Forward-Tracking: A Technique for Searching Beyond Failure

Elena Marchiori[1,3] Massimo Marchiori[2] Joost N. Kok[3]

[1]CWI
P.O. Box 94079, 1090 GB Amsterdam (NL)
elena@cwi.nl

[2]University of Padova
via Belzoni 7, 35131 Padova (I)
max@math.unipd.it

[3]University of Leiden
P.O. Box 9512, 2300 RA Leiden (NL)
joost@wi.leidenuniv.nl

Abstract

In many applications, such as decision support, negotiation, planning, scheduling, etc., one needs to express requirements that can only be partially satisfied. In order to express such requirements, we propose a technique called forward-tracking. Intuitively, forward-tracking is a kind of dual of chronological back-tracking: if a program globally fails to find a solution, then a new execution is started from a program point and a state 'forward' in the computation tree. This search technique is applied to the specific paradigm of constraint logic programming, obtaining a powerful extension that preserves all the useful properties of the original scheme. We report on the successful practical application of forward-tracking to the evolutionary training of (constrained) neural networks.

1 Introduction

One of the most important search mechanism in automated reasoning is chronological back-tracking. Intuitively, the control can during the execution jump backwards in case of failure, in order to try other alternatives. If no alternative is left, then the execution stops. In many applications, however, like decision support, planning, and scheduling, one needs to express also some alternatives to (global) finite failure.

In this paper, we propose a novel technique, called forward-tracking, that allows one to formalize in a simple way problems as those described above, where in case of failure the constraints on the problem are relaxed in order to produce a satisfactory solution. Informally, this technique is based on the following idea: Some points of a program are selected as 'forward-tracking points' (ft-points); when a global failure occurs, the program is restarted from a new point 'forward' in the program, taking as state the 'best' value previously produced in one of the forward-tracking points. In order to define such a *forward-tracking* for a program, one needs a set of ft-points, a 'jump function' to restart the program from another point, and a 'height ordering' (to select the best value). Forward-tracking is useful in all the situations where we are interested in an optimal solution with respect to several requirements, and where we require that a satisfactory partial solution be given in case of failure. These situations include for instance conflicting requirements, over-constrained problems, partial knowledge problems, prioritized reasoning and so on.

In this paper we apply forward-tracking to the specific paradigm of constraint logic programming (cf. [6]), a general scheme that allows one to express in a declarative way reasoning in the presence of constraints. We investigate semi-automatic and (almost) fully-automatic methods for computing forward-trackings. At the most automated level, the user can just use a constraint logic program, which is automatically executed with forward-tracking; in another almost automated level, the user has only to insert one additional predicate, named *separation*, in the locations where (s)he thinks there is a separation of concerns. Forward-tracking is computationally little expensive, since it can be implemented in such a way that: as far as time complexity is concerned, there is only a small (fixed-time) overhead each time a forward-tracking point is reached or a forward jump is performed; as far as space complexity is concerned, forward-tracking only needs a limited fixed-size amount of extra memory w.r.t. the original program. Forward-tracking allows to formalize complex problems that can be subject to various constraints, and cannot be easily tackled with usual techniques, in an elegant and *modular* way as follows. First, the various aspects of the problem are specified by means of modules, each module describing a particular aspect. Next, the forward-tracking technique is used for combining the modules with respect to the specific instance of the problem considered, filtering out possible inconsistencies, in order to obtain satisfactory solutions. This design approach is illustrated in a practical application (Section 6), namely the

training of constrained neural networks.

We turn now to the relation of our proposal to previous work. Related work includes Hierarchical Constraint Solving and the corresponding Hierarchical Constraint Logic Programming (HCLP) systems (see e.g. [1, 9]). In these schemes, constraints hierarchies are used in order to express preferences as well as requirements among constraints. Intuitively, the constraints used in the problem are labeled by means of preferences, which express the strengths of the constraints. During the execution of the program, non-required constraints are accumulated and they are solved only after the execution of the query is successfully terminated, selecting a method that depends on the domain and on the technique used for comparing solutions. The incremental version of this technique ([9]) tries to satisfy constraints as soon as they appear, requiring a special backward algorithm. The main differences of forward-tracking with the above approach are: Forward-tracking does not depend on the particular constraint solver employed; it is applicable also to logic programs, i.e., when the information that we want to discard in case of finite failure is described by some predicates; it is built 'on top' of the program, and does not need, like the previous paradigms, a specific constraint solver to be built from scratch. Moreover, forward-tracking is *transparent* with respect to the original program, in the sense that it is only activated when the program globally fails; in all the cases when the original program gives a solution, forward-tracking does not interfere. Thus forward-tracking can be optionally activated on request, when the original program fails. Finally, forward-tracking, in its *dynamic* version (Subsection 5), allows to give preferences among constraints that are created during the execution of the program (i.e. 'temporal' priorities), which is not possible in the hierarchical constraint solving paradigm.

Other related work concerns the relaxation of constraints in case of conflicts (e.g. [3, 2]). Recently, in order to discard failed subproblems in a constraint satisfaction problem, [4] introduced a method that dynamically discards failed subproblems during forward checking search. This techniques can be successfully integrated with forward-tracking for constraint logic programming since the latter, as said, is a search technique that works on top of a constraint solver.

