A FRAMEWORK TO INCORPORATE NON-MONOTONIC REASONING INTO CONSTRAINT LOGIC PROGRAMMING

JÜRGEN DIX AND FRIEDER STOLZENBURG

▷ Impressive work has been done in the last years concerning the meaning of negation and disjunction in logic programs, but most of this research concentrated on propositional programs only. While it suffices to consider the propositional case for investigating general properties and the overall behaviour of a semantics, we feel that for real applications and for computational purposes an implementation should be able to handle first-order programs without grounding them.

In this paper we present a theoretical framework by defining a calculus of program transformations that apply directly to rules with variables and function symbols. Our main results are that (a) this calculus is weakly confluent for arbitrary programs, (b) it is terminating for Datalog$^{\vee,\neg}$ programs, (c) for finite ground programs it is equivalent to a terminating calculus introduced by Brass and Dix (1995), and (d) it approximates a generalisation of D-WFS for arbitrary programs.

We achieve this by transforming program rules into rules with equational constraints thereby using heavily methods and techniques from constraint logic programming (CLP). In particular, disconnection-methods play a crucial role. In principle, any constraint theory known from CLP can be exploited in the context of non-monotonic reasoning, not only equational constraints over the Herbrand domain. However, the respective constraint solver must be able to treat negative constraints of the considered constraint domain.

In summary, this work yields the basis for a general combination of two paradigms: constraint logic programming and non-monotonic reasoning. ◁

This is a completely revised and extended version of [16], originally presented at the Postconference Workshop on *Non-monotonic Extensions of LP* following JICSLP '96, Bad Honnef, Germany, September 7–8, 1996.

Address correspondence to Jürgen Dix and Frieder Stolzenburg, University of Koblenz, Department of Computer Science, Rheinau 1, D-56075 Koblenz, Germany

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

1. INTRODUCTION

Recent years have seen an enormous number of different semantics of logic programs with negation. Fewer semantics have been developed for disjunctive programs and nearly all approaches assume "without loss of generality"—as most authors put it—that the underlying programs are *grounded*, i.e. they do not contain variables. However, we think that one of the most important advantages of the logic programming paradigm and therefore the success of Prolog is its ability to compute answer-substitutions for a given query with variables.

Although semantics for logic programs with negation are undecidable if function symbols and variables are allowed, we are convinced that query answering mechanisms for the non-ground-case have great advantages over the propositional case. Of course, such procedures can only be sound and not complete. But completeness can hold for certain restricted classes of programs as well as for certain queries.

Our approach is essentially based on a calculus of program transformations that has been recently shown to be confluent and terminating for ground programs [6]. The most important transformation in this calculus is the partial evaluation property (GPPE) introduced for disjunctive programs independently by Sakama/Seki and Brass/Dix ([28, 3]). Unfortunately, GPPE is not sound for rules with variables because of the occurrence of unifiable atoms in the heads of rules (see Section 4).

In this paper we make the GPPE sound—which allows us to use the results of [6]—by introducing *inequality constraints*. This immediately leads us to introduce *constraint disjunctive logic programs* and consequently to extend our transformations to this class of programs. Surprisingly, this framework shares the same nice properties as our original calculus. In fact, we can lift almost all results of [6] to the non-ground case.

The paper is structured as follows. In Section 2 we introduce the basic notation we will use throughout the paper. In Section 3 we consider ground programs. After introducing our calculus of transformations originally defined in [3, 6], we cite its *confluence* and *termination* for *finite* ground programs: this gives rise to the semantics D-WFS. We then generalise D-WFS to arbitrary infinite ground programs by introducing a *Loop-Detection rule* and showing that the calculus still is confluent.

Our overall framework is introduced in Section 4, where we consider arbitrary first-order programs. We extend program rules by a *constraint theory* and investigate how techniques from the CLP field apply. In Section 5, we extend our original transformations using the new framework so that they directly apply to programs with variables. We introduce CD-WFS using the new calculus, prove its weakly confluence, and show how it can be computed and used for *query-answering*. We also show its relation with WFS, GCWA and D-WFS and we show termination for arbitrary function-free disjunctive logic programs with negation (not necessarily range-restricted), i.e. Datalog$^{\vee,\neg}$. We conclude with Section 6.

2. PRELIMINARIES

In the first part of this paper we restrict to propositional programs. We also allow that they are infinite, i.e. they are full instantiations of first order programs. For this, let some fixed finite signature Σ containing the binary connectives $\wedge, \vee, \rightarrow$, the unary connective $\neg$ and the *falsum* $\bot$ be given.

Definition 2.1. [Logic Program Φ]
A logic program Φ is a (possibly infinite) set of rules of the form

$$A_1 \vee \cdots \vee A_l \leftarrow B_1 \wedge \cdots \wedge B_m \wedge \neg C_1 \wedge \cdots \wedge \neg C_n$$

where the A_i, B_i and C_i are Σ-atoms different from $\bot$, $l \geq 1$, $m \geq 0$, $n \geq 0$. We allow empty conjunctions: they are considered as abbreviations for $\neg\bot$ (the *verum*) which we also denote by $\top$.

We identify such a rule with the triple consisting of the following sets of atoms $\mathcal{A} := \{A_1, \ldots, A_k\}$, $\mathcal{B} := \{B_1, \ldots, B_m\}$, $\mathcal{C} := \{C_1, \ldots, C_n\}$, and write it as $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C}$. This means, in particular, that we assume the A_i (respectively the B_i, respectively the C_i) to be pairwise distinct. We write *heads*(Φ) for the set of all atoms occurring in rule heads in Φ: these are atoms that are *possibly true*. By *pure* disjunctions we mean disjunctions consisting solely of positive or solely of negative literals. We also want to allow *integrity constraints*, i.e. rules with empty heads ($l = 0$). This will sometimes lead to inconsistent programs, i.e. a program Φ may semantically imply both a proposition and its negation.

Definition 2.2. [Operator $\mid\!\sim$, Semantics $\mathcal{S}_{\mid\sim}$]
By a semantic operator $\mid\!\sim$ we mean a binary relation between logic programs and pure disjunctions which satisfies the following three arguably obvious conditions:

(1) **Right Weakening:** If $\Phi \mid\!\sim \psi$ and $\psi \subseteq \psi'$ (ψ is a subdisjunction of ψ'), then $\Phi \mid\!\sim \psi'$.

(2) **Necessarily True:** If $(\mathcal{A} \leftarrow\ ) \in \Phi$ for a disjunction $\mathcal{A}$, then $\Phi \mid\!\sim \mathcal{A}$.

(3) **Necessarily False:** If $A \notin$ *heads*(Φ) for some Σ-ground atom A, then $\Phi \mid\!\sim \neg A$.

Given such an operator $\mid\!\sim$ and a logic program Φ, by the semantics $\mathcal{S}_{\mid\sim}(\Phi)$ of Φ determined by $\mid\!\sim$ we mean the set of all pure disjunctions derivable by $\mid\!\sim$ from Φ, i.e. $\mathcal{S}_{\mid\sim}(\Phi) := \{\psi \mid \Phi \mid\!\sim \psi\}$.

Note that both *model-based* as well as *completion-based* approaches fit well into this framework, because these approaches provide in a natural way a set of *derivable* disjunctions. We simply take the sceptical view: truth in all intended models or in all models of the completion. In this respect, we contribute to the discussion about the usefulness of model theory. On the one hand, our approach is not against model theory. But on the other hand, we do not explicitly need a model theory.

In addition to satisfying the general conditions from Definition 2.2, we may want a specific semantic operator $\mid\!\sim$ to be invariant under certain *natural* program transformations. So, we will give a set of reasonable transformation rules in the sequel.

Definition 2.3. [Invariance of $\mid\!\sim$ under a Transformation]
Suppose that a program transformation *Trans* is given. We view such a transformation as a relation (not necessarily a function) which specifies under what conditions a program Φ' is a transformation of another program Φ. To facilitate notation we also write such a transformation sometimes as a function *Trans* : $\Phi \mapsto$ *Trans*(Φ), mapping logic programs into logic programs.

In general, a particular transformation can be applied in different ways to a program so that *Trans*(Φ) is not uniquely defined. We say that the operator $\mid\!\sim$ is invariant under *Trans* (or that *Trans* is a $\mid\!\sim$-equivalence transformation) iff

$$\Phi \mid\!\sim \psi \iff \mathit{Trans}(\Phi) \mid\!\sim \psi$$

for any pure disjunction ψ and any program Φ.

Why do we only consider *pure* disjunctions? This is simply because we are interested in the sceptical viewpoint and therefore we cannot assume that any semantics is given as a set of models. For example, SLDNF-like semantics usually are given in proof-theoretic terms. In order to be as general as possible (i.e. to cover as many semantics as possible in our framework), we can only assume that a semantics provides us at least with the notion of deriving arbitrary conjunctions consisting of either positive or either negative literals and of negations of such conjunctions. But these are exactly the *pure* disjunctions. Some semantics simply do not define what it means to derive a *mixed* disjunction.

3. D-WFS FOR GROUND PROGRAMS

We now describe several transformations *Trans* which will be used later to define the D-WFS semantics. By abuse of language (and to facilitate reading) we will simply say "$\mid\!\sim$ satisfies *Trans*" meaning that "$\mid\!\sim$ is invariant under *Trans*". A more complete discussion of our calculus can be found in [5, 6].

3.1. The Calculus of Transformations

We begin with partial evaluation in the sense of the "unfolding" operation. It is the *generalised principle of partial evaluation* (GPPE) [3, 28]:

Definition 3.1. [GPPE]
A semantics $\mathcal{S}_{\mid\sim}$ satisfies GPPE if it is invariant under the following transformation: *Replace a rule* $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C}$ *where* $\mathcal{B}$ *contains a distinguished atom* B *by the rules*

$$\mathcal{A} \cup \big(\mathcal{A}_i - \{B\}\big) \;\leftarrow\; \big(\mathcal{B} - \{B\}\big) \cup \mathcal{B}_i \;\wedge\; \neg\big(\mathcal{C} \cup \mathcal{C}_i\big) \quad (i = 1, \ldots, k)$$

where $\mathcal{A}_i \leftarrow \mathcal{B}_i \wedge \neg\mathcal{C}_i$ $(i = 1, \ldots, k)$ are *all* the rules with $B \in \mathcal{A}_i$.

Note that we are free to select a specific positive occurrence of an atom B and then perform the transformation: this is just to say that the GPPE-transformation is a relation between programs. The new rules are obtained by replacing B by the bodies of all rules r with head literal B and adding the remaining head atoms of r to the head of the new rule. GPPE also covers the degenerate case when the atom B to be replaced does not appear in any head: then the whole rule is simply deleted.

The next transformation, *elimination of tautologies*, states that tautological clauses like $p \leftarrow p$ do not influence the semantics of a logic program. With the help of *elimination of non-minimal rules* we can get rid of clauses that are subsumed by others.

Definition 3.2. [Elimination of Tautologies, Non-Minimal Rules]
A semantics $\mathcal{S}_{\mid\sim}$ satisfies the *elimination of tautologies* (TAUT), respectively the *elimination of non-minimal rules* (NMIN) if $\mid\sim$ is invariant under the following transformations:

- **TAUT:** Delete a rule $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C}$ with $\mathcal{A} \cap \mathcal{B} \neq \emptyset$.
- **NMIN:** Delete a rule $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C}$ if there is another rule $\mathcal{A}' \leftarrow \mathcal{B}' \wedge \neg\mathcal{C}'$ with $\mathcal{A}' \subseteq \mathcal{A}$, $\mathcal{B}' \subseteq \mathcal{B}$, and $\mathcal{C}' \subseteq \mathcal{C}$.

