Chapter 22 in: Constraint Logic Programming: Selected Research,
Frédéric Benhamou and Alain Colmerauer, eds., The MIT Press, 1993,
pp. 405--419. Previous version as DFKI Research Report RR-91-13

Residuation and Guarded Rules for Constraint Logic Programming

Gert Smolka

German Research Center for Artificial Intelligence (DFKI)
and
Universität des Saarlandes
Stuhlsatzenhausweg 3, 6600 Saarbrücken 11, Germany
smolka@dfki.uni-sb.de

Abstract

Current constraint logic programming languages provide simplification for built-in constraints (e.g., arithmetic or boolean), but do not offer constraint propagation for user-defined predicates. We present two concepts, residuation and guarded rules, for obtaining user-defined constraint propagation.

Residuation is a nonsequential control strategy similar to the so-called Andorra Principle giving priority to the reduction of atoms to which at most one clause applies. It achieves an interesting form of constraint propagation for user-defined predicates, thus reducing the need for backtracking.

Residuation, of course, does not extract all useful constraint propagation rules from the clausal definitions of user-defined predicates. Hence we propose so-called guarded rules as a means by which the programmer can explicitly specify additional constraint propagation rules. Guarded rules are similar to guarded clauses in committed-choice languages, but in contrast to these languages our guarded rules run concurrently with the ordinary clausal definition of a predicate. Our framework can in fact be seen as a combination of constraint logic programming with concurrent committed-choice programming.

The second part of the paper offers a semantical model for the proposed family of languages, where goal reduction amounts to equivalence

transformation and guarded rules appear as logical consequences of the clausal definitions of predicates.

1 Introduction

A major difficulty with logic programming is combinatorial explosion: since goals are solved with possibly indeterminate (i.e., branching) reductions, the resulting search trees may grow wildly. Constraint logic programming systems [5, 12, 7] try to avoid combinatorial explosion by building in strong determinate (i.e., non-branching) reduction in the form of constraint simplification. In this paper we present two concepts, residuation and guarded rules, for further strengthening determinate reduction. Both concepts apply to constraint logic programming in general and yield an operational semantics that coincides with the declarative semantics.

1.1 Residuation

Residuation[1] is a control strategy for constraint logic programming meant to replace the rigid depth first strategy of Prolog, which amounts to eager generation of usually wrong assumptions. Residuation makes determinate reduction the rule and indeterminate reduction the exception that must be requested explicitly by declaring relations as generating. Given a goal, an atom is called *determinate* if reduction with all but possibly one clause defining the atom immediately fails due to constraint simplification. *Residuation* is now the following control strategy:

- given a goal that contains determinate atoms, a determinate atom must be reduced
- given a goal that contains no determinate atoms, an atom whose relation is declared as *generating* must be reduced.

Thus the user controls which atoms can reduce indeterminately by declaring relations as generating. If no relation is declared generating, indeterminate reduction cannot occur. Even with generating relations, indeterminate reduction can only occur if determinate reduction is not possible. A relation is called *residuating* if it is not declared generating. Given a goal, an atom is called *residuated* if it is not determinate and its relation is residuating. An important feature of the residuation strategy is that goals whose atoms are all residuated are taken as answers. Often such complex answers are fine as they are. For instance, if length is a length predicate for lists, the goal

$$\exists \mathsf{N}\ (\mathsf{length}(\mathsf{L},\mathsf{N}) \wedge \mathsf{N} \leq 47)$$

[1] The term residuation was coined by Hassan Aït-Kaci [1] for delaying control schemes.

("L is a list with at most 47 elements") is a perfect answer. If the user is not satisfied with a complex answer, he can request indeterminate reduction of a residuated atom.

Residuation is similar to the control strategy of the Andorra model [8, 9], with the difference that residuation performs indeterminate reduction only on atoms whose relation is explicitly declared as generating. The philosophy behind residuation is that for most relations indeterminate reduction simply does not make sense, and that complex answers are often appropriate.

In the examples of this paper we will assume a constraint system with trees and linear integer arithmetic.