2 Searching Beyond Failure

In this section we explain the technique of forward-tracking. In order to focus more on the technique than on the programming paradigm where it is employed, we shall describe in a rather informal way the operational model. The incorporation of the technique into a specific programming paradigm, namely Constraint Logic Programming, is examined in detail in the next section.

The operational semantics of a program language can be defined in terms of a transition system with rules of the form $(point, state) \rightarrow (point', state')$ if $conditions$, meaning that if the execution reaches a certain (program) $point$ with a certain $state$, and if the $conditions$ are satisfied, then the execution goes to $point'$ with state $state'$. Usually, an initial (*start*) and a final (*end*) point are used to represent input and output, respectively. Given an input state *input*, the set of computations is considered, obtained using the rules of the transition system starting from the pair *(start, input)*. Then the operational semantics of the program is defined, w.r.t. an *input*, by abstracting the set of computations from *(start, input)*, according to the information one is interested in. Finite computations ending in *(end, s)* are called *successful* and s is called output state. Instead, a program *fails* (w.r.t. an input state) if it does not compute any output state, while it *finitely fails* if it fails and all its computations are finite.

Forward-tracking allows to deal with finite failure, by allowing the computation to restart at a suitable point, from a suitable state among those that have been produced during the computations. In order to define a forward-tracking, a suitable subset *FTP* of points, called forward-tracking points (ft-points), is selected, representing the points from which one can jump forward in case of failure. Moreover, a *height* partial ordering $<_{\mathcal{H}}$ on pairs $(\zeta, state)$ consisting of a point and a state is used, in order to choose a proper pair in case of finite failure. Finally, a *jump* function J on ft-points is employed, that specifies how to jump forward. In the sequel, P denotes a program, *PP* and FTP the set of its points and ft-points, respectively. Moreover, $\mathcal{S}$ denotes the set of states. We have the following definitions.

Definition 2.1 A *jump function*, denoted by *J*, is a function *FTP* $\rightarrow$ *FTP*. An *height ordering (w.r.t. FTP)*, denoted by $<_{\mathcal{H}}$, is a partial order on *FTP* $\times$ $\mathcal{S}$. A *forward-tracking FT for P*, is a pair $(J, <_{\mathcal{H}})$, where *J* and $<_{\mathcal{H}}$ are a jump function and a height order on *FTP*, respectively. The triple $(P, J, <_{\mathcal{H}})$ is called *forward-tracking program* (ft-program for short). ∘

The operational semantics of a ft-program can be roughly explained as follows. The execution keeps track of the combined information on ft-points and states, by means of pairs consisting of a ft-point and a state, indicating that the control has passed through that ft-point in that state. During the execution a set of best pairs is accumulated, containing the maximal (w.r.t. $<_{\mathcal{H}}$) pairs among those that have been produced from the begin of the execution till that moment. When the program finitely fails, and the control has passed through some ft-point, i.e. the set of best pairs is not empty, one best pair is selected, say $(\zeta, state)$, and a new execution is started from the ft-point obtained by applying J to ζ, in the state *state*. Observe that there can be more than one way to re-start the program, depending on the maximal element that is selected.

In order to work properly, a forward-tracking has to satisfy some correctness properties (otherwise, we would have a kind of unrestricted 'goto' feature).

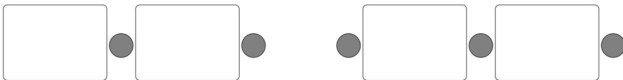
Definition 2.2 A forward-tracking $(J, <_{\mathcal{H}})$ is *correct* if the following conditions hold:
1. For every $\zeta, \eta \in FTP$, if $J(\zeta) = \eta$ then every successful computation of P that reaches ζ passes also through η.
2. If P starting from $(start, s)$ successfully computes an output, then $(P, J, <_{\mathcal{H}})$ starting from $(start, s)$ computes the same output.
3. If $(P, J, <_{\mathcal{H}})$ with input *input* finitely fails, then P with input *input* finitely fails. ∘

The first condition ensures that the flow of control of the original program is respected: in other words, that the jump should be a 'forward' jump, and not a 'backward' jump. The other two conditions ensure that the ft-program behaves at least as well as the corresponding program, i.e., it does not lose successful computations (condition 2.); and it does not add finite failure computations (condition 3.).

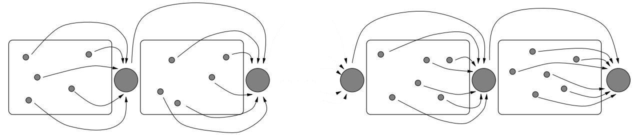
3 Specifying Complex Problems

In this section we present a methodology based on the forward-tracking technique for the treatment of complex problems.