The last two transformations (stated below) allow us to do some simple reductions. We want $\neg L$ to be derivable if L appears in no rule head. Therefore, it should be possible to evaluate the body literal $\neg L$ to true, i.e. to delete $\neg L$ from all rule bodies: this is guaranteed by *positive reduction*. Conversely, if the logic program contains a disjunctive fact $(A_1 \vee \cdots \vee A_k \leftarrow \ )$, at least one of the atoms occurring in it must be true, so a rule body containing $\neg A_1 \wedge \cdots \wedge \neg A_k$ is surely false, so the entire rule is useless, and it should be possible to delete it: this gives us *negative reduction*.

Definition 3.3. [Positive and Negative Reduction]
Semantics $\mathcal{S}_{\mid\sim}$ satisfies *positive reduction* (RED$^+$), respectively *negative reduction* (RED$^-$) if $\mid\sim$ is invariant under the following transformations:

- **RED$^+$:** Replace a rule $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C}$ by $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg(\mathcal{C} \cap \mathit{heads}(\Phi))$.
- **RED$^-$:** Delete a rule $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C}$, if there is a rule $(\mathcal{A}' \leftarrow \ )$ with $\mathcal{A}' \subseteq \mathcal{C}$.

3.2. Finite Ground Programs

If our programs are finite, our calculus has some very nice properties:

Theorem 3.1. [Confluence and Termination [6]]
Our calculus of transformations is confluent for finite ground programs, i.e. if $\Phi \xrightarrow{*} \Phi'$ and $\Phi \xrightarrow{*} \Phi''$ and both Φ', Φ'' are irreducible, then $\Phi' = \Phi''$. In addition, it is also weakly terminating, *i.e. for any program Φ there is an irreducible Φ' with $\Phi \xrightarrow{*} \Phi'$. We call such irreducible Φ' the residual program $res(\Phi)$ from Φ.*

In fact, in [6] we prove a much stronger result, namely *strong termination* for *fair* sequences of transformations. Just for illustration, let us consider the program consisting of the single clause $p \leftarrow p$. GPPE can be applied and leads to the same program. Therefore we get an infinite, nonterminating sequence. We call such a sequence *unfair*, because an application of *elimination of tautologies* can be applied but in fact is never done.

Excluding such behaviour leads to *fair* sequences that always terminate (see [6] for details). Fair sequences can be produced quite obviously by

(1) never applying GPPE to a tautological clause (i.e. first applying *elimination of tautologies*) and

(2) by insisting that, if GPPE has been applied to get rid of an occurrence of atom A, then all other occurrences of A in bodies have to be removed before GPPE is applied to other atoms $B \neq A$.

Originally, the semantics D-WFS has been defined as follows:

Theorem 3.2. [D-WFS [7]]
There exists the weakest semantics $\mathcal{S}_{\mid\!\sim}$ which is invariant under GPPE, TAUT and NMIN, RED^{+} and RED^{-}. Moreover, this semantics is consistent (i.e. it does not derive a literal A and its negation $\neg A$) and closed under logical consequences. Here we mean by closure *the set of all* pure *disjunctions that follow logically. It does not matter if in the derivation non-pure disjunctions are needed. We call it the Disjunctive Well-Founded Semantics, or, briefly, D-WFS. It coincides with the well-founded semantics WFS for normal and with GCWA for positive disjunctive programs.*

In the above theorem, *weakest* semantics stands for the *intersection* of all semantics invariant under our transformations. This implies, in particular, that our transformations are *preserved under intersection* of two semantics—a property, that need not to hold for arbitrary sets of transformations. With the help of our calculus and the residual program, we can compute D-WFS quite nicely:

Theorem 3.3. [Completeness of D-WFS wrt. $res(\Phi)$ [7]]
$\Phi \mid\!\sim_{\mathrm{D-WFS}} \psi$ *iff* *there is a disjunction* $\mathcal{A} \subseteq \psi$ *with* $(\mathcal{A} \leftarrow\ ) \in res(\Phi)$ *or*
there is a literal $\neg A \in \psi$ *and* $A \notin heads(res(\Phi))$.

This theorem is especially important. It tells us that once the residual program $res(\Phi)$ has been produced, the semantics D-WFS can be immediately determined. In fact, $res(\Phi)$ contains enough information to compute the stable semantics by hyper-resolution. See also [4].

3.3. Infinite Ground Programs

What happens if our programs are (countably) infinite? Our transformations are also defined for infinite programs but it is not guaranteed that the calculus always terminates.

Example 3.1. [Infinite Unfounded Set]
We consider the program

$$p(x) \leftarrow p(s(x))$$

or the corresponding infinite propositional program P_{loop}

$$p_0 \leftarrow p_1,\ p_1 \leftarrow p_2,\ \ldots,\ p_i \leftarrow p_{i+1},\ \ldots$$

which can be seen as a propositional version of the full instantiation of the former program.

Our GPPE cannot "unfold" such an infinite loop and eliminate all these rules. To do this, something like an ω-inference rule is needed which is not a constructive rule. Of course we can try to define a particular loop-detection—see Definition 3.4 and the remarks in Section 6.1—, but since the general problem is undecidable, there is no constructive solution in general. If the above loop is finite, this can be recognised in a finite number of steps and D-WFS does this.

Although the very definition of D-WFS given in Theorem 3.2 does not depend on finite programs, we cannot just use this formulation for arbitrary programs. This is because the resulting semantics would not extend WFS or GCWA in that case. The weakest such semantics would simply derive nothing from P_{loop}, instead of $\{\neg p_0, \neg p_1, \ldots, \neg p_i, \ldots\}$ what we should expect. The reason is that for infinite programs we cannot always get rid of all the positive body atoms—like in P_{loop}. But this was the main achievement of the residual program in Theorem 3.3: the residual program contains only negative literals in the bodies of program rules.

For infinite ground programs our GPPE is a transformation replacing a rule by a possibly *infinite* set of other rules. It can now happen that infinite applications of GPPE do not exist, i.e. they do not lead to a well-defined program. Let us shortly illustrate this:

Example 3.2. [Infinite Applications]
We apply GPPE to P_{loop} to replace first p_1 in the first rule. Then we replace p_2 in the second rule and so on (i.e. replacing p_i in the i-th rule). We get a sequence of programs that converges:

$$
\begin{array}{lllclcl}
P_{loop}: & P_1: & P_2: & & P_i: & & P_\omega: \\
p_0 \leftarrow p_1 & p_0 \leftarrow p_2 & p_0 \leftarrow p_2 & & p_0 \leftarrow p_2 & & p_0 \leftarrow p_2 \\
p_1 \leftarrow p_2 & p_1 \leftarrow p_2 & p_1 \leftarrow p_3 & & p_1 \leftarrow p_3 & & p_1 \leftarrow p_3 \\
p_2 \leftarrow p_3 & p_2 \leftarrow p_3 & p_2 \leftarrow p_3 & \ldots & \vdots & \ldots & p_2 \leftarrow p_4 \\
\vdots & \vdots & \vdots & & p_{i-1} \leftarrow p_{i+1} & & \vdots \\
p_i \leftarrow p_{i+1} & p_i \leftarrow p_{i+1} & p_i \leftarrow p_{i+1} & & p_i \leftarrow p_{i+1} & & p_i \leftarrow p_{i+2} \\
\vdots & \vdots & \vdots & & \vdots & & \vdots
\end{array}
$$

But if we first replace p_1 in all rules, and then p_2 in all rules and so on (i.e. replacing p_i in the first i rules with i-fold application of GPPE), we get a non-converging series of programs:

$$
\begin{array}{lllclc}
P_{loop}: & P'_1: & P'_2: & & P'_i: & \\
p_0 \leftarrow p_1 & p_0 \leftarrow p_2 & p_0 \leftarrow p_3 & & p_0 \leftarrow p_i & \\
p_1 \leftarrow p_2 & p_1 \leftarrow p_2 & p_1 \leftarrow p_3 & & p_1 \leftarrow p_i & \\
p_2 \leftarrow p_3 & p_2 \leftarrow p_3 & p_2 \leftarrow p_3 & \ldots & \vdots & \ldots \\
\vdots & \vdots & \vdots & & p_{i-1} \leftarrow p_i & \\
p_i \leftarrow p_{i+1} & p_i \leftarrow p_{i+1} & p_i \leftarrow p_{i+1} & & p_i \leftarrow p_{i+1} & \\
\vdots & \vdots & \vdots & & \vdots &
\end{array}
$$

Nevertheless, we allow the *infinite* application of our rules, if it leads to a well-defined program: let us consider the program $p_0 \leftarrow p_0,\ p_1 \leftarrow p_1,\ \ldots,\ p_i \leftarrow p_i,\ \ldots.$ Only after countably many applications of elimination of tautologies, we get the empty program and can derive $\neg p_i$ for all $i \in \mathbb{N}$.

But there is still the problem with P_{loop}: our transformations (even applied infinitely often) do not allow us to reduce this program to the empty program in order to derive $\neg p_i$ for all $i \in \mathbb{N}$. Can we possibly ensure this with an additional transformation? Indeed, we can formulate a *non-constructive rule* which is similar to condition (3) of Definition 2.2:

Definition 3.4. [Loop-Detection Rule]
Let A be an atom and Φ be a program. Delete all rules $(\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C}) \in \Phi$ with $A \in \mathcal{A}$, if for all Φ' with $\Phi \xrightarrow{*} \Phi'$ we have $\mathcal{B}' \neq \emptyset$ for all $(\mathcal{A}' \leftarrow \mathcal{B}' \wedge \neg\mathcal{C}') \in \Phi'$ with $A \in \mathcal{A}'$.

This rule formalises the intuition that if an atom always (and forever) depends positively on other atoms, then they all should be considered false: this is obviously true for WFS and GCWA. The notation $\xrightarrow{*}$ denotes the transitive closure of the transformations GPPE, TAUT, NMIN, RED^+ and RED^-. Of course, an application of the Loop-Detection rule can allow new applications of the other rules. Therefore, we denote with $\xrightarrow{*}$ from now on the transitive closure of all our rules (including loop-detection). We will later comment on how approximations of the loop-detection rule can be constructively incorporated in our calculus (see Section 6). Note that the Loop-Detection Rule gives us termination of the calculus after possibly infinitely many derivation steps.

We are now in a position to generalise D-WFS and to extend Theorem 3.2 to countably infinite programs. As the following Theorems 3.4 and 3.5 will not be used in this paper anymore and as their proofs closely resemble that given at length in [7], we only sketch the proofs here.

Theorem 3.4. [D-WFS for Infinite Programs]
There exists the weakest semantics $\mathcal{S}_{\mid\sim}$ which is invariant under GPPE, elimination of tautologies and non-minimal rules, positive and negative reduction and satisfies the Loop-Detection rule. Moreover, this semantics is consistent (i.e. it does not derive a literal A and its negation $\neg A$) and closed under logical consequences (as a set consisting of pure disjunctions). It extends D-WFS from Theorem 3.3 and is therefore also called D-WFS. It coincides with the well-founded semantics WFS for normal and with GCWA for positive disjunctive programs.

