Compiling Semiring-based Constraints with `clp(FD,S)`

Yan Georget, Philippe Codognet[*]

INRIA-Rocquencourt, BP 105, 78153 Le Chesnay, France
{Yan.Georget,Philippe.Codognet}@inria.fr

Abstract. In some recent works, a general framework for finite domains constraint satisfaction has been defined, where classical CSPs, fuzzy CSPs, weighted CSPs, partial CSPs and others can be easily cast. This framework, based on a semiring structure, allows, under certain conditions, to compute arc-consistency. Restricting to that case and integrating semiring-based constraint solving in the Constraint Logic Programming paradigm, we have implemented a generic language, `clp(FD,S)`, for semiring-based constraint satisfaction. In this paper, we describe the kernel of the language: the SFD system and our implementation of `clp(FD,S)`. We also give some performance results on various examples.

1 Introduction

In [1, 2], a general framework for finite domains constraint satisfaction and optimization has been defined, where classical CSPs [15, 13, 14], fuzzy CSPs [16, 9, 17], partial CSPs [10] and others can be easily cast. This framework is based on a semiring structure. Moreover, the authors show that local consistency algorithms can be used, provided that certain conditions on the semiring operations are satisfied. Restricting to that case and integrating semiring-based constraint solving in the Constraint Logic Programming paradigm, our goal was to implement a generic language, `clp(FD,S)`, for semiring-based constraint satisfaction using arc-consistency. In order to achieve that goal, we define, as it as been done in [3, 4] for finite domain constraints, a general scheme for compiling semiring-based constraints. The kernel of `clp(FD,S)`, is called SFD and is generic with respect to the semiring. Hence, we are able to generate new languages by specifying semirings, the rest of the implementation being unchanged.
The work the most related to ours is LVCSP [12], a lisp library for constraint solving which is based on the VCSP formalism [18]. An important difference, as stated in [2], is that the semiring-based framework is more general than the VCSP framework because it allows to deal with partial orders. It is also worth mentioning that, in `clp(FD,S)`, the semiring-based solving techniques have been integrated and implemented in a full Constraint Logic Programming (CLP) language, meaning that incrementality is a key feature of the solver. CLP has proved to be very

[*] currently on sabbatical leave at:
SONY Computer Science Laboratory, 6, rue Amyot, 75 005 Paris, France

useful for expressing problems in a concise and natural way and for easily implementing extensions such as optimization predicates.
The rest of the paper is organized as follows: In Section 2, we recall some definitions and results about semiring-based constraint satisfaction. We specialize some results to the case of arc-consistency in Section 3. We introduce the SFD system in Section 4 and give some examples of its use in Section 5. Then, Section 6 describes the implementation while Section 7 presents performances evaluation. Conclusion and perspectives are addressed in Section 8.

2 Semiring-based Constraint Satisfaction

The purpose of this section is to describe briefly the semiring-based framework. For more details, and also for the proofs of the properties, the reader should refer to [2].

2.1 Semirings

Definition 1 (semiring) *A semiring S is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that: A is a set and $\mathbf{0}, \mathbf{1} \in A$; $+$ is closed, commutative, associative, $\mathbf{0}$ is its unit element; $\times$ is closed, associative, distributes over $+$, $\mathbf{1}$ is its unit element, $\mathbf{0}$ is its absorbing element.*

In the following, we will consider semirings with additional properties. Such semirings will be called c-semirings where c stands for constraints, meaning that they are the natural structures to be used when handling constraints.

Definition 2 (c-semiring) *A c-semiring S is a semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that: $+$ is idempotent, $\mathbf{1}$ is its absorbing element.*

We can now define a partial ordering over the set A. For a c-semiring S as defined above, let us consider relation $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$. Then:
- $\leq_S$ is a partial order,
- $+$ and $\times$ are monotone on $\leq_S$,
- $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum,
- $\langle A, \leq_S \rangle$ is a complete lattice and $+$ is its lub.

Additional properties are indeed needed to make it possible to compute local consistency, of which Arc-Consistency (AC [13]) is an instance.

Definition 3 (lc-semiring) *A lc-semiring S is a c-semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that: A is finite; $\times$ is idempotent.*

With the last definition, we get the following results:
- $+$ distributes over $\times$,
- $\langle A, \leq_S \rangle$ is a complete distributive lattice and $\times$ is its glb.

2.2 Constraints

In the following, we suppose given: a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, an ordered set of variables V and a finite set D. We recall the most important definitions of [2].