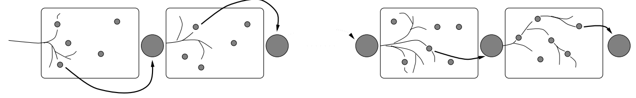
In general, a problem P is made up by several factors, say $P_1, \ldots, P_k$. These sub-problems might be tightly coupled to each other by interdependencies, i.e. not forming a 'modular decomposition' of the general task: in order to get the original problem from the sub-problems, one has to face complex interactions among them. Thus, $P_1, \ldots, P_k$ can form a kind of 'lossy' modular decomposition of the problem. One can 'modularize' P using forward-tracking in the following way. First, separate modules for $P_1, \ldots, P_k$ are developed, say $M_1, \ldots, M_k$. Next, possible conflicts among $P_1, \ldots, P_k$ are 'solved' by means of forward-tracking, by employing the separation given by the modules. This can be realized by inserting *separation* points among the modules, as illustrated in the picture below by big gray circles:



This way each time a module is invoked, it starts solving the specific problem. When it cannot progress further, the control forward-tracks to the next module, and all the work done in the module is preserved. After the automatic calculus of forward-tracking, using e.g. the techniques of Section 5, a situation as the one pictured below is reached, where the small black circles denote the ft-points:



where a possible execution is



Note that modules are applied in a certain order, this way expressing preferences among them: modules appearing prior in the chain will have higher priority than modules appearing later.

4 Forward-Tracking for CLP

In this section we introduce a formalization of forward-tracking for the CLP paradigm (cf. [6]), and provide a sufficient condition for the correctness of a forward-tracking.

A domain $\mathcal{D}$ of constraints is considered, where a constraint c is a (first-order) formula built from primitive constraints. The *states* are the consistent constraints, denoted by c, d. In the sequel, A (possibly subscripted) denotes an atom as well as a constraint; for an atom A, $pred(A)$ indicates its predicate symbol. Queries (i.e., sequences of atoms and constraints) and clauses are supposed to be decorated with distinct *program points*, denoted by ζ, η, ν. For a clause $H \leftarrow {}_{\zeta_0} A_1 {}_{\zeta_1} \cdots {}_{\zeta_{i-1}} A_i {}_{\zeta_i} \cdots {}_{\zeta_{k-1}} A_k.$, ζ_{i-1} is the *calling point* of A_i, denoted by $cp(A_i)$, and $atom(\zeta_{i-1})$ indicates A_i. A constraint logic program (clp) is a set P of clauses augmented with a query Q. The points *start* and *end* are the leftmost and rightmost point of Q, respectively.

In the operational model we are going to describe, the PROLOG left-to-right selection rule is assumed. A transition system for ft-clp's is built on the top of an operational semantics for clp's, that describes also information on ft-points, and is defined w.r.t. an height ordering $<_{\mathcal{H}}$.

We consider a transition system for clp's, defined on two kinds of tuples: either $\langle Q, c, S, \phi \rangle$ or $\langle fail, S, \phi \rangle$, where S is a set of triples of the form (ξ, d, ψ), with ξ an ft-point, d a state, and where ψ and ϕ are functions mapping points into functions that rename the program variables. S and ϕ are used for keeping track of the ft-points encountered during the computation together with the corresponding states, and for each ft-point, of the most recent renamings of the variables of the program clause where it occurs, respectively. The latter information, described by ϕ, is needed when a jump forward is made. In this case, ϕ is used to relate the variables of the clause containing the point where the computation has jumped and the variables occurring in the state used to restart

the computation from that point. There are four transition rules. The two transition rules for steps yielding failure are the standard, except that here the information on S and ϕ is kept, i.e., $\langle Q,\, c,\, S, \phi\rangle \longrightarrow \langle \mathit{fail},\, S, \phi\rangle$. The other two rules treat successful computation steps as follows, where $cp(Q)$ denotes the leftmost calling point of Q.
resolution: $\langle_\zeta\, p(\overline{s})\, Q,\, c,\, S, \phi\rangle \;\longrightarrow\; \langle R\, Q,\, c',\, S', \phi'\rangle$, where $p(\overline{t}) \leftarrow R$ is a renamed apart clause of P, say by means of the function ρ, $c' = (c \wedge \overline{s} = \overline{t})$ is consistent, S' consists of the maximal triples of $S \cup \{(cp(R), c', \phi')\}$, and ϕ' is equal to ϕ except for the points that are in R that are mapped into ρ. Here and in the sequel maximal is intended w.r.t. $<_\mathcal{H}$, when projecting away the third component of the elements.
constraint: $\langle_\zeta\; d\; Q,\, c,\, S, \phi\rangle \longrightarrow \langle Q,\, c \wedge d,\, S', \phi\rangle$, where S' consists of the maximal triples of $S \cup \{(cp(Q), c \wedge d, \phi)\}$.

Observe that in the last two rules we could have chosen S' to be equal to $S \cup \{(cp(R), c', \phi')\}$. However, for efficiency reasons, we have considered a (in general) smaller set.

The operational semantics for a clp $P \cup \{Q\}$ w.r.t. $<_\mathcal{H}$ is defined as follows. Below, I denotes the function mapping each point into the identity renaming ϵ, $\longrightarrow^*$ denotes the reflexive and transitive closure of $\longrightarrow$, and $\Box$ denotes the empty query.