PROOF. To show that the weakest semantics exists, we have to show that our properties are preserved under intersection of two semantics (viewed as a set of pure disjunctions as in Definition 2.2). This is worked out in detail in [7] for the transformations introduced in Section 3.1 and we need to consider here just the Loop-Detection rule. But this is simple, because this rule refers to our old transformations and we can use the result for them. In particular, if the premises of the Loop-Detection rule are satisfied for the intersection of two semantics $\mathcal{S}_{\mid\sim}$ and $\mathcal{S}'_{\mid\sim}$, then they are also satisfied for both $\mathcal{S}_{\mid\sim}$ and $\mathcal{S}'_{\mid\sim}$, and therefore the Loop-Detection rule is valid for the intersection.

Why does D-WFS coincide with WFS for normal programs? Here we have to use a construction of WFS that is given in [14] (Lemma 3.9). The Loop-Detection rule allows us to get rid of the *unfounded atoms*, i.e. to remove the clauses where they appear in the head and therefore to derive their negations.

That D-WFS coincides with GCWA for positive programs is even simpler. Obviously, the Loop-Detection rule preserves the minimal models. But all other transformations do this as well [5]. It is also easy to see that any positive disjunctive program can be reduced (using our transformations) to a set of rules with empty bodies (here we need our Loop-Detection rule). Therefore we end up with a residual program which still has the same minimal models as the program we started with. Consequently, D-WFS coincides with GCWA. □

We also have the following confluence result:

Theorem 3.5. [Confluence for Infinite Programs]
Our set of transformations is confluent for infinite ground programs, i.e. if $\Phi \xrightarrow{*} \Phi'$ *and* $\Phi \xrightarrow{*} \Phi''$ *and both* Φ', Φ'' *are irreducible, then* $\Phi' = \Phi''$*. If a program* Φ' *is irreducible, then D-WFS can be computed by using the characterisation given in Corollary 3.3. Termination in general does not hold.*

PROOF. In this theorem, we do not have termination which facilitates the proof. At first sight, a proof of this theorem seems to follow immediately from the observation that the Loop-Detection rule *commutes* with all other rules. Therefore the confluence reduces to the confluence of the original transformations, which is stated in Theorem 3.1. But this is not quite right, because we allow also (as mentioned before Definition 3.4) for *infinite* applications of our rules. As the Loop-Detection rule also *commutes* with *infinite* applications of our transformations (if they lead to a well-defined program), we are left with the confluence of our original rules, but now allowing infinite applications. But this result can be proved completely analogous to the proof of Theorem 3.1 in [6]. □

There are two important points concerning infinite ground programs that we would like to emphasise:

- The above defined version of D-WFS for infinite ground programs is just a theoretical construct that we will try to approximate with our calculus to be presented in Section 5. Like the WFS for first-order programs, the D-WFS using the Loop-Detection rule is uncomputable in general. The way out of this is to find *constructive* approximations of the Loop-Detection rule (see Section 6) or to ensure termination for particular classes of programs.

- When considering countably infinite programs, we were forced to introduce *infinite* applications of rules. However, such infinite applications do not always lead to a well-defined program. In the first-order version to be presented in the main part of this paper, we only need *finite* applications. This is because one *single application of GPPE to a first-order rule* corresponds to an infinite application of GPPE to the instantiated form of this rule (which is also infinite).

4. NON-GROUND PROGRAMS

What are the problems to extend our transformations to first-order programs? One of the main problems is already *built-in* in Definition 2.1 where we identified a rule with *sets* of atoms: this means that we do not allow for duplicate occurrences. This is not just a technical point: without this restriction, GPPE would not even be a *sound* transformation.

Example 4.1. [Duplicate Occurrences and GPPE]

In the following program Φ we apply the GPPE and obtain $GPPE_B(\Phi)$.

$$\begin{array}{rll} \Phi: & A & \leftarrow B \\ & B \vee B \leftarrow & \end{array} \qquad \begin{array}{rl} GPPE_B(\Phi): & A \vee B \leftarrow \\ & B \vee B \leftarrow \end{array}$$

$GPPE_B(\Phi)$ is irreducible, but equivalent to the program $A \vee B \leftarrow$, $B \leftarrow$, which reduces to $B \leftarrow$ only. $A \vee B \leftarrow$ is deleted because of NMIN. In any case, we can derive B but not A—in the latter case we can even derive $\neg A$)—, if we apply the characterisation given in Theorem 3.3.

But the original program is certainly equivalent to $A \leftarrow B$, $B \leftarrow$, which reduces to $A \leftarrow$, $B \leftarrow$, and therefore not only B but also A is derivable.

Example 4.2. [First-Order Program]
In the last example the problem does not occur if we replace $B \vee B$ by B. But similar problems can occur for first-order programs. Suppose we have the program

$$\begin{array}{rlll} \Phi: & p(x) & \leftarrow q(x) & (1) \\ & q(x) \vee q(a) \leftarrow & & (2) \end{array}$$

and we want to replace $q(x)$ in the first clause. If we do this without taking care of the variables, we get

$$\begin{array}{ll} p(x) \vee q(a) \leftarrow & (1^*) \\ p(a) \vee q(x) \leftarrow & (1^{**}) \\ q(x) \vee q(a) \leftarrow & (2) \end{array}$$

from which $p(a)$ is not derivable, although it should! In order to make the GPPE sound, we should split the rule (2) $q(x) \vee q(a) \leftarrow$ into two rules

$$\begin{array}{rlll} p(x) & \leftarrow q(x) & & (1) \\ q(x) \vee q(a) & \leftarrow & /\ x \neq a & (2') \\ q(a) & \leftarrow & & (2'') \end{array}$$

such that we now can apply the GPPE in a sound way:

$$\begin{array}{rlll} p(x) \vee q(a) & \leftarrow & /\ x \neq a & (1') \\ p(a) \vee q(x) & \leftarrow & /\ x \neq a & (1'') \\ p(a) & \leftarrow & & (1''') \\ q(x) \vee q(a) & \leftarrow & /\ x \neq a & (2') \\ q(a) & \leftarrow & & (2'') \end{array}$$

Therefore we are forced to generalise our rules by adding *equational constraints*. These constraints are not only equations and their negations as in Prolog II [11], but also conjunctions, disjunctions and universally quantified formulae of them. See also [24, 12]. Such constructs arise when we generalise our transformations and take care of the associated constraints. In order to perform the splitting as explained above, we have to introduce a factorisation rule which will be formally defined in Definition 4.3.

By an extended version of the non-minimal rule (see Definition 5.1), we can finally reduce Φ to what one may expect, namely:

$$\begin{array}{ll} p(a) \leftarrow & (1''') \\ q(a) \leftarrow & (2'') \end{array}$$

4.1. Constraint Disjunctive Logic Programs

Equational Constraints are introduced and discussed in detail in [24, 12]. There, it is shown that the satisfiability problem for equational constraints is decidable. [12] presents also a complete simplification algorithm for equational constraints, formulated by some rewrite rules. [24] gives a more abstract quantifier elimination procedure.

We will now state the theoretical framework for computing non-ground disjunctive well-founded semantics and start with the definition of the syntax for equational constraints. The equality theory in our context is just syntactical equality. We will generalise this to arbitrary constraint domains and constraint theories in Section 4.2.

Definition 4.1. [Equational Constraints $\mathcal{R}_{eq}$]
Let V be a set of variables and F a set of function symbols. Then $T(F, V)$ denotes the set of terms over F and V as usual. We define the set of *equational constraints* $\mathcal{R}_{eq}$ as the smallest set satisfying the following properties:

- $\perp, \top \in \mathcal{R}_{eq}$.
- For all $s, t \in T(F, V)$, $s = t \ \in \ \mathcal{R}_{eq}$.
- If $R, R' \in \mathcal{R}_{eq}$ and $x \in V$, then $\neg R, R \wedge R', R \vee R', \exists x R, \forall x R$ are in $\mathcal{R}_{eq}$.

To facilitate reading, we write $s \neq t$ for $\neg(s = t)$, and $R \to R'$ for $\neg R \vee R'$. We are now able to lift the notion of disjunctive logic program to the non-ground case, by defining a *disjunctive logic program* Φ as a finite set of (not necessarily ground) rules

$$(A_1 \vee \cdots \vee A_l) \leftarrow (B_1 \wedge \cdots \wedge B_m) \wedge (\neg C_1 \wedge \cdots \wedge \neg C_n)$$

where $l \geq 1$ and $m, n \geq 0$. It is often abbreviated as $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C}$. We will identify $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ by their sets of atoms $\{A_1, \ldots, A_l\}$, $\{B_1, \ldots, B_m\}$ and $\{C_1, \ldots, C_n\}$, respectively. An atom has the form $p(t_1, \ldots, t_k)$ where $t_1, \ldots, t_k \in T(F, V)$, and p stems from a finite set of predicate symbols. The next definition introduces a further variant with equational constraints.

Definition 4.2. [Non-Ground Constraint Disjunctive Logic Program]
A *constraint disjunctive logic program* Φ (or just *constraint program* for short) is a finite set of *rule*s of the form

$$L \ / \ R$$

where L is a (not necessarily ground) disjunctive rule but without function symbols occurring in L, and R is an equational constraint.

We always assume that all constraint programs and rules are normalised according to Definition 4.5 (see below). For this, we need the definitions of two transformation rules: elimination of tautologies TAUT and factorisation FACTOR. Both rules are *local*, i.e. they do not refer to the whole rule set. Therefore they can be implemented efficiently. Thus, it makes sense to integrate them into the normalisation process. In addition, we introduce the MERGE-rule here that bundles program rules containing the same predicate symbols.

Definition 4.3. [TAUT, FACTOR and MERGE]
We first introduce elimination of tautologies, factorisation and the MERGE-rule. Using these rules we define later—in Definition 4.5—the normalisations of constraint rules and programs.

- **TAUT:** We replace the rule $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R$ by $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R \wedge (A \neq B)$ where $A = p(x_1, \ldots, x_k) \in \mathcal{A}$ and $B = p(y_1, \ldots, y_k) \in \mathcal{B}$ are atoms with the same predicate symbol. Here, $A \neq B$ means $x_1 \neq y_1 \vee \cdots \vee x_k \neq y_k$.

- **FACTOR:** Let $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R$ be a rule, and $A, B \in \mathcal{A}$ atoms with the same predicate symbol, i.e. $A = p(x_1, \ldots, x_k)$ and $B = p(y_1, \ldots, y_k)$. Then, $A = B$ means $x_1 = y_1 \wedge \cdots \wedge x_k = y_k$. The two rules

$$\begin{aligned} \mathcal{A} &\leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R \wedge (A \neq B) \\ (\mathcal{A} - \{B\}) &\leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R \wedge (A = B) \end{aligned}$$

  are the *factorisation* of $L \ / \ R$. This rule has also to be applied to atoms in $\mathcal{B}$ or $\mathcal{C}$.

- **MERGE:** Let $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R$ and $\mathcal{A}' \leftarrow \mathcal{B}' \wedge \neg\mathcal{C}' \ / \ R'$ be variants of program rules such that $\mathcal{A} = \mathcal{A}'$, $\mathcal{B} = \mathcal{B}'$, and $\mathcal{C} = \mathcal{C}'$. Then we can replace the original program rules by the following single one:

$$\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R \vee R'$$

It is important to note that the two rules which are merged by the MERGE-rule may be variants of one and the same rule. We need this in order to establish a strong notion of equivalence for irreducible and normalised constraint programs (see the following Example 4.3).