A length relation for lists can be defined as follows (constraints are written in *italic font*):

$$\begin{array}{rcl} \mathsf{length}(\mathsf{L},\mathsf{N}) & \leftrightarrow & L = nil \wedge N = 0 \\ & \vee & \exists \mathsf{H},\mathsf{R},\mathsf{M}\ (L = H.R \wedge N > 0 \wedge M = N - 1 \\ & & \qquad\qquad \wedge\ \mathsf{length}(\mathsf{R},\mathsf{M})). \end{array}$$

Instead of the conventional definite clause syntax we use definite equivalences, which make more explicit that the relation on the left hand side is in fact defined (we are committed to least model semantics).[2]

Now, given a goal whose constraint is ϕ, an atom $\mathsf{length}(\mathsf{L},\mathsf{N})$ in this goal is determinate if either the constraint $\phi \wedge L = nil \wedge N = 0$ simplifies to $-$, or the constraint $\phi \wedge \exists H, R\ (L = H.R \wedge N > 0)$ simplifies to $-$, where $-$ is the canonical unsatisfiable constraint. Assuming a sufficiently powerful constraint simplifier, the goal $\mathsf{length}(\mathsf{X},\mathsf{N}) \wedge \mathsf{N} \geq 2$ reduces in two steps determinately to the goal

$$\exists \mathsf{Y},\mathsf{Z},\mathsf{U},\mathsf{M}\ (\mathsf{X} = \mathsf{Y.Z.U} \wedge \mathsf{M} = \mathsf{N} - 2 \wedge \mathsf{M} \geq 0 \wedge \mathsf{length}(\mathsf{U},\mathsf{M})),$$

which is an answer if the relation length is residuating. In any case, it would not make sense to reduce this goal further.

Residuation is a simple and powerful alternative to delay primitives such as the delay annotations of IC-Prolog [4], the freeze construct of Prolog II [6], or the wait declarations of MU-Prolog [15]. Major advantages of residuation over these delay primitives are:

- residuation applies to every constraint system (rather than to tree systems only)
- no annotations in clauses are needed—the programmer only decides which relations should be generating

[2] For the special case of Horn clause programming, the translation from the conventional definite clause syntax to definite equivalences is given by Clark's completion [2].

- residuation is much more flexible—even if all relations are declared generating the search space is considerably pruned since determinate reductions are performed first.

An idealized method for solving problems with residuation splits the problem solver in a propagating and a generating part:

- a predicate $\mathsf{propagate}(\mathsf{S})$ that holds if and only if S is a solution of the problem, and that depends only on residuating relations
- a predicate $\mathsf{generate}(\mathsf{S})$ that defines candidates for (partial) solutions and depends on generating relations.

A problem instance is then given as a query

$$\phi \wedge \mathsf{propagate}(\mathsf{S}) \wedge \mathsf{generate}(\mathsf{S}),$$

where the constraint ϕ describes the particular problem instance. With residuation $\phi \wedge \mathsf{propagate}(\mathsf{S})$ will reduce determinately to a constraint propagation network consisting of residuated atoms and a shared constraint. In general, the constraint propagation network alone is too weak to exhibit solutions. Thus $\mathsf{generate}(\mathsf{S})$ is needed to incrementally generate assumptions about the value of the variable S. As soon as an assumption is made, the constraint propagation network will become active since atoms that where residuated before can now fire. Typically, most of the generated assumptions will be invalidated immediately by constraint propagation leading to a failure. To obtain a feasible search space, two things are essential: careful design of the propagation and generation component, and an expressive underlying constraint system.

1.2 Guarded Rules

Guarded rules are logical consequences of the program introducing additional determinate reduction rules. We will see that guarded rules can significantly strengthen the propagation component of a problem solver.

Consider the following definition of list concatenation:

$$\begin{array}{lcl} \mathsf{app}(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \leftrightarrow & \mathsf{X} = \mathsf{nil} \wedge \mathsf{Y} = \mathsf{Z} \\ & \mid & \mathsf{X} = \mathsf{H.R} \wedge \mathsf{Z} = \mathsf{H.U} \wedge \mathsf{app}(\mathsf{R},\mathsf{Y},\mathsf{U}). \end{array}$$

It is written in sugared syntax (indicated by writing $\mid$ rather than $\vee$), which suppresses existential quantification of auxiliary variables and allows nesting of constraint terms.

With this definition the goal $\mathsf{app}(\mathsf{X},\mathsf{Y},\mathsf{Y})$ does not reduce determinately although it is equivalent to $\mathsf{X} = \mathsf{nil}$. In fact, the relation app satisfies the formula

$$\mathsf{Y} = \mathsf{Z} \rightarrow (\mathsf{app}(\mathsf{X},\mathsf{Y},\mathsf{Z}) \leftrightarrow \mathsf{X} = \mathsf{nil}),$$

which validates the determinate reduction of the atom $\mathsf{app}(\mathsf{X},\mathsf{Y},\mathsf{Z})$ to the constraint $\mathsf{X}=\mathsf{nil}$ if the constraint of the goal entails the "guard" $\mathsf{Y}=\mathsf{Z}$.

