A Dataflow Semantics for Constraint Logic Programs

Livio Colussi[1], Elena Marchiori[2], Massimo Marchiori[1]

[1] Dept. of Pure and Applied Mathematics, Via Belzoni 7, 35131 Padova, Italy
{colussi,max}@euler.math.unipd.it
[2] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
elena@cwi.nl

Abstract. This paper introduces an alternative operational model for constraint logic programs. First, a transition system is introduced, which is used to define a trace semantics $\mathcal{T}$. Next, an equivalent fixpoint semantics $\mathcal{F}$ is defined: a dataflow graph is assigned to a program, and a consequence operator on tuples of sets of constraints is given whose least fixpoint determines one set of constraints for each node of the dataflow graph. To prove that $\mathcal{F}$ and $\mathcal{T}$ are equivalent, an intermediate semantics $\mathcal{O}$ is used, which propagates a given set of constraints through the paths of the dataflow graph. Possible applications of $\mathcal{F}$ (and $\mathcal{O}$) are discussed: in particular, its incrementality is used to define a parallel execution model for clp's based on asynchronous processors assigned to the nodes of the program graph. Moreover, $\mathcal{O}$ is used to formalize the Intermittent Assertion Method of Burstall [Bur74] for constraint logic programs.

1 Introduction

In this paper a dataflow semantics for constraint logic programs (clp's for short) is introduced. The importance of dataflow semantics is well-known: they specify the 'functionality' of the program; and hence can be used to transform a program into a functional expression, preserving semantics equality. Or to reason about run-time properties of a program depending on the form of the arguments of program atoms before and after their call. From the practical point of view, dataflow semantics support efficient parallel implementations based on networks, where the nondeterminism of programs is exploited.

In this paper we consider for simplicity 'ideal' CLP systems with Prolog selection rule (cf. [JM94]). The extension of the results to more general systems is given in the last section of the paper. A clp $\mathcal{P}$ is a set of clauses together with a goal-clause. First, a transition system is introduced the configurations of which are pairs consisting of an annotated sequence of atoms and a constraint. Then an operational semantics $\mathcal{T}$ is defined, which assigns to a program $\mathcal{P}$ (with goal-clause G) and a set ϕ of constraints, the set of all partial transition traces starting in (G, α), with α in ϕ.

Next, a fixpoint semantics $\mathcal{F}$, equivalent to $\mathcal{T}$, is introduced. Its definition is based, for a program $\mathcal{P}$, on a *dataflow graph* $dg(\mathcal{P})$: this graph has program points as nodes. The arcs of $dg(\mathcal{P})$ are abstractions of the transition rules where configurations are replaced by program points. This graph is used to define the fixpoint semantics $\mathcal{F}$ of $\mathcal{P}$ w.r.t. a set of constraints: a consequence operator on tuples of sets of constraints is given, based on a predicate transformer for constraints, and the least fixpoint of this operator determines one set of constraints for each node of $dg(\mathcal{P})$. We prove that $\mathcal{F}$ and $\mathcal{T}$ are equivalent, by using a top-down semantics $\mathcal{O}$, which propagates a given set of constraints through the paths of $dg(\mathcal{P})$, by means of the above mentioned predicate transformer.

This is the first time that a fixpoint semantics for a clp viewed as set of program points is given. Related work for logic programs, includes e.g. the models of Mellish [Mel87] and Nilsson [Nil90]. However, they both give a fixpoint semantics in which the operational semantics is contained as a proper subset, while here we give an exact description of $\mathcal{T}$.

The fixpoint semantics $\mathcal{F}$ (and $\mathcal{O}$) is shown to have a number of interesting applications. In particular, the incrementality of $\mathcal{F}$ is used to define an or-parallel execution model for clp's based on asynchronous processors assigned to the nodes of the program graph. Moreover, the intermediate semantics $\mathcal{O}$ is used to formalize the Intermittent Assertion Method of Burstall [Bur74] for clp's. This latter application solves at the same time a problem addressed by the Cousots' in [CC93] on how to formalize the Intermittent Assertion Method for clp's.

The rest of the paper is organized as follows. The next section contains the terminology and the concepts used in the sequel. In Section 3 the operational semantics is given. In Section 4 the notion of dataflow graph is introduced, which is used in Section 5 to define the dataflow semantics $\mathcal{F}$. The equivalence of the two semantics is established in Section 6, where the intermediate semantics is introduced. In Section 7 properties of $\mathcal{F}$ are given. In Section 8 some possible applications are investigated. Finally, in Section 9 the results of this paper are discussed.

2 Preliminaries

Let Var be an (enumerable) set of variables, with elements denoted by x, y, z, u, v, w. We shall consider the set $VAR = Var \cup Var^0 \cup \ldots \cup Var^k \cup \ldots$, where $Var^k = \{x^k \mid x \in Var\}$ contains the so-called *indexed variables* (`i`-variables for short) of *index* k. These special variables will be used to describe the standardization apart process, which distinguishes copies of a clause variable which are produced at different calls of that clause. Thus x^k and x^j will represent the same clause variable at two different calls. This technique is known as 'structure-sharing', because x^k and x^j share the same structure, i.e. x. For an index k and a syntactic object E, E^k denotes the object obtained from E by replacing every variable x with the `i`-variable x^k. We denote by $Term(VAR)$ (resp. $Term(Var)$) the set of terms built on VAR (resp. Var), with elements denoted by r, s, t.

A sequence $E_1, \ldots, E_k$ of syntactic objects is denoted by $\overline{E}$ or $\langle E_1, \ldots, E_k \rangle$, $(s_1 = t_1 \wedge \ldots \wedge s_k = t_k)$ is abbreviated by $\overline{s} = \overline{t}$, and $\tilde{x}$ represents a sequence of distinct variables.

Constraint Logic Programs

The reader is referred to [JM94] for a detailed introduction to Constraint Logic Programming. Here we present only those concepts and notation that we shall need in the sequel.

A constraint c is a (first-order) formula on $Term(VAR)$ built from primitive constraints. We shall use the symbol $\mathcal{D}$ both for the domain and the set of its elements. We write $\mathcal{D} \models c$ to denote that c is valid in all the models of $\mathcal{D}$.

A *constraint logic program* $\mathcal{P}$, simply called program or clp, is a (finite) set of clauses $H \leftarrow A_1, \ldots, A_k$ (denoted by C, D), together with one goal-clause $\leftarrow B_1, \ldots, B_m$ (denoted by G), where H and the A_i's and B_i's are atoms built on $Term(Var)$ (primitive constraints are considered to be atoms as well) and H is not a constraint. Atoms which are not constraints are also denoted by $p(\overline{s})$, and $pred(p(\overline{s}))$ denotes p; for a clause C, $pred(C)$ denotes the predicate symbol of its head. A clause whose body either is empty or contains only constraints is called *unitary*.