Definition 4.4. [Strong Equivalence of Programs: $\Phi \equiv \Phi'$]
Two programs Φ and Φ' are called *strongly equivalent* (written $\Phi \equiv \Phi'$) iff there is a bijection between the respective finite sets of rules such that any corresponding rules are equal. Two rules are called *equal* iff their corresponding atom sets are variants of each other, and the constraints renamed accordingly are equivalent.

Example 4.3. [Special Case of the MERGE-Rule]
Let us consider the two programs

$$\begin{aligned} \Phi_1 : p(x) \vee p(y) \leftarrow &\ \ / \ (x = a \wedge y = b) \vee (x = c \wedge y = d) \\ \Phi_2 : p(x) \vee p(y) \leftarrow &\ \ / \ (x = a \wedge y = b) \vee (x = d \wedge y = c) \end{aligned}$$

which are not equivalent in the strong sense. (This is left as an exercise to the reader.) Nevertheless, their ground instantiations are identical, namely

$$\Phi_{ground} = \{p(a) \vee p(b) \leftarrow \ \ , \ p(c) \vee p(d) \leftarrow \ \ \}$$

which means they are equivalent in a weaker sense. However, if we apply MERGE on Φ_1 and Φ_2, respectively, then we get in both cases

$$p(x) \vee p(y) \leftarrow \ \ / (x{=}a \wedge y{=}b) \vee (x{=}c \wedge y{=}d) \vee (x{=}b \wedge y{=}a) \vee (x{=}d \wedge y{=}c)$$

which means $MERGE(\Phi_1) \equiv MERGE(\Phi_2)$.

We can easily associate with each non-ground constraint disjunctive logic program Φ a possibly infinite ground program. For this, we identify each rule $L \ / \ R$ with all $L\sigma$ where σ is a solution of the constraint R, i.e. a substitution that maps the free variables in R to ground terms. But we also have to think about another problem: how can we translate a non-ground disjunctive logic program to one with constraints? The answer is given by the following definition.

We associate with each disjunctive logic program Φ a constraint disjunctive logic program Φ' as follows. For a rule in Φ, let L be the result of replacing each argument t of a predicate in L by a fresh variable x. Then Φ' contains the rule

$$L \ / \bigwedge_{x \in var(L)} (x = t)$$

for each rule in Φ. Here, $var(L)$ denotes the set of all free variables occurring in the expression L. Furthermore, each rule has to be replaced by its normalisation introduced in Definition 4.3.

Definition 4.5. [Normalisation]
A constraint rule L/R is *normalised* by

- existentially quantifying all free variables in R not occurring in L,
- applying TAUT where possible, and
- performing FACTOR as long as possible.

A constraint program Φ is *normalised* by

- normalising all rules in the program, and
- applying the MERGE-rule as long as possible.

Example 4.4. [Running Example]
Consider the program

$$\begin{array}{lrl} \Phi_0 : & p(0) & \leftarrow \\ & p(x) & \leftarrow \neg p(x) \wedge \neg p(s(x)) \\ & p(s(x)) \vee p(s(s(x))) & \leftarrow \\ & q(0) & \leftarrow p(x) \wedge \neg q(x) \end{array}$$

which is translated into

$$\begin{array}{rlll} \Phi_1: \quad p(x) & \leftarrow & / \; x = 0 & (1) \\ p(x) & \leftarrow \neg p(x) \wedge \neg p(y) & / \; y = s(x) & (2) \\ p(u) \vee p(v) & \leftarrow & / \; \exists x \; u = s(x) \wedge v = s(s(x)) & (3) \\ q(w) & \leftarrow p(x) \wedge \neg q(x) & / \; w = 0 & (4) \end{array}$$

by variable abstraction. This program is almost normalised. TAUT is not applicable anywhere in the program. FACTOR, applied on (3), does not produce new rules, since it holds $u \neq v$ because of $s(x) \neq s(s(x))$ for any x. Finally, we can apply MERGE on (3) with itself (similar as in Example 4.3), which yields

$$p(u) \vee p(v) \leftarrow \; / \; (\exists x \; u = s(x) \wedge v = s(s(x))) \wedge (\exists x \; v = s(x) \wedge u = s(s(x)))$$

In the sequel, we will omit the second part of the constraint for the ease of notation. Furthermore, we will sometimes write rules in a form without variable abstraction in order to keep notation shorter. Our infinite ground program P_{loop} from Section 3.3 can now be written easily as

$$\Phi_{loop} : p(x) \leftarrow p(y) \;\; / \; y = s(x) \quad .$$

One of the most important results of Section 3.2 was the definition of the residual program and Theorem 3.3. Using these notions we can do query answering by just looking at the *head atoms* of the residual program. In order to define an analogue of $heads(\Phi)$ for constraint programs, we need the following definitions.

Definition 4.6. [Projection]
Let $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C}/R$ be a constraint rule, and $p(x_1, \ldots, x_k)$ a literal in $\mathcal{A} \cup \mathcal{B} \cup \mathcal{C}$. Then we call $p(x_1, \ldots, x_k)/\exists y_1 \cdots \exists y_n \; R$, where $\{y_1, \ldots, y_n\} = var(R) - \{x_1, \ldots, x_k\}$, the *projection* of the given rule wrt. $p(x_1, \ldots, x_k)$.

Definition 4.7. [$heads(\Phi)$]
For a constraint program Φ, we define three sets of projections $project_X(\Phi)$, where X may be $\mathcal{A}$, $\mathcal{B}$ or $\mathcal{C}$, by the following procedure. We identify the *possibly true facts*, written $heads(\Phi)$, with $project_{\mathcal{A}}(\Phi)$.

- Take the set of all possible projections wrt. the literals occurring in program rules of Φ in part X (i.e. in heads, in rule bodies or as negative conditions, respectively).
- Normalise this set; in this context, it has to be understood as a constraint program consisting of constraint atoms only.
- For all predicate symbols p (with arity k) that are not used so far, add $p(x_1, \ldots, x_k)/\perp$ to the set.

Note that $heads(\Phi)$ is always finite because Φ is finite. For our Example 4.4, the possibly true facts are as follows. The constraints are given in an already simplified form here.

$$\begin{array}{ll} p(x) & / \;\; x = 0 \vee \exists y \; y = s(x) \vee \exists z \; x = s(z) \vee \exists z \; x = s(s(z)) \\ q(y) & / \;\; y = 0 \end{array}$$

4.2. Reasoning with Arbitrary Constraint Theories

Until now we only considered equational constraints according to Definition 4.1. But our framework can easily be generalised for arbitrary constraint systems and theories. Thus, we do not restrict to equality constraints over the Herbrand domain. We may consider other domains and other constraints predicates besides $=$. Surprisingly, all definitions (including the transformation rules introduced in Section 5) can also be applied to the general case. Here is the general definition of a constraint system.

Definition 4.8. [Constraint System]
A constraint system, often identified by its constraint theory $\mathcal{T}$—see below—consists of four components:

- D: a constraint domain, that is a non-empty set,
- Δ: an alphabet, including a set V of variables, a set F of function symbols, and a set P of predicate symbols, containing at least the predicate $=$, which must be interpreted as the equality relation on D, and the truth constants $\bot$ and $\top$ with the meaning “false” and “true”, respectively,
- $\mathcal{R}$: the set of all Δ-formulae, called constraints in our context, including all atomic Δ-formulae and closed under all logical connectives, in particular negation $\neg$, and
- $\mathcal{T}$: the constraint theory, which can be identified with a Δ-interpretation over the domain D, since we assume $\mathcal{T}$ to be *satisfaction-complete*, i.e. for all constraint formulae $R \in \mathcal{R}$ either $\mathcal{T} \models R$ or $\mathcal{T} \models \neg R$ holds.

The main difference to conventional CLP is the requirement that the set of constraint-formulae $\mathcal{R}$ must be closed not only under conjunction $\wedge$, existential quantification $\exists$ and instantiation of variables—as stated in [20]—but under all logical connectives and quantifiers. We will see that this is necessary because of the first-order versions of the NMIN, RED$^+$ and RED$^-$ rules in Definition 5.1. Thus, the respective constraint solvers must be able to treat negative constraints of the considered constraint domain. For example, we could do non-monotonic reasoning with integer domain constraints. Example 4.4 can easily be viewed as such a problem:

$$
\begin{array}{rll@{\quad}r}
p(x) & \leftarrow & /\ x = 0 & (1) \\
p(x) & \leftarrow \neg p(x) \wedge \neg p(y) & /\ y = x + 1 & (2) \\
p(u) \vee p(v) & \leftarrow & /\ u \geq 1 \wedge v = u + 1 & (3) \\
q(w) & \leftarrow p(x) \wedge \neg q(x) & /\ w = 0 & (4)
\end{array}
$$

Now we are also in a position to extend our notion of a semantics for ground programs (Definition 2.2) straightforwardly to one for constraint disjunctive programs. We say that a constraint disjunctive rule $\mathcal{A}\ /\ R$ *subsumes* another rule $\mathcal{A}'\ /\ R'$ if there is a variant $\mathcal{A}''\ /\ R''$ of $\mathcal{A}\ /\ R$ such that $\mathcal{A}'' \subseteq \mathcal{A}'$ and $\mathcal{T} \models \forall(R'' \rightarrow R')$.

Definition 4.9. [Operator $\mid\!\sim$, Semantics $\mathcal{S}_{\mid\sim}$]
By a semantic operator $\mid\!\sim$ we mean a binary relation between constraint disjunc-

tive logic programs and pure constraint disjunctions which satisfies the following three conditions:

(1) **Right Weakening:** If $\Phi \mid\!\sim L \ / \ R$ and $L \ / \ R$ subsumes $L' \ / \ R'$, then $\Phi \mid\!\sim L' \ / \ R'$.

(2) **Necessarily True:** If $(\mathcal{A} \leftarrow \ \ / \ R) \in \Phi$ for a some disjunction $\mathcal{A} \ / \ R$, then $\Phi \mid\!\sim \mathcal{A} \ / \ R$.

(3) **Necessarily False:** Let $A \ / \ R$ be a constraint atom. If there is a variant A/R' of an atom in $heads(\Phi)$ such that $\mathcal{T} \models \forall(R \rightarrow \neg R')$, then $\Phi \mid\!\sim \neg A/R$.

Given such an operator $\mid\!\sim$ and a constraint disjunctive program Φ, by the semantics $\mathcal{S}_{\mid\sim}(\Phi)$ of Φ (determined by $\mid\!\sim$) we mean the set of all constraint pure disjunctions derivable by $\mid\!\sim$ from Φ, i.e. $\mathcal{S}_{\mid\sim}(\Phi) := \{L \ / \ R \mid \Phi \mid\!\sim L/R\}$.

4.3. Compactness of Constraint Theories

An important feature while handling constraints is to *simplify* them, i.e. rewriting them into "simpler" ones while preserving equivalence. Since the underlying constraint theory is always assumed to be monotonic, simplification can be performed at any time. If a constraint R in a rule reduces to $\perp$, then the respective rule can be discarded.

We discuss now in more detail the constraint theory $\mathcal{T}_{eq}$ for equality constraints as an instance of Definition 4.8. After that, we will distinguish some classes of constraint theories by introducing several notions of compactness, since this affects the complexity and termination of our overall procedure. See also Section 5.3.