A *guarded rule* is a formula

$$\phi \rightarrow (A \leftrightarrow G),$$

for convenience written as

$$\phi \;\Box\; A \;\triangleright\; G,$$

where ϕ is a constraint (called the *guard*), A is an atom, and G is a goal. A guarded rule is *admissible* if it is valid in every model of the declarative semantics (we are committed to least model semantics). Thus admissible guarded rules are redundant as far as the declarative semantics is concerned.

The operational semantics of guarded rules is defined as follows. Given a goal G

$$\exists X(\phi \wedge A \wedge R)$$

and a guarded rule

$$\psi \;\Box\; A \;\triangleright\; G',$$

the goal G can reduce determinately to

$$\exists X(\phi \wedge G' \wedge R)$$

if the constraint ϕ entails the constraint ψ, that is, the implication $\phi \rightarrow \psi$ is valid in every model of the constraint system. Note that $\exists X(\phi \wedge G' \wedge R)$ is logically equivalent to G in all models of the declarative semantics if the guarded rule is admissible. Moreover, $\exists X(\phi \wedge G' \wedge R)$ is a goal up to constraint simplification and minor syntactic rearrangement.

Two further admissible guarded rules for app are

$$\begin{aligned} \mathsf{Y}=\mathsf{nil} \;\Box\; \mathsf{app}(\mathsf{X},\mathsf{Y},\mathsf{Z}) \;&\triangleright\; \mathsf{X}=\mathsf{Z} \wedge \mathsf{list}(\mathsf{X}) \\ \mathsf{X}=\mathsf{Z} \;\Box\; \mathsf{app}(\mathsf{X},\mathsf{Y},\mathsf{Z}) \;&\triangleright\; \mathsf{Y}=\mathsf{nil} \wedge \mathsf{list}(\mathsf{X}), \end{aligned}$$

where the relation list is defined as follows:

$$\mathsf{list}(\mathsf{L}) \quad\leftrightarrow\quad \mathsf{L}=\mathsf{nil} \;\mid \mathsf{L}=\mathsf{H.R} \wedge \mathsf{list}(\mathsf{R}).$$

Admissible guarded rules are a new concept that must not be confused with the guarded clauses of committed-choice languages such as Concurrent Prolog [16] or Parlog [3]. In these languages guarded clauses are used to define agents, while in our framework relations are defined by definite equivalences and admissible guarded rules are logical consequences of the definitions. Moreover, committed-choice languages usually do not have a declarative semantics. Maher [14] has given a declarative semantics for a strongly restricted class of committed-choice languages, where guards must be mutually exclusive. This is usually not the case for guarded rules, as can be seen in the list concatenation example.

Guarded rules have some similarity with the demon predicates of CHIP [7], but are much more general. First, demon predicates in CHIP are defined by guarded rules only, while in our approach the relation is defined independently by clauses. Second, in CHIP guards are restricted to positive tree patterns. Third, in our approach guarded rules can be given for generating relations, while in CHIP demon predicates are residuating by definition. And last not least, CHIP does not even outline a declarative semantics for demon predicates.

In the presence of guarded rules, an atom in a goal is called determinate if it either is determinate as defined before, or if it can reduce with a guarded rule. Residuation is defined as before, except that it now relies on the stronger notion of determinate atoms.

Residuation with guarded rules yields a surprisingly strong constraint propagation mechanism, which we will illustrate with two further examples.

Consider the following relational definition of the Boolean "and" function:

$$\begin{array}{rcl} \mathsf{and}(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \leftrightarrow & \mathsf{X}=1 \wedge \mathsf{Y}=\mathsf{Z} \wedge \mathsf{bool}(\mathsf{Y}) \\ & | & \mathsf{X}=0 \wedge \mathsf{Z}=0 \wedge \mathsf{bool}(\mathsf{Y}) \\ \\ \mathsf{bool}(\mathsf{X}) & \leftrightarrow & \mathsf{X}=1 \ \mid \mathsf{X}=0. \end{array}$$

First note that the definition of and in the presence of residuation already realizes four implicit guarded rules:

$$\begin{array}{rcl} \mathsf{X}\neq 1 \;\Box\; \mathsf{and}(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \rhd & \mathsf{X}=0 \wedge \mathsf{Z}=0 \wedge \mathsf{bool}(\mathsf{Y}) \\ \mathsf{Y}\neq \mathsf{Z} \;\Box\; \mathsf{and}(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \rhd & \mathsf{X}=0 \wedge \mathsf{Z}=0 \wedge \mathsf{bool}(\mathsf{Y}) \\ \mathsf{X}\neq 0 \;\Box\; \mathsf{and}(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \rhd & \mathsf{X}=1 \wedge \mathsf{Y}=\mathsf{Z} \wedge \mathsf{bool}(\mathsf{Y}) \\ \mathsf{Z}\neq 0 \;\Box\; \mathsf{and}(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \rhd & \mathsf{X}=1 \wedge \mathsf{Y}=\mathsf{Z} \wedge \mathsf{bool}(\mathsf{Y}). \end{array}$$

