Universität Potsdam

Ulrich Geske | Armin Wolf (Hrsg.)

Proceedings of the 23rd Workshop on (Constraint) Logic Programming 2009

Universitätsverlag Potsdam

Proceedings of the 23rd Workshop
on (Constraint) Logic Programming 2009

Ulrich Geske | Armin Wolf (Hrsg.)

Proceedings of the 23rd Workshop on (Constraint) Logic Programming 2009

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de/ abrufbar.

Universitätsverlag Potsdam 2010
http://info.ub.uni-potsdam.de/verlag.htm

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 4623 / Fax: 3474
E-Mail: verlag@uni-potsdam.de

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam:
URL http://pub.ub.uni-potsdam.de/volltexte/2010/3797/
URN urn:nbn:de:kobv:517-opus-37977
http://nbn-resolving.org/urn:nbn:de:kobv:517-opus-37977

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-026-7

Preface

The workshops on (constraint) logic programming (WLP) are the annual meeting of the Society of Logic Programming (GLP e.V.) and bring together researchers interested in logic programming, constraint programming, and related areas like databases, artificial intelligence and operations research. In this decade, previous workshops took place in Dresden (2008), Würzburg (2007), Vienna (2006), Ulm (2005), Potsdam (2004), Dresden (2002), Kiel (2001), and Würzburg (2000). Contributions to workshops deal with all theoretical, experimental, and application aspects of constraint programming (CP) and logic programming (LP), including foundations of constraint/logic programming. Some of the special topics are constraint solving and optimization, extensions of functional logic programming, deductive databases, data mining, nonmonotonic reasoning, , interaction of CP/LP with other formalisms like agents, XML, JAVA, program analysis, program transformation, program verification, meta programming, parallelism and concurrency, answer set programming, implementation and software techniques (e.g., types, modularity, design patterns), applications (e.g., in production, environment, education, internet), constraint/logic programming for semantic web systems and applications, reasoning on the semantic web, data modelling for the web, semistructured data, and web query languages.

The topics of the presentations of the 23rd WLP (WLP2009) are grouped into the major areas: Databases, Answer Set Programming, Theory and Practice of Logic Programming as well as Constraints and Constraint Handling Rules.

The topics of the invited talks deal especially with constraint solving. Tom Schrijvers discusseses the advantages of monadic constraint solving which suplies a framework for unifying constraint solving over conjunctive constraints and search in a disjunctive search tree into a common and flexible strategy. Neng-Fa Zhou discusses the importance of choosing the right modeling and programming technique for an efficient solving of constraint problems.

In the **Database** session the declarative aspect of Logic Programming is used to make saver manipulation of relational databases and to extend relational databases to deductive databases. Michael Hanus and Sven Koschnike describe in "An ER-based framework for declarative web programming" a framework which ensures automatically the consistency of a database when the user performs update operations. The basic idea of this framework is an implementation of the conceptual model of a relation database in a declarative programming language. Dietmar Seipel proposes in the contribution "Practical applications of extended deductive databases in DATALOG*" a language DATALOG* for deductive databases which allows - different to classical Datalog - program execution in backward chaining manner. The advantages are given by a greater flexibility formulating programs and requests by using features of the PROLOG language and having available default negation, aggregation and complex data structures.

In the session on **Answer Set Programming** two papers deal with extensions of the ASP formalism and a further paper is devoted to the problem of program analysis.
In the paper on "xpanda: A (Simple) Preprocessor for Adding Multi-valued Propositions to ASP" Martin Gebser, Henrik Hinrichs, Torsten Schaub and Sven Thiele describe a transformation-based approach which makes it possible to model and solve constraint satisfaction problems in the ASP formalism. The authors discuss the problems of grounding in ASP which may lead to large search spaces in their first approach of combing ASP and constraint-solving. The paper on "Existential quantifiers in the rule body" of Pedro Cabalar deals with the extension of ASP for using existential quantifiers and double negations in the bodies of rules. The given introduction to the problem is nice readable and discusses the change over the last years in considering variables in ASP. The author describes a translation algorithm which uses the general technique of introduction of auxiliary predicates but which keeps these predicate hidden from the programmer for avoiding confusion. The third paper on "Kato: A Plagiarism-Detection Tool for Answer-Set Programs" in this session is presented by Johannes Oetsch, Martin Schwengerer and Hans Tompits. For different reasons there is a demand for finding similarities in different programs, the plagiarism detection. The authors present the presumable first such algorithm for ASP programs. Although the algorithm contains language-depended parts, the authors stress the adaptability of the method for other languages (e.g. PROLOG).

The session on **Theory of Logic Programming** starts with a contribution of Heinrich Herre and Axel Hummel on "A Para-consistent Semantics for Generalized Logic Programs". For using logic as a basis for knowledge representation, generalizations of standard logic programs with declarative semantics are needed which take into account the occurrence of contradictions. The authors discuss a generalization where arbitrary quantifier-free formulas in rules, which may contain default negations, may occur. A next step may be admitting quantifiers in the rules. The paper on "Stationary generated models of generalized Logic Programs" by the same authors is devoted to the general problem of finding an adequate declarative semantics for generalized logic programs. The authors introduce for this purpose the notion of a so-called "intended partial model" and prove their properties. Gourinath Banda and John Gallagher present in their paper on a "Constraint-Based Abstraction of a Model Checker for Infinite State Systems" a new approach to verify properties of infinite-state systems by going back to the roots of abstract interpretation and exclusively applying standard techniques for constructing abstractions. The advantage consists in verifying any property of a system and a better efficieny, which originates partly from the integration of constraint solving techniques. The authors present implementation details and demonstrate their approach on two examples. Stefan Brass deals in „Range Restriction for General Formulas" with the gap between the features of SQL and the current standard of Datalog. Some problems which arise by the use of general logic formulas are discussed. The basic question is to answer, which formulas should be allowed in logic rules.

The session **Constraint Handling Rules** contains a paper describing an application of CHR programming for program development and a theoretical contribution dealing with the extension of the CHR paradigm.

Slim Abdennadher, Haythem Ismail and Frederick Khoury discuss in their paper "Transforming Imperative Algorithms to Constraint Handling Rules" the idea using the flexibility and expressiveness of CHR for proving properties of imperative programs. Vice-versa this method could be used for an automatic generation of global constraint solvers. The authors describe the methodology for the conversion of imperative programs into CHR programs and discuss it with examples.

Hariolf Betz, Frank Raiser and Thom Frühwirth deal in "Persistent Constraints in Constraint Handling Rules" with the drawback that CHR as a high-level declarative language has to use a non-declarative token store in its implementation for avoiding trivial non-termination. A solution of this problem is presented which supplies a new operational semantics for CHR programs. Advantages are a higher degree of declarativity with avoidance of non-trivial termination and improved behaviour for concurrency.

The papers of the last session **Practice of Logic Programming** deal with a real world application of (Constraint) Logic Programming and a closer consideration of the difference list formalism.
Hans-Joachim Goltz and Norbert Pieth describe in their paper "A Tool for Generating Partition Schedules of Multiprocessor Systems" the application of Constraint-Logic Programming for modeling and solving a complex scheduling problem. Details for the derivation of constraints from the problem description and the design of the search are discussed. The optimization criterion is a high processor load. A graphical interface supports interactive control of the scheduling process by the user who may change weak constraints. The paper on "Efficiency of Difference-List Programming" by Ulrich Geske and Hans-Joachim Goltz analyses the use of a program construct in PROLOG which has a significant effect on the speed of list processing. The intention of the authors is the promotion of application of this construct, especially for PROLOG novices. Therefore simple syntactical patterns are derived which allow to control the order of elements in a list and an append-free insertion of elements into lists.

Finally, we would like to thank all the authors who have submitted papers, all colleagues who have presented invited talks, and all members of the program committee and external referees for reviewing the submissions and for their contributions to the success of the workshop.

Potsdam, September 15, 2009

Ulrich Geske
Armin Wolf

Organization

Program Chairs

Ulrich Geske (University of Potsdam)
Armin Wolf (Fraunhofer FIRST, Berlin)

Program Committee

Slim Abdennadher (German University Cairo)
Christoph Beierle (FernUniv. Hagen)
Stefan Brass (MLU Halle-Wittenberg)
Jürgen Dix (Clausthal University of Technology)
Tim Furche (LMU München)
Ulrich Geske (University of Potsdam)
Hans-Joachim Goltz (Fraunhofer FIRST, Berlin)
Michael Hanus (CAU Kiel)
Heinrich Herre (University of Leipzig)
Steffen Hölldobler (TU Dresden)
Petra Hofstedt (TU Berlin)
Ulrich John (SIR Plan GmbH)
Michael Leuschel (Univ. Düsseldorf)
Ulrich Neumerkel (TU Wien)
Frank Raiser (University of Ulm)
Georg Ringwelski (Hochschule Zittau/Görlitz)
Sibylle Schwarz (Hochschule Zwickau)
Dietmar Seipel (University of Würzburg)
Michael Thielscher (TU Dresden)
Hans Tompits (TU Wien)
Armin Wolf (Fraunhofer FIRST, Berlin)

Contents

Constraint Handling Rules

Practice of Logic Programming

Overview of the Monadic Constraint Programming Framework

Tom Schrijvers*

Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A
3001 Heverlee, Belgium
tom.schrijvers@cs.kuleuven.be

Abstract. A constraint programming system combines two essential components: a constraint solver and a search engine. The constraint solver reasons about satisfiability of conjunctions of constraints, and the search engine controls the search for solutions by iteratively exploring a disjunctive search tree defined by the constraint program.
The Monadic Constraint Programming framework gives a monadic definition of constraint programming where the solver is defined as a monad threaded through the monadic search tree. Search and search strategies can then be defined as firstclass objects that can themselves be built or extended by composable search transformers. Search transformers give a powerful and unifying approach to viewing search in constraint programming, and the resulting constraint programming system is first class and extremely flexible.

1 Introduction

A constraint programming (CP) [11] system combines two essential components: a constraint solver and a search engine. The constraint solver reasons about conjunctions of constraints and its principal job it to determine unsatisfiability of a conjunction. The search engine controls the search for solutions by iteratively exploring an OR search tree defined by the program. Whenever the conjunction of constraints in one path defined by the search tree is unsatisfiable, search changes to explore another part of the search tree.

Constraint programming is a declarative programming formalism, where the constraints are defined declaratively, but the underlying constraint solvers are highly stateful, and indeed to specify complex search CP programs rely on reflecting state information from the solver. So in that sense constraint programming is not so declarative after all.

In the Monadic Constraint Programming (MCP) framework we give a monadic definition of constraint programming where the solver is defined as a monad threaded through a monadic search tree. We are then able to define search and search strategies as first class objects that can themselves be built or extended

* Post-doctoral researcher of the Fund for Scientific Research - Flanders.

by composable search transformers. Search transformers give a powerful and unifying approach to viewing search in constraint programming. The resulting CP system is first class and extremely flexible.

Our work can be viewed as encapsulating the functional abstractions previously used in constraint programming in a functional programming language, and using the power of functional programming to take a further step in the increasingly abstract view of search and constraint programming. The contributions of the MCP framework are:

- We show how monads provide a powerful tool for implementing constraint programming abstractions, which allows us to build a highly generic framework for constraint programming.
- We define search strategy transformers which are composable transformers of search, and show how we can understand existing search strategies as constructed from more fundamental transformers.
- We open up a huge space of exploration for search transformers.
- The code is available at `http://www.cs.kuleuven.be/~toms/Haskell/`.

The remainder of the paper is organized as follows. Section 2 provides a motivating example of the MCP framework. For those unfamiliar with Haskell type classes and monads, Section 3 introduces them briefly. Then in Sections 4, 5 & 6 the core parts of the MCP framework are presented, respectively the modeling language, the solving process and search strategies. An overview of related work is given in Section 7. Finally, Section 8 concludes.

2 Motivating Example

The n queens problem requires the placing of n queens on an $n \times n$ chessboard, so that no queen can capture another. Since queens can move vertically, horizontally, and diagonally this means that

1. No two queens share the same *column*.
2. No two queens share the same *row*.
3. No two queens share the same *diagonal*.

A standard model of the n queens problem is as follows. Since we have n queens to place in n different columns, we are sure that there is exactly one queen in each column. We can thus denote the row position of the queen in column i by the integer variable q_i. These variables are constrained to take values in the range $1..n$. This model automatically ensures the column constraint is satisfied. We can then express the row constraint as

$$\forall 1 \leq i < j \leq n : q_i \neq q_j$$

and the diagonal constraint as

$$\forall 1 \leq i < j \leq n : q_i \neq q_j + (j - i) \quad \wedge \quad q_j \neq q_i + (j - i)$$

since queens i and j, with $i < j$, are on the same descending diagonal iff $q_i = q_j + (j - i)$, and similarly they are on the same ascending diagonal iff $q_j = q_i + (j - i)$.

A solution to the 8 queens problem is shown in Figure 1. The solution illustrated has $q_1 = 8$, $q_2 = 4$, $q_3 = 1$, $q_4 = 3$, $q_5 = 6$, $q_6 = 2$, $q_7 = 7$, $q_8 = 5$.

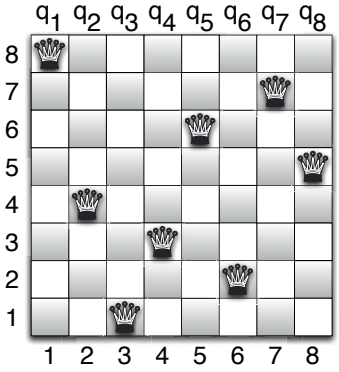


Fig. 1. A solution to the 8 queens problem

The first role of a constraint programming language is to be able to succinctly model problems. We will define constraint programming in Haskell which allows the model of the n queens problem shown in Figure 2. Note how similar it is to the mathematical model.

```
nqueens n =  exist n $ \queens -> model queens n

model queens n = queens `allin` (1,n) /\
                 alldifferent queens  /\
                 diagonals queens

allin queens range =  conj [q `in_domain` range | q <- queens ]

alldifferent queens = conj [ qi @\= qj | qi:qjs <- tails queens,
                                         qj <- qjs ]
diagonals queens =  conj [ qi @\== (qj @+ d) /\ qj @\== (qi @+ d)
                         | qi:qjs  <- tails queens, (qj,d) <- zip qjs [1..]]
```

Fig. 2. Haskell code for modelling n queens.

The next important part of a constraint programming solution is to be able to program the search. We will construct a language for search that allows us to express complex search strategies succinctly, and in a composable manner.

Search is separated into components: specifying the search tree, the basic order for visiting the search tree, and then the search transformers which transform the search tree or the way it is visited. Examples of search orders are depth-first search (`dfs`), breadth-first search (`bfs`) or best-first search. Examples of search transformers are depth bounded search (`db` n never visits nodes at depth below n), node bounded search (`nb` n visits at most n nodes), limited discrepancy search (`ld` n visits only nodes requiring at most n right branches), or branch-and-bound optimization (`bb` f applies a tree transformation f for eliminating non-optimal solutions). These search transformers are composable, so we can apply multiple transformations in order.

For example, using our search framework we can succinctly define complex search strategies. The following calls show how to solve 8 queens with:

- depth first search, first applying a node bound of 100, then a depth bound of 25, then using `newBound` branch and bound
- breadth first search, first applying a depth bound of 25, then a node bound of 100, then using `newBound` branch and bound
- breadth first search, first limited discrepancy of 10, then a node bound of 100, then using `newBound` branch and bound

can be expressed in our framework as:

```
> solve dfs (nb 100 :- db 25 :- bb newBound) $ nqueens 8
> solve bfs (db 25 :- nb 100 :- bb newBound) $ nqueens 8
> solve bfs (ld 10 :- nb 100 :- bb newBound) $ nqueens 8
```

Clearly exploring different search strategies is very straightforward.

3 Haskell Background

The MCP framework relies heavily on Haskell's type system and abstraction mechanism for providing a flexible component-based system.

We assume that the reader is already familiar with the basics of Haskell, such as algebraic data types and (higher-order) functions, but provide a brief introduction to two of the more advanced features, type classes and monads, that MCP heavily relies on.

3.1 Type Classes

Type classes [22] are Haskell's systematic solution to adhoc overloading. Conceptually, a type class C is an n-ary predicate over types that states whether an implementation of the overloaded methods associated to the type class is available for a particular combination of types.

For instance, `Eq` is a unary type class with associated method `(==)` for equality:

```
class Eq a where
  (==) :: a -> a -> Bool
```

The type class constraint `Eq` τ holds if type τ provides an implementation for the method `(==)`. Implementations are provided through type class instances. For instance,

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

provides an implementation of equality for booleans. The function `(==)` has type $\forall a.$`Eq` $a \Rightarrow a \rightarrow a \rightarrow$ `Bool`. This signature expresses that `(==)` applies to argument of any type a that is an instance of the `Eq` type class. This suggests a sequential composition of computations.

Type class constraints propagate from the signature of type class methods to functions that are defined in terms of them. For instance, the function `allEqual` that checks whether all elements of a list are equal inherits the type class constraint from its use of `(==)`:

```
allEqual :: Eq a => [a] -> Bool
allEqual (x:y:zs)  = x == y && allEqual (y:zs)
allEqual _         = True
```

3.2 Monads

Monads [21] are an abstraction used in functional programming languages to represent *effectful computations*. A monad computation `m` is parametrized in the type of the computed result `a`. So `m a` denotes a monadic computation of type `m` that produces a value of type `a`. Monads are captured in the type class:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

The `return` function creates a pure computation that simply returns a given value without any actual effects. The bind operator `(>>=)` composes two monadic computations: the first computation produces a value of type `a` that is consumed by the second computation to produce a value of type `b`. Note that the arrow `>>=` corresponds to the data flow: the left operand produces data, the right one consumes it.

Not captured in Haskell code, but also part of the monad specification, are the monad laws:

$$\begin{aligned} \texttt{return}\ x\ \texttt{>>=}\ f &\equiv f\ x \\ m\ \texttt{>>=}\ \texttt{return} &\equiv m \\ (m\ \texttt{>>=}\ f)\ \texttt{>>=}\ g &\equiv m\ \texttt{>>=}\ (\lambda x.f\ x\ \texttt{>>=}\ g) \end{aligned}$$

Any instance of the `Monad` type class should satisfy these laws, although they are not enforced by the Haskell language.

Haskell provides syntactic sugar of an imperative style for monads, the `do` notation.

$$\texttt{do}\ \begin{matrix} x\ \texttt{<-}\ m_1 \\ m_2 \end{matrix} \equiv m_1\ \texttt{>>=}\ \backslash x\ \texttt{->}\ m_2$$

$$\texttt{do}\ \begin{matrix} m_1 \\ m_2 \end{matrix} \equiv m_1\ \texttt{>>=}\ \backslash\_\ \texttt{->}\ m_2$$

There is a wide range of literature and Haskell libraries related to monad instances, and generic monad infrastructure.

4 Constraint Models

The MCP framework represents constraint models at the core by a separate data type, called `Tree`. This has obvious advantages for manipulating and inspecting the model. On top of this core data type, MCP provides convenient syntactic sugar for expressing models in a higher-level form, closer to mathematical formulas.

4.1 The Model `Tree`

The core data type for representing constraint models is defined as follows:

```
data Tree solver a
   = Return a
   | NewVar (Term solver -> Tree solver a)
   | Add (Constraint solver) (Tree solver a)
   | Try (Tree solver a) (Tree solver a)
   | Fail
   | Dynamic (solver (Tree solver a))
```

It is parametric in two types: 1) the constraint `solver` and its associated constraint domain, and 2) a computed value of type `a`. The former makes most of the constraint model infrastructure independent of the particular constraint domain, and hence reusable in many settings. The latter makes `Tree solver` a monad, and allows Haskell's monad infrastructure to be reused.