The domain D for $\mathcal{T}_{eq}$ is the set of all ground terms over the set F of function symbols. The set of constraints $\mathcal{R}_{eq}$ and implicitly the alphabet Δ has already been defined in Definition 4.1. Now, $=$ has to be interpreted as syntactical equality. This can be axiomatised as follows. We distinguish two cases:

- If the set of function symbols F is infinite, then Clark's axiomatisation of the Herbrand domain yields a satisfaction-complete constraint theory for equational constraints. In this case, basic equational constraints—i.e. unification equations of the form $s = t$, but not general equational constraints—enjoy the *independence of negative constraints* property:

  $$\mathcal{T}_{eq} \models \exists(R \wedge \neg R_1 \wedge \cdots \wedge \neg R_k) \quad \text{iff} \quad \mathcal{T}_{eq} \models \exists(R \wedge \neg R_i) \text{ for } 1 \leq i \leq k \ .$$

  This is equivalent to the *strong compactness property* which is just the contraposition of the proposition stated above. In the CLP context, this means that we can restrict to only one derivation while answering any query, provided that we do not have any inequalities. In our context, it means that in certain cases we can do it without the MERGE-rule. For details, the reader is referred to [25].

- If we assume the set of function symbols F to be finite, then we need the *weak domain closure axiom* in order to make our theory complete [24]:

  $$\forall x \bigvee_{f \in F} \exists z \ x = f(z)$$

This case is mainly considered in [12]. But this point of view is restricted to Herbrand models only and increases the computational complexity to Π_1^1. Therefore, we will assume that F is infinite in the sequel.

Nevertheless, the problem of solving equational constraints is of high complexity: it is known to be PSPACE-hard. Let us now introduce another notion of compactness.

Definition 4.10. [Constraint-Compactness]
A constraint theory $\mathcal{T}$ is called *constraint-compact* if for any set of constraints $\mathcal{S} \subseteq \mathcal{R}$ with only finitely many distinct variables occurring in $\mathcal{S}$, there are finitely many constraints $S_1, \ldots, S_n \in \mathcal{S}$ such that, for all $S \in \mathcal{S}$, it holds:

$$\mathcal{T} \models \forall (S \rightarrow (S_1 \vee \cdots \vee S_n))$$

Constraint systems with this property are exploited by Toman in [32]. Constraint-compactness turns out to be crucial for the termination of our constraint calculus (see Section 5.3). Thus our termination results generalise not only our own results for the ground case of D-WFS (see Theorem 3.1) but also the corresponding results in [32] which are proved there just for normal Datalog$^{\neg}$ programs. Some constraint theories are constraint-compact, but unfortunately, equality constraints do not have this property. The following set of constraints

$$\{x = t \mid t \in D\}$$

is a simple counterexample.

5. A CALCULUS FOR CONSTRAINT PROGRAMS

We have now introduced all necessary machinery to extend our original calculus to constraint programs in Section 5.1: Our main results are Theorems 5.1 and 5.2. In Section 5.2 we consider the computation of the residual program (Theorem 5.4), *query answering* (Theorem 5.5) and we give an approximation of Theorem 3.3 (Theorem 5.6). Finally, in Section 5.3 we consider termination of our calculus for arbitrary Datalog$^{\vee,\neg}$ programs.

5.1. Extending our Original Transformations

We are now able to state the transformation rules for the new calculus with constraints. We will state the rules in a very abstract way, i.e. without mentioning control of the applicability of the rules. But it is clear that our procedure cannot always terminate because of the undecidability of the underlying problem. Note that our former elimination of tautologies rule will be incorporated into GPPE by normalisation.

Definition 5.1. [Transformation Rules]

If

$$\mathcal{A} \leftarrow \mathcal{B} \wedge \neg \mathcal{C} \;/\; R$$

is a rule in the constraint program Φ, then it can be replaced by one of the following (sets of) rules. Constraint simplification can be applied immediately on each newly generated rule.

- **GPPE:** (GPPE plus normalisation of rules)
  Let B be a distinguished atom in $\mathcal{B}$. Then, replace the rule $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R$ by

$$\begin{array}{c} \mathcal{A} \cup (\mathcal{A}_1 - \{B\}) \leftarrow (\mathcal{B} - \{B\}) \cup \mathcal{B}_1 \wedge \neg(\mathcal{C} \cup \mathcal{C}_1) \ / \ R \wedge R_1 \\ \vdots \\ \mathcal{A} \cup (\mathcal{A}_k - \{B\}) \leftarrow (\mathcal{B} - \{B\}) \cup \mathcal{B}_k \wedge \neg(\mathcal{C} \cup \mathcal{C}_k) \ / \ R \wedge R_k \end{array}$$

  where each $\mathcal{A}_i \leftarrow \mathcal{B}_i \wedge \neg\mathcal{C}_i \ / \ R_i$, for $1 \leq i \leq k$, is a variant of a rule in Φ that contains an atom A in its head with the same predicate symbol as B, such that $A = B$ and all variables occurring in other literals are renamed apart. Each of the generated rules is replaced immediately by its normalisation.

- **NMIN:** (Extended Elimination of Non-Minimal Rules)
  Replace the rule $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R$ by

$$\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R \wedge \neg R'$$

  for some variant $\mathcal{A}' \leftarrow \mathcal{B}' \wedge \neg\mathcal{C}' \ / \ R'$ of a rule in Φ such that $\mathcal{A}' \subseteq \mathcal{A}$, $\mathcal{B}' \subseteq \mathcal{B}$ and $\mathcal{C}' \subseteq \mathcal{C}$ hold.

- **RED$^+$:** (Extended Positive Reduction)
  Replace the rule $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R$ by the two rules:

$$\begin{array}{l} \mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R \wedge R' \\ \mathcal{A} \leftarrow \mathcal{B} \wedge \neg(\mathcal{C} - \{C\}) \ / \ R \wedge \neg R' \end{array}$$

  where C is an atom in $\mathcal{C}$ and C'/R' is a variant of a constraint atom in $heads(\Phi)$ such that $C = C'$.

- **RED$^-$:** (Extended Negative Reduction)
  Replace the rule $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R$ by:

$$\mathcal{A} \leftarrow \mathcal{B} \wedge \neg\mathcal{C} \ / \ R \wedge \neg R'$$

  for some variant $\mathcal{A}' \leftarrow \ / \ R'$ of a rule in Φ such that $\mathcal{A}' \subseteq \mathcal{C}$.

We will now state a derivation for Example 4.4. We start with RED$^-$ on (2) with clause (3). This causes the replacement of (2) by:

$$p(x) \leftarrow \neg p(x) \wedge \neg p(y) \ / \ y = s(x) \wedge \neg\exists z(x = s(z) \wedge y = s(s(z))) \qquad (2')$$

After applying the non-minimal rule with (1) which is essentially subsumption, the constraint can be simplified to $y = s(x) \wedge x \neq 0 \wedge \forall z\ x \neq s(z)$. This could be further simplified to $\perp$ if we assume that F is finite, for example $F = \{0, s\}$.

Since for the predicate q only $q(0)$ is in the set of possibly true facts, we can replace (4) with

$$q(w) \leftarrow p(x) \wedge \neg q(x) \ / \ w = 0 \wedge x = 0 \qquad (4')$$

by RED$^+$. Applying the GPPE on $p(x)$ (where $x = 0$) we get finally:

$$q(w) \leftarrow \neg q(x) \ / \ w = 0 \wedge x = 0 \qquad (4'')$$

When the reader compares Definition 5.1 with the corresponding ones in Section 3.1, he will notice that TAUT has been integrated into the GPPE, and RED$^+$ has been turned into a more flexible incremental rule. Unfortunately, neither our constraint calculus (see Definition 5.1) nor the version for infinite ground programs without the Loop-Detection Rule is confluent. The following Example 5.1 reveals this fact. Nevertheless, the calculus is weakly confluent as stated in Theorem 5.1.

Example 5.1. [No Local Confluence]
Consider the following disjunctive logic program. Note that we omit some brackets for better readability and to abbreviate notation.

$$\Phi : \quad p(x) \leftarrow p(fx) \wedge p(gx).$$

Applying the GPPE to the first literal in Φ yields

$$\Phi_1 : \quad p(x) \leftarrow p(ffx) \wedge p(gfx) \wedge p(gx).$$

Let us now consider another derivation starting from Φ. If we apply the GPPE to the second literal in Φ, we get

$$\Phi_2 : \quad p(x) \leftarrow p(fx) \wedge p(fgx) \wedge p(ggx).$$

In both cases, only the GPPE can be applied again and again. We observe that in Φ_1 we cannot get rid of the literal $p(gfx)$ and in Φ_2 we cannot get rid of $p(fgx)$ in the body. This means, our calculus is *not* confluent (not even locally confluent): there is no Φ' that can be reached both from Φ_1 and from Φ_2.

However, there does not exist an irreducible program at all for this example. This example cannot be simulated for the infinite ground case, because of our Loop detection rule which ensures termination. But without it, we get the same problem.

Nevertheless, a weak confluence property holds, which we are going to prove now. For this, we relate a model set to each constraint program Φ in Definition 5.2. Some of these models are partly inconsistent, i.e. there may be ground atoms in the models which are both true and false at the same time. After showing the Lemmata 5.1 and 5.2, we are able to prove weak confluence in Theorem 5.1.

Definition 5.2. [Model Set of a Constraint Program]
Let Φ' be the ground instantiation of the constraint program Φ. Let M be the set of all possible ground atoms, and $M^-, M^+ \subseteq M$. Then, (M^-, M^+) is called *minimal model* of Φ wrt. M^- iff M^+ is minimal wrt. set inclusion such that, for all rules $\mathcal{A} \leftarrow \mathcal{B} \wedge \neg \mathcal{C}$ in Φ', $\mathcal{B} \subseteq M^+$ and $\mathcal{C} \subseteq M^-$ imply $\mathcal{A} \cap M^+ \neq \{\}$. The *model set* $\mathcal{M}$ of Φ is the maximal set of minimal models of Φ obtained by repeatedly applying the following procedure, until the greatest fixpoint is reached:

(1) If a literal $l \notin M^+$ for all $(M^-, M^+) \in \mathcal{M}$, then remove all $(M^-, M^+) \in \mathcal{M}$ with $l \notin M^-$ from $\mathcal{M}$.

(2) If for a set of literals L it is $L \cap M^+ \neq \{\}$ for all $(M^-, M^+) \in \mathcal{M}$, then remove all $(M^-, M^+) \in \mathcal{M}$ with $L \subseteq M^-$ from $\mathcal{M}$.

Lemma 5.1.
The model set of Φ is the same as the one of $Trans(\Phi)$ for any transformation $Trans$ according to Definition 5.1.

PROOF. Let us make a case distinction:

GPPE: GPPE just evaluates a positive atom or removes a tautological rule (via normalisation). But removing a tautology does not affect the model set of Φ. Partial evaluation does not change the model set too as shown in [5].

NMIN: It is trivial, that an application of NMIN does not change the model set of Φ.

RED$^+$: Since it cannot be the case that a literal $l \in M^-$, if $l \notin M^+$ for all $(M^-, M^+) \in \mathcal{M}$—because of step (1) from Definition 5.2, RED$^+$ does not change the model set.