The second and fourth rule could be optimized since under their guards we have $\mathsf{Y}=1$, but residuation will reduce $\mathsf{bool}(\mathsf{Y})$ anyway to $\mathsf{Y}=1$. By exploiting the symmetry of and with respect to its first two arguments we obtain the admissible guarded rules

$$\begin{array}{rcl} \mathsf{Y}\neq 1 \;\Box\; \mathsf{and}(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \rhd & \mathsf{Y}=0 \wedge \mathsf{Z}=0 \wedge \mathsf{bool}(\mathsf{X}) \\ \mathsf{X}\neq \mathsf{Z} \;\Box\; \mathsf{and}(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \rhd & \mathsf{X}=1 \wedge \mathsf{Y}=0 \wedge \mathsf{Z}=0 \\ \mathsf{Y}\neq 0 \;\Box\; \mathsf{and}(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \rhd & \mathsf{X}=\mathsf{Z} \wedge \mathsf{Y}=1 \wedge \mathsf{bool}(\mathsf{X}). \end{array}$$

By adding two further admissible guarded rules

$$\begin{array}{rcl} \mathsf{X}=\mathsf{Y} \;\Box\; \mathsf{and}(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \rhd & \mathsf{X}=\mathsf{Z} \wedge \mathsf{bool}(\mathsf{X}) \\ \mathsf{X}\neq \mathsf{Y} \;\Box\; \mathsf{and}(\mathsf{X},\mathsf{Y},\mathsf{Z}) & \rhd & \mathsf{Z}=0 \wedge \mathsf{bool}(\mathsf{X}) \wedge \mathsf{bool}(\mathsf{Y}), \end{array}$$

we obtain optimal constraint propagation.

For our next example assume that we want to solve a crossword puzzle. For this task a predicate $\mathsf{s(I,U,J,V)}$ is useful that holds if and only if the I's letter of the word U is identical with the J's letter of the word V. This predicate is defined by

$$\begin{array}{rcl} \mathsf{s(I,U,J,V)} & \leftrightarrow & \mathsf{I} = 1 \wedge \mathsf{U} = \mathsf{H.R} \wedge \mathsf{at(J,V,H)} \\ & | & \mathsf{I} > 1 \wedge \mathsf{U} = \mathsf{H.R} \wedge \mathsf{s(I}-1,\mathsf{R,J,V)} \end{array}$$

$$\begin{array}{rcl} \mathsf{at(I,U,X)} & \leftrightarrow & \mathsf{I} = 1 \wedge \mathsf{U} = \mathsf{X.R} \\ & | & \mathsf{I} > 1 \wedge \mathsf{U} = \mathsf{H.R} \wedge \mathsf{at(I}-1,\mathsf{R,X)}. \end{array}$$

Now the goal $\mathsf{s(2,U,J,V)}$ reduces to

$$\exists \mathsf{X,Y,W}\ (\mathsf{U} = \mathsf{X.Y.W} \wedge \mathsf{at(J,V,Y)}),$$

which makes explicit that the word U consists of at least two characters. However, the symmetric goal $\mathsf{s(I,U,2,V)}$ does not reduce determinately. This can be fixed by making the symmetry explicit with the admissible guarded rules

$$\begin{array}{rcl} \mathsf{J} \leq 1\ \Box\ \mathsf{s(I,U,J,V)} & \rhd & \exists \mathsf{H,R}\ (\mathsf{J} = 1 \wedge \mathsf{V} = \mathsf{H.R} \wedge \mathsf{at(I,U,H)}) \\ \mathsf{J} \neq 1\ \Box\ \mathsf{s(I,U,J,V)} & \rhd & \exists \mathsf{H,R}\ (\mathsf{J} > 1 \wedge \mathsf{V} = \mathsf{H.R} \wedge \mathsf{s(I,U,J}-1,\mathsf{R)}) \\ \neg\exists \mathsf{H,R}\ (\mathsf{V} = \mathsf{H.R})\ \Box\ \mathsf{s(I,U,J,V)} & \rhd & -. \end{array}$$

1.3 Nondeclarative Use of Guarded Rules

So far we have only seen admissible guarded rules, that is, guarded rules that were logical consequences of the declarative semantics and whose operational effect was compatible with the declarative semantics. However, the operational semantics obtained by residuation and nonadmissible guarded rules is significantly stronger than what can be captured by classical declarative semantics. In fact, the object-oriented programming techniques developed for Concurrent Prolog [16] become available if determinate atoms are selected for reduction with a fair strategy.

