive functions. The possibility to describe in a general way functions gives us the possibility to extend and unify the work of [BMR95,BMR97] and of [Apt99a,Apt99b] to define general constraint propagation algorithms for soft constraints.

In this section we first represent classical soft local consistency rules as commands in our language, and then we show how the language is able also to capture more extended versions of the local propagation algorithms.

For each local consistency rule that we have in our strategy, we define a suitable assignment that modifies a piece of the constraint store related to the consistency rule itself. We can claim that the semantics of $\mathtt{Sol}(\mathtt{con})$ is the same if we compute it in the constraint store σ or in a constraint store $\mathcal{C}[\![\ \mathtt{Sol}(l, L)\ ]\!]\sigma$ for any l, L, and σ.

Theorem 2 (Equivalence for $\mathtt{Sol}(l, L)$). *Consider a problem $P = \langle C, \mathtt{con}\rangle$ and a local consistency rule $[l \leftarrow L]$. Then we have that*

$$\mathcal{E}[\![\ \mathtt{Sol}(\mathtt{con})\ ]\!]\sigma = \mathcal{E}[\![\ \mathtt{Sol}(\mathtt{con})\ ]\!](\mathcal{C}[\![\ l := \mathtt{Sol}(l, L)\ ]\!]\sigma).$$

Proof. Easily follws from simlar theorem in [BMR97].

5.3 Extending local propagation rules

In the classical CSP environment, a local propagation function is defined (following the approach of [Apt99a,Apt99b]) as a monotone and extensive function that transforms the original problem into a new (possibly equivalent) one. This means that not only a solution over a set of locations $L \cup \{l\}$ and a projection operation over the variables in l are performed but *any* monotone and extensive transformation can be applied to the location l using the locations $L \cup \{l\}$.

For this purpose we have introduced in the language the function $\phi(l, L)$ with the meaning to compute any modification over l by using the constraints over $l \cup L$.

More generally a local propagation function can be seen as a function that transforms a problem in a new one (possibly by modifying only a subpart of it).

Definition 15 (Local propagation). *Given a problem $P = \langle C, \mathtt{con}\rangle$ and the corresponding store σ, a local propagation function $\mathtt{lc}$ is a monotone and extensive (w.r.t. the semiring order $\leq_S$) function that transforms (some of) its costraints obtaining a (possibly equivalent) problem:*

$$\mathtt{lc} : \sigma \rightarrow \sigma$$

The definition of $\mathtt{lc}$ is so general that could match with any of the previously defined local consistency algorithms. We want to characterize it in a more precise way, by using a set of *general local propagation rules* by extending the rules of [BMR95,BMR97]. Note that in [BCGR00] some work in this direction has already been initiated, but no general definition to capture the framework was given.

Definition 16 (general local propagation rule). *A general local propagation rule is defined by a subproblem and a monotone and extensive (w.r.t. the semiring order $\leq_S$) function ϕ. To indicate the application of a function ϕ to a subproblem defined by a rule $r = l \leftarrow L$, we will write $\phi(l, L)$.*

In general since the function $\phi(l, L)$ is monotone and extensive we have:

Theorem 3. *Consider a problem $P = \langle C, con \rangle$ and a general local consistency rule $\phi(l, L)$. Then, for any domain tuple l' we have that*

$$\mathcal{C}[\![\, l := \phi(l, L) \,]\!]\sigma l' \langle l' \rangle \leq_S \sigma l' \langle l' \rangle.$$

6 Conclusion

We have described how the *semiring* framework can be used to both embed the constraint structure and topology in a suitable semiring of functions. Moreover, the functions introduced in the framework enrich the expressivity and the usability of the framework by giving the possibility to express as functions several techniques like dynamic programming and a general version of the local consistency preprocessing methodologies.

We are working to a more extensive use of the semirings to represents SCSPs. In particular, future work will show how the combination and projection of constraint could be seen as special *overloaded* operator of a *generalized semiring*.

References

[Apt99a] K.R. Apt. The essence of Constraint Propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.

[Apt99b] K.R. Apt. The Rough Guide to Constraint Propagation (corrected version). In *Proc. of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99*, number 1713 in LNCS, pages 1–23. Springer-Verlag, 1999. invited lecture.

[BCGR00] S. Bistarelli, P. Codognet, Y. Georget, and F. Rossi. Labeling and partial local consistency for soft constraint programming. In *Proc. of the 2nd International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, number 1753 in LNCS. Springer-Verlag, 2000.

[BD62] R.E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1962.

[Bel57] R.E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[BFM+96] S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and valued CSPs: Basic properties and comparison. In *Over-Constrained Systems*, number 1106 in LNCS. Springer-Verlag, 1996.

[BMR95] S. Bistarelli, U. Montanari, and F. Rossi. Constraint Solving over Semirings. In *Proc. IJCAI95*. Morgan Kaufman, 1995.

[BMR97] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2):201–236, Mar 1997.

[BMR00] S. Bistarelli, U. Montanari, and F. Rossi. Soft constraint logic programming and generalized shortest path problems. *Journal of Heuristics*, 2000.

[DFP93] D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proc. IEEE International Conference on Fuzzy Systems*, pages 1131–1136. IEEE, 1993.

[GMM81] S. Gnesi, A. Martelli, and U. Montanari. Dynamic programming as graph searching: an algebraic approach. *Journal of the ACM*, 28(5):737–751, 1981.

[Mac92] A.K. Mackworth. Constraint satisfaction. In S.C. Shapiro, editor, *Encyclopedia of AI (second edition)*, pages 285–293. John Wiley & Sons, 1992.

[MR86] U. Montanari and F. Rossi. An efficient algorithm for the solution of hierachical networks of constraints. In *Proc. International Workshop on Graph Grammars and their application to Computer Science*, number 291 in LNCS, pages 440–457. Springer-Verlag, 1986.

[Rut94] Zs. Ruttkay. Fuzzy constraint satisfaction. In *Proc. 3rd IEEE International Conference on Fuzzy Systems*, pages 1263–1268, 1994.

[Sch92] T. Schiex. Possibilistic constraint satisfaction problems, or "how to handle soft constraints?". In *Proc. 8th Conf. of Uncertainty in AI*, pages 269–275, 1992.

[Wal96] M. Wallace. Practical applications of constraint programming. *Constraints: An International Journal*, 1(1 and 2), 1996.