RED$^-$: In this case, step (2) from Definition 5.2 is the reason, why the model set is preserved under RED$^-$.

This completes the proof. □

Lemma 5.2.
Let Φ_1 and Φ_2 be two irreducible constraint programs. Then, $\Phi_1 \equiv \Phi_2$ iff Φ_1 and Φ_2 have the same model sets.

PROOF. The direction from left to right is straightforward, since strongly equivalent programs clearly have identical ground instantiations. Let us now prove the contraposition of the direction from right to left, i.e. $\Phi_1 \not\equiv \Phi_2$ implies Φ_1 and Φ_2 have different model sets. Thus, let $\mathcal{A} \leftarrow \neg\mathcal{C}$ be a conditional fact in the ground instantiation of Φ_1 without loss of generality, such that there is no rule $\mathcal{A}' \leftarrow \neg\mathcal{C}'$ in Φ_2, where (a) $\mathcal{A}' \subseteq \mathcal{A}$ and $\mathcal{C}' \subseteq \mathcal{C}$, or (b) $\mathcal{A}' \subseteq \mathcal{C}$ and $\mathcal{C}' = \{\}$.

Now, there is a minimal model (M^-, M^+) of Φ_2 where $\mathcal{A} \cup \mathcal{C} \subseteq M^-$ holds, because conditions (a) and (b) ensure that there is a minimal model, taking step (2) of Definition 5.2 into account. The other step (1) is not critical, because it can only increase M^-. However, (M^-, M^+) is not a minimal model of Φ_1, because $\mathcal{A} \leftarrow \mathcal{C}$ cannot be valid. This means, the model sets are different. □

Theorem 5.1. [Weak Confluence]
Each constraint program Φ has at most one irreducible form, i.e. $\Phi \xrightarrow{*} \Phi_1$ and $\Phi \xrightarrow{*} \Phi_2$, where Φ_1 and Φ_2 are irreducible, imply $\Phi_1 \equiv \Phi_2$.

PROOF. By Lemma 5.1, the model sets of Φ, Φ_1 and Φ_2 are identical. By Lemma 5.2, it follows $\Phi_1 \equiv \Phi_2$, because Φ_1 and Φ_2 are irreducible. □

Theorem 5.2. [CD-WFS]
For the class of all constraint disjunctive programs Φ that possess an irreducible

program Φ' with $\Phi \xrightarrow{*} \Phi'$, there exists the weakest semantics $\mathcal{S}_{\mid\sim}$ which is invariant under GPPE, NMIN, RED$^+$ and RED$^-$. We call it the Constraint Disjunctive Well-Founded Semantics, or, briefly, CD-WFS and abbreviate it by $\mid\!\sim_{\text{CD-WFS}}$.

PROOF. Like in the proof of Theorem 3.2 (given in [7]), it is easy to see that for any two semantics $\mathcal{S}_{\mid\sim}$ and $\mathcal{S}'_{\mid\sim}$ satisfying all our properties, their intersection $\mathcal{S}_{\mid\sim} \cap \mathcal{S}'_{\mid\sim}$ also satisfies all our properties. Therefore a weakest semantics must exist and is just the intersection of all possible semantics. □

We could have also defined the semantics on the class of all programs. The reason we did not so is because then the semantics would be too weak compared with WFS or GCWA. For the program Φ_{loop} it would not derive anything, because Φ_{loop} does not have an irreducible program below it. Note that we have not stated a Loop-Detection rule as we did for infinite ground programs (Definition 3.4). Of course, we aim to extend our definition to a larger class of programs. CD-WFS can be seen as a constructive approximation of D-WFS (see Theorem 5.6).

Suppose a program Φ can be transformed into an irreducible program $res(\Phi)$. An important problem is if any other sequence of transformations applied to Φ will necessarily also lead to $res(\Phi)$. Or can it happen that a *wrong choice* will lead to an infinite derivation $\Phi_1, \Phi_2, \ldots, \Phi_n, \ldots$ such that no Φ_i is irreducible? In fact, this is possible as the following program shows.

Example 5.2. [Infinite Derivation]
We consider a modification of the Φ_{loop} program (introduced after Example 4.4).

$$\begin{array}{rll} \Phi : p(x) \leftarrow p(y), \neg a, \neg b & / & y = s(x) \\ a \vee b \leftarrow & / & \top \end{array}$$

It is possible to apply GPPE again and again on the first clause: this gives an infinite derivation producing programs $\Phi_1, \Phi_2, \ldots$. But obviously, an application of negative Reduction RED$^-$ allows to delete the first clause and we get the residual program $a \vee b$.

In the last example, we can reach the residual program from any of the Φ_i by applying RED$^-$. This means we never reach a program from which the irreducible program is unreachable. This immediately leads to the conjecture that once a program Φ can be transformed into an irreducible program, there is no *wrong* sequence of transformations, i.e. no sequence which leads to a dead-end. This is an important property, because it expresses that during a derivation no backtracking is needed.

Theorem 5.3. [Calculus needs no Backtracking]
Let Φ be a program such that there exists an irreducible program $res(\Phi)$. Let Φ' be any other program that can be obtained from Φ. Then $res(\Phi)$ can be obtained from Φ' by a suitable sequence of transformations.

PROOF. The only way to get an infinite derivation is by applying GPPE on program rules which contain a predicate A both in the head and in the body (this was the situation in Example 5.2). We have to show how such rules can be eliminated by looking at the derivation leading to the residual program. Eliminating rules is only possible with NMIN or RED$^-$.

Let $r_{\Phi'}$ be such a program rule. It must have been derived from a rule r_Φ from Φ by certain GPPE-steps. Since an irreducible program can be reached from Φ, there is a sequence of transformations σ which eliminates a partial evaluation of r_Φ. This can be done either by RED^- or by NMIN.

If RED^- is applied, then we can partial evaluate those atoms in $r_{\Phi'}$ that have been partial evaluated in the sequence σ, and we can remove the rule by RED^- (note that all GPPE-steps *increase* the number of negative body-atoms and therefore negative reduction can still be applied).

If NMIN is applied, then we have to show that the smaller rule used to remove the partial evaluation of r_Φ can also be reproduced from Φ' and thus NMIN can also be applied here. It suffices of course that we can reproduce an even smaller rule. That this is indeed possible can be shown using the following claim:

Every rule in $res(\Phi)$ can be obtained from Φ'.

Let a rule r_{res} from $res(\Phi)$ be given. It is obtained from a rule r of Φ by a sequence σ of GPPE and RED^+ steps. But Φ' must then contain a partial evaluation of r, let us call it r'. Then it is easy to show that r' can be extended (by applications of GPPE and RED^+ using the original sequence σ) to the rule r_{res}. □

5.2. Residual Program and Query Answering

Now we can define a constructive approximation of D-WFS for non-ground programs. Since our procedure is not guaranteed to terminate we cannot always compute the residual program (see Definition 5.3). But when it exists, we can do *query answering* similar as in the case of D-WFS (Theorem 3.3). Before doing so, let us summarise the result of applying the transformation to our example. We end with an irreducible program.

$$
\begin{array}{rlll}
res(\Phi): \quad p(x) & \leftarrow & / \; x = 0 & (1) \\
p(x) & \leftarrow \neg p(x) \wedge \neg p(y) & & \\
 & & / \; y = s(x) \wedge x \neq 0 \wedge \forall z \; x \neq s(z) & (2'') \\
p(u) \vee p(v) & \leftarrow & / \; \exists x : u = s(x) \wedge v = s(s(x)) & (3) \\
q(w) & \leftarrow \neg q(x) & / \; w = 0 \wedge x = 0 & (4'')
\end{array}
$$

Definition 5.3. [Residual Program, Query]

- A constraint program Φ is called a *residual program* iff no transformation rules are applicable to it, i.e. it is irreducible.
- A *query* ψ is a pure disjunction of literals. We identify it with its translation into the rule $L \; / \; R$ by variable abstraction.

Note that in a residual program, no positive body literals can occur. Otherwise, GPPE would be applicable.

Theorem 5.4. [Computation of CD-WFS]
Let Φ be a (variable abstraction of a disjunctive) program with the residual program $res(\Phi)$, and $\psi = L \; / \; R$ be a pure constraint disjunction. Then

$$\Phi \mid\!\sim_{\mathrm{CD-WFS}} \psi$$

iff one of the following two cases applies

(1) there is a rule $(\mathcal{A} \leftarrow\ )\ /\ R'$ in $res(\Phi)$ which subsumes ψ.

(2) L contains a negative literal $\neg A$ such that there exists a variant A/R' of an atom in $heads(res(\Phi))$ and $\forall(R \rightarrow \neg R')$.

PROOF. By definition, we have

$$\Phi \mid\!\sim_{\mathrm{CD-WFS}} \psi \quad \text{iff} \quad res(\Phi) \mid\!\sim_{\mathrm{CD-WFS}} \psi.$$

Now let us define a semantics $\mathcal{S}_{\mid\sim}$ on all programs possessing a residual program by just the two cases mentioned above. This means we look at the residual program and include a positive disjunction in $\mathcal{S}_{\mid\sim}$, if the first case applies and we include a negative disjunction if the second case applies.

We have to show that CD-WFS and $\mathcal{S}_{\mid\sim}$ are identical. By Definition 4.9, *Right Weakening* and *Necessarily True* imply that all positive disjunctions contained in $\mathcal{S}_{\mid\sim}$ are also contained in CD-WFS. *Necessarily False* of Definition 4.9 implies that also all negative disjunctions contained in $\mathcal{S}_{\mid\sim}$ also belong to CD-WFS. Thus CD-WFS is at least as strong as $\mathcal{S}_{\mid\sim}$.

But since CD-WFS is defined as the *least* semantics satisfying our properties, it cannot be stronger than $\mathcal{S}_{\mid\sim}$ and therefore must coincide with it. □

In the last theorem we started with checking a statement $\psi = L\ /\ R$ where R is a set of constraints. A more natural setting is given when we ask a query ψ with free variables and try to instantiate these variables in order to find solutions. Such query-answering can also be done.

Theorem 5.5. [Query Answering]
Let Φ be a (variable abstraction of a disjunctive) program with the residual program $res(\Phi)$, and ψ be a query with variable abstraction $L\ /\ R$. Then

$$\Phi \mid\!\sim_{\mathrm{CD-WFS}} \psi[\delta]$$

iff δ is a solution of one of the following two constraints R^+ and R^-.

(1) There is a rule $(\mathcal{A} \leftarrow\ )\ /\ R$ in $res(\Phi)$, and $\mathcal{A} \subseteq L_1 \cup \cdots \cup L_k$ where $L_1\ /\ R_1, \ldots, L_k\ /\ R_k$ are different variants of ψ and the constraint $R^+ = (R_1 \wedge \cdots \wedge R_k) \wedge R$ is satisfiable.

(2) ψ contains a negative literal $\neg A$ such that there exists a variant A/R' of an atom in $heads(res(\Phi))$ and $\forall(R \rightarrow \neg R')$. Then $R^- = R \wedge \neg R'$ is satisfiable.

PROOF. For the direction from right to left, it is clear by Theorem 5.4 that in case 1 the desired subsumption relation holds with R^+. In case 2, we set $R^- = R \wedge \neg R'$ and get $\forall(R \rightarrow \neg R')$.