For instance, an agent that reads two input streams X, Y and merges them into one output stream Z can be defined by four nonadmissible guarded rules:

$$\begin{array}{rcl} \mathsf{X} = \mathsf{nil}\ \Box\ \mathsf{merge(X,Y,Z)} & \rhd & \mathsf{Y} = \mathsf{Z} \\ \mathsf{X} = \mathsf{H.R}\ \Box\ \mathsf{merge(X,Y,Z)} & \rhd & \exists \mathsf{U}\ (\mathsf{Z} = \mathsf{H.U} \wedge \mathsf{merge(R,Y,U)}) \\ \mathsf{Y} = \mathsf{nil}\ \Box\ \mathsf{merge(X,Y,Z)} & \rhd & \mathsf{X} = \mathsf{Z} \\ \mathsf{Y} = \mathsf{H.R}\ \Box\ \mathsf{merge(X,Y,Z)} & \rhd & \exists \mathsf{U}\ (\mathsf{Z} = \mathsf{H.U} \wedge \mathsf{merge(X,R,U)}). \end{array}$$

Operationally this merge agent will behave just right: as soon as a message appears on one of the two input streams, it can fire and put the message on the output stream.

It is easy to see that there is no relation merge such that the given guarded rules are admissible. For merge this could be cured by modeling streams as bags (i.e., lists whose order does not matter) rather than lists, but this would destroy the declarative semantics of most stream consumers.

1.4 Rest of the Paper

The rest of the paper presents a simple and general framework for declarative constraint logic programming with residuation and admissible guarded rules. The complications of Jaffar and Lassez's framework [11] are avoided by not providing for negation as failure.

2 Reduction Systems

The abstract notion of a well-founded reduction system captures important properties of logic programming. It builds on predicate logic in that it takes for granted first-order structures and formulae with the usual connectives and quantifiers. We assume that $-$ ("falsity") is a variable-free formula that is invalid in every structure.

A **reduction system** consists of the following:

- a set of formulae called **goals** containing **the trivial goal** $-$
- a set of structures called **models** in which the goals are interpreted
- a set of equivalences $G \leftrightarrow G_1 \vee \ldots \vee G_n$ called **reductions** such that:
  - G and $G_1, \ldots, G_n$ are goals, and $G \neq -$
  - $G \leftrightarrow G_1 \vee \ldots \vee G_n$ is valid in every model.

A reduction $G \leftrightarrow G_1 \vee \ldots \vee G_n$ **applies** to the goal G and no other goal. Typically, a reduction system contains many reductions with the same left hand side, that is, more than one reduction applies to a goal. A reduction system can be seen as a rewrite system, which allows to rewrite a disjunction of goals into an equivalent disjunction of goals by replacing a goal according to a reduction. The idea is to rewrite until no further reduction applies. The reduction systems corresponding to logic programs are in general nonterminating, that is, there are goals from which infinite rewrite derivations issue.

A reduction system can be separated into a **declarative component** given by its goals and models, and an **operational component** given by its goals and reductions.

We say that a goal G **reduces in one step to** G' and write $G \Rightarrow G'$ if there exists a reduction $G \leftrightarrow G_1 \vee \ldots \vee G_n$ such that $G' = G_i$ for some i. We say that a goal G **reduces to** G' if $G \Rightarrow^* G'$, where $\Rightarrow^*$ is the reflexive and transitive closure of $\Rightarrow$.

An **interpretation** is a pair consisting of a model $\mathcal{A}$ and a variable valuation α into $\mathcal{A}$. A **solution of a goal** G is an interpretation $(\mathcal{A}, \alpha)$ such that G is valid in $\mathcal{A}$ under α. A goal is **satisfiable** if it has at least one solution.

An **answer** is a goal to which no reduction applies. Note that $-$ is always an answer (the **trivial answer**). An **answer for a goal** G is an answer G' such that $G \Rightarrow^* G'$. A set of answers for a goal G is **complete** if it contains for every solution σ of G an answer G' such that σ is a solution of G'.

The **computational service** to be provided by a reduction system is **solving of goals**, that is, enumeration of a complete set of answers for a given goal. The declarative component of a reduction system specifies a class of problems, where every goal corresponds to a particular problem instance, and the solutions of the goal are the solutions of the problem instance. The operational component of a reduction system specifies a method for solving problem instances, where solving means to enumerate a complete set of answers.