As in the standard operational model states are consistent constraints, i.e. $States \stackrel{\text{def}}{=} \{c \in \mathcal{D} \mid c \text{ consistent}\}$. States are denoted by c or α. We use the two following operators on states:

$$push, pop : States \rightarrow States,$$

where $push(\alpha)$ is obtained from α by increasing the index of all its `i`-variables by 1, and $pop(\alpha)$ is obtained from α by first replacing every `i`-variable of index 0 with a new fresh variable, and then by decreasing the index of all the other `i`-variables by 1. For instance, suppose that α is equal to $(x^1 = f(z^0) \wedge y^0 = g(x^2))$. Then $push(\alpha)$ is equal to $(x^2 = f(z^1) \wedge y^1 = g(x^3))$ and $pop(\alpha)$ to $(x^0 = f(u) \wedge v = g(x^1))$, where u and v are new fresh variables.

3 Operational Semantics

In Table 1 the operational behaviour of a clp by means of a transition system (TS) is given.

In a pair $(\overline{A}, \alpha)$, α is a state, and $\overline{A}$ is a sequence of atoms and possibly of tokens of the form *pop*, whose use is explained below.

The rules of TS describe the standard operational behaviour of a clp (cf. e.g. [JM94]), but for the fact that we fix a suitable standardization apart mechanism: In the standard operational semantics of (C)LP, every time a clause is called it is renamed apart, generally using indexed variables. Here if a clause is called then *push* is first applied to the state, and if it is released then *pop* is applied to the state. To mark the place at which this should happen the symbol *pop* is used. Rule **R** describes a resolution step. Note that, the way the operators *push* and *pop* are used guarantees that every time an atom is called, its variables can be indexed with index equal to 0. Then, in rule **R** the tuple of terms $push(\overline{s}^0)(= \overline{s}^1)$ is considered, because a *push* is applied to the state. Rule **S** describes the situation where an atom has concluded with success its computation, i.e. when the control

R	$(\langle p(\overline{s})\rangle \cdot \overline{A},\ \alpha\,) \longrightarrow (\overline{B} \cdot \langle pop\rangle \cdot \overline{A},\ push(\alpha) \wedge \overline{s}^1 = \overline{t}^0\,)$, if $C = p(\overline{t}) \leftarrow \overline{B}$ is in $\mathcal{P}$ and $push(\alpha) \wedge \overline{s}^1 = \overline{t}^0$ is consistent
S	$(\langle\, pop\rangle \cdot \overline{A},\ \alpha\,) \longrightarrow (\,\overline{A},\ pop(\alpha)\,)$
C	$(\langle\, d\,\rangle \cdot \overline{A},\ \alpha\,) \longrightarrow (\,\overline{A},\ \alpha \wedge d^0\,)$, if d is a constraint and $\alpha \wedge d^0$ is consistent

Table 1. Transition rules for CLP.

reaches a *pop*. In this case, the operator *pop* is applied to the state. Finally, rule **C** describes the execution of a constraint.

This formalization will lead to an elegant definition of the dataflow semantics. Note that we do not describe explicitly failure, because it is not relevant for our dataflow model.

To refer unambiguously to clause variables, the following non-restrictive assumption is used.

Assumption 3.1 Different clauses of a program have disjoint sets of variables.

We write $(\overline{A}, \alpha) \rightarrow (\overline{B}, \beta)$ to denote a generic transition using the rules of Table 1. We call *computation*, denoted by τ, any sequence $\langle conf_1, \ldots, conf_k, \ldots\rangle$ of configurations s.t. for $k \geq 1$ we have that $conf_k \rightarrow conf_{k+1}$. We consider an operational semantics $\mathcal{T}(\mathcal{P}, \phi)$ for a program $\mathcal{P}$ w.r.t. a set ϕ of states, called *precondition*. This semantics describes all the computations starting in $(G,\ \alpha)$ (recall that G denotes the goal-clause of $\mathcal{P}$) with α in ϕ. It is defined as follows. We use $\cdot$ for the concatenation of sequences.

Definition 3.2 (partial trace semantics) $\mathcal{T}(\mathcal{P}, \phi)$ is the least set T s.t. $\langle (G,\ \alpha)\rangle$ is in T, for every $\alpha \in \phi$, and if $\tau = \tau' \cdot \langle (\overline{A},\ \alpha)\rangle$ is in T and $(\overline{A},\ \alpha) \rightarrow (\overline{B},\ \beta)$, then $\tau \cdot \langle (\overline{B},\ \beta)\rangle$ is in T. □

Observe that this is a very concrete semantics: the reason is that it is not meant for the study of program equivalence, but for the study of run-time properties of clp's, and for the definition of models for parallel implementations. These applications are discussed in Section 8.

4 A Dataflow Graph for clp's

To define a dataflow semantics equivalent to $\mathcal{T}(\mathcal{P}, \phi)$, we start by introducing a dataflow graph associated with a clp, whose nodes are the program points, and whose arcs describe in an abstract way the transition rules of Table 1.

In logic programming, program points are (often implicitly) used to describe the operational observables considered. Similar e.g. to [Nil90], we view a program clause $C : H \leftarrow A_1, \ldots, A_k$ as a sequence consisting alternatingly of (labels l of) *program points* (pp's for short) and atoms,

$$H \leftarrow {}_{l_0} A_1 {}_{l_1} \ \cdots \ {}_{l_{k-1}} A_k {}_{l_k}.$$

The labels l_0 and l_k indicate the *entry point* and the *exit point* of C, denoted by $entry(C)$ and $exit(C)$, respectively. For $i \in [1, k], l_{i-1}$ and l_i are called the *calling point* and *success point* of A_i, denoted by $call(A_i)$ and $success(A_i)$, respectively. Notice that $l_0 = entry(C) = call(A_1)$ and $l_k = exit(C) = success(A_k)$. In the sequel $atom(l)$ denotes the atom of the program whose calling point is equal to l. Moreover, for notational convenience the following non-restrictive assumptions are used.

Assumption 4.1 $l_0, \ldots, l_k$ are natural numbers ordered progressively; distinct clauses of a program are decorated with different pp's; the pp's form an initial segment, say $\{1, 2, \ldots, n\}$ of the natural numbers; and 1 denotes the leftmost pp of the goal-clause, called the *entry point of the program*. Finally, to refer unambiguously to program atom occurrences, all atoms occurring in a program are supposed to be distinct.

The following $CLP(\mathcal{R})$ ([JMSY92]) program *Prod* is explicitly labelled with its pp's.

G: $\leftarrow$ ${}_1$ `prod(u,v)` ${}_2$
$C1$: `prod([x|y],z)` $\leftarrow$ ${}_3$ `z=x*w` ${}_4$ `prod(y,w)` ${}_5$
$C2$: `prod([ ],1)` $\leftarrow$ ${}_6$

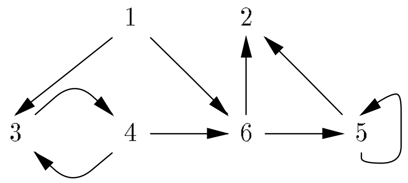
In the sequel, $\mathcal{P}$ denotes a program and $\{1, \ldots, n\}$ the set of its pp's. Program points are used to define the notion of dataflow graph.

Definition 4.2 (dataflow graph) The *dataflow graph* $dg(\mathcal{P})$ *of* $\mathcal{P}$ is the pair $(Nodes, Arcs)$ s.t. $Nodes = \{1, \ldots, n\}$ and $Arcs$ is the subset of $Nodes \times Nodes$ s.t. (i, j) is in $Arcs$ iff it satisfies one of the following conditions:

- i is $call(A)$ and j is $entry(C)$, where A is not a constraint, and $pred(A)$ and $pred(C)$ are equal;
- i is $exit(C)$ and j is $success(A)$, where $pred(A)$ and $pred(C)$ are equal;
- i is $call(A)$ and j is $success(A)$, where A is a constraint.