Definition 4 (constraint) *A constraint is a pair $\langle def, con \rangle$ where $con \subseteq V$ (*type *of the constraint) and $def : D^{|con|} \rightarrow A$.*

A constraint specifies the involved variables and the values "allowed" for them. More precisely, for each tuple of values for the involved variables, a corresponding element of A is given, which can be interpreted as the tuple's weight, or cost, or level of confidence, or anything.

Definition 5 (constraint problem) *A constraint problem is a pair $\langle C, con \rangle$ where $con \subseteq V$ and C is a set of constraints.*

In the following, we will assume that there are not two constraints with the same type (without this hypothesis the results would be a little more complicated).

Definition 6 (tuple projection) *Assuming that V is ordered via ordering $\prec$, consider any k-tuple $t = \langle t_1, \ldots, t_k \rangle$, of values of D and two sets $W = \{w_1, \ldots, w_k\}$ and $W' = \{w'_1, \ldots, w'_m\}$ such that $W' \subseteq W \subseteq V$ and $w_i \prec w_j$ if $i \leq j$ and $w'_i \prec w'_j$ if $i \leq j$. The projection of t from W to W', written $t \downarrow^{W}_{W'}$, is defined as the tuple $t' = \langle t'_1, \ldots, t'_m \rangle$ with $t'_i = t_j$ if $w'_i = w_j$.*

Definition 7 (combination) *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 $def(t) = def_1(t \downarrow^{con}_{con_1}) \times def(t \downarrow^{con}_{con_2})$*

Definition 8 (projection) *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 $def'(t') = \sum_{t / t \downarrow^{con}_{I \cap con} = t'} def(t)$.*

Definition 9 (solution) *The solution of the problem $P = \langle C, con \rangle$ is the constraint $Sol(P) = (\bigotimes C) \Downarrow_{con}$.*

In words, the solution is the constraint induced on the variables in *con* by the whole problem.

Definition 10 (best level of consistency) *The best level of consistency of the problem P is defined by $blevel(P) = Sol(P) \Downarrow_{\emptyset}$. We say that:*

- *P is α-consistent if $blevel(P) = \alpha$,*
- *P is consistent if there exists $\alpha >_S \mathbf{0}$ such that P is α-consistent,*
- *P is inconsistent if it is not consistent.*

Informally, the best level of consistency gives us an idea of how much we can satisfy the constraints of the problem. Note that the best level of consistency of a problem is, in the general case, an upper bound of the values associated with the tuples of its maximal solutions.

Definition 11 (maximal solutions) *Given a constraint problem P, consider $Sol(P) = \langle def, con\rangle$. A maximal solution of P is a pair $\langle t, v\rangle$ satisfying:*

- *$def(t) = v$,*
- *there is no t' such that $v <_S def(t')$.*

Definition 12 (constraint ordering) *Consider two constraints $c_1 = \langle def_1, con\rangle$ and $c_2 = \langle def_2, con\rangle$, with $|con| = k$. Then $c_1 \sqsubseteq_S c_2$ if for all k-tuples t, $def_1(t) \leq_S def_2(t)$.*

The relation $\sqsubseteq_S$ is a partial order.

Definition 13 (problem ordering and equivalence) *Consider two problems P_1 and P_2. Then $P_1 \sqsubseteq_P P_2$ if $Sol(P_1) \sqsubseteq_S Sol(P_2)$. If $P_1 \sqsubseteq_P P_2$ and $P_2 \sqsubseteq_P P_1$, then they have the same solution, thus we say that they are equivalent and we write $P_1 \equiv P_2$.*

The relation $\sqsubseteq_P$ is a preorder. Moreover $\equiv$ is an equivalence relation. Consider two problems $P_1 = \langle C_1, con\rangle$ and $P_2 = \langle C_1 \cup C_2, con\rangle$. Then $P_2 \sqsubseteq_P P_1$ and $blevel(P_2) \leq_S blevel(P_1)$.

2.3 Local Consistency

In the following, we suppose given: a lc-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, an ordered set of variables V and a finite set D.

Definition 14 (local inconsistency) *We say that the problem $P = \langle C, con\rangle$ is locally inconsistent if there exist $C' \subseteq C$ such that $blevel(C') = \mathbf{0}$.*

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 not consistent.