The different constructors of `Tree solver a` have the following meaning. `Return a` is a trivially satisfiable model that returns a value `a`, while `Fail` is an

inconsistent model. `Acc c m` extends a model `t` with an additional constraint `c` at the front. `NewVar f` represents a model `f` with an existentially bound (new) constraint variable. `Try t1 t2` represents a disjunctive model with the alternatives `t1` and `t2`. Finally, `Dynamic m` allows the dynamic creation of the model as a computation in the solver (see later).

Now we can make `Tree solver` a monad instance:

```
instance Monad (Tree solver) where
  return = Return

  (Return x)  >>= f  = f x
  (NewVar g)  >>= f  = NewVar (\v -> g x >>= f)
  (Add c t)   >>= f  = Add c (t >>= f)
  (Try t1 t2) >>= f  = Try (t1 >>= f) (t2 >>= f)
  Fail        >>= f  = Fail
  Dynamic m   >>= f  = Dynamic (do { t <- m ; return (t >>= f)})
```

By straightforward equational reasoning, we can establish that the monad laws hold for this monad instance. For example, the first monad law holds as follows:

$$
\begin{array}{ll}
 & \texttt{return}\ x\ \texttt{>>=}\ f \\
\equiv & \textit{(definition of } \texttt{return}\textit{)} \\
 & \texttt{Return}\ x\ \texttt{>>=}\ f \\
\equiv & \textit{(definition of } \texttt{(>>=)}\textit{)} \\
 & f\ x
\end{array}
$$

4.2 Syntactic Sugar

On top of the core data type, MCP adds various convenient abstractions:

```
true  = Return ()
false = Fail

t1 /\ t2  = t1 >>= \_ -> t2
t1 \/ t2  = Try t1 t2

conj = foldr (/\) true
disj = foldr (\/) false

exists = NewVar

exist n f = aux n []
   where aux 0 vs  = f $ reverse vs
         aux n vs  = exists $ \v -> aux (n-1) (v:vs)
```

Note that `conj` and `exist` are two domain-independent model combinators we have used in the n-queens model.

5 Constraint Solving

The constraint model presented in the previous section is a data type. In order to actually compute solutions for the model, it must be "fed" to a constraint solver. However, MCP refrains from directly exposing the original model to a constraint solver directly. Instead, MCP translates the model into a set of core primitives understood by the solver.

There are several important reasons for this approach:

- It reduces the solver implementor's burden, who must only provide core functionality for his solver.
- The MCP framework retains control over the translation process and exposes it to the framework user.
- Much of the translation logic can be reused for different constraint solvers.

5.1 The Solver Interface

The interface that constraint solvers must support is captured in the `Solver` type class:

```
class Monad solver => Solver solver where
  type Constraint solver  :: *
  type Term solver        :: *
  newvar  :: solver (Term solver)
  add     :: Constraint solver -> solver Bool
  run     :: solver a -> a
  type Label solver :: *
  mark :: solver (Label solver)
  goto :: Label solver -> solver ()
```

First line states that a solver must be a monad. Indeed, in general we assume that the solver encapsulates a stateful computation, where the state consists of the solver's constraint store. Two associated types of the solver define its constraint domain: `Constraint solver` is the type of constraints supported by the solver, and `Term solver` is the type of terms that the constraints range over.

The two methods `newvar` and `add` are the respective counterparts of the `NewVar` and `Add` constructors of the model tree. The former returns a new constraint variable as a solver computation. The latter adds a constraint to the solver state and returns a boolean indicating whether the constraint store is still consistent (`True`) as far as the solver can tell or has become definitely inconsistent (`False`). The `run` method allows extracting the values from a solver computation.

Finally, the remaining three members of the `Solver` class are related to disjunctions. The solver interface for disjunction is much more primitive than the high-level `Try` constructor of the model. The `Label solver` type represents a label for a solver state; the label for the current solver state can be requested with

the `mark` operation. The `goto` operation restores the solver state of a given label. On top of these two operations various search strategies can be programmed.

From the side of the solver, different strategies can be used to implement the primitive operations. In a state copying approach, the labels are simply copies of the state and the operations obvious. In the case of recomputation, a label is a trace of the operations that led up to the solver state, and `goto` replays the trace. Backtracking involves a more intricate strategy.

5.2 A Simple Finite Domain Solver

To illustrate the solver interface, we present a simple instantiation, without going into the implementation details.

Our solver type is called `FD` and its instance of the `Solver` class is:

```
instance Solver FD where
  type Constraint FD  = FDConstraint
  type Term FD        = FDTerm
  newvar  = newvarFD
  ...
```

The `FDTerm` type is abstract, and of course the details of the member functions are not exposed. All the programmer needs to know are the details of the `FDConstraint` type. Our small `FD` solver only supports three constraints:

```
data FDConstraint  =  FDIn FDTerm (Int,Int)
                   |  FDEQ FDTerm Int
                   |  FDNE FDTerm FDTerm Int
```

Formally, the semantics can be expressed as:

$$\begin{aligned} [\![\texttt{FDIn } t \texttt{ (}l\texttt{,}u\texttt{)}]\!] &= [\![t]\!] \in \{l, \ldots, u\} \\ [\![\texttt{FDEQ } t \ d]\!] &= [\![t]\!] = d \\ [\![\texttt{FDNE } s \ t \ i]\!] &= [\![s]\!] \neq [\![t]\!] + i \end{aligned}$$

We use Overton's FD solver [13] for the concrete implementation.

On top of this interface, convenient syntactic sugar such as that used in the n-queens model, is easily defined.

```
x @\= y  = Add (FDNE x y 0) true
```

5.3 From Models to Solver Computations

The `eval` function turns a model into a solver computation.

```
eval :: Solver solver => Tree solver a -> solver [a]
eval model  = eval' model []
```

```
eval' (Return x) wl  = do xs <- continue wl
                          return (x:xs)
eval' (Add c t)  wl  = do b <- add c
                          if b then eval' t  wl
                               else continue wl
eval' (NewVar f) wl  = do v <- newvar
                          eval' (f v) wl
eval' (Try l r)  wl  = do now <- mark
                          eval' l ((now,r):wl)
eval' Fail       wl  = continue wl

continue []              = return []
continue ((past,t):wl)   = do goto past
                              eval' t wl
```

The `eval'` function is the main workhorse, that has a worklist of labels as an additional parameter. When a disjunction (`Try`) is encountered, the label of the current solver state is pushed onto the worklist together with the right branch for later processing, while the left branch is processed immediately. The `continue` function is invoked whenever the end of a branch is reached, or an inconsistency is detected by the solver. Then a deferred branch is popped from the worklist, its state is restored and processing continues until the worklist is empty.

Putting everything together, the list of solutions is extracted from the evaluated model with the solver's `run` method:

```
solve :: Solver solver => Tree solver a -> [a]
solve =  run . eval
```

6 Search

MCP makes search much more flexible in a number of ways, summarized in this section.

6.1 Dynamic Variable Enumeration

Often search is used to complete incomplete propagation strategies of constraint solvers. In particular, for finite domain (FD) solvers, the possible assignments for variables are enumerated in disjunctions. For instance, the following code augments the n-queens model with such enumeration.

```
nqueens n = exist n $ \queens -> model queens n /\
                                 enumerate queens [1..n]

enumerate qs values =  conj [ enum q values | q <- qs ]

enum var values = disj [ var @= value | value <- values ]
```

Note however, that this enumeration is based on the variable's static domain. The generated search tree can be much more compact, if the dynamic domain of variables, reduced by the solver's propagation, is used.

If the `FD` solver exposes a function `domain :: Term FD -> FD [Int]` to query a variable's dynamic domain, the `Dynamic` model constructor allows generating the enumeration part of the search tree dynamically.

```
nqueens n = exist n $ \queens -> model queens n /\
                                 enumerate queens

enumerate  = Dynamic . label

label []      = return ()
label (v:vs)  = do d <- domain v
                   return $ enum v d /\ enumerate vs
```

Many other dynamic enumeration strategies can be captured in a similar way.

6.2 Queueing Strategies

The `eval` function above implements depth-first search using a stack as the worklist. The MCP framework generalizes this by means of a `Queue` type class, which allows other queue-like data structures to be used to implement strategies like breadth-first search and best-first search.

```
class Queue q where
  type Elem q :: *
  emptyQ   :: q -> q
  isEmptyQ :: q -> Bool
  popQ     :: q -> (Elem q,q)
  pushQ    :: Elem q -> q -> q
```

6.3 Search Transformers

Advanced search strategies can be implemented on top of the `eval` loop and queueing strategy. For this purpose, MCP introduces the concept of *search transformers*. Examples of search transformers are various forms of pruning (node-bounded, depth-bounded, limited discrepancy), randomly flipping branches of the search tree, iterative deepening, restart optimization and branch-and-bound. MCP employs the technique of functional mixins to open the recursion of the `eval` loop and to allow a search transformer to intercept each recursive call.

In addition to basic search transformers, MCP also provides search transformer `combinators` for building advanced transformers from basic ones. The most important such combinator is the composition operator `(:-)`, which sequentially composes two transformers. For instance, the sequential composition

of a node-bounded and a depth-bounded pruner explores the seach tree up to a certain depth and up to a certain number of nodes. Another example of a combinator is an iterative restarting combinator, which generalizes both iterative deepening and restart optimization.

7 Related Work

Since our approach combines constraint and functional programming there is a broad spectrum of related work.

Constraint Programming Constraint logic programming languages allow programmable search using the builtin search of the paradigm. Each system provides predicates to define search, analogous to the `Dynamic` nodes in the model tree. For instancen, ECLiPSe [23] provides a search library which allows: user programmable variable and value selection as well as different search transformers including depth bounded search, node bounded search, limited discrepancy search, and others. One transformation cannot be applied to another, although one can change strategy for example when the depth bound finishes to another strategy. The user cannot define their own search transformers in the library, though they could be programmed from scratch.

The Oz [16] language was the first language to truly separate the definition of the disjunctive constraint model from the search strategy used to explore it [14]. Here computation spaces capture the solver state, as well as possible choices (effectively the `Dynamic` nodes). Search strategies such as DFS, BFS, LDS, Branch and Bound and Best first search are constructed by copying the computation space and committing to one of the choices in the space. Search strategies themselves are monolithic, there is no notion of search transformers.

The closest work to this paper is the search language [19] of Comet [18]. Search trees are specified using `try` and `tryall` constructs (analogous to `Try` and `Dynamic` nodes), but the actual exploration is delegated to a search controller which defines what to do when starting or ending a search, failing or adding a new choice. The representation of choices is by continuations rather than the more explicit tree representation we use. The `SearchController` class of Comet is roughly equivalent to the `Transformer` class. Complex search hybrids can be constructed by building search controllers. The Comet approach shares the same core idea as our monadic approach, to allow a threading of state through a complex traversal of the underlying search tree using functional abstractions, and using that state to control the traversal. The Comet approach does not support a notion of composable search transformers. Interestingly the Comet approach to search can also be implemented in C++ using macros and continuations [12].

Functional (Constraint) Logic Programming Several programming languages have been devoted to the integration of Functional Programming and (Constraint) Logic Programming. On the one hand, we have CLP languages with support for a functional notation of predicates, such as Mercury [17] and Ciao

[4]. MERCURY allows the user to program search strategies by using the underlying depth-first search, much like any CLP language. CIAO offers two alternative search strategies, breadth-first search and iterative deepening, in terms of depth-first search by means of program transformation.

On the other hand, we have functional programming languages extended with logic programming features (non-determinism, logical variables). The most prominent of these is the CURRY language, or language family. The PACS CURRY compiler is implemented on top of SICSTUS PROLOG and naturally offers access to its constraint solver libraries; it has a fixed search strategy. However, the KICS CURRY system, implemented in HASKELL, does not offer any constraint solvers; yet, it does provide reflective access to the program's search tree [3], allowing programmed or *encapsulated* search. As far as we can tell, their implementation technique prevents this programmed search from being combined with constraint solving.

Embedding Logic Programming in Functional Programming As far as we know, Constraint Programming has gotten very little attention from mainstream Functional Programming researchers. Most effort has gone towards the study of the related domain of Logic Programming, whose built-in unification can be seen as an equality constraint solver for Herbrand terms.

There are two aspects to Logic Programming, which can and have been studied either together or separately: logical variables and unification on the one hand and (backtracking) search on the other hand.

The former matter can be seen as providing an instance of a Herbrand term equality constraint solver for our `Solver` type class. However, it remains an open issue how to fit the works of Claessen and Ljunglöf [5] and Jansson and Jeuring [9] for adding additional type safety to solver terms into our solver-independent framework.

Logic Programming and Prolog have also inspired work on search strategies in Functional Programming. That is to say, work on Prolog's dedicated search strategy: depth-first search with backtracking. Most notable is the list-based backtracking monad—which Wadler pioneered before the introduction of monads [20]—upon which various improvements have been made, e.g. breadth-first search [15], Prolog's pruning operator *cut* [8], and fair interleaving [10].

The Alma-0 [1] has a similar objective in an imperative setting: it adds Prolog-like depth-first search and pruning features to Modula-2.

FaCiLe is a finite domain constraint library for OCaml, developed as part of the Ph.D. thesis of Nicolas Barnier [2]. FaCiLe's fixed search stratgy is depth-first search; on top of this, optimization is possible by means of both the branch-and-bound and restart strategies. The implementation relies on mutable state.

8 Conclusion and Future Work

We have given a monadic specification of constraint programming in terms of a monadic constraint solver threaded through a monadic search tree. We show how

the tree can be dynamically constructed through so called labelling methods, and the order in which the nodes are visited controlled by a search strategy. The base search strategy can be transformed by search transformers, and indeed these can be constructed as composable transformations. Our framework allows the simple specification of complex search strategies, and illustrates how complex search strategies, like branch-and-bound, or iterative deepening can be built from smaller components. It also gives great freedom to explore new search strategies and transformers, for example the optimistic branch-and-bound search.

Overall by trying to be as generic and modular as possible in defining monadic constraint programming we have a powerful tool for experimentation and understanding of search in constraint programming.

8.1 Future Work:

There are many challenges ahead of the MCP framework. To name just a few important ones: 1) to generalize our search framework to arbitrary search problems, 2) to integrate a Haskell implementation of Constraint Handling Rules [6] with the framework to provide the combination of programmable **s**earch and programmable **s**olving, and 3) to explore the performance characteristics of the framework. Currently, we are integrating the Gecode solver [7] in MCP [24].

Moreover, we think it is an important challenge for Prolog implementations to offer more flexible, programmed search strategies. The stack freezing functionality available in tabulated Prolog systems seems promising to implement the `label` and `goto` methods of the MCP framework and make this possible.

Acknowledgements

I am grateful to the collaborators on the MCP framework: Peter Stuckey, Philip Wadler and Pieter Wuille. Thanks to Christian Schulte for his feedback on the Gecode instance of the MCP framework.

References

1. Krzysztof R. Apt, Jacob Brunekreef, Vincent Partington, and Andrea Schaerf. Alma-o: an imperative language that supports declarative programming. *ACM Trans. Program. Lang. Syst.*, 20(5):1014–1066, 1998.
2. Nicolas Barnier. *Application de la programmation par contraintes à des problèmes de gestion du trafic aérien*. PhD thesis, Institut National Polytechnique de Toulouse, December 2002.
3. Bernd Brassel and Frank Huch. On a tighter integration of Functional and Logic Programming. In Zhong Shao, editor, *5th Asian Symposium on Programming Languages and Systems (APLAS'07)*, volume 4807 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 2007.
4. Amadeo Casas, Daniel Cabeza, and Manuel V. Hermenegildo. A syntactic approach to combining functional notation, lazy evaluation and higher-order in LP systems. In *8th International Symposium on Functional and Logic Programming (FLOPS'06)*, pages 146–162. Springer, 2006.

5. Koen Claessen and Peter Ljunglöf. Typed logical variables in Haskell. In *Proc. of Haskell Workshop* . ACM SIGPLAN, 2000.
6. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
7. Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.
8. Ralf Hinze. Prolog's control constructs in a functional setting - axioms and implementation. *International Journal of Foundations of Computer Science*, (12(2)):125–170, 2001.
9. Patrik Jansson and Johan Jeuring. Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, 1998.
10. Oleg Kiselyov, Chung chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). *SIGPLAN Not.*, 40(9):192–203, 2005.
11. K. Marriott and P.J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.
12. Laurent Michel, Andrew See, and Pascal Van Hentenryck. High-level nondeterministic abstractions in. In Frédéric Benhamou, editor, *CP*, volume 4204 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2006.
13. David Overton. Haskell FD library. `http://overtond.blogspot.com/2008/07/pre.html`, 2008.
14. Christian Schulte. Programming constraint inference engines. In *Principles and Practice of Constraint Programming - CP97, Proceedings*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533. Springer, 1997.
15. Silvija Seres and Michael J. Spivey. Embedding Prolog into Haskell. In *Haskell Workshop'99*, Septembr 1999.
16. Gert Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer, 1995.
17. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 1996.
18. Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. MIT Press, 2005.
19. Pascal Van Hentenryck and Laurent Michel. Nondeterministic control for hybrid search. *Constraints*, 11(4):353–373, 2006.
20. Philip Wadler. How to replace failure by a list of successes. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
21. Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52, London, UK, 1995.
22. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM.
23. Mark Wallace, Stefano Novello, and Joachim Schimpf. ECLiPSe: A platform for constraint logic programming, 1997.
24. Pieter Wuille and Tom Schrijvers. The FD-MCP framework. Report CW 562, Departement of Computer Science, K.U.Leuven, August 2009.

What I Have Learned From All These Solver Competitions

Neng-Fa Zhou

CUNY Brooklyn College & Graduate Center
zhou@sci.brooklyn.cuny.edu

In this talk, I would like to share my experiences gained from participating in four CSP solver competitions and the second ASP solver competition. In particular, I'll talk about how various programming techniques can make huge differences in solving some of the benchmark problems used in the competitions. These techniques include global constraints, table constraints, and problem-specific propagators and labeling strategies for selecting variables and values. I'll present these techniques with experimental results from B-Prolog and other CLP(FD) systems.