An element (i, j) of $Arcs$ is called *(directed) arc from* i *to* j. □

Arcs of $dg(\mathcal{P})$ are graphical abstractions of the transition rules of Table 1. Rule **R** is abstracted as an arc from the calling point of an atom to the entry point of a clause. Rule **S** is abstracted as an arc from the exit point of a clause to a success point of an atom. Finally, rule **C** is abstracted as an arc from the calling point of a constraint to its success point. Below the dataflow graph $dg(Prod)$ of $Prod$ is pictured.



Remark 4.3 Our notion of dataflow graph differs from other graphical representations of (c)lp's, as for instance the *predicate dependency graph* [Kun87] or the *U-graph* [WS94], mainly because of the presence in $dg(\mathcal{P})$ of those arcs from exit points of clauses to success points of atoms, such as the arc from 5 to 2 in $dg(Prod)$. These arcs are crucial to obtain an exact fixpoint description of the operational semantics. For instance, in $dg(Prod)$ there is one arc from 5 to 5 and one from 5 to 2, one from 6 to 2 and one from 6 to 5.

Remark 4.4 One can refine this definition by using also semantic information, i.e. by pruning the arcs stemming from the first two conditions if $\mathcal{D} \models \neg(\overline{s} = \overline{t})$, i.e. if $p(\overline{s})$ and $p(\overline{t})$ do not 'unify', where $p(\overline{s})$ is A and $p(\overline{t})$ is (a variant of) the head of C.

A *path of* $\mathcal{P}$ is a non-empty sequence of pp's forming a (directed) path in $dg(\mathcal{P})$. Paths are denoted by π, and concatenation of paths by $\cdot$. Moreover, $path(i, j)$ denotes the set of all the paths from i to j, and $path(i)$ the set of all the paths from 1 to i.

5 Dataflow Semantics

In this section a dataflow semantics $\mathcal{F}$ for clp's is given, w.r.t. a given 'precondition' ϕ which is associated with the entry point 1 of the program. This semantics determines for every node l of $dg(\mathcal{P})$ a suitable set ϕ_l of states. In Section 6 it will be shown that $\mathcal{F}$ is equivalent to $\mathcal{T}$, i.e. that ϕ_l is the set of the final states of all partial derivations, with initial state in ϕ, ending in l. This semantics describes the run-time behaviour of a clp, i.e. the form of the body atoms of the program (goal-)clauses at the moment when they are called and after their execution. The importance of this information is well-known: it can be used for instance to determine for which class of goals a program terminates and for which class of goals the computation is sufficiently efficient. It will be shown in Section 7 that $\mathcal{F}$ enjoys two relevant properties: it is incremental and and-compositional. Incrementality allows us to compute the semantics of the union of two clp's $\mathcal{P}$ and $\mathcal{P}'$, by computing first the semantics of one of them, say $\mathcal{F}(\mathcal{P})$ of $\mathcal{P}$, and then by using $\mathcal{F}(\mathcal{P})$ to determine the semantics of their union $\mathcal{P} \cup \mathcal{P}'$. Also, from the practical point of view, the incrementality of $\mathcal{F}$ allows us to define parallel execution models of clp's based on asynchronous processors, as explained in

Section 8. And-compositionality allows us to compute the semantics of a goal $\leftarrow \overline{A}, \overline{B}$ from the semantics of $\leftarrow \overline{A}$ and of $\leftarrow \overline{B}$.

To define $\mathcal{F}$, first constraints are described as predicate transformers, by lifting the transition rules to sets of states. Thus one can view a constraint c as a map $sp.c : 2^{States} \rightarrow 2^{States}$ (sp stands for strongest postcondition) defined as follows.

Definition 5.1 For a constraint c and for a set ϕ in 2^{States},

$$sp.c.\phi = \ \{\alpha \wedge c \in States \ \mid \ \alpha \in \phi\}. \qquad \square$$

This definition corresponds to the rule **C** of TS. Observe that it also describes the rule **R**, by taking the constraint c to be equal to $(\overline{s}^1 = \overline{t}^0)$.

Sets of states are denoted by ϕ, ψ, where *false* stands for $\emptyset$, and $\neg\phi$ for $States \setminus \phi$. The set

$$free(x) = \{\alpha \mid \mathcal{D} \models \alpha \rightarrow \forall x.\alpha\}$$

of states will be used in the sequel, describing those states where x is a free variable. The intuition is that x is free in a state if it can be bound to any value without affecting that state. For instance, $y = z$ is in $free(x)$, because x does not occur in the formula. Also $y = z \wedge x = x$ is in $free(x)$, because $\mathcal{D} \models (y = z \wedge x = x) \rightarrow \forall x\,(y = z \wedge x = x)$. The definitions of *pop* and *push* are extended in the natural way to sets of states, where $push(\phi)$ is equal to $\{push(\alpha) \mid \alpha \in \phi\}$. Analogously for $pop(\phi)$. It is convenient to make the following assumptions on non-unitary (goal-)clauses.

Assumption 5.2 The body of every non-unitary clause does not contain two atoms with equal predicate symbol; and at least one argument of its head is a variable.

Notice that every program can be transformed into one satisfying Assumption 5.2. Although the transformation can modify the semantics of the original program (the set of pp's changes and new predicates could be introduced), it is easy to define a syntactic transformation that allows us to recover the semantics of the original program.

These assumptions are used to simplify the definition of the dataflow semantics. Because of the second one, one can fix a variable-argument of the head of a non-unitary clause C, that we call *the characteristic variable of* C, denoted by x_C. Also, a new fresh variable x_G is associated with the goal-clause G, called *the characteristic variable of* G. These variables play a crucial role in the following definitions, to be explained below.

We can introduce now, for a program $\mathcal{P}$ with set $\{1, \ldots, n\}$ of pp's, the immediate consequence operator Ψ on n-tuples of sets of states, defined w.r.t. a given set ϕ of states associated with the entry point of $\mathcal{P}$. For a node j of $dg(\mathcal{P})$, let $input(j)$ denote the set of the nodes i s.t. (i, j) is an arc of $dg(\mathcal{P})$. Because every pp is either an entry point of a clause, or a success point of an atom, it is enough to distinguish these two cases in the following definition of Ψ. In the sequel, Ψ_k denotes the k-th projection of Ψ.