Definition 15 (location) *A typed location l is a set of variables. Given a problem $P = \langle C, con \rangle$, the value $[l]_P$ of the location l in P is $\langle def, l \rangle$ if it belongs to C, $\langle \mathbf{1}, l \rangle$ otherwise. The value $[\{l_1, \ldots, l_n\}]_P$ of the set of locations $\{l_1, \ldots, l_n\}$ is the set $\{[l_1]_P, \ldots, [l_n]_P\}$.*

Definition 16 (assignment) *An assignment is a pair $l := c$ where $c = \langle def, l \rangle$. Given a problem $P = \langle C, con \rangle$, the result of the assignment $l := c$ is defined as $[l := c](P) = \langle \{\langle def', con' \rangle \in C / con' \neq l\} \cup c, con \rangle$.*

Definition 17 (local consistency rule) *A local consistency rule is $l \leftarrow L$ where l is a location, L a set of locations and $l \notin L$.*

Definition 18 (rule application) *The result of applying the rule $l \leftarrow L$ to the problem P is $[l \leftarrow L](P) = [l := Sol(\langle [L \cup \{l\}]_P, l \rangle)](P)$. The application of a sequence of rules $r; R$ is defined by $[r; R](P) = [R]([r](P))$.*

Observe that, given a problem P and a rule r, $P \equiv [r](P)$.

Definition 19 (stable problem) *Given a problem P and a set R of rules for P, P is said to be stable w.r.t R if, for each $r \in R$, $[r](P) = P$.*

Definition 20 (strategy) *Given a set R of rules, a strategy for R is an infinite sequence of R^∞. A strategy T is fair if each rule of R occurs in T infinitely often.*

Definition 21 (local consistency algorithm) *Given a problem P, a set of rules R and a fair strategy T for R, a local consistency algorithm applies to P the rules in R in the order given by T. The algorithm stops when the current problem is stable w.r.t R. In that case, we note $lc(P, R, T)$ the resulting problem.*

The following theorems are proven in [2]. The application of a local consistency algorithm terminates. If $P' = lc(P, R, T)$ then $P \equiv P'$. $lc(P, R, T)$ does not depend on T. Given a problem P and a value v assigned to a tuple in a constraint of P, consider $P' = lc(P, R, T)$ and the value v' assigned to the same tuple of the same constraint in P', then $v' \leq_S v$.

3 Application to Arc-Consistency

In this section, we will instantiate the results concerning local consistency to the particular case of AC.

3.1 Formal Approach

We suppose given a problem $P = \langle C, con\rangle$. Because we want to compute AC, we are interested only in the application of rules of the form:

$$r = \{x\} \leftarrow \{\{x, y_1, \ldots, y_n\}, \{y_1\}, \ldots, \{y_n\}\}$$

where $x, y_1, \ldots, y_n \in con$. We call such a rule an AC-rule, we note AC the set of AC-rules and $ac(P) = lc(P, AC)$. For the AC-rule r previously considered, we have:

$$[r](P) = [\{x\} := Sol(\langle[\{\{x\}, \{x, y_1, \ldots, y_n\}, \{y_1\}, \ldots, \{y_n\}\}]_P\rangle)](P)$$

Noting $C' = \{[\{x, y_1, \ldots, y_n\}]_P, [\{x\}]_P, [\{y_1\}]_P, \ldots, [\{y_n\}]_P\}$, this reduces to:

$$[r](P) = [\{x\} := Sol(\langle C', \{x\}\rangle)](P)$$

Or equivalently:

$$[r](P) = [\{x\} := (\bigotimes C') \Downarrow_{\{x\}}](P)$$

We note $[\{x, y_1, \ldots, y_n\}]_P = \langle def, \{x, y_1, \ldots, y_n\}\rangle$, $[\{x\}]_P = \langle def_x, \{x\}\rangle$ and $\forall i \in [1, n], [\{y_i\}]_P = \langle def_{y_i}, \{y_i\}\rangle$. We assume $x \prec y_1 \prec \cdots \prec y_n$. Finally, the application of the rule r reduce to replacing def_x by its product with:

$$\lambda i_x. \sum_{i_{y_1}, \ldots, i_{y_n}} def(i_x, i_{y_1}, \ldots, i_{y_n}) \times def_{y_1}(i_{y_1}) \times \ldots \times def_{y_n}(i_{y_n})$$

Note that the previous formula is a generalization of Zadeh's extension principle for fuzzy logic [11, 21].