What I Have Learned From All These Solver Competitions

Neng-Fa Zhou
CUNY Brooklyn College and Graduate Center

Outline

- Preparations
  - CSP and CLP(FD)
  - Global constraints
  - Table constraints
  - Action rules
- Programming techniques
  - Using global constraints
  - Using table constraints
  - Using specialized propagators
  - Using problem-specific labeling strategies
- Conclusion

Constraint Satisfaction Problems

- CSP
  - *A set of variables $V=\{V_1,\ldots,V_n\}$*
  - *Each variable has a domain $V_i :: D_i$*
  - *A set of constraints*
- *Example*
  - A:{0,1}, B:{0,1}, C:{0,1}
  - C = A and B
- Solution to CSP
  - *An assignment of values to the variables that satisfies all the constraints*

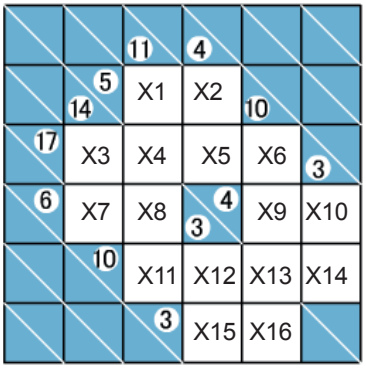
CLP(FD)

- CLP(FD) language
  - An extension of Prolog that provides built-ins for describing and solving CSPs
- CLP(FD) systems
  - B-Prolog, CHIP, ECLiPSe, GNU-Prolog, IF/Prolog, Prolog-IV, SICStus, SWI-Prolog, YAP, …

CLP(FD) - B-Prolog

- Domain constraints
  - X in D
  - X notin D
- Unification and arithmetic constraints
  - Exp R Exp
    - R is one of the following: #=, #\=, #>, #>=, #<, #=
    - Exp may contain +, -, *, /, //, mod, sum, min, max
- Boolean constraints
  - Exp R Exp
    - R is one of the following: #/\, #\/, #=>, #<=>, #\
- Global constraints
- Labeling built-ins

Example



A Kakuro puzzle

```
go:-
    Vars=[X1,X2,…,X16],
    Vars :: 1..9,
    word([X1,X2],5),
    word([X3,X4,X5,X6],17),
    …
    word([X10,X14],3),
    labeling(Vars),
    writeln(Vars).
word(L,Sum):-
    sum(L) #= Sum,
    all_different(L).
```

Global Constraints

- all_different(L)
- all_distinct(L)
- circuit(L)
- cumulative(Starts,Durations,Resources,Limit)

all_different(L) and all_distinct(L)

- all_different(L)
  - Let L=[X_1,…,X_n].
    For each i,j$\in$1..n (i<j) $X_i \neq X_j$.
- all_distinct(L)
  - Maintains some sort of hyper-arc consistency
    - Hall-set finding (B-Prolog and ECLiPSe)
    - Maximal-matching (SICStus)

circuit(L)

- Let L=[X_1,…,X_n], where $X_i \in 1..n$. An assignment
  (X_1/a_1,…,X_n/a_n)
  satisfies this constraint if
  $\{1 \rightarrow a_1, \ldots, n \rightarrow a_n\}$
  forms a Hamiltonian cycle.
- Propagation algorithms
  - Remove non-Hamiltonian arcs as early as possible
    - Avoid sub-cycles
  - Reachability test

cumulative(Starts,Durations,Resources,Limit)

- Starts = [S_1,…,S_n],
  Durations = [D_1,…,D_n],
  Resources = [R_1,…,R_n],
  The resource limit cannot be exceeded at any time
- When Resources=[1,…,1] and Limit=1

  ⇓

  serialized(Starts,Durations)
  - Disjunctive scheduling
    - Edge-finding algorithms are used

Table Constraints

- Positive constraints

```
(X,Y,Z) in [(0,1,1),
            (1,0,1),
            (1,1,0)]
```

- Negative constraints

```
(X,Y,Z) notin [(0,1,1),
               (1,0,1),
               (1,1,0)]
```


Action Rules

Agent, Condition, {EventSet} => Action

- Events
  - Instantiation: `ins(X)`
  - Domain
    - `bound(X)`, `dom(X)`, `dom(X,E)`, `dom_any(X)`, and `dom_any(X,E)`
  - Time : `time(X)`
  - GUI
    - `actionPerformed(X)`, `mouseClicked(X,E)`…
  - General
    - `event(X,O)`

Applications of Action Rules

- Lazy evaluation

```
freeze(X,G), var(X), {ins(X)} => true.
freeze(X,G) => call(G).
```

- Constraint propagators

```
'X in C-Y_ac'(X,Y,C),var(X),var(Y),
      {dom(Y,Ey)}
      =>
      Ex is C-Ey,
      exclude(X,Ex).
'X in C-Y_ac'(X,Y,C) => true.
```


Using Global Constraints
all_distinct(L) (1)

- Graph coloring
  - Model-1 (neq)
    - For each two neighbors i and j, $C_i \neq C_j$
  - Model-2 (all_distinct)
    - For each complete subgraph $\{i_1,i_2,\ldots,i_k\}$, all_distinct([Ci_1, Ci_2,…, Ci_k])
    - post_neqs(Neqs) in B-Prolog

Using Global Constraints
all_distinct(L) (2)

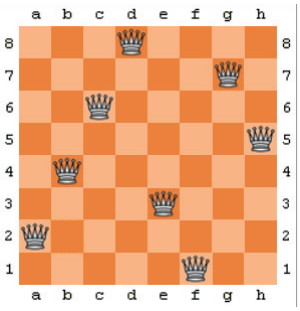
- Benchmarking results (seconds)

Benchmark	Model-1(neq)	Model-2 (all_distinct)
color-1-FullIns-5	> 3600	> 3600
color-3-FullIns-5	> 3600	> 3600
color-4-FullIns-4	> 3600	10.81
color-4-FullIns-5	> 3600	2091.74
color-5-FullIns-4	> 3600	41.59

Source of benchmarks:
Agostino Dovier, Andrea Formisano, and Enrico Pontelli,
A comparison of CLP(FD) and ASP solutions to NP-complete problems,
ICLP'05.

Using Global Constraints
all_distinct(L) (3)

- N-Queens problem



- Model-1 (neq)
  - For $i,j \in 1..n$ $(i < j)$
    $Q_i \neq Q_i$
    $Q_i - Q_j \neq (j-i)$
    $Q_j - Q_i \neq (j-i)$
- Model-2 (all_distinct)
  - all_distinct($[Q_1,\ldots,Q_n]$),
    all_distinct($[Q_1,Q_2-1,\ldots,Q_n-n]$),
    all_distinct($[Q_1,Q_2+1,\ldots,Q_n+n]$)

Using Global Constraints
all_distinct(L) (4)

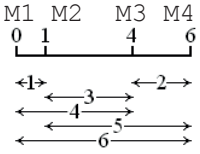
- Benchmarking results (ms)

Benchmark	Model-1(neq)	Model-2 (all_distinct)
blockedqueens.28.1449787798	46	16
blockedqueens.28.1449787894	32	31
blockedqueens.28.1449787934	15	31
blockedqueens.28.1449787988	16	16
blockedqueens.28.1449788117	31	62
blockedqueens.28.1449788237	141	219
blockedqueens.28.1449788307	31	16
blockedqueens.28.1449789281	31	62
blockedqueens.28.1449789491	16	31
blockedqueens.28.1449789909	62	94
blockedqueens.28.1449790187	47	47
blockedqueens.28.1449790413	63	109
blockedqueens.28.1449790708	172	16
blockedqueens.28.1449791337	93	156
blockedqueens.28.1449791430	0	16
blockedqueens.28.1449791733	63	78
blockedqueens.28.1449791778	47	78
blockedqueens.28.1449791905	16	0
blockedqueens.28.1449792036	15	15
blockedqueens.28.1449793568	94	110

Source of benchmarks: 2nd ASP Competition

Using Global Constraints
all_distinct(L) (5)

- Some times all_different(L) is faster than all_distinct(L)
- An example: Golomb ruler




```
all_different([M2-M1,M3-M1,M4-M1,
               M3-M2,M4-M2,
               M4-M3])
```

Using Global Constraints
all_distinct(L) (6)

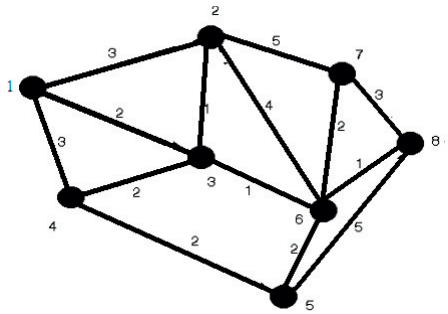
- Benchmarking results (seconds)

Benchmark	all_different	all_distinct
golomb-8-positions-100-16	0.03	0.05
golomb-8-positions-50-16	0.03	0.06
golomb-10-positions-100-36	2.20	5.09
golomb-10-positions-125-36	2.20	5.11
golomb-10-positions-75-36	2.19	5.11
golomb-11-positions-100-35	47.72	125.39
golomb-11-positions-125-35	47.22	125.87
golomb-11-positions-75-35	45.45	121.12
golomb-12-positions-100-48	476.53	> 600
golomb-12-positions-125-48	478.11	> 600
golomb-12-positions-150-48	479.34	> 600

Source of benchmarks: 2nd ASP Competition

Using Global Constraints
circuit(L)

- The Traveling Salesperson Problem




```
tsp(Vars):-
      Vars=[V1,V2,…,V8],
      V1 :: [2,3,4],
      V2 :: [1,3,6,7],
      …
      V8 :: [5,6,7],
      circuit(Vars).
```

Using Global Constraints serialized(Starts,Durations) (1)

- Scheduling
  - Model-1: use disjunctive constraints

```
VV5+97#=<VV17#\/VV17+52#=<VV5,
VV5+97#=<VV26#\/VV26+59#=<VV5,
VV5+97#=<VV32#\/VV32+41#=<VV5,
VV5+97#=<VV49#\/VV49+63#=<VV5,
                          ….
```

  - Model-2: use global constraints
    - `post_disjunctive_tasks(Disjs)` in B-Prolog
      `Disjs=[disj_tasks(S1,D1,S2,D2),…]`
      - converts disjunctive constraints into serialized

Using Global Constraints serialized(Starts,Durations) (2)

- Benchmarking results

Benchmark	Model-1(dis)	Model-2 (serialized)
os-taillard-15-95-0	> 600	0.22
os-taillard-15-95-1	> 600	0.22
os-taillard-15-95-2	> 600	0.22
os-taillard-15-95-3	> 600	0.20
os-taillard-15-95-4	> 600	0.20
os-taillard-15-95-5	> 600	0.20
os-taillard-15-95-6	> 600	0.22
os-taillard-15-95-7	> 600	0.20
os-taillard-15-95-8	> 600	> 600
os-taillard-15-95-9	> 600	0.27

Source of benchmarks:
www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html

Using Table Constraints (1)

- The Schur number problem
  - Partition n positive integers into m sets such that all of the sets are sum-free.
- Model-1 (sum-free triplet)

$$S_i = S_j \rightarrow S_{i+j} \neq S_i$$

- Mode-2 (use redundant constraints)

$$S_i=S_j \rightarrow S_{i+j}\neq S_i,\ S_i=S_{i+j} \rightarrow S_j\neq S_i,\ S_j=S_{i+j} \rightarrow S_i\neq S_j.$$

- Mode-3 (use table constraints)

$$(S_i,S_j,S_{i+j})\ \text{notin}\ [(1,1,1),(2,2,2),\ldots]$$

Using Table Constraints (2)
The Schur number problem

- Benchmarking results (seconds)

Benchmark	Model-1	Model-2 (redundant)	Model-3 (table)
15.1.schur.lp	> 600	> 600	441.72
15.10.schur.lp	> 600	> 600	432.56
15.14.schur.lp	> 600	> 600	> 600
15.16.schur.lp	> 600	> 600	> 600
15.19.schur.lp	> 600	> 600	> 600
15.20.schur.lp	> 600	> 600	328.96
15.3.schur.lp	> 600	> 600	252.20
15.4.schur.lp	> 600	> 600	> 600
15.5.schur.lp	> 600	> 600	393.90

Source of benchmarks: 2nd ASP Competition

Using Table Constraints (3)
The Knights Problem

- Model-1: use disjunctive constraints

```
(abs(P1//N-P2//N)#=1 #/\ abs(P1 mod N-P2 mod N)#=2) #\/
(abs(P1//N-P2//N)#=2 #/\ abs(P1 mod N-P2 mod N)#=1)
```

- Model-2: use table constraints

```
(P1,P2) in [(0,6),(0,9),(1,7),(1,8),(1,10),…]
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Using Table Constraints (4)
The Knights Problem

- Benchmarking results (seconds)

Benchmark	Model-1 (dis)	Model-2(table)
knights-10-5	25.20	0.02
knights-12-5	74.84	0.03
knights-12-9	> 600	0.09
knights-15-5	283.42	0.09
knights-15-9	> 600	0.28
knights-20-5	> 600	0.28
knights-20-9	> 600	1.00
knights-25-5	> 600	0.67
knights-25-9	> 600	2.64
knights-50-25	> 600	181.51
knights-50-5	> 600	9.35
knights-50-9	> 600	40.53
knights-8-5	6.44	0.02

Source of benchmarks:
www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html

Not Using Table Constraints (1)

- The Black Hole problem

```
(X,Y) notin [(0,0),(1,1),(2,2),…]
```

- Transform table constraints

```
(X,Y) notin [(0,0),(1,1),(2,2),…],
          ↓
       X #\= Y
          ↓
  all_distinct(…)
```


Not Using Table Constraints (2)
The Black Hole problem

- Benchmarking results (seconds)

Benchmark	table	all_distinct
BlackHole-4-13-e-1_ext	>1800	2.641
BlackHole-4-13-e-2_ext	>1800	2.719
BlackHole-4-13-e-3_ext	>1800	2.703
BlackHole-4-13-m-0_ext	>1800	2.735
BlackHole-4-13-m-1_ext	>1800	2.703
BlackHole-4-13-m-2_ext	>1800	2.657
BlackHole-4-4-e-0_ext	>1800	0.031
BlackHole-4-4-e-1_ext	>1800	0.016
BlackHole-4-4-e-2_ext	>1800	0.032
BlackHole-4-4-e-3_ext	>1800	0.031
BlackHole-4-4-e-4_ext	>1800	0.016
BlackHole-4-4-e-5_ext	>1800	0.015
BlackHole-4-4-e-6_ext	>1800	0.031

www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html

Using Specialized Propagators(1)

- Example 1: abs(X-Y)#=N

```
fd_abs_diff_ins(X,Y,N),var(X),{ins(X)} => true.
fd_abs_diff_ins(X,Y,N) =>
    Ey1 is X-N,
    Ey2 is X+N,
    Y in [Ey1,Ey2].

fd_abs_diff_dom(X,Y,N),var(X),var(Y),
   {dom_any(X,Ex)}
=>
    Ey1 is Ex-N, Ex1 is Ey1-N,
    (fd_false(X,Ex1)->fd_set_false(Y,Ey1);true),
    Ey2 is Ex+N, Ex2 is Ey2+N,
    (fd_false(X,Ex2)->fd_set_false(Y,Ey2);true).
```


Using Specialized Propagators(2)
The Schur Number Problem

- Model-4 (use specialized propagators)

```
not_the_same(X,Y,Z),n_vars_gt(3,1),
       {ins(X),ins(Y),ins(Z)}
=>
       true.
not_the_same(X,Y,Z),X==Y => fd_set_false(Z,X).
not_the_same(X,Y,Z),X==Z => fd_set_false(Y,X).
not_the_same(X,Y,Z),Y==Z => fd_set_false(X,Y).
not_the_same(X,Y,Z) => true.
```

Using Specialized Propagators(3)

- Benchmarking results (seconds)

Benchmark	Model-3 (table)	Model-4 (specialized)
15.1.schur.lp	441.72	155.23
15.10.schur.lp	432.56	256.78
15.14.schur.lp	> 600	435.89
15.16.schur.lp	> 600	348.46
15.19.schur.lp	> 600	486.20
15.20.schur.lp	328.96	128.00
15.3.schur.lp	252.20	106.32
15.4.schur.lp	> 600	> 600
15.5.schur.lp	393.90	138.04

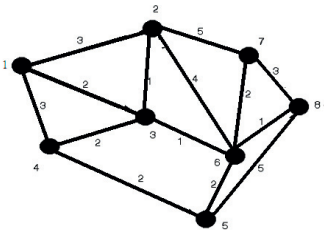
Source of benchmarks: 2nd ASP Competition

Using Problem-Specific Labeling Strategies

- Variable selection
  - queens: labeling([ff],Vars)
  - golomb: labeling([],Vars)
- Value selection
  - tsp: select an edge with the lowest weight

Value Selection

- Select an edge with the lowest weight



```
tsp(Vars):-
      Vars=[V1,V2,…,V8],
      V1 :: [2,3,4],
     put_attr_no_hook(V1,nbs,[3,2,4]),
     V2 :: [1,3,6,7],
     put_attr_no_hook(V2,nbs,[3,1,6,7]),
      …
      V8 :: [5,6,7],
     pub_attr_no_book(V8,nbs,[6,7,5]),
      circuit(Vars).
```