Definition 5.3 For a program $\mathcal{P}$ with set $\{1,\ldots,n\}$ of pp's, and for a given set ϕ of states (the precondition), the operator $\Psi : (2^{States})^n \to (2^{States})^n$ is defined as follows. For $\overline{\psi} = (\psi_1,\ldots,\psi_n)$:

- $\Psi_1(\overline{\psi}) = \phi$;
- for $k \in [2,n]$:
  1. if k is $entry(C)$ then

$$\Psi_k(\overline{\psi}) = \bigcup_{j \in input(k)} sp.\,(\overline{s}^1 = \overline{t}^0).\,push(\psi_j),$$

     where $p(\overline{t})$ is the head of C and $p(\overline{s})$ is $atom(j)$;
  2. if k is $success(A)$ and A is not a constraint then

$$\Psi_k(\overline{\psi}) = \bigcup_{j \in input(k)} pop(\psi_j) \cap \neg free(x_C^0),$$

     where C is the clause containing A;
  3. if k is $success(A)$ and A is a constraint then

$$\Psi_k(\overline{\psi}) = sp.\,A^0.\,\psi_{k-1}.$$

□

Because $sp.\ c.\ \bigcup_i \psi_i = \bigcup_i (sp.\ c.\ \psi_i)$ it follows that Ψ is a continuous operator on the complete lattice $((2^{States})^n, \subseteq)$, where $\subseteq$ denotes componentwise inclusion. Hence by the Knaster-Tarski theorem it has a least fixpoint $\mu(\Psi) = \bigcup_{k=0}^{\omega} \Psi^k(\bot)$, where $\bot$ stands for the least element $(\emptyset,\ldots,\emptyset)$ of $(2^{States})^n$.

Definition 5.4 (dataflow semantics) Let ϕ be s.t. $\phi \subseteq \neg free(x_G^0)$, and $\phi \subseteq free(x_C^0)$ for every non-goal, non-unitary clause C. Then the *semantics* $\mathcal{F}(\mathcal{P},\phi)$ *of* $\mathcal{P}$ *with respect to* ϕ is the least fixpoint $\mu(\Psi)$. □

Let us comment on the above definitions. The operational intuition behind the definition of Ψ can be explained using the transition system of Table 1: let $\overline{A}$ be a generic sequence of atoms and/or *pop* tokens. Then in case 1. $entry(C)$ 'receives' those states obtained by applying rule **R** to $(\langle atom(j)\rangle \cdot \overline{A}, \alpha)$, for every α in ψ_j, and for every j s.t. the arc $(j, entry(C))$ is in the dataflow graph. In case 2. $success(A)$ 'receives' those states obtained by applying the rule **S** to $(\langle pop\rangle \cdot \overline{A}, \alpha)$, for every α in ψ_j, for every j s.t. the arc $(j, success(A))$ is in the dataflow graph. Finally, in case 3. $success(A)$ 'receives' those states obtained by applying the transition rule **C** to $(\langle A\rangle \cdot \overline{A}, \alpha)$, for every α in $\psi_{call(A)}$. In Definition 5.4 the operator Ψ is iterated ω times starting from $\bot$.

The characteristic variables of the program are used in case 2. of Definition 5.3, where the result is intersected with $\neg free(x_C^0)$, and in the two conditions in Definition 5.4. They are of crucial importance for obtaining a dataflow semantics which is equivalent to $\mathcal{T}$. In fact, they are used to rule out all those paths which are not semantic, i.e. which do not describe partial traces.

Informally, whenever a state is propagated through a semantic path the characteristic variable x_C^0 of a non-unitary clause is initially free (by assumption). Then, the index of x_C is increased and decreased by means of the applications of the *push* and *pop* operators. When C is called, then x_C^0 is bound (because by assumption it occurs in the head of C), hence x_C^0 is not free. From that moment on its index will be increased and decreased and it will become 0 *only* if the success point of an atom of the body of C is reached. Concerning the characteristic variable x_G^0 of the goal, it is initially not free (by assumption). Then, its index is increased and decreased by means of the applications of the *push* and *pop* operators and it will become 0 *only* if the success point of an atom of G is reached. In that case, for each other clause C, x_C^0 is free, because either C was never called, or x_C^0 has been replaced with a fresh variable by an application of *pop*. Observe that Assumptions 3.1 and 5.2, and those of Definition 5.4 are needed.

Example 5.5 We illustrate how $\mathcal{F}$ is determined by computing $\mathcal{F}(Prod, \phi)$, where ϕ is the set $\{(u^0 = [\,] \wedge x_G^0 = 1),\ (u^0 = [r] \wedge x_G^0 = 1)\}$ (with r a variable). We choose x as characteristic variable of $C1$ and the fresh variable x_G as the one of G. For every $k \geq 0$, we have that Ψ_1^k is ϕ. Then in the following steps, Ψ_1^k is not mentioned. Moreover, the other Ψ_i^k's which are omitted are assumed to be equal to $\emptyset$. Finally, the abbreviation $s_1 = s_2 = \ldots = s_m$ stands for $s_1 = s_2 \wedge \ldots \wedge s_{m-1} = s_m$, and the brackets for singleton sets are omitted.

- Ψ_3^1 is $(u^1 = [r] \wedge x_G^1 = 1 \wedge y^0 = [\,] \wedge v^1 = z^0)$;
  Ψ_6^1 is α, where α is $u^1 = [\,] \wedge x_G^1 = 1 \wedge v^1 = 1$.
- Ψ_2^2 is $pop(\alpha)$;
  Ψ_4^2 is $u^1 = [r] = [x^0] \wedge x_G^1 = 1 \wedge y^0 = [\,] \wedge v^1 = z^0 \wedge z^0 = x^0 * w^0$;
  Ψ_i^2 is Ψ_i^1 , for $i = 3, 6$. Observe that while $pop(\alpha)$ is added to Ψ_2^2, it is not added to Ψ_5^2 (which remains empty), because x^0 does not occur in $pop(\alpha)$, hence $pop(\alpha)$ intersected with $\neg free(x^0)$ yields the empty set.
- Ψ_i^3 is Ψ_i^2 , for $i = 2, 3, 4$;
  Ψ_6^3 is $\{\alpha,\ \beta\,\}$, where β is $u^2 = [r] = [x^1] \wedge x_G^2 = 1 \wedge y^1 = [\,] \wedge v^2 = z^1 = x^1$.
- Ψ_i^4 is equal to Ψ_i^3, for $i = 2, 3, 4, 6$;
  Ψ_5^4 is $pop(\beta)$. Observe that here $pop(\beta)$ is added to Ψ_5^4 but not to Ψ_2^4, because x_G^0 does not occur in $pop(\beta)$.
- Ψ_i^5 is Ψ_i^4 , for $i = 3, \ldots, 6$;
- Ψ_2^5 is $\{\, pop(pop(\beta)),\ pop(\alpha)\,\}$. Observe that here $pop(pop(\beta))$ is added to Ψ_2^5, but not to Ψ_5^5, because x^0 does not occur in $pop(pop(\beta))$.
- Ψ^6 is Ψ^5.

□

Remark 5.6 In order to illustrate how to compute $\mathcal{F}$, we have assumed to deal with an ideal system. However, in $CLP(\mathcal{R})$ the constraint $z = x * w$ is delayed until it becomes linear (cf. [JMSY92]). In Section 9 we shall discuss how to modify the dataflow semantics to deal with such systems, and to handle this example.

6 Equivalence of $\mathcal{T}$ and $\mathcal{F}$

To prove the equivalence of $\mathcal{T}$ and $\mathcal{F}$, an intermediate semantics $\mathcal{O}$ is introduced, which propagates sets of states through the paths of $dg(\mathcal{P})$ by means of the predicate transformer *sp*. This semantics is not only useful to prove the above mentioned equivalence. It also allows us to define the Burstall Intermittent Assertion Method for clp's, as will be described in Section 8.