The direction from left to right is also clear by Theorem 5.4. In case 1, there cannot be more instances of the positive disjunctions. In case 2, only one variant of ψ suffices, since the characterisation for negative disjunctions refers to only one literal. □

The solutions δ of the constraints are called *answers* of the query. If in the former case $k > 0$ holds, then the answer is called *disjunctive* or *indefinite*. For our example it holds $p(0)$ and $\neg q(s(z))$, but neither $q(0)$ nor $\neg q(0)$ because $(4'')$ is a conditional fact. Sometimes it is not sufficient to look for only one rule while answering queries. For example, the query

$$p(x) \vee p(y)\ /\ (x = 0 \wedge y = 0) \vee \exists z(x = s(z) \wedge y = s(s(z)))$$

fails, although one should expect one positive answer. But we can split the query by factorisation and constraint simplification into:

$$p(x) \vee p(y) \;\;/\;\; x = 0 \wedge y = 0$$
$$p(x) \vee p(y) \;\;/\;\; \exists z(x = s(z) \wedge y = s(s(z))) \;\; \wedge x \neq y$$

Then the former query has the answer $x = 0 \wedge y = 0$ and the latter $\exists z(x = s(z) \wedge y = s(s(z)))$. Both answers can be combined to the desired disjunctive constraint of the original query. Nevertheless, the MERGE-rule helps us to bundle answers.

Example 5.3. [Bundling Answers Using MERGE]
Consider the program:

$$\begin{array}{ll} p(x) \leftarrow & (1) \\ p(a) \leftarrow & (2) \end{array}$$

If we apply NMIN on (1) and variable abstraction on (2) we get:

$$\begin{array}{ll} p(x) \leftarrow \;/\; x \neq a & (1') \\ p(x) \leftarrow \;/\; x = a & (2') \end{array}$$

But this would give us two answers for the query $p(z)$. However, the MERGE-rule yields

$$p(x) \leftarrow \;\;/\;\; x = a \vee x \neq a$$

which is clearly equivalent to (1) after constraint simplification. This is exactly what one should expect, because (2) can be deleted from the original program by NMIN on (2) with (1). Thus, the query $p(z)$ can now be answered positively by looking at the only remaining rule (1).

Theorem 5.6. [CD-WFS Related to Other Semantics]
Let Φ be a first-order disjunctive logic program, and ψ be a purely positive or negative clause. We identify Φ with its variable abstracted form, i.e. with its corresponding constraint program. We also assume that $res(\Phi)$ exists. Then the following holds:

(1) Let Φ_{inst} be the possibly infinite instantiation of Φ and ψ_{inst} the set of all ground instances of ψ. Then $\Phi \mid\!\sim_{\mathrm{CD-WFS}} \psi$ iff $\Phi_{inst} \mid\!\sim_{\mathrm{D-WFS}} \psi_{inst}$. In particular, if Φ is a ground program (i.e. there are only nullary predicate symbols in Φ), then $\Phi \mid\!\sim_{\mathrm{CD-WFS}} \psi$ iff $\Phi \mid\!\sim_{\mathrm{D-WFS}} \psi$).

(2) If Φ is a positive disjunctive program (i.e. there are no negative literals in the body of any rule in Φ), then $\Phi \mid\!\sim_{\mathrm{CD-WFS}} \psi$ iff $\Phi \mid\!\sim_{\mathrm{GCWA}} \psi$.

(3) If Φ is a normal program (i.e. any rule in Φ contains exactly one literal in its head), then $\Phi \mid\!\sim_{\mathrm{CD-WFS}} \psi$ iff $\Phi \mid\!\sim_{\mathrm{WFS}} \psi$.

PROOF.

(1) We note first that the existence of $res(\Phi)$ implies the existence of $res(\Phi_{inst})$ as introduced in Section 3.3, Theorem 3.4, where we allow infinite applications of our rules in the ground case. Therefore and because of the definition

of CD-WFS and D-WFS, it suffices to show

$$res(\Phi) \mathrel{|\!\!\sim}_{\mathrm{CD-WFS}} \psi \quad \text{iff} \quad res(\Phi_{inst}) \mathrel{|\!\!\sim}_{\mathrm{D-WFS}} \psi_{inst}.$$

Moreover, we have

$$(res(\Phi))_{inst} = res(\Phi_{inst}),$$

so that (because $(res(\Phi))_{inst} \mathrel{|\!\!\sim}_{\mathrm{CD-WFS}} \psi_{inst}$ iff $res(\Phi) \mathrel{|\!\!\sim}_{\mathrm{CD-WFS}} \psi_{inst}$ for all ground Ψ_{inst}) we finally get

$$res(\Phi) \mathrel{|\!\!\sim}_{\mathrm{CD-WFS}} \psi_{inst} \quad \text{iff} \quad res(\Phi_{inst}) \mathrel{|\!\!\sim}_{\mathrm{CD-WFS}} \psi_{inst}.$$

But $res(\Phi_{inst}) \mathrel{|\!\!\sim}_{\mathrm{CD-WFS}} \psi_{inst}$ iff $res(\Phi_{inst}) \mathrel{|\!\!\sim}_{\mathrm{D-WFS}} \psi_{inst}$ because both relations are computed according to the same evaluation method (see Theorem 3.3 as well as Theorem 5.4).

(2/3) It is easy but quite tedious to verify that both GCWA and WFS satisfy all our properties. This has been proved for propositional programs in [7] and the corresponding proofs for first-order constraint programs are very similar. Therefore it remains to show

$$\begin{aligned}(2) : \Phi \mathrel{|\!\!\sim}_{\mathrm{CD-WFS}} \psi \;\; \text{iff} \;\; \Phi \mathrel{|\!\!\sim}_{\mathrm{GCWA}} \psi \\ (3) : \Phi \mathrel{|\!\!\sim}_{\mathrm{CD-WFS}} \psi \;\; \text{iff} \;\; \Phi \mathrel{|\!\!\sim}_{\mathrm{WFS}} \psi\end{aligned}$$

We first discuss (2). Since $res(\phi)$ consists of constraint disjunctions (bodies are all empty), GCWA entails exactly those disjunctions as characterised by CD-WFS using Theorem 5.4.

Let us now consider (3). Here, $res(\phi)$ consists of rules of the form

$$A \leftarrow \neg\mathcal{C} \;\; / \;\; R,$$

where again no positive body-atoms are present. To show (3), we choose the 3-valued computation of WFS due to Przymusinski (see also [13, Lemma 3.9]). This procedure works, roughly said, by *reducing* the given program. A reduction consists of replacing first all atoms A with a rule $A \leftarrow \; \in P$ (those that are true), and then replacing all $\neg B$ occurrences if there are no rules with head B. The reduced program may give rise to further reductions in the next round.

But since our program is in normal form, it is immediate that no reductions can be applied: such reductions would be applications of RED$^{\pm}$. Therefore exactly those literals as characterised by CD-WFS using Theorem 5.4 are entailed by WFS. □

5.3. Termination

Although the constraint calculus cannot always terminate in general, it terminates for special classes of constraints, namely for constraint-compact constraint theories. In Section 3.2, we remarked that termination of the ground calculus can only be guaranteed for *fair* derivation sequences. Let us try transferring this result to the constraint calculus. At first, we note that the GPPE is never applied to a tautological clause, because TAUT is built into the GPPE in Definition 5.1.

However, since there may be infinitely many different ground instances of constraint atoms in rule bodies, it is not obvious how we can systematically get rid of all these atoms. Recall that, in the ground case, we can do this by insisting on the following: if the GPPE has been applied to get rid of an occurrence of atom A, then all other occurrences of A in bodies have to be removed before the GPPE is applied to other atoms $B \neq A$. We are now going to show that this idea can be adapted for the constraint calculus, provided the considered constraint theory is constraint-compact.

After a careful inspection of Definition 5.1, we observe that NMIN, RED$^+$ and RED$^-$ only "delete" atoms, because, for all these rules, existing constraints are always restricted further (by conjunction with other constraints). Let us state this more formally:

Lemma 5.3.
If we consider constraint atoms $A/R \in project_X(\Phi)$ and $A/R' \in project_X(\Phi')$ (for $X = \mathcal{A}, \mathcal{B}, \mathcal{C}$), where the constraint program Φ' is obtained from the original program Φ by a sequence of transformations (except GPPE), then it holds $\mathcal{T} \models \forall(R \rightarrow R')$. This means, the projection sets decrease monotonically.

The GPPE does not always decrease the projection sets, it never increases it nevertheless. However, since each constraint-compact constraint theory can have only a finite domain—as the following Example 5.4 shows—, after at most finitely many applications of the GPPE they will decrease. But we have to take the right control discipline.

Theorem 5.7. [Weak Termination for Datalog$^{\vee,\neg}$]
Let Φ be a constraint program wrt. a constraint-compact constraint theory. Then, there always exists an irreducible constraint program Φ' with $\Phi \xrightarrow{*} \Phi'$. This means, the calculus from Definition 5.1 is weakly terminating.

PROOF. We can systematically get rid of all body literals by the following control discipline for the GPPE: take maximally many variants of projections of positive body literals $A/R_1, \ldots, A/R_n$ in Φ, such that $R = \exists(R_1 \wedge \cdots \wedge R_n)$ is consistent, and replace all the respective atoms in program rules by the GPPE, before applying the GPPE to other body literals. This works eventually, since once a constraint atom is deleted, it cannot come in again.

By this procedure we eventually arrive at an irreducible program after finitely many transformation steps, because there are no infinite sequences of constraints with (a) $R_1 \rightarrow R_2 \rightarrow \cdots \rightarrow R_n \rightarrow \cdots$ and (b) $R_1 \not\leftarrow R_2 \not\leftarrow \cdots \not\leftarrow R_n \not\leftarrow \cdots$. The reason for this is that constraint compactness and (a) imply the following: there is a constraint R_k such that, for all $i \in \mathbb{N}$, it holds $\forall(R_i \rightarrow R_k)$. However, this contradicts (b), because all constraints R_i must be equivalent for $i \geq k$. □

Example 5.4. [GPPE and Projection]
Consider the simple program $p(x) \leftarrow p(y)$ whose normalisation is:

$$p(x) \leftarrow p(y) \;/\; x \neq y$$

Applying the GPPE yields $p(x) \leftarrow p(y) \;/\; \exists z(x \neq y \wedge y \neq z \wedge x \neq z)$. After n-times

GPPE, we get $p(x) \leftarrow p(y)$ / $\exists z_1 \cdots \exists z_n$ *alldistinct*$\{x, y, z_1, \ldots, z_n\}$ where it holds:

$$alldistinct\{t_1, \ldots, t_n\} = \bigwedge_{1 \leq i < j \leq n} (t_i \neq t_j)$$

Because of the argument from Theorem 5.7, it becomes clear that there can be at most finitely many elements in the domain D for constraint-compact theories.

Constraint programs with constraint-compact constraint theories correspond to Datalog$^{\vee,\neg}$, i.e. general disjunctive logic programs without function symbols. This means, there are only finitely many individuals in the domain. Our result is therefore a generalisation of the termination result in [32] to the case with disjunctions. However, it remains an open question how to incorporate the control mechanism used in Theorem 5.7 into an actual implementation of our calculus.

6. CONCLUSIONS

The careful reader may have noticed that we did not explicitly incorporate the *Loop-Detection rule* into our constraint framework. The reason is that this rule is essentially the only non-constructive rule that can only be approximated if the constraint system is not compact. For example, Chen and Warren do such loop-detection in their SLG or XSB system [9, 8, 10] for non-disjunctive programs (under WFS and STABLE) by *tabling techniques*. However, the main focus of this paper was to introduce the constraint machinery and to combine it with our former calculus. Loop-detection is beyond the scope of this paper and we only give some hints in the next subsection.