```
Use get_attr(V,nbs,Nbs),member(V,Nbs)
    rather than indomain(V) to label V.
```


Conclusion

- Techniques
  - Using global constraints
  - Using table constraints
  - Using specialized propagators
  - Using problem-specific labeling strategies
- More techniques and systems to explore
  - Integrating CLP(FD) with SAT and ASP solvers
- Thanks!
  - Organizers of the solver competitions
  - Program committee of WLP'09

An ER-based Framework for Declarative Web Programming*

Michael Hanus Sven Koschnicke

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de sven@koschnicke.de

Abstract. We describe a framework to support the implementation of web-based systems to manipulate data stored in relational databases. Since the conceptual model of a relational database is often specified as an entity-relationship (ER) model, we propose to use the ER model to generate a complete implementation in the declarative programming language Curry. This implementation contains operations to create and manipulate entities of the data model, supports authentication, authorization, session handling, and the composition of individual operations to user processes. Furthermore and most important, the implementation ensures the consistency of the database w.r.t. the data dependencies specified in the ER model, i.e., updates initiated by the user cannot lead to an inconsistent state of the database. In order to generate a high-level declarative implementation that can be easily adapted to individual customer requirements, the framework exploits previous works on declarative database programming and web user interface construction in Curry.

Note: The full version of the paper will appear in the Proceedings of the Twelfth International Symposium on Practical Aspects of Declarative Languages (PADL 2010), Springer Lecture Notes in Computer Science

* This work was partially supported by the German Research Council (DFG) under grant Ha 2457/5-2.

Practical Applications of Extended Deductive Databases in DATALOG*

Dietmar Seipel

University of Würzburg, Department of Computer Science
Am Hubland, D – 97074 Würzburg, Germany
seipel@informatik.uni-wuerzburg.de

Abstract. A wide range of additional forward chaining applications could be realized with deductive databases, if their rule formalism, their immediate consequence operator, and their fixpoint iteration process would be more flexible.
Deductive databases normally represent knowledge using stratified DATALOG programs with default negation. But many practical applications of forward chaining require an extensible set of user–defined built–in predicates. Moreover, they often need function symbols for building complex data structures, and the stratified fixpoint iteration has to be extended by aggregation operations.
We present an new language DATALOG*, which extends DATALOG by stratified meta–predicates (including default negation), function symbols, and user–defined built–in predicates, which are implemented and evaluated top–down in PROLOG. All predicates are subject to the same backtracking mechanism. The bottom–up fixpoint iteration can aggregate the derived facts after each iteration based on user–defined PROLOG predicates.

Keywords.
Deductive databases, PROLOG, forward / backward chaining, bottom–up, top–down, built–in predicates, stratification, function symbols, XML

1 Introduction

Deductive databases allow for efficiently deriving inferences from flat tables using forward chaining and relational database technology. Most deductive database systems support the representation language of stratified DATALOG$^{\mathsf{not}}$ [8, 13]. Disallowing function symbols and enforcing the condition of range–restrictedness (safety)[1] guarantees that the inference process will always terminate for DATALOG. In principle, the fixpoint interation based on forward chaining can also be applied to the extension by function symbols, but termination is not always guaranteed, if function symbols can occur in rule heads. Some DATALOG extensions allow for a very limited set of built–in predicates under an extended safety condition. In DATALOG$^{\mathsf{not}}$, stratified[2] default negation is possible.

[1] all variable symbols of a rule must occur in the positive body
[2] there is no cycle including default negation in the predicate dependency graph

DATALOG rules extended by existentially quantified variables in rule heads are known as *tuple generating dependencies*. They have been used, e.g., for enabling ontological knowledge representation and efficient reasoning [7] and for specifying generalized schema mappings in databases [12]. In the former paper, also stratified default negation has been considered.

For handling non–stratified default negation and disjunctive rule heads, *answer set programming* (ASP) can be used [2, 11]; the advantage of an efficient query evaluation for large relational databases is partly lost in ASP, but much more complex problems – of a high computational complexity – can be encoded elegantly. Today, even larger graph–theoretic problems, such as graph colouring, can be encoded in a very compact way and solved by ASP in reasonable time.

On the other hand, *logic programming* in PROLOG [5, 9] is not as declarative as deductive databases and ASP, but backward chaining and side effects make it a fully fledged programming language. PROLOG allows for function symbols, and it can handle stratified default negation. However, recursion can lead to non–termination, even if there are no function symbols. Although the PROLOG extension XSB can solve this termination problem using tabling (memoing) techniques, there exist many applications where backward chaining is not suitable.

Usually, dedicated special–purpose problem solvers are used for more general *forward chaining* problems. Due to their lack of declarativity, these solvers often are hard to maintain, difficult to extend and port to similar application domains. Thus, we are proposing *extended deductive databases* (EDDB) based on a DATALOG extension named DATALOG*, which combines declarative forward chaining with meta–predicates, function symbols, and built–in predicates implemented in PROLOG. So far, we have used DATALOG* for the following forward chaining applications:

- diagnostic reasoning in medical or technical domains, e.g., *d3web* [14] or root cause detection in computer networks,
- anomaly detection in ontologies extended by rules, such as the extension SWRL of the ontology language OWL, and
- meta–interpreters, e.g. for disjunctive reasoning with the hyperresolution consequence operator $\mathcal{T}^s_\mathcal{P}$ in disjunctive deductive databases.

These practical EDDB applications require PROLOG meta–predicates (such as default negation `not/1` and the list predicates `findall/3` and `maplist/2,3`), recursion on cyclic data, and function symbols for representing complex data structures, such as lists or semi–structured data and XML [1]. Sometimes, the standard conjunctive rule bodies are not adequate: the knowledge representation becomes too complicated, and the evaluation is unnecessarily complex due to redundancy, if rules with non–conjunctive rule bodies are normalized to sets of rules with conjunctive rule bodies. Recently, [4] has defined an extended version of range–restricted DATALOG rules with non–conjunctive rule bodies.

In general, meta–predicates need to be stratified to ensure termination. It is easy to decide if a DATALOG$^{\mathsf{not}}$ program can be stratified. But DATALOG*

programs with arbitrary PROLOG meta–predicates have to be analysed carefully using heuristics based on extended call graphs to find out which predicates call which other predicates through meta–predicates. Although this problem is undecidable in general, suitable heuristics have been published in [17], even for the more general context of PROLOG.

The rest of this paper is organized as follows: In Section 2, we indicate how DATALOG* mixes forward chaining with PROLOG's backward chaining for built–in predicates. Section 3 presents three case studies for DATALOG*. In Section 4, we describe a possible meta–interpreter for DATALOG*, which we have implemented in PROLOG. Finally, we give conclusions and sketch some future work.

2 The General Idea of DATALOG*

We distinguish between DATALOG* rules and PROLOG rules. Syntactically, DATALOG* rules are PROLOG rules; i.e., they may contain function symbols (in rule heads and bodies) as well as negation, disjunction, and PROLOG predicates in rule bodies. As forward chaining rules, DATALOG* rules are evaluated bottom–up, and all possible conclusions are derived. The supporting PROLOG rules are evaluated top–down, and – for efficiency reasons – only on demand, and they can refer to DATALOG* facts. The PROLOG rules are also necessary for expressivity reasons: they are used for computations on complex terms, and – more importantly – for computing very general aggregations of DATALOG* facts.

DATALOG* rules cannot be evaluated in PROLOG or DATALOG alone for the following reasons: Current DATALOG engines cannot handle function symbols and non–ground facts, and they do not allow for the embedded computations (arbitrary built–in predicates), which we need for our practical applications. Standard PROLOG systems may loop in recursion forever (e.g., when computing the transitive closure of a cyclic graph), and they may be inefficient, if there are subqueries that are posed and answered multiply. Thus, they have to be extended by some DATALOG* facilities (our approach) or memoing/tabling facilities (the approach of the PROLOG extension XSB).

Since we need forward chaining, and since the embedding system DDK [15] is developed in SWI–PROLOG, we have implemented a new inference machine in standard PROLOG that can handle mixed, stratified DATALOG*/PROLOG rule systems. The evaluation of a DATALOG* program $\mathcal{D}$ mixes forward–chained evaluation of DATALOG with SLDNF–resolution of PROLOG, see Figure 1. The body atoms B_i of a DATALOG* rule $A \leftarrow B_1 \wedge \ldots \wedge B_n$ are evaluated backward in PROLOG based on previously derived facts and based on a PROLOG program $\mathcal{P}$ using SLDNF–resolution.

Due to the PROLOG evaluation of rule bodies, variable symbols appearing only under default negation are implicitely quantified as *existential*. In contrast to [6], we do not need an explicit program transformation. For *range–restrictedness*, we just require that all variable symbols appearing in the head of a rule must also occur in at least one positive body atom. Moreover, similarly to PROLOG,

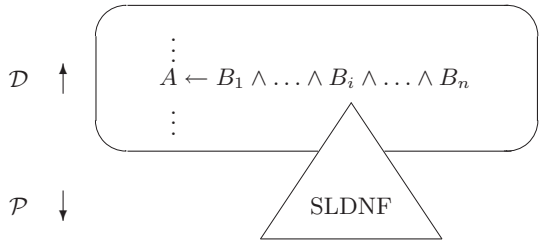


Fig. 1. Mixing Forward and Backward Chaining.

the programmer has to ensure that the standard left–to–right evaluation of the rule bodies will be adequate; in our case, on backtracking it should instantiate all variable symbols of the head in finitely many ways. This will guarantee, that every single iteration of the fixpoint process for DATALOG* derives only a finite set of ground facts. In contrast to [4], we do not allow a reordering of the body literals during the evaluation, since built–in predicates in DATALOG* can be arbitrary PROLOG predicates including meta–predicates and predicates with side effects.

Because of the embedded calls to PROLOG, a formal definition of the semantics of DATALOG* would be rather technical and difficult to understand. Instead, we will describe a compact meta–interpreter for DATALOG* in Section 4.

3 Case Studies for DATALOG*

In the following, we present three case studies for DATALOG* that require forward chaining together with built–in and user–defined PROLOG predicates. These practical applications could not be implemented so elegantly in DATALOG or PROLOG alone.

3.1 Diagnostic Reasoning

Diagnostic reasoning in *d3web* [14] requires forward chaining and built–in predicates for invoking user dialogs. We have implemented diagnostic reasoning in a declarative way using DATALOG*, cf. [18]. The DATALOG* rules use meta–predicates (such as `not/1`, `m_to_n/2`, `maplist/3`, and `findall/3`), and after each iteration of the immediate consequence operator, the derived facts are aggregated by combining the scores of different derivations of the same diagnosis.

For example, the first DATALOG* rule below assigns the value `1` to the intermediate diagnose `I3` by combining the answers to the questions `Q1`, `Q2`, and `Q3` during an interactive examination dialog. The second diagnositic rule assigns a score `16` to the diagnose `D2` based on the intermediate diagnose `I3` and the answer to the question `Q4`.

```
finding('I3' = 1) :-
   condition('Q1' = 2),
   ( condition('Q2' = 3)
   ; condition('Q3' = 2) ).

diagnosis('D2' = 16) :-
   condition('I3' = 1),
   condition('Q4' = 5).
```

The questions are asked while the DATALOG* rules are evaluated. An example of an interactive user dialog is given in Figure 2.

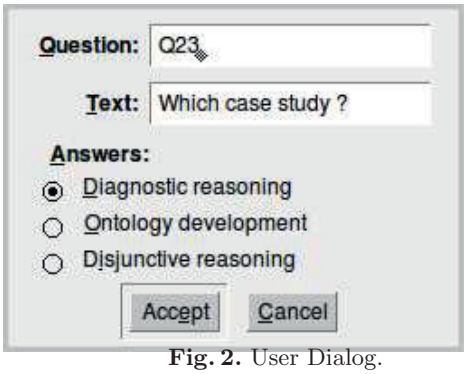


Fig. 2. User Dialog.

If a question `Qid` has not been asked yet, then a body atom `condition(Qid` Θ `V)` in a DATALOG* rule causes the call `dialog(Qid = Val)` of a suitable PROLOG dialog for determining the answer `Val` to the question. This value `Val` is then asserted in the PROLOG database as an atom `finding(Qid = Val)`, and it can be compared with `V` using the operator Θ, which is called `Comparator` below.

```
condition(C) :-
   C =.. [Comparator, Qid, V],
   ( finding(Qid = Val)
   ; dialog(Qid = Val),
     assert(finding(Qid = Val)) ),
   apply(Comparator, [Val, V]).
```

We can prevent the same instance of a DATALOG* rule from firing twice. Given a range–restricted DATALOG* rule `Head :- Body`, the following modified rule will fire at most once:

```
Head :-
   Body,
   not( has_fired(Head, Body) ),
   assert( has_fired(Head, Body) ).
```

The modified rules can be obtained automatically by a simple program transformation.

For diagnostic reasoning in *d3web*, all rules are ground, and it is necessary that each rule can fire at most once. The following aggregation predicate adds the scores in the list of derived facts for the same diagnosis; the other facts remain unchanged:

```
d3_aggregate_facts(I, J) :-
   findall( diagnosis(D=S),
      ( bagof( T, member(diagnosis(D=T), I), Ts ),
        add(Ts, S) ),
      J1 ),
   findall( A,
      ( member(A, I),
        not( functor(A, diagnosis, 1) ),
      J2 ),
   append(J1, J2, J).
```

3.2 Ontology Development

For the development of practical semantic applications, ontologies are commonly used with rule extensions. The integration of ontologies creates new challenges for the design process of such ontologies, but also existing evaluation methods have to cope with the extension of ontologies by rules. Since the verification of OWL ontologies with rule extensions is not tractable in general, we propose to *verify* and *analyze* ontologies at the symbolic level by using a declarative approach based on DATALOG*, where known *anomalies* can be easily specified and tested in a compact manner [3].

Our DATALOG* implementation requires meta–predicates such as `setof/3` and `maplist/2` for aggregation in rule bodies; moreover, for convenience, the junctor or (";") is used in addition to and (",") in rule bodies. The DATALOG* program can be *stratified* into two layers $\mathcal{D}_1$ and $\mathcal{D}_2$ of DATALOG* rules; below, we show a few of the rules. The rules for the predicates `anomaly/2` and `tc_derives/2` are part of the upper layer $\mathcal{D}_2$, and the rules for `derives/2`, `sibling/2`, and `disjoint/2` are part of the lower layer $\mathcal{D}_1$. $\mathcal{D}_1$ is applied to the DATALOG* facts for the basic predicates (such as `subclass_of/2`), which have to be derived from an underlying rule ontology. The resulting DATALOG* facts

are the input for $\mathcal{D}_2$. The stratification is necessary, because $\mathcal{D}_2$ refers to $\mathcal{D}_1$ through negation and aggregation.

```
anomaly(circularity, C) :-
   tc_derives(C, C).

anomaly(lonely_disjoint, C) :-
   class(C), siblings(_, Cs), disjoints(C, Cs),
   not( sibling(C, M), disjoint(C, M) ).
```

Firstly, an obvious equivalence exists between a subclass relationship between two classes C and D and a rule $A \leftarrow B$ with a single body atom B, such that A and B have the same arguments and the unary predicate symbols D and C, respectively. Thus, we combine them into the single formalism `derives/2` and compute the transitive closure in DATALOG*. Every class C contained in a cycle forms an anomaly, which is detected as `tc_derives(C, C)`.

```
tc_derives(X, Y) :-
   derives(X, Y).
tc_derives(X, Y) :-
   derives(X, Z), tc_derives(Z, Y).
```

Secondly, a class C is called a *lonely disjoint*, if it is disjoint to a set of siblings, and it does not have a sibling M with which it is disjoint. The first of the following PROLOG rules *aggregates* the siblings `Y` of a class `X` to a list `Ys` using the meta–predicate `setof/3`, and the second PROLOG rule tests if a given class `X` is disjoint to all classes in the list `Ys` using the meta–predicate `maplist/2`:

```
siblings(X, Ys) :-
   setof( Y, sibling(X, Y), Ys ).

disjoints(X, Ys) :-
   maplist( disjoint(X), Ys ).
```

The call to `setof/3` succeeds for every class `X` having siblings, and it computes the list `Ys` of all siblings `Y` of `X`; on backtracking, the siblings of the other classes `X` are computed. This means, `setof/3` does a *grouping* on the variable `X`. The rule for `siblings/2` could also be evaluated as a forward rule, but the rule for `disjoints/2` could not, since it is not range–restricted.

The lower layer $\mathcal{D}_1$ contains the following rule. We treat it as a DATALOG* rule, instead of a PROLOG rule, since we want to derive all pairs of siblings.

```
sibling(X, Y) :-
   subclass_of(X, Z), subclass_of(Y, Z), X \= Y.
```

3.3 Disjunctive Reasoning

In disjunctive deductive databases [11], the definite consequence operator $\mathcal{T}_\mathcal{P}$ has been generalized to the disjuntive hyperresolution operator $\mathcal{T}^s_\mathcal{P}$:

$$\mathcal{T}^s_\mathcal{P}(S) = \{\, C \vee C_1 \vee \ldots \vee C_m \mid C, C_1, \ldots, C_m \in \mathit{DHB}_\mathcal{P} \text{ and there is a rule } C \leftarrow B_1 \wedge \ldots \wedge B_m \in \mathit{gnd}\,(\mathcal{P}) : \forall\, i \in \langle\, 1, m \,\rangle : B_i \vee C_i \in S \,\}.$$

Encoding disjunctive reasoning in DATALOG* requires built–in predicates for standard operations on disjunctions and disjunctive Herbrand states $S \subseteq \mathit{DHB}_\mathcal{P}$, such as disjunction, union, and subsumption. A disjunction can be represented as a list of atoms. In [16], a disjunctive rule $r = C \leftarrow B_1 \wedge \ldots \wedge B_m$ without default negation is translated to a definite DATALOG* rule

$$\mathit{dis}(C_0) \leftarrow \mathit{dis}(B_1, C_1) \wedge \ldots \wedge \mathit{dis}(B_m, C_m) \wedge \mathit{merge}([C, C_1, \ldots, C_m], C_0),$$

where $B_1, \ldots, B_n$ are the body atoms of r, the list C represents the head of r, and $C_0, C_1, \ldots, C_n$ are distinct fresh variables.

The PROLOG calls $\mathit{dis}(B_i, C_i)$ ground instantiate the atoms B_i, and they instantiate the variables C_i to lists of ground atoms. If r is range–restricted – i.e., all variable symbols in C occur in at least one of the body atoms B_i – then this will also instantiate C to a list of ground atoms. The call $\mathit{dis}(B_i, C_i)$ finds already derived disjunctions containing the atom B_i and returns the list C_i of the remaining atoms as follows:

```
dis(B, C) :-
   dis(D), delete_atom(B, D, C).
delete_atom(B, D, C) :-
   append(D1, [B|D2], D), append(D1, D2, C).
```

The PROLOG predicate *merge/2* computes the disjunction of a list of disjunctions. After each iteration of the immediate consequence operator, an *aggregation* operator eliminates subsumed disjunctions.

Alternatively, the disjunctive rule $r = C \leftarrow B_1 \wedge \ldots \wedge B_m$ can be represented as a DATALOG* fact of the form $\mathit{rule}(C\text{–}[B_1, \ldots, B_n])$; then, a single, generic DATALOG* rule is sufficient, namely the following rule:

```
dis(D) :-
   rule(C-Bs),
   maplist( dis, Bs, Cs ),
   merge([C|Cs], D).
```

In [16] it is shown, that also disjunctive rules with default negation can be translated while preserving the stable model semantics. It is known that the well–founded model is a subset of all stable models (considered as sets of literals). Thus, by computing the well–founded semantics of the resulting program, we could approximate the stable model semantics based on DATALOG*.

4 A Meta–Interpreter for DATALOG*

In the following, we sketch an inference engine for stratified DATALOG*, which we have used successfully for our particular applications. We have implemented it as a meta–interpreter using the well–known PROLOG system SWI [19]; the user dialogs have been built with its publicly available graphical API.

4.1 The Immediate Consequence Operator

The generalized immediate consequence operator $\mathcal{T}_{\mathcal{D},\mathcal{P}}$ operates on a forward program $\mathcal{D}$ and an auxilliary PROLOG program $\mathcal{P}$. In the implementation below, $\mathcal{D}$ is given as a list `Datalog` of rules, whereas $\mathcal{P}$ is stored in the modules `M` and `user` (the standard module) of the PROLOG database. The following predicate calls all rule bodies of `Datalog`; the calls are executed in the module `M`. The set `Facts` of derived head facts will be stored in `M` only afterwards.

```
tp_operator(Datalog, M, Facts) :-
   findall( Head,
      ( member(Head :- Body, Datalog),
        call(M:Body) ),
      Facts ).
```

For all body predicates of $\mathcal{D}$ there have to be either rules (or facts) in $\mathcal{D} \cup \mathcal{P}$ or `dynamic` declarations in $\mathcal{P}$; otherwise, a call to such a predicate would raise an exception. And there can be rules in both $\mathcal{D}$ and $\mathcal{P}$. A body predicate that is solely defined by rules in $\mathcal{P}$ which do not refer to predicates from $\mathcal{D}$ could be considered as a built–in predicate of $\mathcal{D}$. The rules of $\mathcal{P}$ are evaluated top–down using PROLOG's SLDNF–resolution.

Since the evaluated forward program $\mathcal{D}$ is not part of the PROLOG database, the forward rules do not call each other recursively within a single $\mathcal{T}_{\mathcal{D},\mathcal{P}}$–operator, and they cannot be called from backward rules. The recursion is only reflected in the bottom–up fixpoint iteration of $\mathcal{T}_{\mathcal{D},\mathcal{P}}$.

4.2 Managing and Aggregating Facts in a Module

We use two elementary predicates for asserting/retracting a given list `Facts` of facts in a module `Module` of the PROLOG database using the predicate `do/2` from the well–known `loops` package of Schimpf:

```
assert_facts(Module, Facts) :-
   foreach(A, Facts) do assert(Module:A).
retract_facts(Module, Facts) :-
   foreach(A, Facts) do retract(Module:A).
```

For adding a list `Facts` of facts to `Module`, we need to know the list `I` of facts that are already stored in `Module`. First, these facts are retracted from `Module`, then `I` and `Facts` are aggregated using a user–defined plugin predicate (if there is no such predicate, then no aggregation is done), and finally the result is asserted in `Module`.