Definition 4.1 The (input-output) *operational semantics* $\mathcal{O}(P \cup \{Q\})$ *of* $P \cup \{Q\}$ *(w.r.t.* $<_\mathcal{H}$*)* is the set $\mathcal{SS}(Q) \cup \mathcal{FS}(Q)$, where $\mathcal{SS}(Q)$ is the set of pairs (c, c') s.t. $\langle Q,\, c,\, \{(\mathit{start}, c, I)\}, I\rangle \longrightarrow^* \langle \Box, c', S, \phi\rangle$ for some S, ϕ, and $\mathcal{FS}(Q)$ is the set of tuples (c, S, ϕ) s.t. $\langle Q,\, c,\, \{(\mathit{start}, c, I)\}, I\rangle \longrightarrow^* \langle \mathit{fail}, S, \phi\rangle$. ∘

It is not difficult to check that the standard input-output semantics of (ideal) clp's with left-to-right selection rule can be obtained as an abstraction of $\mathcal{O}(P \cup \{Q\})$, by adding to $\mathcal{SS}(Q)$ the set of pairs (c, fail) with $(c, S, \phi) \in \mathcal{FS}(Q)$.

A derivation is defined in the usual way, as a sequence of elementary steps obtained using the transition system. The (standard) notion of finite failure is defined as follows. (Q, c) *finitely fails* if every its derivation is finite and $\mathcal{SS}(Q) \cap \{(c, c') \mid c' \in \mathcal{D}\} = \emptyset$.

The transition system for ft-clp's uses $\mathcal{O}(P \cup \{Q\})$ to specify the applicability conditions of the rules. It is defined either on tuples of the form $\langle Q, c\rangle$ or $\langle \mathit{fail}\rangle$. It consists of the three following rules:
success: $\langle Q, c\rangle \longrightarrow_{ft} \langle \Box,\, c'\rangle$ if $(c, c') \in \mathcal{SS}(Q)$.
forward-tracking: $\langle Q, c\rangle \longrightarrow_{ft} \langle Q', c'\rangle$ if (Q, c) finitely fails, (ζ, c', ϕ) is maximal among the elements of all the sets S such that $(c, S, \mathcal{R})$ is in $\mathcal{FS}(Q)$; R is the clause-body (or the query) containing $J(\zeta)$, $\rho = \phi(J(\zeta))$ and Q' is the suffix of $R\rho$ starting at $J(\zeta)$.
failure: $\langle Q, c\rangle \longrightarrow_{ft} \langle \mathit{fail}\rangle$ if (Q, c) finitely fails and there is no maximal element among the elements of all the sets S s.t. $(c, S, \mathcal{R})$ is in $\mathcal{FS}(Q)$.

The input-output semantics of a ft-clp can be defined as follows.

Definition 4.2 The operational semantics $\mathcal{O}_{P \cup \{Q\}}(FT)$ of $(P \cup \{Q\}, FT)$ is the set $\mathcal{SS}(Q, FT) \cup \mathcal{FS}(Q, FT)$, where $\mathcal{SS}(Q, FT)$ is the set of pairs (c, c') s.t. $\langle Q, c\rangle \longrightarrow^*_{ft} \langle \Box, c'\rangle$, and $\mathcal{FS}(Q, FT)$ is the set of pairs (c, fail) s.t. $\langle Q, c\rangle \longrightarrow^*_{ft} \langle \mathit{fail}\rangle$. ∘

Example 4.3 A Hamiltonian path of a graph is an acyclic path containing all the nodes of the graph. The following (fragment of the) general logic program *hamiltonian* defines hamiltonian paths.

$ham(G, P) \leftarrow path(N1, N2, G, P)\;_\xi\; cov(P, G).$
$cov(P, G) \leftarrow \neg notcov(P, G).$
$notcov(P, G) \leftarrow node(X, G), \neg member(X, P).$

The relation $ham(g, p)$ is specified in terms of *path* and *cov*: it is true if p is an acyclic path of g that covers all its nodes. Acyclic paths of a graph are specified by means of the predicate *path*.

We assume that negation by (finite) failure is used for resolving negated atoms. Using a forward-tracking, the behaviour of *hamiltonian* can be 'relaxed' in order to obtain useful answers also when there is no hamiltonian path. Consider the order $\leq$ on substitutions s.t. $\alpha \leq \beta$ if $\beta = \alpha\gamma$ for some γ. Define $J(\xi) = \mathit{end}$, and $(\xi, \alpha) <_\mathcal{H} (\xi, \beta)$ if $\alpha < \beta$. Then, the ft-program $(\mathit{hamiltonian} \cup \{ham(g, X)\}, J, <_\mathcal{H})$ produces answers also for those graphs containing no hamiltonian paths, where *ham(g,X)* will bind X to an acyclic path of g of maximal length. We will show in the next section that the resulting ft-program is 'correct'. ∘

Correct Forward-Trackings

We introduced now a sufficient criterion for the correctness of the forward-tracking for clp's. To this aim, we assume that a jump function must respect a certain *forward ordering* $<_\mathcal{F}$. This ordering imposes reasonable conditions on possible jumps, preventing the introduction of cycles in the derivations, and ensuring that the flow of control of the original program is 'respected': a jump from ζ to η is allowed only if, in order to reach η the control (in the original clp) has first to reach a point that is in the same clause of η and occurring to its left, and then to reach ζ. For instance, consider the program clauses $p \leftarrow q, r.,\;\; s \leftarrow c.,\; c \leftarrow .,$ $r \leftarrow .$ One can jump from $cp(q)$ to $cp(r)$, but not from $cp(q)$ to $cp(c)$, because the corresponding clauses are completely unrelated. The effect of the latter kind of jumps is similar to that of the *goto*-like statements in the imperative paradigm, and therefore they are discarded.