Definition 6.1 Consider a path π in $dg(\mathcal{P})$. The *path strongest postcondition* $psp.\ \pi.\ \phi$ *of* π *w.r.t.* ϕ is inductively defined as follows:

- If π is of the form $\langle l \rangle$ then
$$psp.\ \pi.\ \phi = \phi.$$
- Otherwise, if π is of the form $\pi' \cdot \langle l_k \rangle$, where π' is $\langle l_1, \ldots, l_{k-1} \rangle$ and $k \geq 2$, then:
  1. if l_k is *entry*(C) and l_{k-1} is *call*(A), where A is an atom, say $p(\overline{s})$, then
  $$psp.\pi.\phi = sp.\ (\overline{s}^1 = \overline{t}^0).\ push(psp.\pi'.\phi),$$
  where $p(\overline{t})$ is the head of C;
  2. if l_k is *success*(A) and l_{k-1} is *exit*(D), where A is not a constraint and D is a clause, then
  $$psp.\pi.\phi = pop(psp.\pi'.\phi) \cap \neg free(x_C^0),$$
  where C is the clause containing A;
  3. if l_k is *success*(A), where A is a constraint, then
  $$psp.\pi.\phi = sp.\ A^0.\ (psp.\pi'.\phi).$$

□

Definition 6.2 Let $\mathcal{P}$ be a program with set $\{1, \ldots, n\}$ of pp's, and let ϕ be s.t. $\phi \subseteq \neg free(x_G^0)$, and $\phi \subseteq free(x_C^0)$ for every non-goal, non-unitary clause C. The *semantics* $\mathcal{O}(\mathcal{P}, \phi)$ *of* $\mathcal{P}$ *w.r.t.* ϕ is the n-tuple:
$$(\phi,\ \cup_{\pi \in path(2)} psp.\pi.\phi,\ \ldots,\ \cup_{\pi \in path(n)} psp.\pi.\phi).$$
□

Recall that $path(i)$ denotes the set of all the paths of $dg(\mathcal{P})$ from 1 to i. The operational intuition behind the definition of $psp.\pi.\phi$ can be illustrated using the transition rules of Table 1: case 1. corresponds to the application of rule **R**, case 2. to the application of rule **S** and case 3. to the application of rule **C**. Then the semantics $\mathcal{O}(\mathcal{P}, \phi)$ associates with every node of $dg(\mathcal{P})$ the union, over all the paths π from the entry point of $\mathcal{P}$ to that node, of the strongest postconditions of the π's w.r.t. ϕ. The characteristic variables have here the same function as in the definition of $\mathcal{F}$. The following example illustrates the crucial role of these variables to discriminate those paths which are not semantic.

at pp	x^0_G	x^0_{C1}	x^0_{C2}
1	not free	free	free
3	free	not free	free
4	free	not free	free
6	free	free	not free
2	free	not free	free

Table 2. Characteristic variables of index 0 through π

Example 6.3 Consider again the program *Prod*. Let π be $\langle 1, 3, 4, 6, 2\rangle$ and let α be $x^0_G = 0$, where 0 is a constant. The behaviour, with respect to freeness, of the characteristic variables of index 0 during the propagation of α through π is described in Table 2. Observe that, at program point 2, the `i`-variable x^0_G is free. Then, Definition 6.1 is not applicable. In fact, π does not describe a computation, because it 'jumps' to the success point of the goal before finishing the execution of the called clause $C1$. To describe a computation, π has to be modified by replacing 2 with 5. In fact, x^0_{C1} is not free at pp 5.

□

We now show that $\mathcal{T}$ and $\mathcal{F}$ are equivalent, by proving that $\mathcal{T}$ and $\mathcal{O}$ are isomorphic ($\mathcal{T} \sim \mathcal{O}$), and that $\mathcal{F}$ and $\mathcal{O}$ are equal. To define the isomorphism between $\mathcal{T}$ and $\mathcal{O}$, we use a relation *Rel* relating partial traces and paths.

We write *conf*, possibly subscripted, to denote a configuration $(\overline{A}, \alpha)$ used in the rules of TS. The relation *Rel* is defined inductively on the number of elements of a partial trace as follows.

The base case is $\langle(\langle p(\overline{s})\rangle \cdot \overline{A}, \alpha)\rangle \; Rel \; \langle call(p(\overline{s}))\rangle$, and the induction case is as follows. Suppose that $\tau' \cdot \langle conf_1\rangle \; Rel \; \pi$ and that τ is $\tau' \cdot \langle conf_1, conf_2\rangle$ (by definition this implies $conf_1 \rightarrow conf_2$). Then:

- $\tau \; Rel \; \pi \cdot \langle entry(C)\rangle$,
  if $conf_1$ is $(\langle p(\overline{s})\rangle \cdot \overline{A}, \alpha)$ and C is the selected clause;
- $\tau \; Rel \; \pi \cdot \langle success(A)\rangle$,
  if $conf_1$ is $(\langle pop\rangle \cdot \overline{A}, \alpha)$, and if the atom A satisfying the following condition exists: Let π be of the form $\langle l_1, \ldots, l_k\rangle$. Then for some $i \in [1, k]$, $call(A)$ is equal to l_i, and for every B in $\mathcal{P}$, the sets $I_{call(B)}$ and $I_{success(B)}$ have the same cardinality, where $I_\star$ is the set $\{j \mid i < j \leq k, l_j = \star\}$, for $\star$ in $\{call(B), success(B)\}$.
- $\tau \; Rel \; \pi \cdot \langle success(d)\rangle$,
  if $conf_1 = (\langle d\rangle \cdot \overline{A}, \alpha)$.

Informally, the isomorphism $\sim$ first extracts from an element τ of $\mathcal{T}$ of the form $\tau' \cdot \langle(\overline{A}, \beta)\rangle$ its final state β, and maps it into the l-th component ϕ_l of $\mathcal{O}$, where l is the last node of a path π s.t. $\tau \; Rel \; \pi$ holds. Vice versa, $\sim$ maps a β

in ϕ_l, with $l \in \{1, \ldots, n\}$, into the partial trace τ of $\mathcal{T}$ of the form $\langle (G, \alpha)\rangle \cdot \tau'$, s.t. for some π in $path(l)$, we have that τ *Rel* π, and $\{\beta\}$ is $psp.\pi.\{\alpha\}$.

Theorem 6.4 **($\mathcal{T} \sim \mathcal{O}$)** *Let ϕ be s.t. $\phi \subseteq \neg free(x^0_G)$, and $\phi \subseteq free(x^0_C)$ for every non-goal, non-unitary clause C. Then $\mathcal{T}(\mathcal{P}, \phi)$ and $\mathcal{O}(\mathcal{P}, \phi)$ are isomorphic.*

Theorem 6.5 **($\mathcal{F} = \mathcal{O}$)** *Let ϕ be s.t. $\phi \subseteq \neg free(x^0_G)$, and $\phi \subseteq free(x^0_C)$ for every non-goal, non-unitary clause C. Then $\mathcal{F}(\mathcal{P}, \phi) = \mathcal{O}(\mathcal{P}, \phi)$.*

This result can be proven by showing that for every $k \geq 0$, $\Psi^k_i(\perp)$ is equal to the union of the path strongest postconditions w.r.t. ϕ of all the paths π which start in 1 and have length less or equal than k.