```
aggregate_facts(Module, I, Facts, J) :-
   retract_facts(Module, I),
   ( aggregate_facts(I, Facts, J)
   ; ord_union(I, Facts, J) ),
   assert_facts(Module, J).
```

`aggregate_facts/3` is a plugin predicate that can be specified using application specific PROLOG rules. For example, for diagnostic reasoning in *d3web*, we use the following plugin predicate:

```
aggregate_facts(I, Facts, J) :-
   append(I, Facts, K),
   d3_aggregate_facts(K, J).
```

Also Δ–iteration with subsumption [10] could be implemented by a suitable plugin.

4.3 The Fixpoint Iteration with Aggregation

For a given set `Datalog` of forward rules, `tp_iteration/3` derives a set `Facts` in module `M`:

```
tp_iteration(Datalog, M, Facts) :-
   tp_iteration(Datalog, M, [], Facts).

tp_iteration(Datalog, M, I, K) :-
   tp_operator(Datalog, M, Facts),
   ( tp_terminates(I, Facts) -> K = I
   ; aggregate_facts(M, I, Facts, J),
     tp_iteration(Datalog, M, J, K) ).

tp_terminates(I, Facts) :-
   not(member(A, Facts), not(member(A, I))).
```

Given a set `I` of facts, `tp_iteration/4` derives a new set `K` in module `M`. The derived facts are stored in `M` and kept on the argument level (`I`, `J`, `K`). The

iteration terminates, if all derived facts have already been known. The derived facts in `M` are necessary for `tp_operator/3`. Since they are mixed with the facts and the rules of the auxilliary PROLOG program $\mathcal{P}$ in `M`, they cannot be extracted from `M` at the end of the iteration; therefore, they also have to be kept on the argument level.

4.4 Stratification

We have implemented a PROLOG library for the stratification of DATALOG* programs. Sophisticated, heuristic methods of program analysis, which can also be extended by the user, are used to determine embedded calls in PROLOG meta–predicates based on suitably extended call graphs, cf. [17]. Thus, we can partition a DATALOG* program into strata, which can be evaluated successively.

For stratification, we have to analyze $\mathcal{D} \cup \mathcal{P}$; thus, $\mathcal{P}$ has to be available on the argument level, too. In practice, all PROLOG rules in the system could be used by the immediate consequence operator; but we only need to analyze the portion $\mathcal{P}'$ of the rules that access the facts for DATALOG* predicates in the PROLOG database.

In DATALOG*, most PROLOG meta–predicates require a stratified evaluation (e.g., `not/1`, `findall/3`, `setof/3`, and `maplist/2,3`). Only the ASP extension of deductive databases and logic programming can handle non–stratified default negation (`not/1`) as well; but, ASP solvers do not support function symbols and general built–in predicates.

4.5 Side Effects in Forward Rules

In our current implementation, the forward rules are fired successively, and it is possible that the forward rules update the PROLOG database module using `assert/1` and `retract/1` in rule bodies. We call these updates *side effects* – in contrast to the assertions of derived facts done by `assert_facts/4`, that are inherent to our approach.

The temporary facts asserted by side effects need not be derived by the rules, but they can nevertheless be used by other forward rules. These temporary facts are hidden in the PROLOG database module, they are not derived by the immediate consequence operator, and normally they will not be part of the final result of the fixpoint iteration. However, if desired, the user can bring these hidden facts to the surface by suitable helper rules for deriving them. For example, the second rule in the DATALOG* program below is a helper rule for deriving the atom b that was asserted by the first rule:

```
a :- assert(b).
b :- b.
```

During fixpoint iteration, DATALOG* makes asserted facts available to further derivations by subsequent rules within the same iteration. In DATALOG, this could be simulated by the so–called *Gauss–Seidel* evaluation of deductive databases, cf. [8]. Consequently, a larger part of the transitive closure is computed during a single iteration for the following DATALOG* program:

```
tc(X, Y) :-
   arc(X, Y), assert(tc(X, Y)).
tc(X, Y) :-
   arc(X, Z), tc(Z, Y), assert(tc(X, Y)).
```

For example, given a graph containing – amoung others – the two edges $arc(a, b)$ and $arc(b, c)$, the transitive edge $tc(a, c)$ is already derived in iteration 1, since the asserted fact $tc(b, c)$ of the first rule can be used in the second rule together with $arc(a, b)$, whereas – without assert – it is only derived in iteration 2 under the standard *Jacobi* evaluation of deductive databases.

Due to PROLOG's evaluation, an `assert` statement in a DATALOG* rule $r = A \leftarrow B_1 \wedge \ldots \wedge B_n$, where $B_i = assert(X)$, is relevant for all subsequent rules and for all atoms within the body of the same rule r. This can be simulated without `assert` by Gauss–Seidel evaluation: let $\beta = B_1 \wedge \ldots \wedge B_{i-1}$, then we can replace r by a helper rule $r' = X \leftarrow \beta$ for deriving X followed by a reduced rule $r'' = A \leftarrow \beta \wedge B_{i+1} \wedge \ldots \wedge B_n$ without B_i. By rule extraction we could of course avoid the redundancy caused by the double evaluation of the conjunction β in r' and r''. The only remaining differences are, that r asserts the atom X only as a temporary fact, whereas r' derives X, and that r' is fully evaluated before r'', which makes all asserts of r' available to r''.

For avoiding problems, a more controlled use of update predicates for the PROLOG database could be required. For example, in our DATALOG* implementation of diagnostic reasoning, `assert` is used only after interactive dialogs. This does not effect the derivation process so drastically; the computation simply behaves as if all findings had been known before the computation.

5 Conclusions

We have presented an extension of deductive databases with a generalized immediate consequence operator and fixpoint iteration, called DATALOG*, that is useful for implementing practical EDDB applications nicely in a compact way. In future work, we will investigate the theoretical properties of the extensions.

The described *meta–interpreter* could efficiently handle the case studies on diagnostic reasoning and ontology development with a few thousand facts and rules. The main advantage of the PROLOG–based approach was the flexibility in modelling applications requiring more general concepts of forward reasoning than deductive databases usually offer. Moreover, the meta–interpreter can be easily

extended to fit further needs. For other potentially very large applications, such as disjunctive reasoning, we will conduct experimental evaluations to compare our proposal to other approaches in the future.

Based on the generalized immediate consequence operator $\mathcal{T}_{\mathcal{D},\mathcal{P}}$, it also seems to be posssible to implement an extended form of the magic sets transformation method for rules with non–conjunctive rule bodies in a very simple way.

So far, only stratified evaluation is possible for DATALOG*. But, it would be interesting to extend $\mathcal{T}_{\mathcal{D},\mathcal{P}}$ to handle non–stratified negation using ASP technology (stable or well–founded models). Sometimes, guessing strategies on the truth of special atoms can be used, whereas the whole extension of the called predicates has to be guessed, before a call to a meta–predicate such as `setof/3` can be evaluated.

Logic programming and extended deductive databases can be used as a declarative *mediator technology* between different data sources (like relational databases, XML databases/documents, and EXCEL sheets) and tools. We are planning to integrate similar diagnostic problem solvers by mapping them to DATALOG*, and to combine data mining tools by processing their input and output, such that a declarative data mining workflow can be specified in DATALOG*.

References

1. *S. Abiteboul, P. Bunemann, D. Suciu:* Data on the Web – From Relations to Semi–Structured Data and XML, Morgan Kaufmann, 2000.
2. *C. Baral:* Knowledge Representation, Reasoning and Declarative Problem Solving, Cambridge University Press, 2003.
3. *J. Baumeister, D. Seipel:* Smelly Owls – Design Anomalies in Ontologies, Proc. 18th International Florida Artificial Intelligence Research Society Conference, FLAIRS 2005, AAAI Press, 2005, pp. 215–220.
4. *S. Brass:* Range Restriction for General Formulas, Proc. 22nd Workshop on (Constraint) Logic Programming, WLP 2009.
5. *I. Bratko:* PROLOG– Programming for Artificial Intelligence, 3rd Edition, Addison–Wesley, 2001.
6. *P. Cabalar:* Existential Quantifiers in the Rule Body, Proc. 22nd Workshop on (Constraint) Logic Programming, WLP 2009.
7. *A. Cali, G. Gottlob, T. Lukasiewicz:* A General Datalog–Based Framework for Tractable Query Answering over Ontologies, Proc. International Conference on Principles of Database Systems, PODS 2009, pp. 77–86.
8. *S. Ceri, G. Gottlob, L. Tanca:* Logic Programing and Databases, Springer, 1990.
9. *W.F. Clocksin, C.S. Mellish:* Programming in PROLOG, 5th Edition, Springer, 2003.
10. *G. Köstler, W. Kießling, H. Thöne, U. Güntzer:* Fixpoint Iteration with Subsumption in Deductive Databases, Journal of Intelligent Information Systems, Volume 4, Number 2, Springer, 1995.
11. *J. Lobo, J. Minker, A. Rajasekar:* Foundations of Disjunctive Logic Programming, MIT Press, 1992.

12. *B. Marnette:* Generalized Schema–Mappings: From Termination To Tractability, Proc. International Conference on Principles of Database Systems, PODS 2009, pp. 13–22.
13. *J. Minker (Ed.):* Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, 1987.
14. *P. Puppe et al.:* D3. `http://d3web.informatik.uni-wuerzburg.de/`
15. *D. Seipel:* The DisLog Developers' Kit (DDK), `http://www1.informatik.uni-wuerzburg.de/databases/DisLog`
16. *D. Seipel:* Using Clausal Deductive Databases for Defining Semantics in Disjunctive Deductive Databases. Annals of Mathematics and Artificial Intelligence, vol. 33, Kluwer Academic Publishers, 2001, pp. 347-378.
17. *D. Seipel, M. Hopfner, B. Heumesser:* Analyzing and Visualizing Prolog Programs based on Xml Representations. Proc. International Workshop on Logic Programing Environments, WLPE 2003.
18. *D. Seipel, J. Baumeister:* Declarative Specification and Interpretation of Rule–Based Systems, Proc. 21st International Florida Artificial Intelligence Research Society Conference, FLAIRS 2008, AAAI Press, 2008.
19. *J. Wielemaker:* Swi–Prolog 5.0 Reference Manual and *J. Wielemaker, A. Anjewierden:* Programming in Xpce/Prolog, `http://www.swi-prolog.org/`

xpanda: A (Simple) Preprocessor for Adding Multi-Valued Propositions to ASP

Martin Gebser, Henrik Hinrichs, Torsten Schaub*, and Sven Thiele

Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract. We introduce a simple approach extending the input language of Answer Set Programming (ASP) systems by multi-valued propositions. Our approach is implemented as a (prototypical) preprocessor translating logic programs with multi-valued propositions into logic programs with Boolean propositions only. Our translation is modular and heavily benefits from the expressive input language of ASP. The resulting approach, along with its implementation, allows for solving interesting constraint satisfaction problems in ASP, showing a good performance.

1 Introduction

Boolean constraint solving technologies like Satisfiability Checking (SAT;[1]) and Answer Set Programming (ASP;[2]) have demonstrated their efficiency and robustness in many real-world applications, like planning [3, 4], model checking [5, 6], and bioinformatics [7, 8]. However, many applications are more naturally modeled by additionally using non-Boolean propositions, like resources or functions over finite domains. Unlike in SAT, however, where such language extensions are application-specific, ASP offers a rich application-independent modeling language. The high level of expressiveness allows for an easy integration of new language constructs, as demonstrated in the past by preferences [9] or aggregates [10]. Interesting examples of language extensions illustrating the utility of mixing Boolean and non-Boolean propositions can be found in [11–13], dealing with reasoning about actions.

In fact, a Boolean framework seems to offer such an elevated degree of efficiency that it becomes also increasingly attractive as a target language for non-Boolean constraint languages. This is for instance witnessed by the system Sugar [14], an award-winning SAT-based constraint solver. This motivated us to pursue a translational approach rather than an integrative one, as proposed in [15, 16] or, in more generality, in the field of SAT modulo theories.

In what follows, we expect the reader to be familiar with ASP (cf. [2]) as well as the input language of *lparse* [17, 18] or *gringo* [19, 20].

* Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada, and the Institute for Integrated and Intelligent Systems at Griffith University, Brisbane, Australia.

2 Approach

Our approach takes a logic program with multi-valued propositions and translates it into a standard logic program having Boolean propositions only. A multi-valued proposition is a variable taking exactly one value out of a pre-defined range. Currently, the range of multi-valued propositions is fixed to integer intervals. A multi-valued proposition is subject to two functional conditions, namely, it takes at most and at least one value. This is conveniently expressed by means of cardinality constraints. Let us make this precise by looking at the current input syntax.

For instance, the multi-valued proposition v taking values between 1 and 3 is declared as follows:

$$\#\texttt{variables}\ v = 1..3.$$

Such a declaration is translated into the following expressions:

$$\begin{array}{l} \_x\_dom(v, 0, 1..3). \\ 1\ \{val(v,\ Val) : \_x\_dom(v, 0,\ Val)\}\ 1. \end{array} \tag{1}$$

The new ternary predicate $\_x\_dom$ captures the fact that v has arity 0 and ranges over 1 to 3. This predicate is hidden in the output via $\#\texttt{hide}\ \_x\_dom(X,\ Y, Z)$. The value assignment to v is captured by the binary predicate val. Thus passing the logic program in (1) to an ASP system yields three answer sets, given by $\{val(v, 1)\}$, $\{val(v, 2)\}$, and $\{val(v, 3)\}$, each representing a valid assignment to variable v.

The declaration $\#\texttt{variables}$ also allows for more fine-grained specifications, like:

$$\begin{array}{l} \#\texttt{variables}\ u, v = 1..3 \mid 10..20. \\ \#\texttt{variables} f(X) = 1..10\ \texttt{:-}\ p(X). \end{array}$$

The first declaration shows how multiple variables can be specified, sharing a non-consecutive range of values. The second one shows how terms can be incorporated. For this, a term's domain must be guarded by domain predicates, like $p(X)$. These domain predicates are then added as body literals to the resulting cardinality constraint, viz.:

$$1\ \{val(f(X),\ Val) : \_x\_dom(f, 1,\ Val)\}\ 1\ \texttt{:-}\ p(X).$$

Interestingly, the mere possibility of defining multi-valued propositions opens up the possibility of specifying and solving simple constraint satisfaction problems. As an example, take two variables u, v ranging over $\{1, 2, 3\}$ and being subject to the constraint $u + v \leq 3$. This can be expressed by means of the following program:

$$\begin{array}{l} \#\texttt{variables}\ u, v = 1..3. \\ u + v \leq 3. \end{array}$$

This program is then translated as follows. While the declaration of u and v is given as in (1), the constraint $u + v \leq 3$ is expressed via an integrity constraint:

$$\texttt{:-}\ val(u,\ Val_u), val(v,\ Val_v),\ Val_u + Val_v > 3.$$

Note that the original constraint $u + v \leq 3$ appears negated as $u + v > 3$ within the integrity constraint. The idea is to exclude assignments to u and v such that $u + v > 3$. Finally, we note that passing the result of the compilation to an ASP system yields three answer sets, containing

$$\{val(u,1), val(v,1)\}, \quad \{val(u,1), val(v,2)\}, \quad \text{and} \quad \{val(u,2), val(v,1)\}.$$

The above transformation applies whenever an arithmetic expression involving multi-valued propositions appears in the head of a rule. Appearances of such expressions as body literals can be dealt with in a similar way. Unlike above, however, the original constraint is not negated. For instance, the rule within the program

$$\#\texttt{variables}\ u, v = 1..3.$$
$$p(u,v)\ \texttt{:-}\ u + v \leq 3.$$

is turned into

$$p(\mathit{Val}_u, \mathit{Val}_v)\ \texttt{:-}\ val(u, \mathit{Val}_u), val(v, \mathit{Val}_v), \mathit{Val}_u + \mathit{Val}_v \leq 3.$$

The resulting program has nine answer sets, reflecting all possible value assignments to u and v. However, among them, only three contain a single instance of predicate p, namely, $p(1,1)$, $p(1,2)$, and $p(2,1)$.

Finally, our approach provides a dedicated treatment of the popular *alldistinct* constraint, expressing that all involved variables must take pairwisely different values. As before, this constraint is easily mapped onto cardinality constraints. To illustrate this, consider the program:

$$\#\texttt{variables}\ u, v, w = 1..3.$$
$$\#\texttt{alldistinct}\ u, v, w.$$

The declaration of u, v, and w is dealt with as in (1). The *alldistinct* constraint yields the following program rules:

$$\#\texttt{hide}\ \_x\_distinct\_0\_var(\mathit{Var}). \qquad \#\texttt{hide}\ \_x\_distinct\_0\_val(\mathit{Val}).$$
$$\_x\_distinct\_0\_var(u). \qquad \_x\_distinct\_0\_val(\mathit{Val})\ \texttt{:-}\ \_x\_dom(u, 0, \mathit{Val}).$$
$$\_x\_distinct\_0\_var(v). \qquad \_x\_distinct\_0\_val(\mathit{Val})\ \texttt{:-}\ \_x\_dom(v, 0, \mathit{Val}).$$
$$\_x\_distinct\_0\_var(w). \qquad \_x\_distinct\_0\_val(\mathit{Val})\ \texttt{:-}\ \_x\_dom(w, 0, \mathit{Val}).$$
$$\texttt{:-}\ \_x\_distinct\_0\_val(\mathit{Val}), 2\ \{val(\mathit{Var}, \mathit{Val}) : \_x\_distinct\_0\_var(\mathit{Var})\}.$$

The predicates *_x_distinct_0_var* and *_x_distinct_0_val* are unique for each *alldistinct* constraint, fixing the sets of involved variables and values, respectively. The integrity constraint is violated whenever there are at least two variables sharing a value.

Our translation tool *xpanda* is written in Python and best used via Unix' pipes, e.g.:

```
cat simple.lp | xpanda.py | gringo | clasp 0.
```

A prototype version implementing a subset of the above transformations is (presently) available at http://files.mutaphysis.de/xpanda.zip. It works with ASP systems supporting the input language of *lparse* [17, 18] or *gringo* [19, 20]. Clearly, the translation of *xpanda* can easily be modified to using disjunction and explicit counting aggregates rather than cardinality constraints, and then be used by ASP systems like *dlv* [21].

3 A (Little) Case Study: *SEND+MORE=MONEY*

Let us conduct a brief case-study reflecting the scalability of our approach. To this end, we consider the *SEND+MORE=MONEY* puzzle. The task is to assign distinct numbers from $\{0,\ldots,9\}$ to the variables S,E,N,D,M,O,R,Y such that the addition of the decimal numbers $SEND$ and $MORE$ results in the decimal number $MONEY$. By convention, leading digits of decimal numbers must not be 0. This eliminates 0 from the domains of S and M. Moreover, we know that M cannot be greater than 1 because it occurs as carry. Hence, the value of M must be 1, effectively reducing the variables to S,E,N,D,O,R,Y. For clarity, however, we below use variable notation for M too.

A first and apparently compact representation of this problem is the following one:

```
#variables                m = 1.
#variables                s = 2..9.
#variables     e,n,d,o,r,y = 2..9 | 0.
#alldistinct s,e,n,d,o,r,y.

            s*1000+e*100+n*10+d
  +         m*1000+o*100+r*10+e
 == m*10000+o*1000+n*100+e*10+y.
```

The result of the compilation is given in Appendix A. Unfortunately, the grounding blows up in space because the (non-ground) integrity constraint resulting from the actual *SEND+MORE=MONEY* constraint leads to $8 * 9^6$ ground integrity constraints.

This extreme blow-up is avoided in the following representation, using column-wise addition and three carry variables to express the *SEND+MORE=MONEY* constraint:

```
#variables                m = 1.
#variables                s = 2..9.
#variables     e,n,d,o,r,y = 2..9 | 0.
#alldistinct s,e,n,d,o,r,y.
#variables        n1,e1,y1 = 0..1.

 d+e    == y+y1*10.
 n+r+y1 == e+e1*10.
 e+o+e1 == n+n1*10.
 s+m+n1 == o+ m*10.
```

The result of the compilation is given in Appendix B. Unlike a single constraint with seven variables, this formalization relies on four constraints with at most five variables. This reduces the resulting ground program to 7172 rules, which the ASP solver *clasp* (1.2.1) solves in milliseconds. The overall runtime, including *xpanda*, *gringo* (2.0.3), and *clasp*, is less than half a second when enumerating all solutions. In fact, this example has a unique solution containing:

```
 val(s,9)  val(e,5)  val(n,6) val(d,7)
 val(m,1)  val(o,0)  val(r,8)
val(n1,0) val(e1,1) val(y1,1) val(y,2).
```

4 Conclusion

We have provided a simple transformation-based approach to incorporating multi-valued propositions into ASP. Our translation is modular and heavily benefits from the expressive input language of ASP, providing variables and aggregate statements such as cardinality constraints. Once multi-valued propositions are available, it is possible to formulate and solve interesting constraint satisfaction problems by appeal to ASP technology. As with many ASP applications, the bottleneck of the approach manifests itself in grounding. We have seen that constraints involving too many variables may result in a space blow-up. This phenomenon can to some extent be controlled by the user since the number of variables remains the same in the initial specification and the resulting compilation. Of course, large domains may still be problematic.

Many open questions remain, concerning encoding optimizations, further language constructs, etc., and are subject to future research.

Acknowledgments. We are grateful to Wolfgang Faber for commenting on this paper. This work was partially funded by DFG under Grant SCHA 550/8-1 and by the GoFORSYS[1] project under Grant 0313924.

A First *SEND+MORE=MONEY* Representation: Compilation

```
:- val(s,Val_s), val(e,Val_e), val(n,Val_n), val(d,Val_d),
   val(m,Val_m), val(o,Val_o), val(r,Val_r), val(y,Val_y),
                  Val_s*1000+Val_e*100+Val_n*10+Val_d
    +             Val_m*1000+Val_o*100+Val_r*10+Val_e
   != Val_m*10000+Val_o*1000+Val_n*100+Val_e*10+Val_y.

#hide _x_distinct_0_var(X).
#hide _x_distinct_0_val(X).