A reduction system is **well-founded** if there exists a well-founded ordering on pairs of goals and interpretations such that for every reduction $G \leftrightarrow G_1 \vee \ldots \vee G_n$ and every solution σ of G there exist an $i = 1, \ldots, n$ such that $(G, \sigma) > (G_i, \sigma)$ and σ is a solution of G_i. A well-founded reduction system has two important properties:

- every goal has a complete set of answers
- a complete set of answers for a goal G can be enumerated as follows: if no reduction applies to G, then $\{G\}$ is a complete set of answers; otherwise, choose don't care any reduction $G \leftrightarrow G_1 \vee \ldots \vee G_n$ and solve the goals $G_1, \ldots, G_n$ in parallel.

We will see that every Horn clause program yields a well-founded reduction system.

A reduction is **determinate** if its right hand side is a single goal. We say that G **reduces determinately to** G' if G reduces to G' using only determinate reductions. If G reduces determinately to G', then G and G' have exactly the same solutions. A reduction system is **determinate** if it has only determinate reductions. Note that in well-founded and determinate reduction systems there exist no infinite reduction chains $G \Rightarrow G_1 \Rightarrow G_2 \Rightarrow G_3 \cdots$ issuing from a satisfiable goal G.

A reduction system is **terminating** if there exists no infinite chain $G \Rightarrow G_1 \Rightarrow G_2 \Rightarrow G_3 \Rightarrow \cdots$ of reduction steps. Note that a terminating reduction system is always well-founded, but not vice versa. Even a well-founded and determinate reduction system may not terminate on unsatisfiable goals.

3 Constraint Systems

A **constraint system** is a terminating and determinate reduction system whose goals are closed under conjunction, existential quantification, and variable renaming. In a constraint system we call the goals **constraints**, the answers **simplified constraints**, and the process of reducing a constraint to a simplified constraint **constraint simplification**. Note that in a constraint system one can compute for every constraint a simplified constraint. Moreover, if a constraint simplifies to the trivial constraint $-$, it must be unsatisfiable. A constraint system is called **complete** if a constraint is unsatisfiable if and only if it simplifies to $-$. Thus constraint simplification in a complete constraint system is a decision algorithm for satisfiability of constraints.

The operational component of a constraint system is called a **constraint simplifier**, and the operational component of a complete constraint system is called a **constraint solver**. Our framework for constraint logic programming does not require that the underlying constraint system is complete. Given a set of constraints with the corresponding models, one may prefer in practice an incomplete constraint simplifier since a (tractable) constraint solver may not exist.

Our notion of a constraint system is deliberately very general: every set of formulae with a corresponding class of models can be seen as a constraint system if we provide no reductions and close the formulae under conjunction, existential quantification and variable renaming. Such trivial constraint systems providing no computational service are of course not what we want in practice.

4 Definite Construction

We now introduce definite construction, which is the principle underlying constraint logic programming. We obtain a very simple framework for constraint logic programming with residuation. The two theorems given in this section are consequences of the results in [10].

We assume that a constraint system and a set of **definite relation symbols** are given, where the definite relation symbols take a fixed number of arguments and do not occur in the constraint system.

An **atom** takes the form $r(x_1, \ldots, x_n)$, where r is a definite relation symbol taking n arguments and $x_1, \ldots, x_n$ are pairwise distinct variables. A **definite goal** takes the form

$$\exists X\ (\phi \wedge R),$$

where X is a possibly empty set of existentially quantified variables, ϕ is a constraint, and R is a possibly empty conjunction of atoms. Note that the definite goals containing no atoms are exactly the constraints. A **definite equivalence** takes the form

$$A \leftrightarrow G_1 \vee \ldots \vee G_n,$$

where A is an atom and $G_1, \ldots, G_n$ are definite goals called the **clauses** of A. A **definite specification** is a set of definite equivalences containing for every definite relation symbol r exactly one equivalence with r appearing at the left hand side.

In the following we assume that a definite specification is given. Moreover, we assume that ϕ and ψ range over constraints, A over atoms, R over possibly empty conjunctions of atoms, and G over definite goals. We will construct a reduction system for definite goals by defining definite models (the declarative semantics) and definite reductions (the operational semantics).

For convenience, we will often refer to definite goals simply as goals.

A **definite structure** is a structure that can be obtained from a model of the constraint system by adding interpretations for the definite relation symbols. Definite structures are partially ordered as follows: $\mathcal{A} \leq \mathcal{B}$ iff $\mathcal{A}$ and $\mathcal{B}$ extend the same constraint model and $r^{\mathcal{A}} \subseteq r^{\mathcal{B}}$ for every definite relation symbol r. A **definite quasi-model** is a definite structure that is a model of the definite specification. A **definite model** is a minimal definite quasi-model. The following theorem validates our declarative semantics.