Corollary 6.6 **($\mathcal{T} \sim \mathcal{F}$)** *Let ϕ be s.t. $\phi \subseteq \neg free(x^0_G)$, and $\phi \subseteq free(x^0_C)$ for every non-goal, non-unitary clause C. Then $\mathcal{F}(\mathcal{P}, \phi) \sim \mathcal{T}(\mathcal{P}, \phi)$.*

7 Properties of $\mathcal{F}$

We show here that $\mathcal{F}$ enjoys some important properties, namely it is incremental, monotonic and and-compositional. Incrementality is important because, for instance, it allows us to compute the semantics of the union of two clp's $\mathcal{P}$ and $\mathcal{P}'$, by computing first the semantics of one of them, say $\mathcal{F}(\mathcal{P})$ of $\mathcal{P}$, and then by using $\mathcal{F}(\mathcal{P})$ to determine the semantics of their union $\mathcal{P} \cup \mathcal{P}'$. Also, from the practical point of view, incrementality allows us to define parallel execution models of clp's based on asynchronous processors, as explained in Section 8. And-compositionality allows us to compute the semantics of a goal $\leftarrow \overline{A}, \overline{B}$ from the semantics of $\leftarrow \overline{A}$ and of $\leftarrow \overline{B}$. The and-compositionality of $\mathcal{F}$ is used in the next section to define using $\mathcal{F}$ a goal-independent semantics.

Formally, let S be a subset of $\{1, \ldots, n\}$. We define $\Psi_S : (2^{States})^n \to (2^{States})^n$, called *the restriction of Ψ to the pp's in S*, as in Definition 5.3 except that for every pp l which is not in S, $(\Psi_S)_l(\overline{\psi})$ is set to be ψ_l.

Lemma 7.1 **(Incrementality)** *Let S be a subset of $\{1, \ldots, n\}$. If $\overline{\psi} \subseteq \mu\Psi$ then $\bigcup_{k=0}^{\omega} \Psi^k_S(\overline{\psi}) \subseteq \mu\Psi$.*

This lemma says that to compute $\mathcal{F}$ one can first restrict to a subset S of the pp's of the program, and iterate Ψ a number of times, using only the pp's of S; then the result $\overline{\psi}$ obtained can be incremented by iterating Ψ starting from $\overline{\psi}$ instead than $\perp$.

Lemma 7.2 **(Monotonicity)** *If $\phi \subseteq \phi'$ then $\mathcal{F}(\mathcal{P}, \phi) \subseteq \mathcal{F}(\mathcal{P}, \phi')$.*

A program without a goal is called *pure*.

Lemma 7.3 **(And-compositionality)** *Let $G = \leftarrow A_1, \ldots, A_\ell, B_1, \ldots, B_m$ and let $\mathcal{P}$ be a pure program. Suppose that:*

$\mathcal{F}(\{\leftarrow A_1, \ldots, A_\ell\} \cup \mathcal{P}, \phi_1) = (\phi_1, \phi_2, \ldots, \phi_{\ell+1}, \phi_{\ell+2}, \ldots, \phi_{\ell+k})$,

$\mathcal{F}(\{\leftarrow B_1, \ldots, B_m\} \cup \mathcal{P}, \phi_{\ell+1}) = (\psi_1, \psi_2, \ldots, \psi_{m+1}, \psi_{m+2}, \ldots, \psi_{m+k})$. *Then* $\mathcal{F}(\{G\} \cup \mathcal{P}, \phi_1) = (\phi_1, \ldots, \phi_{\ell+1}, \psi_2 \ldots, \psi_{m+1}, \phi_{\ell+2} \cup \psi_{m+2}, \ldots, \phi_{\ell+k} \cup \psi_{m+k})$.

The Monotonicity lemma follows by the monotonicity of Ψ, while the proofs of the other lemmas use the intermediate semantics $\mathcal{O}$, and can be found in the full version of the paper. The Monotonicity and the And-Compositionality Lemmas are used in the next section to define a goal-independent dataflow semantics for clp's.

A Goal-Independent Semantics

$\mathcal{F}$ is defined w.r.t. a set of input states describing a set of initial bindings for the goal, hence *lifting* to sets of goals the so called goal-dependent analysis, where only one goal is considered. In logic programming other semantics, like those based on the s-semantics ([BGLM94]), perform an analysis which is goal-independent, i.e. they refer to *pure* (viz. without goal) programs. These two different kinds of analysis can be nicely reconciled, since one can (finitely) define for a pure clp $\mathcal{P}$ a goal-independent semantics $\widehat{\mathcal{F}}(\mathcal{P})$.

Let $\{G\} \cup \mathcal{P}$ be a program. Define the *restriction of* $\mathcal{F}(\{G\} \cup \mathcal{P}, \phi)$ *to* $\mathcal{P}$, written $\mathcal{F}(\{G\} \cup \mathcal{P}, \phi)_{|\mathcal{P}}$, to be the tuple obtained from $\mathcal{F}(\{G\} \cup \mathcal{P}, \phi)$ by deleting those elements which are associated with the pp's of G.

Then the *goal-independent semantics* $\widehat{\mathcal{F}}(\mathcal{P})$ of a pure clp $\mathcal{P}$ is

$$\widehat{\mathcal{F}}(\mathcal{P}) = \bigcup_{p \text{ in } pred(\mathcal{P})} \mathcal{F}(\{G_p\} \cup \mathcal{P}, \phi_{G_p})_{|\mathcal{P}},$$

where $pred(\mathcal{P})$ is the set of predicate symbols occurring in $\mathcal{P}$, G_p is $\leftarrow p(\tilde{x})$, and ϕ_{G_p} is the set $\neg free(x^0_{G_p}) \cap free(x^0_{C1}) \cap \ldots \cap free(x^0_{Ck})$, where $C1, \ldots, Ck$ are the non-unitary clauses of $\mathcal{P}$.

Then $\widehat{\mathcal{F}}$ is the best goal-independent dataflow semantics, in the following sense:

Theorem 7.4 *For every pure program* $\mathcal{P}$, $\widehat{\mathcal{F}}(\mathcal{P}) = \bigcup_{\substack{G \text{ a goal} \\ \phi \subseteq \phi_G}} \mathcal{F}(G \cup \mathcal{P}, \phi)_{|\mathcal{P}}$.

Proof. By the Monotonicity and And-compositionality Lemmas. □

8 Applications

The dataflow semantics $\mathcal{F}$ allows us to view a program as a dataflow, where a node l receives states from the set $input(l)$ of all the nodes l' s.t. (l', l) is an arc of the dataflow graph. This description of the semantics of a clp is important for various reasons. $\mathcal{F}$ can be used to study run-time properties of clp's, as done e.g. in [DM88, CM91, DM93] for logic programs. For instance, we have used $\mathcal{F}(\mathcal{P}, \phi)$ in [CMM95] to develop a sound and complete method to prove termination of a clp w.r.t. a precondition ϕ. In this section we give two other possible applications of the dataflow semantics. In the first one $\mathcal{F}$ is used to define a parallel execution model based on asynchronous processors. In the second one the semantics $\mathcal{O}$ is used to define an *à la* Burstall [Bur74] intermittent assertions method for clp's.