_x_distinct_0_var(s). _x_distinct_0_val(Val) :- _x_dom(s,0,Val).
_x_distinct_0_var(e). _x_distinct_0_val(Val) :- _x_dom(e,0,Val).
_x_distinct_0_var(n). _x_distinct_0_val(Val) :- _x_dom(n,0,Val).
_x_distinct_0_var(d). _x_distinct_0_val(Val) :- _x_dom(d,0,Val).
_x_distinct_0_var(o). _x_distinct_0_val(Val) :- _x_dom(o,0,Val).
_x_distinct_0_var(r). _x_distinct_0_val(Val) :- _x_dom(r,0,Val).
_x_distinct_0_var(y). _x_distinct_0_val(Val) :- _x_dom(y,0,Val).

:- _x_distinct_0_val(Val),
   2{ val(Var,Val) : _x_distinct_0_var(Var) }.

#hide _x_dom(X,Y,Z).

_x_dom(m,0,1).
1{ val(m,X_D_Val) : _x_dom(m,0,X_D_Val) }1.
```

[1] http://www.goforsys.org

```
_x_dom(s,0,2..9).
1{ val(s,X_D_Val) : _x_dom(s,0,X_D_Val) }1.

_x_dom(e,0,2..9). _x_dom(e,0,0).
1{ val(e,X_D_Val) : _x_dom(e,0,X_D_Val) }1.

_x_dom(n,0,2..9). _x_dom(n,0,0).
1{ val(n,X_D_Val) : _x_dom(n,0,X_D_Val) }1.

_x_dom(d,0,2..9). _x_dom(d,0,0).
1{ val(d,X_D_Val) : _x_dom(d,0,X_D_Val) }1.

_x_dom(o,0,2..9). _x_dom(o,0,0).
1{ val(o,X_D_Val) : _x_dom(o,0,X_D_Val) }1.

_x_dom(r,0,2..9). _x_dom(r,0,0).
1{ val(r,X_D_Val) : _x_dom(r,0,X_D_Val) }1.

_x_dom(y,0,2..9). _x_dom(y,0,0).
1{ val(y,X_D_Val) : _x_dom(y,0,X_D_Val) }1.
```

B Second *SEND+MORE=MONEY* Representation: Compilation

```
:- val(d,Val_d), val(e,Val_e), val(y,Val_y), val(y1,Val_y1),
                               Val_d+Val_e != Val_y+Val_y1*10.
:- val(n,Val_n), val(r,Val_r), val(y1,Val_y1), val(e,Val_e),
   val(e1,Val_e1),    Val_n+Val_r+Val_y1 != Val_e+Val_e1*10.
:- val(e,Val_e), val(o,Val_o), val(e1,Val_e1), val(n,Val_n),
   val(n1,Val_n1),    Val_e+Val_o+Val_e1 != Val_n+Val_n1*10.
:- val(s,Val_s), val(m,Val_m), val(n1,Val_n1), val(o,Val_o),
                       Val_s+Val_m+Val_n1 != Val_o+Val_m*10.

#hide _x_distinct_0_var(X).
#hide _x_distinct_0_val(X).

_x_distinct_0_var(s). _x_distinct_0_val(Val) :- _x_dom(s,0,Val).
_x_distinct_0_var(e). _x_distinct_0_val(Val) :- _x_dom(e,0,Val).
_x_distinct_0_var(n). _x_distinct_0_val(Val) :- _x_dom(n,0,Val).
_x_distinct_0_var(d). _x_distinct_0_val(Val) :- _x_dom(d,0,Val).
_x_distinct_0_var(o). _x_distinct_0_val(Val) :- _x_dom(o,0,Val).
_x_distinct_0_var(r). _x_distinct_0_val(Val) :- _x_dom(r,0,Val).
_x_distinct_0_var(y). _x_distinct_0_val(Val) :- _x_dom(y,0,Val).

:- _x_distinct_0_val(Val),
   2{ val(Var,Val) : _x_distinct_0_var(Var) }.

#hide _x_dom(X,Y,Z).

_x_dom(m,0,1).
```

```
1{ val(m,X_D_Val) : _x_dom(m,0,X_D_Val) }1.

_x_dom(s,0,2..9).
1{ val(s,X_D_Val) : _x_dom(s,0,X_D_Val) }1.

_x_dom(e,0,2..9). _x_dom(e,0,0).
1{ val(e,X_D_Val) : _x_dom(e,0,X_D_Val) }1.

_x_dom(n,0,2..9). _x_dom(n,0,0).
1{ val(n,X_D_Val) : _x_dom(n,0,X_D_Val) }1.

_x_dom(d,0,2..9). _x_dom(d,0,0).
1{ val(d,X_D_Val) : _x_dom(d,0,X_D_Val) }1.

_x_dom(o,0,2..9). _x_dom(o,0,0).
1{ val(o,X_D_Val) : _x_dom(o,0,X_D_Val) }1.

_x_dom(r,0,2..9). _x_dom(r,0,0).
1{ val(r,X_D_Val) : _x_dom(r,0,X_D_Val) }1.

_x_dom(y,0,2..9). _x_dom(y,0,0).
1{ val(y,X_D_Val) : _x_dom(y,0,X_D_Val) }1.

_x_dom(n1,0,0..1).
1{ val(n1,X_D_Val) : _x_dom(n1,0,X_D_Val) }1.

_x_dom(e1,0,0..1).
1{ val(e1,X_D_Val) : _x_dom(e1,0,X_D_Val) }1.