In order to formalize the notion of forward ordering, we use the following dependency relation DEP on points. In the sequel, we write η is (immediately) to the right/left of ζ as a shorthand for ζ and η occur in the same clause and η is (immediately) to the right/left of ζ.

Definition 4.4 *DEP* is the transitive closure of the relation *dep* on program points defined by $(\zeta, \eta) \in dep$ if either $atom(\eta)$ and the head of the clause containing ζ have the same predicate symbol, or $atom(\eta)$ is a constraint and η is immediately to the right of ζ. ◦

Example 4.5 Consider the program clauses
$p \leftarrow_{\zeta_1} q_{\zeta_2} r.,\ q \leftarrow_{\zeta_3} s.,\ r \leftarrow_{\zeta_4} c_{\zeta_5} r.$
Then $DEP = \{(\zeta_3, \zeta_1), (\zeta_4, \zeta_2), (\zeta_5, \zeta_4), (\zeta_4, \zeta_5), (\zeta_5, \zeta_2)\}$. ◦

The following definition of safe pair of points formalizes when a jump respects the control flow of the program. For a subset T of points, we say that T has a maximum ν if $\nu \in T$ and for every $\zeta \neq \nu$ occurring in T we have that $(\nu, \zeta) \in DEP$ but $(\zeta, \nu) \notin DEP$.

Definition 4.6 A pair (ζ, η) of points is *safe* if there exists a set $\mathcal{C}$ of points s.t.:
1. $\zeta \in \mathcal{C}$;
2. $\mathcal{C}$ has a maximum, say ν;
3. for each $\zeta \in \mathcal{C}$, if $\zeta \neq \nu$ and $(\zeta, \eta) \in dep$ then $\eta \in \mathcal{C}$;
4. η is to the right of ν. ◦

Observe that if η is to the right of ζ then (ζ, η) is safe (take $\mathcal{C} = \{\zeta\}$). This formalizes that it is possible to jump from a point to another point of the same query in the direction of the flow of control, i.e. from left to right, unless the jump introduces a cycle. This latter requirement is expressed by condition 2. of the following definition of forward ordering.

Definition 4.7 The *forward ordering* $\leq_{\mathcal{F}}$ is the greatest partial ordering $\leq$ on points such that if $\zeta \leq \eta$ then
1. (ζ, η) is safe;
2. $(\zeta, \eta) \notin DEP$;
3. if $\zeta = end$ then $\eta = end$. ◦

We illustrate by an example how condition 2. prevents the introduction of cycles.

Example 4.8 Consider the program clauses $p \leftarrow_{\zeta_1} q_{\zeta_2} r.$, $r \leftarrow_{\zeta_3} p$. If we would not have condition 2. in the definition of forward ordering, then the set $\{(\zeta_1, \zeta_2)\}$ would be a legal ordering. But following the intended interpretation, this ordering yields the infinite computation $p, q, r, p, \ldots$, while in the original program all derivations are finite. ◦

We say that a ft-clp $(P \cup \{Q\}, J, <_{\mathcal{H}})$ *respects the forward ordering* $<_{\mathcal{F}}$ if the jump function J is such that $\zeta <_{\mathcal{F}} J(\zeta)$ (provided $\zeta \neq end$). The following result holds.

Theorem 4.9 *If a ft-clp respects the forward ordering, then it is correct.*

As far as the first condition of correct forward-tracking is concerned, an ft-clp respecting $<_{\mathcal{F}}$ satisfies even more:

Lemma 4.10 *Let $(P \cup \{Q\}, J, <_{\mathcal{H}})$ be a ft-clp respecting the forward ordering. Let $\zeta, \eta \in FTP$ and suppose that $J(\zeta) = \eta$. Every computation of Q that reaches ζ passes also through a point to the left of η.*

The above lemma is important, because it ensures that when a jump is performed from a given point equipped with a state, to another point, the state can be safely attached to the new point, because it describes information on (some renaming of) the variables of the clause containing that point.

Another important property of forward-trackings respecting $<_{\mathcal{F}}$ is that they behave well w.r.t. termination, in the sense that they preserve strong termination. Recall that a (ft-)clp P is strongly terminating if every Q has only finite computations. Moreover, passing through ft-points eliminates finite failure. Thus, the following results hold.

Theorem 4.11 *Suppose P is strongly terminating. For every query Q, if FT is a forward-tracking for $P \cup \{Q\}$ respecting $<_{\mathcal{F}}$ then $(P \cup \{Q\}, FT)$ has only finite computations.*

Proposition 4.12 *Every computation of a ft-clp that passes through a ft-point is either non-terminating or successful.*

Example 4.13 It is easy to check that the ft-program $(hamiltonian \cup \{ham(g, X)\}, J, <_{\mathcal{H}})$ given in Example 4.3 respects the forward ordering, hence by Theorem 4.9 it is correct. ◦

5 Computing Forward-Trackings

In this section, we investigate various techniques for constructing forward trackings. First, we assume a jump function given, and introduce automatic methods for generating a corresponding height ordering. Next, we investigate automatic techniques for constructing both the jump function and the height ordering.