8.1 A Parallel Execution Model

The Incrementality Lemma 7.1 for $\mathcal{F}$ suggests a possible parallel execution model $\mathcal{M}$ of clp's based on a network of processors, defined as follows:

Network Let N be the set of pp's of $\mathcal{P}$. For $l \in N$, a *processor* P_l is associated with l.

Communication among processors is realized by means of channels, as follows:

Communication Processors are connected by the following *channels*:

- $c_{env}^{entry(G)}$ from the environment env to $P_{entry(G)}$ and $c_{exit(G)}^{env}$ from $P_{exit(G)}$ to the environment;
- c_i^j from i to j for every i, j such that there is an arc from i to j in $dg(\mathcal{P})$.

A channel c_i^j is called an *input channel of* P_j and an *output channel of* P_i. Each channel is supposed to have a *memory* that contains a queue of states whose policy is fair (e.g. first in first out).

The execution model allows the processors to run in parallel and asynchronously:

Execution Model Processors in the network execute asynchronously the following *algorithms*:

- $P_{entry(G)}$ takes an α from $c_{env}^{entry(G)}$ and sends it to all its output channels.
- $P_{entry(C)}$ selects with *fair choice* from one of its input channels, say $c_{call(A)}^{entry(C)}$, an α, and it computes $push(\alpha) \wedge \overline{s}^1 = \overline{t}^0$, where $A = p(\overline{s})$ and $p(\overline{t})$ is the head of H; then $P_{entry(C)}$ sends $push(\alpha) \wedge \overline{s}^1 = \overline{t}^0$ to every its output channel.
- $P_{success(A)}$, where A is not a constraint and is contained in the clause C, selects with fair choice from one of its input channels, say $c_{exit(D)}^{success(A)}$, an α; then it computes $pop(\alpha)$; if $pop(\alpha)$ is in $\neg free(x_C^0)$ then $P_{success(A)}$ sends $pop(\alpha)$ to every its output channel.
- $P_{success(A)}$, where A is a constraint, takes an α from its input channel and computes $\alpha \wedge A^0$, then $P_{success(A)}$ sends $\alpha \wedge A^0$ to every its output channel.

This model describes a sound and complete implementation of $\mathcal{O}$, as stated in the following theorem.

Theorem 8.1 (Adequacy of $\mathcal{M}$) *If the input channel c_{env}^e of $\mathcal{M}$ is feed with the set of states ϕ s.t. $\phi \subseteq \neg free(x_G^0)$, and $\phi \subseteq free(x_C^0)$ for every non-goal, non-unitary clause C, then $\bigcup_{\pi \in path(l)} psp.\pi.\phi$ is the set of states that P_l in $\mathcal{M}$ sends on its output channels.*

This result can be proven using $\mathcal{O}$. For the completeness part, observe that, intuitively, since the choice of the state to be processed is fair, no state will be delayed forever.

Remark 8.2 Our execution model assigns one processor to each program point. However, because the processors work asynchronously, in case there are less processors than program points, then a single processor can be assigned to a number of pp's, which can be encoded as distinct tasks to be executed with a fair schedule discipline. This will still yield a complete and asynchronous model.

8.2 Burstall's Intermittent Assertions Method

We show how the intermittent assertions method of Burstall [Bur74] can be adapted to clp's. The advantages of the Intermittent Assertion Method, and of Temporal Logic (TL) in general, for instance to prove *liveness properties*, *termination*, *total correctness* etc. are well known (see for instance [CC93]). So far, finding a suitable presentation of the intermittent assertion method for logic programming was still an open problem ([CC93]). In this section we show how one can give a solution to this problem for clp's, by means of the intermediate semantics $\mathcal{O}$. For lack of space, the presentation is rather sketchy: We mention the main ingredients of the system, and give an example to illustrate its application. The complete specification of the corresponding formal system is the subject of another forthcoming paper.

For simplicity, assertions are denoted by ϕ, ψ, thus identifying an assertion with the set of states it denotes. Implication is interpreted as set inclusion, i.e. $\phi \Rightarrow \psi$ iff $\phi \subseteq \psi$. Also, conjunction and disjunction are interpreted set-theoretically as intersection and union, respectively. The assertion $push(\phi)$ is obtained by replacing each `i`-variable x^i in ϕ by the `i`-variable x^{i+1}; and $pop(\phi)$ is obtained by first renaming with fresh variables all the `i`-variables of index 0 and then replacing each remaining `i`-variable x^i with x^{i-1}.

Here, an 'intermittent rule' is a formula in temporal logic of the form $\Box\big(\phi \wedge \mathrm{at}(i) \Rightarrow \Diamond(\psi \wedge \mathrm{at}(j))\big)$, where $\Box$ and $\Diamond$ are the 'always' and 'sometime' operators, and $\mathrm{at}(i)$ indicates that execution is at program point i. The intended meaning of this formula is: for every state α which satisfies ϕ, there is at least one execution of the program starting in the pp i with state α, which reaches the pp j in a state which satisfies ψ. The set of proof rules we consider contains a formalization of the induction principle (Burstall's "little induction"), a suitable axiomatization of TL (cf. [Sti92, CC93]), plus the following *path rule*, which formalizes the "hand simulation" part of the method:

$$(\pi \in path(i,j) \wedge psp.\pi.\phi \neq false) \Rightarrow \Box\big(\phi \wedge \mathrm{at}(i) \Rightarrow \Diamond(psp.\pi.\phi \wedge \mathrm{at}(j))\big)$$

A *sound* and *relatively complete* proof system w.r.t. $\mathcal{F}$ can be defined using these tools.

We illustrate by means of an example how the method can be applyed to prove total correctness of a clp. The following *composition rule* will be used:

$$\frac{\Box\big(\phi \wedge \mathrm{at}(i) \Rightarrow \Diamond(\psi \wedge \mathrm{at}(j))\big) \qquad \Box\big(\psi \wedge \mathrm{at}(j) \Rightarrow \Diamond(\chi \wedge \mathrm{at}(k))\big)}{\Box\big(\phi \wedge \mathrm{at}(i) \Rightarrow \Diamond(\chi \wedge \mathrm{at}(k))\big)} \qquad (1)$$

It enables us to compose intermittent assertions (note that this is a particular case of the 'chain rule' which is one of the basic tools in the proof system presented in [MP83]).

Example 8.3 Consider again the program *Prod*. Let the initial assertion ϕ be $u^0 = [r_0, \ldots, r_k] \wedge \neg free(x^0_G) \wedge free(x^0_{C1}) \wedge \mathrm{at}(1)$.