Theorem 4.1 *For every model of the constraint system there exists exactly one definite model extending it.*

Next we define the operational semantics. We assume that the order in which atoms are written in a definite goal does not matter.

An equivalence $G \leftrightarrow D$ is a **definite reduction** iff the following conditions are satisfied:

- $G = \exists X(\phi \wedge A \wedge R)$ is a definite goal
- $A \leftrightarrow \bigvee_{i=1}^{n} \exists Y_i \ (\phi_i \wedge R_i)$ is obtained from a definite equivalence of the definite specification by variable renaming such that only the variables in A are shared with G
- obtain for every clause $\exists Y_i \ (\phi_i \wedge R_i)$ of the definite equivalence the goal

$$G_i := \begin{cases} - & \text{if } \phi \wedge \phi_i \text{ simplifies to } - \\ \exists X \cup Y_i \ (\psi_i \wedge R_i \wedge R) & \text{if } \phi \wedge \phi_i \text{ simplifies to } \psi_i \neq - \end{cases}$$

- D is the disjunction of all $G_i \neq -$; if all G_i's are $-$, then $D = -$.

Note that our definition of definite reductions corresponds exactly to SLD-resolution [13] for the special case of Horn clauses.

Given a constraint system $\mathcal{C}$ and a definite specification $\mathcal{D}$ over $\mathcal{C}$, we define $\mathcal{R}(\mathcal{C}, \mathcal{D})$ as the reduction system whose goals are the corresponding definite goals, whose models are the corresponding definite models, and whose reductions are the corresponding definite reductions *together* with the reductions of the constraint system $\mathcal{C}$. It is easy to verify that $\mathcal{R}(\mathcal{C}, \mathcal{D})$ is in fact a reduction system.

Theorem 4.2 *$\mathcal{R}(\mathcal{C}, \mathcal{D})$ is a well-founded reduction system whose answers are exactly the simplified constraints.*

It is now straightforward to build in **residuation**. We only have to discard unnecessary indeterminate reductions:

- discard all indeterminate reductions for goals that do have determinate reductions
- discard all indeterminate reductions obtained by reduction upon a residuating atom (an atom whose relation is not declared generating).

Let us call the thus obtained reduction system $\mathcal{R}^*(\mathcal{C}, \mathcal{D}, \mathcal{G})$, where $\mathcal{G}$ is the set of generating relation symbols. Clearly, $\mathcal{R}^*(\mathcal{C}, \mathcal{D}, \mathcal{G})$ is still a well-founded reduction system. Moreover, let $\mathcal{R}^*(\mathcal{C}, \mathcal{D})$ be the reduction system $\mathcal{R}^*(\mathcal{C}, \mathcal{D}, \mathcal{G})$ where all definite relations are declared generating. Then $\mathcal{R}^*(\mathcal{C}, \mathcal{D})$ is well-founded and has again exactly the simplified constraints as answers (follows immediately from the above theorem). The important difference between $\mathcal{R}(\mathcal{C}, \mathcal{D})$ and $\mathcal{R}^*(\mathcal{C}, \mathcal{D})$ is that $\mathcal{R}^*(\mathcal{C}, \mathcal{D})$ has significantly smaller search spaces (even for the case of Horn clauses), a fact that has only been realized recently in the Andorra model [8, 9].

5 Guarded Rules

Let a constraint system $\mathcal{C}$ and a definite specification $\mathcal{D}$ over $\mathcal{C}$ be given. A *guarded rule* is a formula

$$\phi \rightarrow (A \leftrightarrow G),$$

where ϕ is a constraint (called the *guard*), A is an atom, and G is a definite goal. A guarded rule is *admissible* if it is valid in every definite model.

Let $\mathcal{F}$ be a set of admissible guarded rules. Then $G \leftrightarrow G'$ is called a *forward reduction* iff the following conditions are satisfied:

- $G = \exists X(\phi \wedge A \wedge R)$ is a definite goal
- $\psi \rightarrow (A \leftrightarrow \exists Y(\phi' \wedge R'))$ is obtained from a guarded rule in $\mathcal{F}$ by variable renaming such that only the variables in the atom A are shared with G
- $\phi \wedge \neg\psi$ is a constraint that simplifies to $-$
- $G' = \begin{cases} - & \text{if } \phi \wedge \phi' \text{ simplifies to } - \\ \exists X \cup Y \ (\phi'' \wedge R' \wedge R) & \text{if } \phi \wedge \phi' \text{ simplifies to } \phi'' \neq -. \end{cases}$

If $\phi \wedge \neg\psi$ simplifies to $-$, then ϕ entails ψ, that is, the implication $\phi \rightarrow \psi$ is valid in every model of the constraint system. Moreover, if the constraint system is complete, then $\phi \wedge \neg\psi$ simplifies to $-$ if and only if ϕ entails ψ.