Semi-Automatic Techniques

Assume a jump function J respecting the forward ordering is given. Then one can automatically derive a forward-tracking as follows. Consider the partial ordering $\lhd$ on *FTP* s.t.:

$$\forall \zeta, \eta \in FTP,\ \forall c, c'.\ (\zeta, c) \lhd (\eta, c') \Leftrightarrow \zeta <_{\mathcal{F}} \eta.$$

Then $(J, \lhd)$ is a correct forward-tracking. An interesting feature of forward-trackings having $\lhd$ as height ordering is that they do not depend on the constraint solver, and therefore they can be 'universally' implemented, in the sense that, under reasonable assumptions, there is an automatic compilation of ft-clp's into clp's. This way, every ft-clp written in a particular CLP system can be (automatically) implemented using the same system.

Automatic Techniques

We have seen how forward-trackings can be derived semi-automatically, where the user provides a J function, without bothering about the height ordering. However, this may still be a tedious task, especially for large and complex programs. In this subsection, we present a technique for deriving forward-trackings almost automatically.

The idea is that the user is only concerned with the choice of a suitable sequence of pp's, indicating those points that separate different tasks, and defined as follows.

Definition 5.1 A sequence of points $\mathbf{S}=\nu_1, \ldots, \nu_n$ is a *separation* if $\nu_1 <_{\mathcal{F}} \nu_2 <_{\mathcal{F}} \ldots <_{\mathcal{F}} \nu_n = end$. $\circ$

>From a separation $\mathbf{S}=\nu_1, \ldots, \nu_n$ one can derive automatically a jump function $J_{\mathbf{S}}$ as follows. Consider the set *FTP* of ft-points that can 'reach' (via $<_{\mathcal{F}}$) a separation point, i.e., the points ζ such that $\zeta <_{\mathcal{F}} \nu_i$ (for some $i \in [1, n-1]$). From each of these points we can jump to the 'nearest' (w.r.t. $<_{\mathcal{F}}$) separation point, by means of a function ι characterized as follows.

Proposition 5.2 *For every $\zeta \in FTP$ there is a* unique *number $\iota(\zeta) \in [1, n]$ such that $\zeta <_{\mathcal{F}} \nu_{\iota(\zeta)}$, and if $\iota(\zeta) > 1$ then $\zeta \not<_{\mathcal{F}} \nu_{\iota(\zeta)-1}$.*

Then $J_{\mathbf{S}}$ is defined by $\forall \zeta \in FTP.\ J_{\mathbf{S}}(\zeta) = \nu_{\iota(\zeta)}$. It can be proven that $J_{\mathbf{S}}$ is a jump function respecting the forward ordering. Therefore, one can utilize the semi-automatic approach introduced in the previous subsection to automatically obtain a corresponding $\lhd$ height ordering.

The specification of a separation can be integrated in the syntax of the program using a distinguished predicate name *separation*. This predicate is transparent in the sense that it doesn't affect the normal execution of the CLP (i.e., it is defined by means of the fact $separation \leftarrow .$). Its role is only to indicate the corresponding separation point. This way, the user indicates the separation points of the program by inserting the *separation* predicates in the corresponding points. A syntactical checker can be designed that tests if the so far inserted separation predicates form a separation. Since the notion of separation relies on the syntactically defined ordering $<_{\mathcal{F}}$, a facility that visually specifies what positions that are allowed for the next insertion of a separation can be incorporated into the checker.

Note that one can perform forward-tracking also in a completely automatic way (i.e. not requiring the user to do a separation), by using the trivial separation consisting only of the *end* point. This means that the final output will be the 'best' (w.r.t. the height ordering) result produced by the program during the execution (cf. Example 4.3).

Dynamic Forward-Tracking

There can be situations where we know that some progress is made, but it is hard to recover this information from the state. This is due to the fact that the height ordering has to be provided *statically*. In this section, we propose a technique for the dynamic construction of forward-tracking, i.e. the height ordering is built *dynamically* during the execution of the program.

An interface to the user can be defined, similar to the *separation* predicate of the previous section, by means of a distinguished predicate *progress*. In order to assist the inference of the height ordering, this predicate is inserted in those points where the execution is considered (by the user) to progress. Like *separation*, *progress* is transparent, so it doesn't affect the normal execution of the CLP. Using the information provided by *progress*, a dynamic height ordering is automatically defined by extending $\lhd$ as follows. Suppose a forward-tracking has to be activated. This means that an execution of the (original) program has failed. Suppose that the execution passes first through a point ζ in a state c; that the *progress* predicate is also executed; and then that the execution passes through a point η in a state c'. We add to the ordering the condition that (η, c') is greater than (ζ, c), provided this doesn't clash with the existing ordering. That is, when a *progress* predicate is encountered all the subsequently obtained pairs (program point, state) are considered to be higher than those previously obtained.