3.2 Exploiting The Shape Of Constraints

The previous formula involves heavy computations. We show that these computations can be simplified for certain types of constraints.
Let us consider, for example, three constraints $c_E = \langle def_E, E\rangle$, with $E \in \{\{x\}, \{y\}, \{x, y\}\}$, $x \prec y$, $def_{\{x,y\}}(i, j) = \alpha$ (resp. β) if $i \geq j$ (resp. $i < j$), and $\alpha \geq_S \beta$.
The constraints $c_{\{x\}}$ and $c_{\{y\}}$ define the domains of x and y, while $c_{\{x,y\}}$ can be seen as $x \geq y$ with values α and β[2]. Here, we are interested in computing one step of AC over variable x. Using the formula of 3.1, the new domain of x is defined by:

$$def_{\{x\}}(i) \times (\sum_j def_{\{x,y\}}(i, j) \times def_{\{y\}}(j))$$

$$= def_{\{x\}}(i) \times (\alpha \times \sum_{j \leq i} def_{\{y\}}(j) + \beta \times \sum_j def_{\{y\}}(j))$$

[2] It extends the classical arithmetic constraint by replacing the boolean truth values with value α for satisfaction and β for violation, while keeping the shape of the constraint. This scheme will be used in Section 5.1 for encoding high-level constraint.

Hence, the new domain of x is easily computed using $\lambda i. \sum_{j \leq i} def_{\{y\}}(j)$ and $\sum_j def_{\{y\}}(j)$. This means that it is possible to specialize the computation to the particular shape of the constraint. This principle can be used for every mathematical relation or function with a "regular shape". We will now design a language of operators well-suited for the encoding of such simple-shaped constraints, but where, more generally, any kind of multi-valued constraint can be cast, by combining primitive operators.

4 The SFD System

Let us now detail the set of indexicals and operators (the SFD system) that allows the compilation of usual mathematical constraints.

4.1 Syntax Of The Language

Definition 22 (valued domain) *A* valued domain *(or domain) is a function from D to A.*

Definition 23 (range) *Let $\mathcal{V}$ be a set of variables. A* range *is a syntactic domain defined by Table 1, where $Y \in \mathcal{V}$.*

Definition 24 (constraint) *Let $\mathcal{V}$ be a set of variables. A* constraint *is a formula of the form X* `in` *r where $X \in \mathcal{V}$ and r is a range.*

4.2 Semantics Of The Constraint Language

In the following, r will be a valued domain and r_i the semiring value associated to integer i. Note that all the indexes take their values in D. Let us first detail the operators and indexicals appearing in the category of ranges:

- The indexical `def` gives the valued domain of a variable.
- The following classes of operators are used for optimization:
  - Given a relation R over the semiring: $r' =$`cut_`$R(\mathtt{r}, \alpha)$ is defined by $r'_j = \mathbf{0}$ if $R(r_j, \alpha)$, $r'_j = r_j$ otherwise.
  - Given a relation R over the semiring: $r' =$`keep_`$R(\mathtt{r}, \alpha)$ is defined by $r'_j = r_j$ if $R(r_j, \alpha)$, $r'_j = \mathbf{0}$ otherwise.
- Semiring operations (between ranges and between a range and a semiring value):
  - $r'' = r + r'$ is defined by $r''_j = r_j + r'_j$.
  - $r'' = r \times r'$ is defined by $r''_j = r_j \times r'_j$.
  - $r' = r + a$ is defined by $r'_j = r_j + a$.
  - $r'' = r \times a$ is defined by $r'_j = r_j \times a$.
- Two operators are used to build domains from integers:
  - An operator for building intervals : $r = \mathtt{i_1..i_2}$ is defined by $r_j = \mathbf{1}$ if $j \in [i_1, i_2]$, $r_j = \mathbf{0}$ if $j \notin [i_1, i_2]$.

Table 1. Syntax of SFD constraints.

$c ::=$	X `in` r		$r ::=$	`def`(Y)	(indexical def)
				`cut_`$R(r,at)$	(cut)
				`keep_`$R(r,at)$	(keep)
				r `+` r	($+$ operation)
$it ::=$	i	(integer)		r `*` r	($\times$ operation)
	`infinity`	(greatest value)		r `+` at	($+$ operation)
	C	(integer parameter)		r `*` at	($\times$ operation)
				it`..`it	(interval)
				`comp`(it)	(exclusion)
				`d`(r)	(different)
$at ::=$	a	(semiring value)		`le`(r)	(less or equal)
	at `+` at	($+$ operation)		`l`(r)	(less)
	at `*` at	($\times$ operation)		`ge`(r)	(greater or equal)
	`pi`(Y)	(indexical pi)		`g`(r)	(greater)
	`sigma`(Y)	(indexical $sigma$)		`add`(r,r)	(addition)
	A	(value parameter)		`sub`(r,r)	(substraction)
				`mul`(r,r)	(multiplication)
				`div`(r,r)	(division)
				`add`(r,it)	(addition)
				`sub`(r,it)	(substraction)
				`mul`(r,it)	(multiplication)
				`div`(r,it)	(division)

- An operator for excluding one index : $r = \texttt{comp(i)}$ is defined by $r_j = \mathbf{1}$ if $j \neq i$, $r_j = \mathbf{0}$ is $j = i$.