_x_dom(y1,0,0..1).
1{ val(y1,X_D_Val) : _x_dom(y1,0,X_D_Val) }1.
```

References

1. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. IOS Press (2009)
2. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
3. Kautz, H., Selman, B.: Planning as satisfiability. In Neumann, B., ed.: Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92), John Wiley & Sons (1992) 359–363
4. Lifschitz, V.: Answer set planning. In De Schreye, D., ed.: Proceedings of the Sixteenth International Conference on Logic Programming (ICLP'99), MIT Press (1999) 23–37
5. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19**(1) (2001) 7–34
6. Heljanko, K., Niemelä, I.: Bounded LTL model checking with stable models. Theory and Practice of Logic Programming **3**(4-5) (2003) 519–550
7. Lynce, I., Marques-Silva, J.: Efficient haplotype inference with Boolean satisfiability. In Gil, Y., Mooney, R., eds.: Proceedings of the Twenty-first National Conference on Artificial Intelligence (AAAI'06), AAAI Press (2006) 104–109

8. Erdem, E., Türe, F.: Efficient haplotype inference with answer set programming. In Fox, D., Gomes, C., eds.: Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08), AAAI Press (2008) 436–441
9. Delgrande, J., Schaub, T., Tompits, H.: Logic programs with compiled preferences. In Baral, C., Truszczyński, M., eds.: Proceedings of the Eighth International Workshop on Non-Monotonic Reasoning (NMR'00), arXiv (2000)
10. Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T., Ielpa, G.: Design and implementation of aggregate functions in the DLV system. Theory and Practice of Logic Programming **8**(5-6) (2008) 545–580
11. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence **153**(1-2) (2004) 49–104
12. Lee, J., Lifschitz, V.: Describing additive fluents in action language C+. In Gottlob, G., Walsh, T., eds.: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03), Morgan Kaufmann Publishers (2003) 1079–1084
13. Dovier, A., Formisano, A., Pontelli, E.: Multivalued action languages with constraints in CLP(FD). In Dahl, V., Niemelä, I., eds.: Proceedings of the Twenty-third International Conference on Logic Programming (ICLP'07). Volume 4670 of Lecture Notes in Computer Science., Springer-Verlag (2007) 255–270
14. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. In Benhamou, F., ed.: Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP'06). Volume 4204 of Lecture Notes in Computer Science., Springer-Verlag (2006) 590–603
15. Mellarkod, V., Gelfond, M.: Integrating answer set reasoning with constraint solving techniques. In Garrigue, J., Hermenegildo, M., eds.: Proceedings of the Ninth International Symposium on Functional and Logic Programming (FLOPS'08). Volume 4989 of Lecture Notes in Computer Science., Springer-Verlag (2008) 15–31
16. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In Hill, P., Warren, D., eds.: Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09). Volume 5649 of Lecture Notes in Computer Science., Springer-Verlag (2009) 235–249
17. Syrjänen, T.: Lparse 1.0 user's manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz
18. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
19. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. http://potassco.sourceforge.net
20. Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., Thiele, S.: On the input language of ASP grounder *Gringo*. In Erdem, E., Lin, F., Schaub, T., eds.: Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09). Volume 5753 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2009) 502–508
21. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562

Existential Quantifiers in the Rule Body

Pedro Cabalar*

Department of Computer Science,
Corunna University (Corunna, Spain),
cabalar@udc.es

Abstract. In this paper we consider a simple syntactic extension of Answer Set Programming (ASP) for dealing with (nested) existential quantifiers and double negation in the rule bodies, in a close way to the recent proposal RASPL-1. The semantics for this extension just resorts to Equilibrium Logic (or, equivalently, to the General Theory of Stable Models), which provides a logic-programming interpretation for any arbitrary theory in the syntax of Predicate Calculus. We present a translation of this syntactic class into standard logic programs with variables (either disjunctive or normal, depending on the input rule heads), as those allowed by current ASP solvers. The translation relies on the introduction of auxiliary predicates and the main result shows that it preserves strong equivalence modulo the original signature.

1 Introduction

One of the traditional limitations of Answer Set Programming (ASP) in the past has been the need of resorting to a ground instantitation of program rules. Starting from the original definition of Stable Models [1] in terms of a propositional language, ASP solvers were designed following a two step process: first, removing variables in favour of all their ground instances; and second, computing the stable models of the resulting ground program. Variables were somehow an "external" element that was not directly treated in the semantics. It is not surprising, in this way, that quantification was not paid too much attention in the past although, paradoxically, most practical applications of ASP deal in one way or another with some limited use of quantified variables, using auxiliary predicates to capture the intended meaning.

This general picture has experienced a drastical change in the last years thanks to the introduction of Quantified Equilibrium Logic [2] (QEL) or the equivalent definition of stable models for first-order formulas proposed in [3]. These approaches provide a logic-programming interpretation for any arbitrary first-order theory, so that syntactic restrictions do not play a role in the semantic definition any more. Some recent results have been obtained in applying this semantics to programs with variables, without resorting to grounding. For instance,

* This research was partially supported by Spanish MEC project TIN-2006-15455-C03-02 and Xunta de Galicia project INCITE08-PXIB105159PR.

[4] treats the problem of strong equivalence (i.e., programs that have the same equilibrium models, even when included in a greater, common context), whereas in [5] QEL is used to analyse rule redundancy and the completeness of rule subsumption under a given substitution. On the other hand, much work remains to be done yet in exploring the intuition, under a logic-programming perspective, of the QEL interpretation of formulas with arbitrary syntax or belonging to new syntactic classes. Several works have followed this direction: we can mention [6], that has studied the extension of the concept of *safety* for arbitrary theories; [7], which considers an extension for dealing with partial functions; or [8], that proposes a logic-programming language RASPL-1 for counting and choice that can be translated into first-order expressions under QEL by introducing existential quantifiers and double negations in the rule bodies.

In this paper we analyse an extension of logic programs with variables where, similarly to first-order theories resulting from the RASPL-1 translation, we introduce existential quantifiers and double negations in the rule bodies, further allowing a way of nesting these new constructs (something not considered in [8]). We provide some intutitions of the utility of this extension and explain how these features are already used in the current ASP programming style by a suitable introduction of auxiliary predicates. In fact, we propose an automated translation that relies on this technique of auxiliary predicates and reduces the proposed extension to regular logic programs with variables as those accepted by current ASP grounding tools. This translation is shown to be *strongly equivalent*, that is, the original set of rules and the result of the translation will yield the same (non-monotonic) consequences, even when they are part of a greater context or program (of course, in the original language without the auxiliary predicates). Apart from providing a more readable and compact representation, the advantage of dealing with the extended syntax is avoiding a potential source of errors in the introduction of auxiliary predicates, not only due to a possible programmer's mistake in the formulation, but especially because auxiliary predicates must be guaranteed to be hidden and limited to their original use.

The rest of the paper is organised as follows. In the next section we introduce some motivating examples and explain the paper goals. In Section 3 we provide an overview of Quantified Equilibrium Logic to proceed in the next section with the introduction of the syntactic subclass we study in this paper. Section 5 presents the translation of this class into regular logic programs, proving its correctnes. Section 6 discusses some related work and finally, Section 7 concludes the paper.

2 Motivation

Consider the following example. Given the extent of predicates $person(X)$, $parent(X, Y)$ (X is a parent of Y) and $married(X, Y)$ which is a symmetric relation, suppose we want to represent that a person is happy when all his/her offsprings are married. A typical piece of program representing this problem in ASP would

probably look like:

$$\begin{aligned} \mathit{has\_spouse}(Y) &\leftarrow \mathit{married}(Y, Z) \\ \mathit{has\_single\_offs}(X) &\leftarrow \mathit{parent}(X, Y), \mathit{not}\ \mathit{has\_spouse}(Y) \\ \mathit{happy}(X) &\leftarrow \mathit{person}(X), \mathit{not}\ \mathit{has\_single\_offs}(X) \end{aligned}$$

Notice how predicates *has_spouse* and *has_single_offs* are *not* in the problem enunciate. Their name suggest that we are capturing an existential quantifier: note also the occurrence of free variables in the bodies that do not occur in the heads. A more compact way of representing this program could just be:

$$\mathit{happy}(X) \leftarrow \mathit{person}(X), \mathit{not}\ \exists Y(\mathit{parent}(X, Y), \mathit{not}\ \exists Z\ \mathit{married}(Y, Z)) \quad (1)$$

We will show that, in fact, both representations are strongly equivalent under QEL if we restrict the use of the auxiliary predicates *has_spouse* and *has_single_offs* to the above mentioned rules. Notice, however, the importance of this second representation. We, not only, get a more readable formula and avoid auxiliary predicates not included in the original problem: we also avoid a possible mistake in the use of these predicates in another part or module of the program, something that could radically change their intended meaning for the example.

As another typical example of an implicit existential quantifier, consider the frequent formalisation of the inertia default in ASP:

$$\mathit{holds}(F, V, \mathit{do}(A, S)) \leftarrow \mathit{holds}(F, V, S), \mathit{not}\ \mathit{ab}(F, V, A, S) \quad (2)$$

$$\mathit{ab}(F, V, A, S) \leftarrow \mathit{holds}(F, W, \mathit{do}(A, S)), W \neq V \quad (3)$$

where the complete rule bodies would also include the atoms $\mathit{action}(A)$, $\mathit{situation}(S)$, $\mathit{fluent}(F)$, $\mathit{range}(F, V)$ and $\mathit{range}(F, W)$ to specify the sorts of each variable Again, predicate *ab* is introduced to capture the meaning: "there exists a value for F other than V." That is, the formula could have been written instead as:

$$\mathit{holds}(F, V, \mathit{do}(A, S)) \leftarrow \mathit{holds}(F, V, S), \mathit{not}\ \exists W(\mathit{holds}(F, W, \mathit{do}(A, S)), W \neq V)$$

Something similar happens with choice-like pairs of rules for generating possible solutions. They typically have the form of even negative loops, like in the example:

$$\begin{aligned} \mathit{in}(X) &\leftarrow \mathit{vertex}(X), \mathit{not}\ \mathit{out}(X) \\ \mathit{out}(X) &\leftarrow \mathit{vertex}(X), \mathit{not}\ \mathit{in}(X) \\ \bot &\leftarrow \mathit{in}(X), \mathit{in}(Y), X \neq Y, \mathit{not}\ \mathit{edge}(X, Y), \mathit{not}\ \mathit{edge}(Y, X) \end{aligned}$$

intended for generating a *clique*[1] in terms of predicate $\mathit{in}(X)$. It seems clear that predicate $\mathit{out}(X)$ is auxiliary and thus its use should be limited to this pair of rules (adding other rules with $\mathit{out}(X)$ as a head may change the intended

[1] A clique is a set of vertices that are pairwise adjacent.

meaning). In fact, if the use of *out* is limited in that way, the first two rules become strongly equivalent (wrt the language without *out*) to the rule:

$$in(X) \vee \neg in(X) \leftarrow vertex(X) \tag{4}$$

Once again, the interest of the extended syntax is that it can be translated into traditional logic programs while it avoids the explicit use of auxiliary predicates which become hidden in the translation.

3 Overview of Quantified Equilibrium Logic

Following [5], Quantified Equilibirum Logic (QEL) is defined in terms of a models selection criterion for the intermediate logic of Quantified Here-and-There. In the paper, we will deal with a version of this logic dealing with static domains and decidable equality, calling it QHT for short.

Let $\mathcal{L} = \langle C, F, P\rangle$ be a first-order language where C is a set of *constants*, F a set of *functions* and P a set of *predicates*. First-order formulae for $\mathcal{L}$ are built up in the usual way, with the same syntax of classical predicate calculus. As in Intuitionistic Calculus, the formula $\neg\varphi$ will actually stand for $\varphi \rightarrow \bot$. We write $Atoms(C, P)$ to stand for the set of atoms built with predicates in P and constants in C. Similarly, $Terms(C, F)$ denote the set of ground terms built from functions in F and constants in C.

We will adopt a logical writing for logic programming connectives, so that constructions like (α, β), $(not\ \alpha)$ and $(\alpha \leftarrow \beta)$ are respectively written as $(\alpha \wedge \beta)$, $(\neg\alpha)$ and $(\beta \rightarrow \alpha)$. We also adopt lower-case letters for variables and functions, and upper-case for predicates and constants. In this way, a rule like (2) becomes the formula: $Holds(f, v, s) \wedge \neg Ab(f, v, a, s) \rightarrow Holds(f, v, do(a, s))$

We use boldface letters $\mathbf{x}, \mathbf{y}$ to denote tuples of variables, and similarly $\mathbf{d}$ for tuples of domain elements.

Definition 1 (QHT-interpretation). *A QHT-*interpretation *for a language $\mathcal{L} = \langle C, F, P\rangle$ is a tuple $\langle (D, \sigma), H, T\rangle$ where:*

1. *D is a nonempty set of constant names identifying each element in the interpretation universe. For simplicity, we take the same name for the constant and the universe element.*
2. *$\sigma : Terms(D \cup C, F) \rightarrow D$ assigns a constant in D to any term built with functions in F and constants in the extended set of constants $C \cup D$. It must satisfy: $\sigma(d) = d$ for all $d \in D$.*
3. *H and T are sets of atoms such that $H \subseteq T \subseteq Atoms(D, P)$.* □

An interpretation of the form $\langle (D, \sigma), T, T\rangle$ is said to be *total* and can be seen as the classical first-order interpretation $\langle (D, \sigma), T\rangle$. In fact, we will indistinctly use both notations. Furthermore, given any arbitrary $\mathcal{M} = \langle (D, \sigma), H, T\rangle$ we will define a corresponding total (or classical) interpretation $\mathcal{M}_T \stackrel{\text{def}}{=} \langle (D, \sigma), T\rangle$.

Satisfaction of formulas is recursively defined as follows. Given an interpretation $\mathcal{M} = \langle (D, \sigma), H, T\rangle$, the following statements are true:

- $\mathcal{M} \models p(t_1, \ldots, t_n)$ if $p(\sigma(t_1), \ldots, \sigma(t_n)) \in H$.
- $\mathcal{M} \models t_1 = t_2$ if $\sigma(t_1) = \sigma(t_2)$.
- $\mathcal{M} \not\models \bot$.
- $\mathcal{M} \models \alpha \wedge \beta$ if $\mathcal{M} \models \alpha$ and $\mathcal{M} \models \beta$. Disjunction $\vee$ is analogous.
- $\mathcal{M} \models \alpha \rightarrow \beta$ if both:
  (i) $\mathcal{M} \not\models \alpha$ or $\mathcal{M} \models \beta$ and
  (ii) $\mathcal{M}_T \models \alpha \rightarrow \beta$ in classical logic
- $\mathcal{M} \models \forall x\ \alpha(x)$ if both: (i) $\mathcal{M} \models \alpha(d)$, for each $d \in D$;
  and (ii) $\mathcal{M}_T \models \forall x\ \alpha(x)$ in classical logic.
- $\mathcal{M} \models \exists x\ \alpha(x)$ if for some $d \in D$, $\mathcal{M} \models \alpha(d)$. □

In the proofs, we will make use of the following property:

Proposition 1. *If $\mathcal{M} \models \varphi$ then $\mathcal{M}_T \models \varphi$.* □

Nonmonotonic entailment is obtained by introducing a models-minimisation criterion. Let us define the following ordering relation among interpretations:

Definition 2. *An interpretation $\mathcal{M} = \langle (D, \sigma), H, T \rangle$ is said to be* smaller *than an interpretation $\mathcal{M}' = \langle (D, \sigma), H', T \rangle$, written $\mathcal{M} \preceq \mathcal{M}'$, when $H \subseteq H'$.* □

That is, to be comparable, $\mathcal{M}$ and $\mathcal{M}'$ must only differ in their H component, so that $\mathcal{M} \preceq \mathcal{M}'$ iff $H \subseteq H'$. Notice that, as a consequence, $\mathcal{M} \preceq \mathcal{M}_T$. As usual, we write $\mathcal{M} \prec \mathcal{M}'$ when $\mathcal{M} \preceq \mathcal{M}'$ and $\mathcal{M} \neq \mathcal{M}'$ (that is $H \subset H'$).

We say that $\mathcal{M}$ is a *model* of a theory Γ if $\mathcal{M}$ satisfies all the formulas in Γ. If $\mathcal{M}$ is total, it is easy to check that: $\mathcal{M} \models \Gamma$ iff $\mathcal{M}_T \models \Gamma$ in classical logic. The next definition introduces the idea of minimal models for QHT.

Definition 3 (Equilibrium model). *A total model $\mathcal{M}$ of a theory Γ is an* equilibrium model *if there is no smaller model $\mathcal{M}' \prec \mathcal{M}$ of Γ.* □

Note that an equilibrium model is a total model, i.e., a classical model of Γ. We name *Quantified Equilibrium Logic* (QEL) the logic induced by equilibrium models. As said in the Introduction, equilibrium models coincide with the concept of stable models (usually defined in terms of program reducts) for all syntactic classes of programs. In fact, Equilibrium Logic has inspired the *General Theory of Stable Models* (introduced and shown to be equivalent in [3]) which extends the definition of stable model to any first order theory.

Given an interpretation $\mathcal{M}$ for a given language, and a sublanguage $\mathcal{L}$, we write $\mathcal{M}|_{\mathcal{L}}$ to denote the projection of $\mathcal{M}$ modulo $\mathcal{L}$. We say that two theories Γ_1, Γ_2 for language $\mathcal{L}'$ are *strongly equivalent* with respect to a given sublanguage $\mathcal{L}$ of $\mathcal{L}'$, written $\Gamma_1 \equiv_s^{\mathcal{L}} \Gamma_2$, when for any theory Γ in $\mathcal{L}$, the sets of equilibrium models (modulo $\mathcal{L}$) for $\Gamma_1 \cup \Gamma$ and $\Gamma_2 \cup \Gamma$ coincide. When $\mathcal{L} = \mathcal{L}'$ we just write $\Gamma_1 \equiv_s \Gamma_2$ and, in fact, this has been proved [9] to correspond to the QHT-equivalence of Γ_1 and Γ_2.

A *Herbrand QHT-interpretation* $\mathcal{M} = \langle (D, \sigma), H, T \rangle$ is such that D corresponds to $Terms(C, F)$ and $\sigma = id$, where id is the identity relation. In [9] it was shown that $\mathcal{M}$ is a Herbrand equilibrium model of a logic program Π iff T is a stable model of the (possibly infinite) ground program $gr_D(\Pi)$ obtained by replacing all variables by all terms in D in all possible ways.

4 Bodies with Existential Quantifiers

In this section we introduce the syntactic extension of logic programs we are interested in. We define a *body* as conjunction of *conditions*, where a condition, in its turn, recursively defined as:

i) a *predicate atom* $P(\mathbf{t})$ where $\mathbf{t}$ is a tuple of terms;
ii) an *equality atom* $t = t'$ with t, t' terms;
iii) $\exists \mathbf{x}\ (\psi)$ where $\mathbf{x}$ is a tuple of variables and ψ is a body in its turn;
iv) $\neg C$ where C is a condition;

Conditions of the form i) and ii) are called *atoms*: the former are predicate atoms and the latter, equality atoms. A *literal* is also a condition, with the form of an atom or its negation; the rest of conditions are called *non-literal*. A literal like $\neg(t = t')$ will be abbreviated as $t \neq t'$. Without loss of generality, we can assume that we handle two consecutive negations at most, since $\neg\neg\neg C \leftrightarrow \neg C$ is a QHT-tautology. Conditions beginning (resp. not beginning) with $\neg$ are said to be *negative* (resp. *positive*). Given a body B, we define its *positive* (resp. *negative*) *part*, B^+ (resp. B^-) as the conjunction of positive (resp. negative) conditions in B. We assume that $\exists x_1 \dots x_n\ \psi$ is a shorthand notation for $\exists x_1 \dots \exists x_n\ \psi$.

A *rule* is an expression like $B \rightarrow Hd$ where Hd is a (possibly empty) disjunction of predicate atoms (called the rule *head*) and B is a body. We assume that an empty disjunction corresponds to $\bot$. All free variables in a rule are implicitly universally quantified. The following are examples of rules:

$$P(x) \wedge \neg\neg Q(x) \wedge \neg\exists y\ \big(R(x,y) \wedge \exists z\ \neg R(y,z)\big) \rightarrow S(x) \vee R(x,x) \quad (5)$$

$$Person(x) \wedge \neg\exists y(Parent(x,y) \wedge \neg\exists z\ Married(y,z)) \rightarrow Happy(x) \quad (6)$$

$$Vertex(x) \wedge \neg\neg In(x) \rightarrow In(x) \quad (7)$$

Rules (6) and (7) are just different ways of writing (1) and (4) respectively. A rule is said to be *normal* if Hd just contains one atom. If $Hd = \bot$ the rule is called a *constraint*. A rule is said to be *regular* if its body is a conjunction of literals (i.e. it does not contain double negations or existential quantifiers). A set of rules of the general form above will be called a *logic program with existential quantifiers in the body* or $\exists$-logic program, for short. A program is said to be *normal* when all its rules are normal. The same applies for a *regular* program.

5 A Translation into Regular Logic Programs

The translation of a rule $r : B \rightarrow Hd$ into a regular logic program r^* will consist in recursively translating all the negative conditions in the rule body B *with respect to its positive part* B^+. This will possibly generate a set of additional rules dealing with new auxiliary predicates.

Definition 4 (Translation of conditions). *We define the translation of a condition C with respect to a positive body B^+ as a pair $\langle C^\bullet, \Pi(C, B^+)\rangle$ where $C^\bullet$ is a formula and $\Pi(C, B^+)$ a set of rules.*

1. *If C is a literal or has the form $\exists \mathbf{x}\ \alpha(\mathbf{x})$ then $C^\bullet = C$, $\Pi(C, B^+) = \emptyset$.*
2. *Otherwise, the condition has the form $C = \neg\alpha(\mathbf{x})$ being $\mathbf{x}$ the free variables in C. Then $C^\bullet = \neg Aux(\mathbf{x})$ and $\Pi(C, B^+) = (B^+ \wedge \alpha(\mathbf{x}) \rightarrow Aux(\mathbf{x}))^*$ where Aux is a new fresh predicate and * is the translation of rules in Def. 5.* □

The translation of a conjunction of conditions D with respect to a positive body B^+ is defined as expected $\langle D^\bullet, \Pi(D, B^+)\rangle$ where $D^\bullet$ is the conjunction of all $C^\bullet$ for each C in D, and $\Pi(D, B^+)$ the union of all rules $\Pi(C, B^+)$ for each C in D.

Definition 5 (Translation of a rule). *The translation of a rule r, written r^* is done in two steps:*

i) *We begin replacing all the positive conditions $\exists \mathbf{x}\ \varphi$ in the body of r by $\varphi[\mathbf{x}/\mathbf{y}]$ being $\mathbf{y}$ a tuple of new fresh variables*[2] *and repeat this step until no condition of this form is left. Let $B \rightarrow \mathrm{Hd}$ denote the resulting rule.*

ii) *We then obtain the set of rules:*

$$r^* \stackrel{\text{def}}{=} \{\mathrm{Hd} \leftarrow B^+ \wedge (B^-)^\bullet\} \cup \Pi(B^-, B^+) \qquad \square$$

The translation of an $\exists$-logic program Π is denoted Π^* and corresponds to the logic program $\bigcup_{r\in\Pi} r^*$ as expected. As an example of this translation, consider the rule $r_1 = (6)$. We would have:

$$r_1^* = \{Person(x) \wedge \neg Aux_1(x) \rightarrow Happy(x)\} \cup \Pi(B(r_1)^-, Person(x))$$

where $B(r_1)^- = \neg\exists y(Parent(x, y) \wedge \neg\exists z(Married(y, z)))$ and so, $\Pi(B(r_1)^-, Person(x))$ contains the translation of the rule:

$$Person(x) \wedge \exists y(Parent(x, y) \wedge \neg\exists z(Married(y, z))) \rightarrow Aux_1(x)$$

We remove the positive existential quantifier to obtain r_2:

$$Parent(x, y) \wedge \neg\exists z(Married(y, z)) \wedge Person(x) \rightarrow Aux_1(x)$$

and now

$$\begin{aligned} r_2^* = \{&Parent(x, y) \wedge Person(x) \wedge \neg Aux_2(x, y) \rightarrow Aux_1(x)\} \\ &\cup \Pi(B(r_2)^-, Parent(x, y) \wedge Person(x)) \end{aligned}$$

This yields the rule

$$Parent(x, y) \wedge Person(x) \wedge \exists z(Married(y, z)) \rightarrow Aux_2(x, y)$$

in which, again, we would just remove the positive existential quantifier. To sum up, the final complete translation r_1^* would be the (regular) logic program:

$$\begin{aligned} Person(x) \wedge \neg Aux_1(x) &\rightarrow Happy(x) \\ Parent(x, y) \wedge Person(x) \wedge \neg Aux_2(x, y) &\rightarrow Aux_1(x) \\ Parent(x, y) \wedge Person(x) \wedge Married(y, z) &\rightarrow Aux_2(x, y) \end{aligned}$$

[2] The introduction of new variables $\mathbf{y}$ can be omitted when $\mathbf{x}$ does not occur free in the rest of the rule.