This execution model is sound since it can be proved that at any stage of the execution such dynamically built ordering is an height ordering that, together with *J*, forms a correct forward-tracking.

An automatic implementation of this dynamic approach can be provided by just inserting progress predicates before each point in FTP (the heuristic is that the more predicates are invoked, the more work is done): We call *dynamic separation via* $\lhd$ the automatic generation of a forward-tracking from a separation by applying this approach to $\lhd$. At first sight such dynamic generation of the height ordering seems to be *extremely* expensive. However, this is not the case. For instance, a special global variable can be used that memorizes only the best-so-far pairs (relatively to *progress*): these are updated only when a *progress* predicate is reached. Also, relative information on the 'progress' ordering is maintained as a counter in the constraint store (and again, it is updated only when a *progress* predicate is reached): this way, backtracking automatically does the job of 'undoing' progress information. The neat effect of such an implementation is that there is only a small (fixed-time) overhead when a *progress* predicate is executed, and then the selection of the maximal program point w.r.t. the height ordering is performed without overhead (since we have available the 'best-so-far' result, and so actually we are not maintaining the whole ordering but only the best couple to select for forward-tracking).

Refining the Height Ordering

The automatic height ordering $\lhd$ used so far can be safely replaced in all the previous techniques by a more sophisti-

cated ordering, provided that the new ordering does not conflict with $\lhd$. The height ordering $\lhd$ depends only on the points (i.e., on the *program structure*). We investigate now an extension of $\lhd$ obtained by exploiting some semantic information on the states associated with points. Somehow, one should define an ordering on states too, that provides a measure of the computational information that is represented by a constraint. To infer such information, one has to rely on the particular CLP system. Therefore, we assume that the considered CLP system allows to compute the character length of the representation of the current state (this condition is satisfied by many CLP systems). For a state c, let $|c|$ denote its character length. An height ordering $\blacktriangleleft$ extending $\lhd$ can be defined by:

$\forall \zeta, \eta \in FTP, \forall c, c' : (\zeta, c) \blacktriangleleft (\eta, c') \Leftrightarrow$
either $\zeta \lhd \eta$ or ζ and η $\lhd$-incomparable and $|c| < |c'|$.

As expected, we have that if J is a jump function respecting the forward ordering, then $(J, \blacktriangleleft)$ is a correct forward-tracking. This technique can be parameterized using different orderings on states. Here we have employed the ordering induced by the $|\cdot|$ mapping. Other predefined orderings can be used, for instance the one induced by the map counting how many different variables are present in the state. As for $\lhd$, $\blacktriangleleft$-based ft-clp's can be automatically compiled on the considered CLP system (under reasonable assumptions, and provided, as said, that the $|\cdot|$ mapping can be written in it).

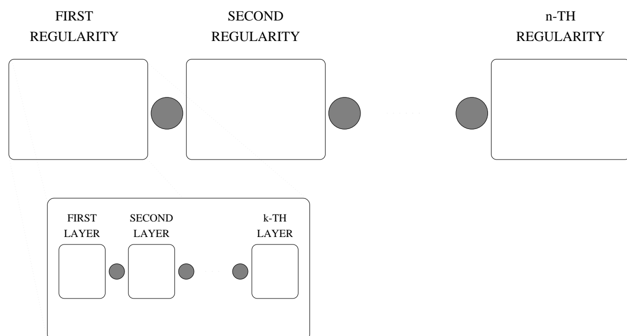
6 Constraining Neural Networks

In this section we illustrate an application of the forward-tracking technique to solve an interesting problem in the field of neural networks, namely the training of a constrained neural network.

Neural networks have been used in many applications, for example in planning, control, content-addressable memory, optimization, constraint satisfaction, and classification (see e.g. [5]). They are being promoted for their robustness, massive parallelism, and ability to learn. The training of a neural network is a major design step: Roughly, one has to find a set of weights that minimizes the neural network's error on an initial set of input-output examples called the training set. The standard method for that problem is a local gradient search method known as the back-propagation algorithm. Alternative approaches based on Genetic Algorithms (GA's) have been proposed, which apply for instance to recurrent networks, or to networks with non-differentiable error criteria (for example due to non-differentiable transfer functions, error measures that use absolute values, bonuses for small weights etc.). However, there is a well-known problem with the application of genetic algorithms to neural networks, called the *competing conventions problem*. When one chooses a representation (in this case for a neural network), then it can be the case that the same individual has more than one representation. (i.e. many structurally different networks can nevertheless represent the same functional mapping). In order to solve the competing conventions problem, a novel approach has been introduced in [8]. The idea is to develop suitable constraints (via a CLP) that avoid multiple representations of individuals, and then performing an evolutionary training of the neural network by using only chromosomes satisfying these constraints. In [7] the study of regularities of the error function has revealed that single regularities are in some cases solvable: for example when the network has a single (hidden) layer. However, in order to cope with a generic network and with generic regularities of the error function, one has to face two assembling problems: *composing layers*, where the constraints generated for each layer are composed; and *composing regularities*, where the constraints generated for each regularity are composed. So, this is an example of hard and complex problem for which a decomposition in smaller factors is known (the regularities, the layers), but such a decomposition is not immediately useful from a modularization point of view.