– We have the functional versions of usual relations:
  - Less or equal : $r' = \texttt{le(r)}$ is defined by $r'_j = \sum_{k \leq j} r_k$.
  - Less : $r' = \texttt{l(r)}$ is defined by $r'_j = \sum_{k<j} r_k$.
  - Greater or equal : $r' = \texttt{ge(r)}$ is defined by $r'_j = \sum_{k \geq j} r_k$.
  - Greater : $r' = \texttt{g(r)}$ is defined by $r'_j = \sum_{k>j} r_k$.
  - Different : $r' = \texttt{d(r)}$ is defined by $r'_j = \sum_{k \neq j} r_k$.

– Arithmetic operations (between ranges and between a range and an integer):
  - $r'' = \texttt{add(r,r}')$ is defined by $r''_j = \sum_{k+k'=j} r_k \times r'_{k'}$.
  - $r'' = \texttt{sub(r,r}')$ is defined by $r''_j = \sum_{k-k'=j} r_k \times r'_{k'}$.
  - $r'' = \texttt{mul(r,r}')$ is defined by $r''_j = \sum_{kk'=j} r_k \times r'_{k'}$.
  - $r'' = \texttt{div(r,r}')$ is defined by $r''_j = \sum_{k=jk'} r_k \times r'_{k'}$.
  - For each `op` in $\{\texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{div}\}$ `op(r,i)` is defined by `op(r,r')` where $r'_i = \mathbf{1}$ and $r'_j = \mathbf{0}$ if $j \neq i$.

Let us details the indexicals appearing in the category of semiring values:

- The indexical `sigma` is defined by: `sigma(Y)`$=\sum_i Y_i$. `sigma(Y)` is an upper bound of the truth value of `Y`. As the truth values can only decrease during the (monotonic) computation, this is also an upper bound of the truth value that variable `Y` will have when instantiated.
- The indexical `pi` is defined by: `pi(Y)`$=\prod_{i/Y_i \neq \mathbf{0}} Y_i$. `pi(Y)` is a lower bound of the truth value of `Y`. Observe that, for the same reason as above, this is not a lower bound of the truth value that variable `Y` will have when instantiated.

5 Using The SFD System

5.1 Encoding High-Level Constraints

Using the SFD language, it is very easy to encode high-level constraints. Given a classical CSP constraint c (i.e. defined on a boolean semiring), we can derive a semiring constraint with the same shape but with value α for satisfaction and β for violation ($\alpha \geq_S \beta$), noted $c : (\alpha, \beta)$. In Section 3.2 for instance, we have defined the constraint $x \geq y : (\alpha, \beta)$, that could be expressed using SFD as follows:

```
X in ge(def(Y))*Alpha + sigma(Y)*Beta
Y in le(def(X))*Alpha + sigma(X)*Beta
```

We can define other arithmetic constraints similarly.
The constraint $x = y + c : (\alpha, \beta)$ is encoded by:

```
X in add(def(Y),C)*Alpha + sigma(Y)*Beta
Y in sub(def(X),C)*Alpha + sigma(X)*Beta
```

The constraint $x \leq y + z : (\alpha, \beta)$ is encoded by:

```
X in le(add(def(Y),def(Z)))*Alpha + sigma(Y)*sigma(Z)*Beta
Y in ge(sub(def(X),def(Z)))*Alpha + sigma(X)*sigma(Z)*Beta
Z in ge(sub(def(X),def(Y)))*Alpha + sigma(X)*sigma(Y)*Beta
```

More generally, it is possible to encode all the n-ary constraints involving a relation among ($\leq$, $<$, $=$, $>$, $\geq$, $\neq$) and two expressions built with operators among ($+$,$-$,$\times$,$/$). Moreover, other types of constraints can be defined (as in `clp(FD)`), such as, for instance, a weak form of constructive disjunction between constraints [4].

5.2 Computing Maximal Solutions

We will now focus on the computation of maximal solutions.