Another improvement is to apply the *relevance-property* [5]. Relevance is the condition that a query does only depend on the subprogram formed by the call-graph below that query. This property holds for D-WFS [4] and therefore, given a query Q we do not have to consider the whole residual program of ϕ, but only the relevant part of it with respect to the query. This often has the effect that this subprogram is much smaller than the original one.

6.1. Explicit Loop-Detection

Obviously, by the undecidability of D-WFS, there are always cases where our procedure will not terminate. But sometimes we can overcome this problem. Let us consider the following example:

$$\begin{array}{rl} \Phi: \quad p(x) & \leftarrow p(s(x)) \\ p(s(0))) & \leftarrow \end{array}$$

Applying the GPPE and NMIN we get:

$$\begin{array}{rll} \Phi': \quad p(x) & \leftarrow p(s(s(x))) & / \ x \neq 0 \wedge x \neq s(0) \quad (*) \\ p(s(0)) & \leftarrow & \\ p(0) & \leftarrow & \end{array}$$

We notice, that the GPPE rule can be applied on $(*)$ again and again, although it is clear that only $p(0))$ and $p(s(0))$ hold in the D-WFS semantics for this example (and $\neg p(s^i(0))$ for all $i \geq 2$), because both facts cannot be replaced by any of the transformation rules. Here we need a sort of loop-detection. As far as we know that GPPE can only be applied to the clause itself and that the corresponding term of the predicate to be replaced gets larger and larger, we can immediately delete such a rule—it will only cause an infinite loop that is not grounded. Such a deletion is in fact nothing but an approximation of the Loop-Detection rule. Finally we get the following residual program

$$\begin{aligned} res(\Phi) : p(s(0)) &\leftarrow \\ p(0) \ &\leftarrow \end{aligned}$$

from which D-WFS follows immediately. Of course such approximations need further investigation.

6.2. Related Work

We know the approach [19, 18] that also considers non-ground disjunctive programs. In this approach, the authors define a Gelfond-Lifschitz transformation for first-order programs and consider instantiation by need techniques. It is related to [21], where definite non-ground programs are treated. But the latter approach only considers normal logic programs, i.e. without disjunctions. In addition, they do not consider the case with function symbols in its full generality.

[31] also considers normal logic programs only, employs constructive negation and gives a clean approach to negation in CLP. [27] makes use of rewrite techniques to tackle the problem of computing answers for normal CLP programs with negation, based on the Kunen/Fitting semantics which is weaker than the semantics treated here.

Another approach is presented in [26]. There, the complexity of several propositional non-monotonic logics enhanced by quantified Boolean formulas over propositional variables as constraints are investigated.

Other approaches with constraints for Datalog$^{\vee,\neg}$ are described in [22, 32]. Last but not least, we want to mention [2, 29]. These papers present methods based on partial deduction—as also done here—for normal logic programs and disjunctive logic programs, respectively, but both do not aim at incorporating CLP.

6.3. Outlook

Disjunctive logic programs are natural candidates for e.g. knowledge representation tasks. Often WFS for non-disjunctive programs is too weak. The possibility to explicitly write down disjunctive statements extends the expressive power and makes the formulation of many statements quite naturally. On the other hand, when such information is not needed, our semantics D-WFS coincides with WFS and hence shares its computational advantages. In contrast to the stable semantics, D-WFS is consistent and goal-directed since the GPPE acts top-down. For this, it seems a good idea to make the GPPE more incremental, i.e. to drop the requirement that all definitions of one predicate have to be inserted at once.

A CLP implementation is currently undertaken in ECLiPSe-Prolog [17] at the University of Koblenz as part of the 4-year project on *Disjunctive Logic Programming* (see [1] and http://www.uni-koblenz.de/ag-ki/DLP/). We expect to improve efficiency of non-monotonic systems by exploiting CLP. For positive disjunctive logic programming, the advantage of such a combination has been shown e.g. in [30]. Termination criteria and control are currently elaborated more precisely and are subject of future research. But here for the first time, we have considered a framework for a non-ground disjunctive semantics with negation.

We also mentioned in Section 4.2 that our framework can be even more generalised. It is possible to introduce more constraint predicates than just equality $=$. In contrast to usual CLP, the constraint language must be closed not only under conjunction and existential quantification, but under all logical connectives. In this respect, our work yields the basis for a general combination of two paradigms: constraint logic programming and non-monotonic reasoning. This would lead to a very powerful programming language.

Acknowledgements

We thank our colleagues of the artificial intelligence group at Koblenz. Thanks also to anonymous referees, who read some passages rather carefully.

REFERENCES

1. Chandrabose Aravindan, Jürgen Dix, and Ilkka Niemelä. Report on the DisLoP-project on Disjunctive Logic Programming. *Computational Logic*, 3(2), 1997.
2. Chandrabose Aravindan and Phan Minh Dung. Partial deduction of logic programs wrt well-founded semantics. *New Generation Computing*, 13:45–74, 1994.
3. Stefan Brass and Jürgen Dix. A disjunctive semantics based on unfolding and bottom-up evaluation. In Bernd Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen,* (IFIP '94-Congress, Workshop FG2: Disjunctive Logic Programming and Disjunctive Databases), pages 83–91, Berlin, 1994. Springer.
4. Stefan Brass and Jürgen Dix. A General Approach to Bottom-Up Computation of Disjunctive Semantics. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Non-monotonic Extensions of Logic Programming*, LNAI 927, pages 127–155. Springer, Berlin, 1995.
5. Stefan Brass and Jürgen Dix. Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. *Journal of Logic Programming*, 32(3):207–228, 1997. (Extended abstract appeared in: Characterizations of the Stable Semantics by Partial Evaluation *LPNMR, Proceedings of the Third International Conference, Kentucky*, pages 85–98, 1995. LNCS 928, Springer.).
6. Stefan Brass and Jürgen Dix. Characterizations of the Disjunctive Well-founded Semantics: Confluent Calculi and Iterated GCWA. *Journal of Automated Reasoning*, to appear, 1997. (Extended abstract appeared in: Characterizing D-WFS: Confluence and Iterated GCWA. *Logics in Artificial Intelligence, JELIA '96*, pages 268–283, 1996. Springer, LNCS 1126.).
7. Stefan Brass and Jürgen Dix. Semantics of Disjunctive Logic Programs Based on Partial Evaluation. *Journal of Logic Programming*, accepted for publication, 1997. (Extended abstract appeared in: Disjunctive Semantics Based upon Partial and Bottom-Up Evaluation, *Proceedings of the 12-th International Logic Programming Conference, Tokyo*, pages 199–213, 1995. MIT Press.).

8. Weidong Chen, Terrance Swift, and David S. Warren. Efficient Top-Down Computation of Queries under the Well-Founded Semantics. *Journal of Logic Programming*, 24(3):219–245, 1995.

9. Weidong Chen and David S. Warren. Computing of Stable Models and its Integration with Logical Query Processing. *IEEE Transactions on Knowledge and Data Engineering*, 17:279–300, 1995.

10. Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

11. Alain Colmerauer. Theoretical model of Prolog II. In Michel van Canegham and David H. D. Warren, editors, *Logic programming and its applications*, pages 3–31. Ablex Publishing Corporation, Norwood, NJ, 1986.

12. Hubert Comon and Pierre Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.

13. Jürgen Dix. A Classification-Theory of Semantics of Normal Logic Programs: I. Strong Properties. *Fundamenta Informaticae*, XXII(3):227–255, 1995.

14. Jürgen Dix. A Classification-Theory of Semantics of Normal Logic Programs: II. Weak Properties. *Fundamenta Informaticae*, XXII(3):257–288, 1995.

15. Jürgen Dix, Donald Loveland, Jack Minker, and David. S. Warren. Disjunctive Logic Programming and databases: Nonmonotonic Aspects. Technical Report Dagstuhl Seminar Report 150, IBFI GmbH, Schloß Dagstuhl, 1996.

16. Jürgen Dix and Frieder Stolzenburg. Computation of Non-Ground Disjunctive Well-Founded Semantics with Constraint Logic Programming (preliminary report). In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 1216, pages 202–226. Springer, Berlin, 1997.

17. ECRC GmbH, München. *ECLiPSe 3.5: User Manual – Extensions User Manual*, 1995.

18. T. Eiter, J. Lu, and V. S. Subrahmanian. Computing Non-Ground Representations of Stable Models. In J. Dix, U. Furbach, and A. Nerode, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Fourth International Conference*, LNAI, Berlin, July 1997. Springer.

19. Georg Gottlob, Sherry Marcus, Anil Nerode, Gernot Salzer, and V. S. Subrahmanian. A non-ground realization of the stable and well-founded semantics. *Theoretical Computer Science*, 166(1-2):221–262, 1996.

20. Joxan Jaffar and Michael J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19,20:503–581, 1994.

21. Vadim Kagan, Anil Nerode, and V. S. Subrahmanian. Computing Minimal Models by Partial Instantiation. *Theoretical Computer Science*, 155:157–177, 1995.

22. Paris C. Kanellakis, M. Kuper, and Peter Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51:26–52, 1995.

23. David B. Kemp, Peter J. Stuckey, and Divesh Srivastava. Magic Sets and Bottom-Up Evaluation of Well-Founded Models. In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the 1991 Int. Symposium on Logic Programming*, pages 337–351. MIT, June 1991.

24. Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, pages 348–359. Computer Society Press, 1988.

25. Michael J. Maher. A logic programming view of CLP. In David S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 737–753, Budapest, 1993. MIT Press, Cambridge, MA, London.

26. Chris Pollett and Jeffrey B. Remmel. Nonmonotonic reasoning with quantified Boolean constraints. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 18–39, Schloß Dagstuhl, 1997. Springer, Berlin, Heidelberg, New York. In LNAI.

27. Gilles Richard and Frédéric Saubion. Answer synthesis for CLP programs with negation. *Journal of the IGPL*, 5(3):339–373, 1997.

28. Chiaki Sakama and Hirohisa Seki. Partial Deduction of Disjunctive Logic Programs: A Declarative Approach. In *Logic Program Synthesis and Transformation – Meta Programming in Logic*, LNCS 883, pages 170–182, Berlin, 1994. Springer.

29. Chiaki Sakama and Hirohisa Seki. Partial Deduction in Disjunctive Logic Programming. *Journal of Logic Programming*, 32(3):229–245, 1997.

30. Frieder Stolzenburg and Bernd Thomas. Analysing rule sets for the calculation of banking fees by a theorem prover with constraints. In *Proceedings of the 2nd International Conference on Practical Application of Constraint Technology*, pages 269–282, London, 1996. Practical Application Company.

31. Peter J. Stuckey. Constructive negation for constraint logic programming. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science*, pages 328–339, 1991.

32. David Toman. Computing the well-founded semantics for constraint extensions of datalog$^{\neg}$. In Volker Gaede, Alexander Brodsky, Oliver Günther, Divesh Srivastava, Victor Vianu, and Mark Wallace, editors, *Proceedings of the 2nd International Workshop on Constraint Database Systems 1997*, pages 64–79, Delphi, 1996. Springer, Berlin, Heidelberg, New York. In LNAI 1197.