The reduction system $\mathcal{R}(\mathcal{C},\mathcal{D},\mathcal{F})$ is obtained from $\mathcal{R}(\mathcal{C},\mathcal{D})$ by adding the forward reductions defined by the admissible guarded rules in $\mathcal{F}$. It is easy to verify that $\mathcal{R}(\mathcal{C},\mathcal{D},\mathcal{F})$ is in fact a reduction system, and that every goal of $\mathcal{R}(\mathcal{C},\mathcal{D},\mathcal{F})$ has a complete set of answers.

In general, $\mathcal{R}(\mathcal{C},\mathcal{D},\mathcal{F})$ is not well-founded; consider, for instance, the admissible guarded rule $\neg - \rightarrow (A \leftrightarrow A)$. It is the responsibility of the programmer to design the guarded rules in $\mathcal{F}$ such that $\mathcal{R}(\mathcal{C},\mathcal{D},\mathcal{F})$ is well-founded. Further research is necessary to find good sufficient conditions for the well-foundedness of $\mathcal{R}(\mathcal{C},\mathcal{D},\mathcal{F})$.

Residuation for $\mathcal{R}(\mathcal{C},\mathcal{D},\mathcal{F})$ is defined as before.

6 Conclusions

Residuation is a control strategy for CLP meant to replace the rigid depth first strategy of Prolog, which amounts to eager generation of usually wrong assumptions. Residuation makes determinate reduction the rule and indeterminate reduction the exception that must be requested explicitly by declaring relations as generating. Consequently, residuation may produce complex answers containing residuated atoms.

Guarded rules are logical consequences of programs adding otherwise unavailable determinate reductions. Together with residuation guarded rules yield a general and powerful constraint propagation mechanism resulting in drastically smaller search spaces.

Residuation overcomes the strictly sequential computation strategy of Prolog. With residuation every determinate atom can be reduced next, which amounts to multiple threads of computation if a fair selection strategy is used.

The operational semantics of residuation and nonadmissible guarded rules is more expressive than what can be captured by classical declarative semantics. In fact, the object-oriented programming techniques developed for Concurrent Prolog [16] can be expressed.

Topics for further research include: investigation of abstract incrementality properties ensuring efficient implementation if satisfied by constraint simplifiers; design of an abstract machine separating control from constraint simplification; and investigation of parallel reduction strategies.

Acknowledgments. The research reported in this paper was inspired by my collaboration with Hassan Aït-Kaci and Andreas Podelski on the semantics of LIFE. I'm also thankful to Ralf Scheidhauer who contributed to the examples.

References

[1] H. Aït-Kaci and R. Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89, 1989.

[2] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, NY, 1978.

[3] K. Clark and S. Gregory. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.

[4] K. L. Clark and F. G. McCabe. The control facilities of IC-PROLOG. In D. Mitchie, editor, *Expert Systems in the Micro-Electronic Age*. Edinburgh University Press, Edinburgh, Scotland, 1979.

[5] A. Colmerauer. An introduction to PROLOG III. *Communications of the ACM*, pages 70–90, July 1990.

[6] A. Colmerauer, H. Kanoui, and M. V. Caneghem. Prolog, theoretical principles and current trends. *Technology and Science of Informatics*, 2(4):255–292, 1983.

[7] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan, Dec. 1988.

[8] S. Haridi. A logic programming language based on the Andorra model. *New Generation Computing*, 7:109–125, 1990.

[9] S. Haridi and S. Janson. Kernel Andorra Prolog and its computation model. In D. Warren and P. Szeredi, editors, *Logic Programming, Proceedings of the 7th International Conference*, pages 31–48, Cambridge, MA, June 1990. The MIT Press.

[10] M. Höhfeld and G. Smolka. Definite relations over constraint languages. LILOG Report 53, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, Germany, Oct. 1988.

[11] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, Jan. 1987.

[12] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In J.-L. Lassez, editor, *Proceedings of the 4th International Conference on Logic Programming*, Cambridge, MA, 1987. The MIT Press.

[13] J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Springer-Verlag, Berlin, Germany, 1984.

[14] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 858–876, Cambridge, MA, 1987. The MIT Press.

[15] L. Naish. Automating control for logic programs. *Journal of Logic Programming*, 3:167–183, 1985.

[16] E. Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing*, 1:24–48, 1983.