We suppose given a problem $P = \langle C, con \rangle$. We assume that $con = \{x_1, \ldots, x_n\}$ with $x_1 \prec \cdots \prec x_n$. In the following, we will note $c_{x,i,v}$ the constraint $\langle def_x, \{x\} \rangle$ such that $def_x(j) = v$ if $j = i$ and $def_x(j) = \mathbf{0}$ otherwise. Note that $c_{x,i,v}$ instantiate x to i with truth value v.

We follow the same approach as in traditional CLP: for each n-tuple t, we are going to solve the subproblem $P_t = \langle C_t, con \rangle$ where $C_t = C \cup \{c_{x_i,t_i,\mathbf{1}} / i \in [1,n]\}$. It is easy to prove that the maximal solutions of P can be found among those of the subproblems. For each t, we have to compute $Sol(P_t)$. Let $c_{x_i,t_i,v_i^t} = [\{x_i\}]_{ac(P_t)}$, because of the AC-computation, constraints c_{x_i,t_i} may have been replaced, so we get c_{x_i,t_i,v_i^t} instead of $c_{x_i,t_i,\mathbf{1}}$. It is easy to prove that $Sol(P_t) = \langle def, \{x_1, \ldots, x_n\} \rangle$ with $def(t') = \prod_i v_i^t$ if $t' = t$ and $def(t') = \mathbf{0}$ otherwise. Thus, the unique maximal solution of P_t is $\langle t, \prod_i v_i \rangle = \langle t, blevel(P_t) \rangle$. Note also that: $\prod_i v_i >_S v \Rightarrow \forall i, v_i >_S v$.

The two last results can be used to find a maximal solution of P efficiently: given t_1 and t_2, we can discard P_{t_2} as soon as one of the $v_i^{t_2}$ is not strictly greater than $blevel(P_{t_1})$.

Because we have the full power of a CLP language, it is very easy to write an optimization predicate that computes a maximal solution of given problem. Here is a definition, using a Prolog syntax:

```
maximal_solution(Vars) :-
   semiring_zero(Zero),
   g_assign(max,Zero),

   % first, computes the blevel (proof of optimality)
   repeat,                     % creates a choice point
   g_read(max,Alpha),
   (keep(greater,Vars,Alpha), % for each V in Vars:
                               % V in keep_greater(def(X),Alpha)
   labeling(Vars)
   ->
      blevel(Vars,Blevel),
      g_assign(max,Blevel),
      fail
   ;
      !,
      % then, computes the maximal solutions
      g_read(max,Beta),
      keep(greater_or_equal,Vars,Beta),
      labeling(Vars)).
```

Note that this optimization predicates is fully generic (it does not depend on the semiring).

6 Implementation

Let us first define an abstract encoding scheme for the semiring and then go into the implementation details.

6.1 Semiring Representation

In order to have a generic implementation, we have chosen to work on a representation of the semiring. Consider a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and:

- $B = [0, |A| - 1]$, i.e. the integer range between 0 and $|A| - 1$,
- ϕ a bijective function from A to B,
- $+_\phi = \lambda xy.\phi(\phi^{-1}(x) + \phi^{-1}(y))$ and
- $\times_\phi = \lambda xy.\phi(\phi^{-1}(x) \times \phi^{-1}(y))$.

Then, $S' = \langle B, +_\phi, \times_\phi, \phi(\mathbf{0}), \phi(\mathbf{1}) \rangle$ is a c-semiring and ϕ is a morphism from A to B. We call S' a representation of S.

For example, since binary numbers with 3 bits range from 0 to 7, $\langle \mathcal{P}(\{a, b, c\}), \cup, \cap, \emptyset, \{a, b, c\} \rangle$ can be represented by $\langle [0, 7], |, \&, 0, 7 \rangle$ (where $|$ and $\&$ are the C bitwise operators over integers).

Hence, if the user wants to use the semiring S, he has to:

- define S' (actually, knowing $|A|$, $+_\phi$ and $\times_\phi$ is sufficient),
- recompile a small part of the system (this could be avoided, but we have chosen this solution for efficiency reasons),
- then, he can write his `clp(FD,S)` programs using the semiring S'.

With this approach, the implementation remains totally generic. Moreover, the same example (and same code!) can be tested using different semirings.

6.2 Implementation Of X `in` r

We describe very briefly the implementation of X `in` r constraints in `clp(FD,S)`. This implementation is very close to the one of X `in` r constraints in `clp(FD)`. Thus, for more details, the reader should refer to [4]. Let us detail the three main data structures used to manage X `in` r constraints:

Constraint Frame: A constraint frame is created for every constraint. The informations recorded in a constraint frame are: the address of the associated code, the address of the FD variable it constrains, and the pointer to the environment in which to evaluate its range. The environment is just a list of as many arguments as are in the range of the constraint: each argument may be an integer, a semiring value, a (pointer to a) FD variable or a (pointer to a) range.