In order to cope with the general, intractable case one can use the forward-tracking technique, following the methodology proposed in Section 3. We consider constrained networks, e.g. neural networks with shared weights, constraints on the weights - for example domain constraints for hardware implementation - etc. Moreover, other constraints are generated ensuring that in most cases each network is specified by exactly one chromosome. Thus the problem becomes a constrained optimization problem. The optimization criterion is to optimize the error of the network (usually this error includes a sum over all the training patterns of the network), and the constraints specify the domain constraints on the weights and those constraints used for avoiding the competing conventions problem. More precisely, a ft-clp is used, for producing constraints on the weights such that each network has in most cases a unique chromosome representation. This way, the CLP system also checks for the satisfiability of these constraints. Then the obtained constraints and the optimization function are given as input to a GA system, that searches for an optimal solution that satisfies the constraints. We have used the CLP system ECL^{i}PS^e (ECRC Common Logic Programming System) and the GA system GENOCOP [10] (GEnetic algorithm for Numerical Optimization for COnstrained Problems). Since the weights are real numbers, and they are constrained, a natural choice is to employ a system that can handle constraints and where the data are encoded using real numbers, instead of the original bit-encoding. The GENOCOP system satisfies both these requirements. Following the methodology in Section 3, we developed modules solving particular regularities. These modules have then been composed via the separation structure. For each layer of the network we built up a constraint generator solving the regularity, and

then merged these modules in a separation structure. The situation is illustrated below:



This approach allows an easy and flexible modular control over the constraint generations: if the user is aware of some new specific regularity due to the particular activation function chosen, or due to symmetries in the data, etc., (s)he can safely add a submodule performing this specific task. This is also the case if (s)he knows that some cases will *not* occur, in which case some submodules which are not needed and that will maybe restrict too much the search space can safely be removed.

Also, the user is free to experiment with every combination of modules (the first choice could be for instance to give priority in the chain to modules according to the importance of the corresponding regularity, i.e. how much it affects the search space; then, some other combinations could also be tested).

To this respect, empirical tests showed that forward-tracking behaves very well, in the sense that the performance remains practically unaltered when different orderings are chosen for the modules in the chain (i.e. using different orderings leads to minimal differences). We tested the system using both randomly generated neural networks and networks with regular structures like unconstrained feed-forward ones, on various problems. Moreover, we also experimented with several non-differentiable error criteria.

The results show that solving the regularities problem using the constraints generated by our FT-CLP system *considerably* improves the convergence. In all the tests the automatic approach of separations based on $\lhd$ was used. We also tried $\blacktriangleleft$, and the dynamic separations via $\lhd$ and $\blacktriangleleft$. When refining the forward-tracking using these techniques, small improvements were achieved. We were able to design some particular constrained networks where these techniques lead to significant improvements: this, in our opinion, shows that (at least in the treatment of this problem) the simpler approach of separations based on $\lhd$ suffices for most of the cases, where the other more sophisticated techniques should be used if one wants to reach the highest performance in all the cases.

7 Conclusion

In this paper a novel search technique called forward-tracking, has been introduced, that allows to obtain solutions to programs whose original executions fail. Forward-tracking supports a modular approach of programming, where in order to formalize a problem, its various parts are specified separately, and the relations and priorities among these parts are described by means of forward-trackings. As a consequence, forward-tracking is applicable to a large class of problems, like planning, scheduling, decision support, negotiation: instances of these problems that are over-specified are admissible, and in such cases a satisfactory relaxation of the conflicting parts has to be provided. From a software engineering point of view, forward-tracking supports software *reusability* (cf. [11]). This was illustrated in the application we have considered, where programs developed to solve a particular case (one regularity, one layer) have been re-used with almost null modifications, obtaining a system able to cope with the general case of neural networks.

Acknowledgement The work of Elena Marchiori was partially supported by NWO, the Dutch Organization for Scientific Research, under grant 612-32-001.

References

[1] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5:223–270, 1992.

[2] M.S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*. Morgan Kaufmann, 1987.

[3] E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 65:363–376, 1992.

[4] E.C. Freuder and P.D. Hubbe. Extracting constraint satisfaction subproblems. In *IJCAI'95*, pp. 548-555.

[5] S. Haykin. *Neural Networks, A Comprehensive Foundation*. Macmillan, 1994.

[6] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *J. of Logic Programming*, 19,20:503–581, 1994.

[7] J.N. Kok, E. Marchiori, M. Marchiori, and C. Rossi. Constraining of weights using regularities. In *European Symp. on Artificial Neural Networks (ESANN'96)*, pp. 267-272.

[8] J.N. Kok, E. Marchiori, M. Marchiori, and C. Rossi. Evolutionary training of clp-constrained neural networks. In *Int. Conf. on the Practical Application of Constraint Technology (PACT'96)*, pp. 129-142.

[9] F. Menezes and P. Barahona. Defeasible constraint solving. In *CP'95 post-workshop on over-constrained systems*, 1995.

[10] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1994.

[11] I. Sommerville. *Software Engineering*. Addison-Wesley, 1992.