Suppose that we want to prove that *Prod* satisfies the following assertion:

$$\Box(\phi \Rightarrow \Diamond(v^0 = r_0 * \ldots * r_k \wedge \mathrm{at}(2))) \tag{2}$$

which says that for every state α of ϕ, at least one execution of the goal $\leftarrow prod(u, v)$ starting in α terminates (i.e. reaches the pp 2) and its final state binds v to $r_0 * \ldots * r_k$. Using the path rule we obtain the following (simplified) assertions:

$\Box\big(\phi \Rightarrow \Diamond(v^1 = z^0 = r_0 * w^0 \wedge y^0 = [r_1, \ldots, r_k]) \wedge \mathrm{at}(4))\big)$
with path $\langle 1, 3, 4\rangle$;

$\Box\big(v^{k+1} = z^k = r_0 * \ldots * r_k * w^0 \wedge y^0 = [\,] \wedge \mathrm{at}(4) \Rightarrow$
$\Diamond(v^{k+1} = z^k = r_0 * \ldots * r_k \wedge y^0 = [\,] \wedge \mathrm{at}(5))\big)$
with path $\langle 4, 6, 5\rangle$;

$\Box\big(v^1 = z^0 = r_0 * \ldots * r_k \wedge \mathrm{at}(5) \Rightarrow$
$\Diamond(v^0 = r_0 * \ldots * r_k \wedge \mathrm{at}(2))\big)$
with path $\langle 5, 2\rangle$;

The following assertions can be proven by straightforward induction:

$\Box\big(v^{m+1} = z^m = r_0 * \ldots * r_m * w^0 \wedge y^0 = [r_{m+1}, \ldots, r_k] \wedge m < k \wedge \mathrm{at}(4) \Rightarrow$
$\Diamond(v^{k+1} = z^k = r_0 * \ldots * r_k * w^0 \wedge y^0 = [\,] \wedge \mathrm{at}(4))\big)$
using as path $\pi = \langle 4, 3, 4\rangle$, and

$\Box\big(v^{k+1} = z^k = r_0 * \ldots * r_k \wedge y^0 = [\,] \wedge \mathrm{at}(5) \Rightarrow$
$\Diamond(v^1 = z^0 = r_0 * \ldots * r_k \wedge \mathrm{at}(5))\big)$
using as path $\pi = \langle 5, 5\rangle$.

Then, the repeated application of rule (1) to compose the above assertions yields (2). □

9 Discussion

In this paper an alternative operational model for clp's was proposed, where a program is viewed as a dataflow graph and a predicate transformer semantics transforms a set of states associated with a fixed node of the graph (corresponding to the entry-point of the program) into a tuple of set of states, one for each node of the graph. To the best of our knowledge, this is the first predicate transformer semantics for clp's based on dataflow graphs. The dataflow graph provides a static description of the flow of control of a program, where sets of constraints 'travel' through its arcs. The relevance of this approach was substantiated in the Applications section.

We would like to conclude this paper by giving an extension of its results to more general CLP systems. We have considered 'ideal' CLP systems. With slight modifications, the dataflow semantics $\mathcal{F}$ (and all its applications) can be adapted to deal also with 'quick-check' and 'progressive' systems (cf. [JM94]), which are those more widely implemented. This can be done as follows. States are considered to be pairs (c_1, c_2) of constraints, instead than constraints, where c_1 denotes the active part and c_2 the passive part.

$$\mathit{States} = \{(c_1, c_2) \mid c_1 \text{ and } c_2 \text{ are constraints s.t. } \mathit{consistent}(c_1)\},$$

where the test $\mathit{consistent}(c_1)$ checks for (an approximation of) the consistency of c_1. Then rules **R** and **C** of Table 1 have to be changed as illustrated below, where a state $\alpha = (c_1, c_2)$ is also denoted by (α_1, α_2):

$$\mathbf{R} \quad (\langle\, p(\overline{s})\rangle \cdot \overline{A},\, \alpha\,) \longrightarrow (\,\overline{B} \cdot \langle pop\rangle \cdot \overline{A},\, \mathit{infer}(\alpha'_1, \alpha'_2 \wedge \overline{s}^1 = \overline{t}^0)\,),$$

with $\alpha' = \mathit{push}(\alpha)$, if $C = p(\overline{t}) \leftarrow \overline{B}$ is in $\mathcal{P}$.

$$\mathbf{C} \quad (\langle\, d\,\rangle \cdot \overline{A},\, \alpha\,) \longrightarrow (\,\overline{A},\, \mathit{infer}(\alpha_1, \alpha_2 \wedge d^0)\,),$$

if d is a constraint. Finally, the definition of sp has to be changed in:

$$sp.c.\phi = \; \{\alpha' \in \mathit{States} \;\mid\; \alpha' = \mathit{infer}(\alpha_1, \alpha_2 \wedge c) \text{ and } \alpha \in \phi\}.$$

The operator *infer* computes from the current state (c_1, c_2) a new active constraint c'_1 and passive constraint c'_2, with the requirement that $c_1 \wedge c_2$ and $c'_1 \wedge c'_2$ are equivalent constraints. The intuition is that c_1 is used to obtain from c_2 more active constraints; then c_2 is simplified to c'_2. For instance, in the example of Section 5.5, in the state of Ψ^2_4 the constraint $z^0 = x^0 * w^0$ would be passive, because the equation is not linear (cf. [JMSY92]). Then, in Ψ^3_6 this constraint is transformed by applying first *push* to it and then *infer* . So $z^1 = x^1 * w^1$ becomes active, because w^1 is bound to 1 and hence the equation becomes linear.

Acknowledgments: We would like to thank Jan Rutten and the anonymous referees for their useful comments. The research of the second author was partially supported by the Esprit Basic Research Action 6810 (Compulog 2).

References

[BGLM94] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: theory and applications. *The Journal of Logic Programming*, 19,29: 149–197, 1994.

[Bur74] R.M. Burstall. Program proving as hand simulation with a little induction. *Information Processing*, 74:308–312, 1974.

[CC93] P. Cousot and R. Cousot. "A la Burstall" Intermittent Assertions Induction Principles for Proving Inevitability Properties of Programs. *Theoretical Computer Science*, 120:123–155, 1993.

[CM91] L. Colussi and E. Marchiori. Proving correctness of logic programs using axiomatic semantics. In *Proceedings of the Eight ICLP*, pages 629–644. MIT Press, 1991.

[CMM95] L. Colussi, E. Marchiori and M. Marchiori. On Termination of Constraint Logic Programs. In *Proc. First International Conference on Principles and Practice of Constraint Programming*. LNCS, Springer–Verlag, 1995. To appear.

[DM88] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *TCS*, 59(1):133–155, 1988.

[DM93] P. Deransart and J. Małuszyński. A Grammatical View of Logic Programming. The MIT Press, 1993.

[JMSY92] J. Jaffar, S. Michaylov, P.J. Stuckey and R.H.C. Yap. The CLP($\mathcal{R}$) Language and System. *ACM TOPLAS*, 14(3):339–395, 1992.

[JM94] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *JLP* 19,20: 503-581, 1994.

[Kun87] K. Kunnen. Signed Data Dependency in Logic Programs. *Computer Science Technical Report 719*, University of Wisconsin - Madison, 1987.

[Mel87] C. Mellish. Abstract interpretation of Prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of declarative languages*, pp. 181–198. Ellis Horwood, 1987.

[MP83] Z. Manna and A. Pnueli. How to cook a proof system for your pet language. In *Proceedings 10th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pp. 141–154, 1983.

[Nil90] U. Nilsson. Systematic semantics approximations of logic programs. In *Proc. PLILP*, pp. 293–306. Eds. P. Deransart and J. Małuszyński, Springer Verlag, 1990.

[Sti92] C. Stirling. Modal and Temporal Logics. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 477–563, 1992.

[WS94] B. Wang and R.K. Shyamasundar. A methodology for proving termination of logic programs. *JLP* 21(1): 1–30, 1994.