FD Variable Frame: The frame associated to an FD variable X is divided into two main parts: the valued domain (it is stored in a bit vector, but, as soon as X is instantiated, we switch to another representation, which is a pair index-value) and a dependency list pointing to the constraint frames of the constraints depending on X. These two parts are modified at different times during computation: the dependency lists are created when the constraints are *installed* and then are never changed, while the domain can be updated during execution. Actually, several distinct dependency lists are used:

– one related to `def`,
– one related to `sigma`,
– one related to `pi`,
– one related to `sigma` and `pi`.

This helps avoiding useless propagation.

Propagation Queue: The propagation phase consists of awakening and executing some constraints that could themselves select some new constraints to be awaken. To do this, we maintain an explicit propagation queue. A simple optimization consists in not pushing constraints but only pairs of the form $< X, mask >$ where X is the variable causing the propagation (that is, the one whose domain has been changed) and *mask* is a bit-mask of dependency lists to awake.

Finally, the implementation of SFD is very small: 5000 C lines and 500 Prolog lines in addition to the code for the Prolog compiler and run-time engine, taken from the original `clp(FD)` system.

7 Examples

In this section, we will test some of the instances of `clp(FD,S)`, using the following lc-semirings:

– $Bool = \langle\{false, true\}, \vee, \wedge, false, true\rangle$,
– $Fuzzy = \langle\{0.0, 0.1, \ldots, 1.0\}, max, min, 0.0, 1.0\rangle$,
– $Set = \langle\mathcal{P}(U), \cup, \cap, \emptyset, U\rangle$.

7.1 Boolean Examples

We compare `clp(FD,`*Bool*`)` and `clp(FD)`. We consider the problems `five` (Lewis Carroll's Zebra puzzle), originally presented in [19], and `cars` (a car sequencing problem), originally presented in [8]. We have run both implementations on a Sun Sparc 5. The results[3], given in Table 2, show that `clp(FD,`*Bool*`)` is slower than `clp(FD)`. The slow-down results from the loss of some specific optimizations, e.g. optimized boolean operations on bit-vectors. Observe also that, since `clp(FD)` indeed uses partial AC (propagating minimal and maximal values of domains), it could be much faster than `clp(FD,S)` on benchmarks heavily using arithmetic constraints, viz. linear equations. We plan to integrate a similar technique in `clp(FD,S)` in the future.

[3] `five` has been run a hundred times.

Table 2. `clp(FD)` versus `clp(FD,`*Bool*`)`

Problem	Time in ms using:	
	`clp(FD)`	`clp(FD,`*Bool*`)`
100×`five`	310	680
`cars`	20	70

7.2 Fuzzy Examples

We compare `clp(FD,`*Fuzzy*`)` and CON'*FLEX*, a system dedicated to fuzzy constraint solving [7]. We consider the problems `cantatrice` and `menu` given with the CON'*FLEX* distribution. Both `clp(FD,`*Fuzzy*`)` and CON'*FLEX* have been run on a Pentium Pro 200. The results, given in Table 3, show that `clp(FD,`*Fuzzy*`)` performs well compared to a dedicated system.

Table 3. CON'*FLEX* versus `clp(FD,`*Fuzzy*`)`

Problem	Time in ms using:	
	CON'*FLEX*	`clp(FD,`*Fuzzy*`)`
`cantatrice`	60	10
`menu`	30	10

7.3 Set-based Examples

We used `clp(FD,`*Set*`)` to solve time-tabling problems with preferences [5]. More precisely, we used the set-based framework to encode global preferences. We compare our results with those of [5]. Caseau and Laburthe use a global constraint, based on the Hungarian method, dedicated to weighted matching problems and additional redundant constraints (CLAIRE2), they also provide results using simple constraint propagation (CLAIRE1). Both `clp(FD,`*Set*`)` and the CLAIRE [6] codes have been run on a Sun Sparc 5. The results are given in Table 4. Our method is more efficient than simple constraint propagation for the three problems (both in time and backtrack number), it is more efficient than the weighted matching technique for problems 1 and 3 (both in time and backtrack number) but less efficient for problem 2 (for which the satisfiability is easier).

Table 4. CLAIRE versus `clp(FD,`*Set*`)`.

	CLAIRE1		CLAIRE2		clp(FD,*Set*)	
Problem	Backtracks	Time (s)	Backtracks	Time (s)	Backtracks	Time (s)
1	189000	1541	3500	29	899	18.6
2	286000	1040	234	2.6	31921	254
3	43000	246	17000	120	1797	9.7

8 Conclusion

We have defined a generic scheme for compiling semiring-based constraints. Using this kernel and a semiring representation, we can generate a new solver dedicated to the chosen semiring. Because of the generality of our approach, we lose some optimizations that could be introduced in the solver (for example, in the "boolean" one) but we still are efficient with respect to dedicated systems (this is due to the fact that we have captured the essential notions of the computation). Another important point is that we can now imagine (and trivially implement!) new solvers (for example, by combining different semirings).
Future work will focus on extending the expressive power of the SFD system and introducing partial AC (instead of the exact computation of AC which might be too costly in arithmetic examples). This roughly means moving from AC-3 to AC-5 [20] as in `clp(FD)`. The problem in our case is to find a good and generic approximation of a valued domain.
Finally, note that the `clp(FD,S)` system is freely available by:

`http://pauillac.inria.fr/~georget/clpfds.html`.

Acknowledgments

We would like to thank Daniel Diaz, Francesca Rossi and Stefano Bistarelli for fruitful discussions and comments.

References

1. S. Bistarelli, U. Montanari and F. Rossi. Constraint Solving over Semirings. In *Proceedings of IJCAI'95*, Morgan Kaufman, 1995.
2. S. Bistarelli, U. Montanari and F. Rossi. Semiring-based Constraint Solving and Optimization. Journal of ACM, vol.44, n.2, pp. 201-236, March 1997.
3. P. Codognet and D. Diaz. A Minimal Extension of the WAM for `clp(FD)`. In *Proceedings of ICLP'93, 10th Int. Conf. on Logic Programming*, Budapest, Hungary, MIT Press 1993.

4. P. Codognet and D. Diaz. Compiling Constraints in `clp(FD)`. *Journal of Logic Programming*, vol. 27, no. 3, 1996.
5. Y. Caseau and F. Laburthe. Solving Various Weighted Matching Problems with Constraints. In *Proceedings of CP'97, 3rd Int. Conf. on Constraint Programming*, Springer Verlag, 1997.
6. Y. Caseau and F. Laburthe. The Claire documentation. LIENS Report 96-15, Ecole Normale Superieure, Paris, 1995.
7. CON'*FLEX*, manuel de l'utilisateur Laboratoire de Biométrie et d'Intelligence Artificielle, INRA, France, 1996
8. M. Dincbas, H. Simonis, P. Van Hentenryck. Solving the Car-Sequencing Problem In Constraint Logic Programming. In *Proceedings of ECAI'88.* Munich, West Germany, August 1988.
9. D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proceedings of IEEE International Conference on Fuzzy Systems.* IEEE, 1993.
10. E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *AI Journal*, 58, 1992, pp.21–70.
11. L. Gacôgne. Elements de logique floue. Hermes, 1997.
12. M. Lemaître and L. Lobjois. Bibliothèque d'algorithmes de résolution de problèmes et d'optimisation sous contraintes. CERT, FRANCE, 1997.
13. A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence 8 (1977)*, pp 99-118.
14. A.K. Mackworth. Constraint satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of AI (second edition)*, volume 1, pages 285–293. John Wiley & Sons, 1992.
15. U. Montanari. Networks of constraints: Fundamental properties and application to picture processing. *Information Science*, 7, 1974.
16. A. Rosenfeld, R.A. Hummel, and S.W. Zucker. Scene labelling by relaxation operations. *IEEE Transactions on Systems, Man, and Cybernetics*, 6(6), 1976.
17. Z. Ruttkay. Fuzzy constraint satisfaction. In *Proceedings of 3rd International Conference on Fuzzy Systems*, 1994.
18. T. Schiex, H. Fargier, and G.Verfaillie. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In *Proceedings of IJCAI'95.* Morgan Kaufmann, 1995.
19. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.
20. P. Van Hentenryck, Y. Deville and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence 57 (1992)*, pp 291-321.
21. L.A. Zadeh. Calculus of fuzzy restrictions. in K. Tanaka, L.A. Zadeh, K.S Fu and M. Shimura editors, *Fuzzy sets and their applications to cognitive and decision processes.* Academic Press, 1975.

This article was processed using the LaTeX macro package with LLNCS style