We can informally read $Aux_2(x, y)$ as "*y is a married child of x*" and $Aux_1(x)$ as "*x has some single child.*"

It is easy to see that the translation is modular (we translate each rule independently) and that its size is polynomial with respect to the original input.

Proposition 2. *Given a rule r containing A atoms in its body and N subformulas of one of the forms* $(\neg\exists\mathbf{x}\ \alpha)$ *or* $(\neg\neg\alpha)$*, the translation* r^* *contains* $N+1$ *regular rules whose bodies contain at most* $A+N$ *atoms.* □

It might be thought that, as we always have a way of removing positive existential quantifiers, these are unnecessary. However, we must take into account that they are useful when nested in another expression.

As an example with double negation, it can be easily checked that the translation of rule (7) becomes the program:

$$Vertex(x) \wedge \neg Aux(x) \rightarrow In(x) \qquad Vertex(x) \wedge \neg In(x) \rightarrow Aux(x)$$

Theorem 1 (Main result). *Let* Π *be an* $\exists$*-logic program for language* $\mathcal{L}$*. Then* $\Pi \equiv_s^{\mathcal{L}} \Pi^*$. □

Theorem 2. *If* Π *is a disjunctive (resp. normal)* $\exists$*-logic program then* Π^* *is a disjunctive (resp. normal) regular logic program.* □

The reason for making the definition of new auxiliary predicates depend on the positive body of the original rule has to do with the following property, that will guarantee a correct grounding of the program resulting from the translation.

Definition 6 (Restricted variable). *A variable X is said to be* restricted *by some positive literal* $p(\mathbf{t})$ *occurring in a conjunction of literals* β *when:*

1. *either X directly occurs in* $p(\mathbf{t})$*;*
2. *or there exists a positive literal* $X = Y$ *or* $Y = X$ *in* β *and Y is restricted by* $p(\mathbf{t})$ *in* β*.*

We just say that X is restricted *in* β *if it is restricted by some* $p(\mathbf{t})$ *in* β. □

Definition 7 (Safe rule). *A rule* $r : B \rightarrow \mathrm{Hd}$ *is said to be safe when both:*

a) Any free variable occurring in r also occurs free and restricted in B.
b) For any condition $\exists x\ \varphi$ *in B, x occurs free and restricted in* φ. □

For instance, rule (6) is safe: its only free variable x occurs in the positive body $Person(x)$. In fact, all the rules we used in the previous sections are safe. However, rules like:

$$\neg\neg Mark(x) \rightarrow Mark(x) \qquad \exists y\ Q(y) \rightarrow P(x) \qquad \exists x\ \neg P(x) \rightarrow A$$

are not safe. Notice that, for regular programs (i.e. those exclusively containing literal conditions) only case *a)* of Definition 7 is applicable and, in fact, this coincides with the usual concept of safe rule in ASP.

Theorem 3. *If* Π *is safe then* Π^* *is safe.* □

6 Related Work

The technique of replacing quantifiers by auxiliary predicates was already introduced in Lloyd and Topor's paper [10] for Prolog extended programs. That work contained a closely similar translation[3] for removing existential quantifiers. In the case of ASP, however, many of Lloyd and Topor's transformations are not valid: for instance treating implications as disjunctions, removing double negations or replacing $\forall x \varphi$ by $\neg \exists x \neg \varphi$, cannot be done in ASP, as it can be expected from the intuitionistic nature of its logical characterisation in terms of QHT. So, in principle, Lloyd and Topor's treatment of existential quantifiers needed not to be correct in the case of ASP – we have proved it is so.

In fact, the correctness of this technique for ASP has also been independently found in the recent work [11] where, moreover, they implemented a system called `F2LP` for dealing with quantifiers. This system allows computing answer sets for first order theories that satisfy some syntactic restrictions: informally speaking, existential[4] quantifiers must be in the antecedent of an implication or in the scope of negation. In fact, the current approach deals with a syntactic subclass of that of `F2LP` where we do not nest conjunctions, disjunctions and implications. On the other hand, although `F2LP` handles a more general syntax, no safety condition has been defined for it (until now) in such a way that an arbitrary theory results in a (quantifier-free) logic program that is safe in the format accepted by current solvers, as happens with the syntactic subclass proposed in the current work.

As commented in the Introduction, this work is directly related to the recently introduced language RASPL-1 [8]. In fact, that language is defined in terms of a translation into first order sentences that fit into the syntax extension we study here (existential quantifiers and double negations in the body).

The use of $\exists$-logic programs was actually forwarded in [7] where an extension of QEL for dealing with partial functions was introduced (in fact, the main result of the current paper was conjectured in that work). A less related approach that has also considered the use of body quantifiers is [12], although the semantics was only defined for stratified programs.

7 Conclusions

We have presented an extension of logic programming that allows dealing with (possibly nested) existential quantifiers and double negations in the rule bodies. We have shown how this new syntactic class captures several typical representation problems in ASP allowing a more compact and readable formulation and avoiding the use of auxiliary predicates. In fact, we presented a translation that reduces this new syntax to that of regular logic programs by automatically generating these auxiliary predicates, which are kept hidden to avoid programmer's errors.

[3] The main difference is that, in our case, the rule for the auxiliary predicate inherits the body where the existential quantifier occurred, so that safety can be preserved.

[4] `F2LP` also allows universal quantifiers, but only when they are strongly equivalent to negations of existential quantifiers.

Acknowledgements Many thanks to anonymous reviewers for their helpful suggestions and for pointing out some clearly missing references to related work.

References

1. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. of the 5th Intl. Conf. on Logic Programming. (1988) 1070–1080
2. Pearce, D., Valverde, A.: Towards a first order equilibrium logic for nonmonotonic reasoning. In: Proc. of the 9th European Conf. on Logics in AI (JELIA'04). (2004) 147–160
3. Ferraris, P., Lee, J., Lifschitz, V.: A new perspective on stable models. In: Proc. of the International Joint Conference on Artificial Intelligence (IJCAI'07). (2004) 372–379
4. Lifschitz, V., Pearce, D., Valverde, A.: A characterization of strong equivalence for logic programs with variables. In: Proc. of the 9th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'07). (2007) 188–200
5. Pearce, D., Valverde, A.: Quantified equilibrium logic and foundations for answer set programs. In: Proc. of the 24th Intl. Conf. on Logic Programming (ICLP'08). (2008) 547–560
6. Lee, J., Lifschitz, V., Palla, R.: Safe formulas in the general theory of stable models (preliminary report). In: Proc. of the 24th Intl. Conf. on Logic Programming (ICLP'08). (2008) 672–676
7. Cabalar, P.: Partial functions and equality in answer set programming. In: Proc. of the 24th Intl. Conf. on Logic Programming (ICLP'08). (2008) 392–406
8. Lee, J., Lifschitz, V., Palla, R.: A reductive semantics for counting and choice in answer set programming. In: Proc. of the 23rd AAAI Conference on Artificial Intelligence. (2008) 472–479
9. Pearce, D., Valverde, A.: Quantified equilibrium logic and the first order logic of here-and-there. Technical Report MA-06-02, University of Málaga, Spain (2006)
10. Lloyd, J., Topor, R.: Making PROLOG more expressive. Journal of Logic Programming **1**(3) (1984) 225–240
11. Lee, J., Palla, R.: System F2LP - computing answer sets of first-order formulas. In: Proc. of the 10th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'09). (2009) to appear.
12. Eiter, T., Gottlob, G., Veith, H.: Modular logic programming and generalized quantifiers. In: Proc. of the 4th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'97). (1997) 290–309
13. Cabalar, P., Pearce, D., Valverde, A.: Reducing propositional theories in equilibrium logic to logic programs. In: 12th Portuguese Conference on Artificial Intelligence (EPIA 2005). (2005) 4–17
14. Lifschitz, V., Tang, L.R., Turner, H.: Nested expressions in logic programs. Ann. Math. Artif. Intell. **25**(3-4) (1999) 369–389

Appendix. Proofs

Proof (Proposition 2). It is easy to see that, excepting for the first step, in which the original rule r is considered, each time we introduce a new rule is for univoquely defining an auxiliary predicate $Aux(\mathbf{x})$ that corresponds to one of the subexpressions of the form $\neg\exists\mathbf{x}\ \alpha$ or $\neg\neg\alpha$ that occurred in r. So, the total number of rules is $N+1$. As for the body size of each rule, we always go keeping a (usually strict) subset of the original number of atoms A occurring in r, plus additional literals $\neg Aux(\mathbf{x})$ corresponding to replaced conditions of the form $\neg\exists\mathbf{x}\ \alpha$ or $\neg\neg\alpha$. As a result, we get the upper bound $N+A$. □

To prove our main result, we will use several QHT valid equivalences (many of them already commented in [13]) and introduce several lemmas. For instance, we will frequently make use of the following QHT valid formula (see [13])

$$\alpha \rightarrow (\beta \rightarrow \gamma) \leftrightarrow (\alpha \wedge \beta \rightarrow \gamma) \tag{8}$$

Similarly, the following is a QHT-theorem:

$$\alpha \wedge \neg(\alpha \wedge \beta) \leftrightarrow \alpha \wedge \neg\beta \tag{9}$$

whose proof can be obtained from transformations in [14, 13].

Lemma 1. *Let $\mathcal{M}$ be an equilibrium model of Γ, and $\mathcal{M} \models \alpha$. Then $\mathcal{M}$ is an equilibrium model of $\Gamma \cup \{\alpha\}$.* □

Theorem 4 (Equivalent subformula replacement). *Given the equivalence:*

$$\forall\mathbf{x}(\alpha(\mathbf{x}) \leftrightarrow \beta(\mathbf{x})) \tag{10}$$

where $\mathbf{x}$ is the set of free variables in α or β, and a given formula γ containing a subformula $\alpha(\mathbf{t})$, then (10) implies $\gamma \leftrightarrow \gamma[\alpha(\mathbf{t})/\beta(\mathbf{t})]$. □

Theorem 5 (Defined predicate removal). *Let Γ_1 be a theory for language $\mathcal{L}$, α a formula in that signature and Aux a predicate not in $\mathcal{L}$. If Γ_2 is Γ_1 plus*

$$\forall\mathbf{x}(Aux(x) \leftrightarrow \alpha(\mathbf{x})) \tag{11}$$

then $\Gamma_1 \equiv_s^{\mathcal{L}} \Gamma_2$. □

Lemma 2. *Let $\mathcal{M}_1 = \langle(D,\sigma), H, T\rangle$ be a model of the formulas*

$$\forall\mathbf{x}(\alpha(\mathbf{x}) \rightarrow Aux(\mathbf{x})) \tag{12}$$

$$\forall\mathbf{x}(\neg Aux(\mathbf{x}) \rightarrow \beta(\mathbf{x})) \tag{13}$$

where α and β do not contain predicate Aux, and let $\mathcal{M}_2 = \langle(D,\sigma), H', T\rangle$ be such that $H \setminus H' = \{Aux(\mathbf{d}) \mid \mathbf{d} \in \mathcal{D}\}$ for some set of tuples of domain elements $\mathcal{D} \neq \emptyset$ satisfying $\mathcal{M}_1 \models Aux(\mathbf{d})$ and $\mathcal{M}_1 \not\models \alpha(\mathbf{d})$. Then $\mathcal{M}_2 \models (12) \cup (13)$. □

Theorem 6. *Let $\mathcal{L}$ denote a signature not containing predicate Aux, and let $\alpha(\mathbf{x}), \beta(\mathbf{x})$ be a pair of formulas for $\mathcal{L}$. Given $\Gamma_1 = (12) \cup (13)$ and Γ_2 consisting of Γ_1 plus:*

$$\forall \mathbf{x}(Aux(\mathbf{x}) \rightarrow \alpha(\mathbf{x})) \tag{14}$$

then $\Gamma_1 \equiv_s^{\mathcal{L}} \Gamma_2$. □

Theorem 7. *Let Γ_1 be a theory consisting of the single formula*

$$\forall \mathbf{x}\big(\ \alpha(\mathbf{x}) \wedge \neg\beta(\mathbf{x}) \rightarrow \gamma(\mathbf{x})\ \big) \tag{15}$$

for language $\mathcal{L}$, being $\mathbf{x}$ a tuple with all the variables that occur free in the antecedent or in the consequent. Then $\Gamma_1 \equiv_s^{\mathcal{L}} \Gamma_2$ where Γ_2 is the pair of formulas:

$$\forall \mathbf{x}\big(\ \alpha(\mathbf{x}) \wedge \neg Aux(\mathbf{x}) \rightarrow \gamma(\mathbf{x})\ \big) \tag{16}$$

$$\forall \mathbf{x}\big(\ \alpha(\mathbf{x}) \wedge \beta(\mathbf{x}) \rightarrow Aux(\mathbf{x})\ \big) \tag{17}$$

and $Aux(\mathbf{x})$ is a fresh auxiliary predicate not included in $\mathcal{L}$. □

Lemma 3. *Let x be a variable that does not occur free in β. Then, the following is a QHT-tautology:*

$$(\exists x\ \alpha(x) \rightarrow \beta) \leftrightarrow \forall x(\alpha(x) \rightarrow \beta) \tag{18}$$

Proof (Lemma 1). Obviously, $\mathcal{M} \models \Gamma \cup \{\alpha\}$. There cannot be a smaller model $\mathcal{M}' \prec \mathcal{M}$ for $\Gamma \cup \{\alpha\}$, because it would also be a model of Γ and this would contradict minimality of $\mathcal{M}$ for Γ. □

Proof (Theorem 4). It is easy to check that, given any tuple $\mathbf{d}$ of domain elements and any model $\mathcal{M} = \langle (D, \sigma), H, T \rangle$, $\mathcal{M} \models (10)$ implies that:

1. $\mathcal{M} \models \alpha(\mathbf{d})$ iff $\mathcal{M} \models \beta(\mathbf{d})$
2. $\mathcal{M}_T \models \alpha(\mathbf{d})$ iff $\mathcal{M}_T \models \beta(\mathbf{d})$

Looking at the satisfaction of formulas, this means that for any model of (10), $\alpha(\mathbf{t})$ and $\beta(\mathbf{t})$ for any tuple of terms $\mathbf{t}$ are semantically equivalent and can be interchanged. □

Proof (Theorem 5). Let Γ denote an arbitrary theory for $\mathcal{L}$ and take $\mathcal{M}$ an equilibrium model of $\Gamma \cup \Gamma_1$ and signature $\mathcal{L}$. We will show that there exists an equilibrium model $\mathcal{M}'$ of $\Gamma \cup \Gamma_2$ such that $\mathcal{M}'|_{\mathcal{L}} = \mathcal{M}$. It is clear we can take $\mathcal{M}'$ equal to $\mathcal{M}$ for all predicates in $\mathcal{L}$ and fix the extent of Aux such that $\mathcal{M}' \models Aux(\mathbf{d})$ iff $\mathcal{M} \models \alpha(\mathbf{d})$ for any tuple of elements $\mathbf{d}$. Obviously, by construction, $\mathcal{M}' \models \Gamma \cup \Gamma_2$. It must also be minimal, since any $\mathcal{M}'' \prec \mathcal{M}'$ that $\mathcal{M}'' \models \Gamma \cup \Gamma_2$ is also a model of $\Gamma \cup \Gamma_1$ and this would contradict the minimality of $\mathcal{M}$ for that theory.

For the other direction, take some $\mathcal{M}' = \langle (D, \sigma), T' \rangle$ equilibrium model of $\Gamma \cup \Gamma_2$. Clearly, $\mathcal{M}' \models \Gamma \cup \Gamma_1$ and, since this theory does not contain Aux, its projection $\mathcal{M}'|_{\mathcal{L}} = \mathcal{M} = \langle (D, \sigma), T \rangle$ must also be a model for $\Gamma \cup \Gamma_1$. Take another model of this theory, $\mathcal{M}_2 = \langle (D, \sigma), H, T \rangle$ with $H \subset T$, that is $\mathcal{M}_2 \prec \mathcal{M}$. But then, we can construct $\mathcal{M}'_2 = \langle (D, \sigma), H', T' \rangle$ such that H' consists of H and the set of atoms $Aux(\mathbf{d})$ for which $\mathcal{M}_2 \models \alpha(\mathbf{d})$. Notice that H' must be a subset of T' because $\mathcal{M}_2 \models \alpha(\mathbf{d})$ implies $\mathcal{M} \models \alpha(\mathbf{d})$ and this implies $\mathcal{M}' \models \alpha(\mathbf{d})$, that together with $\mathcal{M}' \models (11)$ implies and $\mathcal{M}' \models Aux(\mathbf{d})$. But as $H \subset T$ we get $H' \subset T'$ and so $\mathcal{M}'_2 \prec \mathcal{M}'$. On the other hand, by construction of $\mathcal{M}'_2$ together with $\mathcal{M}' \models (11)$, we obtain $\mathcal{M}'_2 \models (11)$. In this way, $\mathcal{M}'_2 \models \Gamma \cup \Gamma_2$ while $\mathcal{M}'_2 \prec \mathcal{M}'$ reaching a contradiction with minimality of $\mathcal{M}'$ for this theory. □

Proof (Lemma 2). Note first that, for any tuple $\mathbf{d} \notin \mathcal{D}$, $\mathcal{M}_1$ and $\mathcal{M}_2$ coincide both for $Aux(\mathbf{d})$, $\alpha(\mathbf{d})$ and $\beta(\mathbf{d})$. Then $\mathcal{M}_1 \models (12)$ and $\mathcal{M}_1 \models (13)$ allow us to conclude $\mathcal{M}_2 \models \alpha(\mathbf{d}) \rightarrow Aux(\mathbf{d})$ and $\mathcal{M}_2 \models \neg Aux(\mathbf{d}) \rightarrow \beta(\mathbf{d})$, respectively. We remain to prove that the same holds for tuples $\mathbf{d} \in \mathcal{D}$. Consider $\mathcal{M}_T = \langle (D, \sigma), T \rangle$, that is, the total model above $\mathcal{M}_1$ and $\mathcal{M}_2$. For any $\mathbf{d} \in \mathcal{D}$, we have $\mathcal{M}_1 \models Aux(\mathbf{d})$ and thus $\mathcal{M}_T \models Aux(\mathbf{d})$, but then $\mathcal{M}_2 \not\models \neg Aux(\mathbf{d})$. On the other hand, $\mathcal{M}_1 \models (13)$ also implies $\mathcal{M}_T \models (13)$ and, in particular, $\mathcal{M}_T \models \neg Aux(\mathbf{d}) \rightarrow \beta(\mathbf{d})$. The latter, together with $\mathcal{M}_2 \not\models \neg Aux(\mathbf{d})$, implies $\mathcal{M}_2 \models \neg Aux(\mathbf{d}) \rightarrow \beta(\mathbf{d})$, for any $\mathbf{d} \in \mathcal{D}$.

Similarly, $\mathcal{M}_1 \models (12)$ implies $\mathcal{M}_T \models (12)$ and, in particular, $\mathcal{M}_T \models \alpha(\mathbf{d}) \rightarrow Aux(\mathbf{d})$ for $\mathbf{d} \in \mathcal{D}$. On the other hand, as $\mathcal{M}_1$ and $\mathcal{M}_2$ do not differ for $\alpha(\mathbf{d})$, we conclude $\mathcal{M}_2 \not\models \alpha(\mathbf{d})$, and thus, $\mathcal{M}_2 \models \alpha(\mathbf{d}) \rightarrow Aux(\mathbf{d})$. □

Proof (Theorem 6). Let Γ denote an arbitrary theory for $\mathcal{L}$ and take $\mathcal{M} = \langle (D, \sigma), T \rangle$ an equilibrium model of $\Gamma \cup \Gamma_1$. For proving that $\mathcal{M}$ is equilibrium model of $\Gamma \cup \Gamma_2$, by Lemma 1, it suffices to show that $\mathcal{M} \models (14)$. Assume this does not hold. As $\mathcal{M}$ is a total model, this just means that for some tuple of domain elements $\mathbf{d}$, $\mathcal{M} \models Aux(\mathbf{d})$ and $\mathcal{M} \not\models \alpha(\mathbf{d})$. Let us take now a model $\mathcal{M}' = \langle (D, \sigma), H, T \rangle$ where H is equal to T excepting that the extension of Aux does not include the tuple $\mathbf{d}$. Notice that $H \subset T$ and $\mathcal{M}' \prec \mathcal{M}$. In fact, we can observe that Lemma 2 is applicable taking $\mathcal{M}_1 = \mathcal{M}$, $\mathcal{M}_2 = \mathcal{M}'$ and $\mathcal{D} = \{\mathbf{d}\}$ to conclude $\mathcal{M}' \models (12) \cup (13)$, i.e., $\mathcal{M}' \models \Gamma_1$. Furthermore, as $\mathcal{M}'$ only differs from $\mathcal{M}$ in Aux, $\mathcal{M}' \models \Gamma$. But this contradicts the minimality of $\mathcal{M}$ as equilibrium model of $\Gamma \cup \Gamma_1$.

For the other direction, let $\mathcal{M}$ be an equilibrium model of $\Gamma \cup \Gamma_2$. Since $\Gamma_1 \subset \Gamma_2$, obviously $\mathcal{M} \models \Gamma \cup \Gamma_1$. We remain to prove that $\mathcal{M}$ is minimal. Suppose we had some other model $\mathcal{M}' \prec \mathcal{M}$ of $\Gamma \cup \Gamma_1$. If $\mathcal{M}' \models (14)$ we would have $\mathcal{M}' \models \Gamma \cup \Gamma_2$ and this would contradict the minimality of $\mathcal{M}$ for that theory. So, assume $\mathcal{M}' \not\models (14)$. Let $\mathcal{D}$ be the set of tuples $\mathbf{d}$ for which $\mathcal{M}' \not\models Aux(\mathbf{d}) \rightarrow \alpha(\mathbf{d})$ (note that this set cannot be empty). As $\mathcal{M} \models (14)$ we must have $\mathcal{M}' \models Aux(\mathbf{d})$ and $\mathcal{M}' \not\models \alpha(\mathbf{d})$ for all $\mathbf{d} \in \mathcal{D}$. Now take $\mathcal{M}''$ equal to $\mathcal{M}'$ excepting that, for all

$\mathbf{d} \in \mathcal{D}$, $\mathcal{M}'' \not\models Aux(\mathbf{d})$. We can apply Lemma 2 taking $\mathcal{M}_1 = \mathcal{M}'$, $\mathcal{M}_2 = \mathcal{M}''$ and $\mathcal{D}$ to conclude $\mathcal{M}'' \models (12) \cup (13)$, i.e., $\mathcal{M}'' \models \Gamma_1$. Furthermore, as $\mathcal{M}''$ only differs from $\mathcal{M}'$ in the extent of Aux, we obtain $\mathcal{M}'' \models \Gamma \cup \Gamma_1$. Now, as $\mathcal{M}'' \not\models Aux(\mathbf{d})$ and we have $\mathcal{M} \models (14)$ we conclude $\mathcal{M}'' \models Aux(\mathbf{d}) \rightarrow \alpha(\mathbf{d})$. For tuples $\mathbf{c} \notin \mathcal{D}$ we had $\mathcal{M}' \models Aux(\mathbf{c}) \rightarrow \alpha(\mathbf{c})$ by definition of $\mathcal{D}$, but $\mathcal{M}'$ and $\mathcal{M}''$ coincide in $Aux(\mathbf{c})$ and $\alpha(\mathbf{c})$. As a result, $\mathcal{M}'' \models (14)$ too, and since $\mathcal{M}'' \prec \mathcal{M}$ we obtain a contradiction with minimality of $\mathcal{M}$ for $\Gamma \cup \Gamma_2$. □

Proof (Theorem 7). By (8), the formula (16) is strongly equivalent to:

$$\forall \mathbf{x}\big(\ \neg Aux(\mathbf{x}) \rightarrow (\alpha(\mathbf{x}) \rightarrow \gamma(\mathbf{x}))\ \big) \tag{19}$$

so that, we can apply Theorem 6 on Γ_2 to transform the implication in (17) into a double implication:

$$\forall \mathbf{x}\big(\ \alpha(\mathbf{x}) \wedge \beta(\mathbf{x}) \leftrightarrow Aux(\mathbf{x})\ \big) \tag{20}$$

As a result, Γ_2 is strongly equivalent (modulo $\mathcal{L}$) to the theory consisting of (16) and (20). By Theorem 4, this is strongly equivalent, in its turn, to (20) plus:

$$\forall \mathbf{x}\big(\ \alpha(\mathbf{x}) \wedge \neg(\alpha(\mathbf{x}) \wedge \beta(\mathbf{x})) \rightarrow \gamma(\mathbf{x})\ \big) \tag{21}$$

Due to (9), the latter is strongly equivalent to (15). Finally, by Theorem 5, we can remove (20), since it is a definition for predicate Aux which does not belong to $\mathcal{L}$. □

Proof (Lemma 3). As (18) is a classical tautology, we remain to prove that, for any interpretation $\mathcal{M} = \langle (D, \sigma), H, T \rangle$, $\mathcal{M} \models \exists x\ \alpha(x) \rightarrow \beta$ iff $\mathcal{M} \models \forall x(\alpha(x) \rightarrow \beta)$. For the left to right direction, assume $\mathcal{M} \models \exists x\ \alpha(x) \rightarrow \beta$ but $\mathcal{M} \not\models \forall x(\alpha(x) \rightarrow \beta)$. The latter means there exists some element d for which $\mathcal{M} \not\models \alpha(d) \rightarrow \beta$. Since $\mathcal{M} \models \exists x\ \alpha(x) \rightarrow \beta$ we have that $\mathcal{M}_T$ also satisfies that formula and so $\mathcal{M}_T \models \forall x(\alpha(x) \rightarrow \beta)$ since it is a classically equivalent formula. Therefore, the only possibility is $\mathcal{M} \models \alpha(d)$ and $\mathcal{M} \not\models \beta$. But from the former we get $\mathcal{M} \models \exists x\ \alpha(x)$ and this contradicts $\mathcal{M} \models \exists x\ \alpha(x) \rightarrow \beta$.

For the right to left direction, suppose $\mathcal{M} \models \forall x(\alpha(x) \rightarrow \beta)$. As $\mathcal{M}_T$ also satisfies that formula it must also satisfy the classically equivalent formula $\exists x\ \alpha(x) \rightarrow \beta$. We remain to prove that $\mathcal{M} \models \exists x\ \alpha(x)$ implies $\mathcal{M} \models \beta$. Assume that the former holds. Then, for some element d, $\mathcal{M} \models \alpha(d)$. As $\mathcal{M} \models \forall x(\alpha(x) \rightarrow \beta)$, in particular, $\mathcal{M} \models \alpha(d) \rightarrow \beta$, but this together with $\mathcal{M} \models \alpha(d)$ implies $\mathcal{M} \models \beta$.

□

Proof (Theorem 1. Main result). We prove the result by induction on the successive application of $\cdot^*$ in each group of newly generated rules. If a rule r is regular it can be easily checked that $r^* = r$ and the result of strong equivalence is straightforward. If r contains a double negation or an existential quantifier,

we will show that the two steps in Definition 5 preserve strong equivalence. Step i) is the result of the successive application of Lemma 3, that allows us to remove a positive existential quantifier in the body, provided that the quantified variable does not occur free in the rest of the formula. Notice that this lemma can be applied to a larger body like $\exists x\ \alpha(x) \wedge \gamma \rightarrow \beta$ (again, with x not free in γ) because the latter is QHT-equivalent to $\exists x\ \alpha(x) \rightarrow (\gamma \rightarrow \beta)$. For Step ii), consider any rule $r : B \rightarrow Hd$ with some non-literal negative condition $\neg\beta_1(\mathbf{x})$. We can write r as $B^+(\mathbf{x}) \wedge \neg\beta_1(\mathbf{x}) \wedge B'(\mathbf{x}) \rightarrow Hd(\mathbf{x})$, being $B'(\mathbf{x})$ the rest of conjuncts in the negative body, that is, $B^-(\mathbf{x})$ excepting $\neg\beta_1(\mathbf{x})$. This expression can be equivalently written as $B^+(\mathbf{x}) \wedge \neg\beta_1(\mathbf{x}) \rightarrow (B'(\mathbf{x}) \rightarrow Hd(\mathbf{x}))$ and so, we can apply Theorem 7 taking $\alpha(\mathbf{x})$ to be the positive body $B^+(\mathbf{x})$, and $\gamma(\mathbf{x})$ the implication $B'(\mathbf{x}) \rightarrow Hd(\mathbf{x})$ to conclude that r is strongly equivalent (modulo its original language $\mathcal{L}$) to the conjunction of $B^+(\mathbf{x}) \wedge \neg Aux_1(\mathbf{x}) \wedge B'(\mathbf{x}) \rightarrow Hd(\mathbf{x})$ plus $B^+(\mathbf{x}) \wedge \beta_1(\mathbf{x}) \rightarrow Aux_1(\mathbf{x})$ being Aux_1 a new fresh predicate. We can repeat this step for the rest of non-literal negative conditions in B^- until the original rule becomes $B^+(\mathbf{x}) \wedge \neg Aux_1(\mathbf{x}) \wedge \cdots \wedge \neg Aux_n(\mathbf{x}) \wedge B''(\mathbf{x}) \rightarrow Hd(\mathbf{x})$, i.e., what we called $B^+ \wedge (B^-)^\bullet \rightarrow Hd$ in Definition 5. Finally, the correctness of the translation of the newly generated rules $B^+(\mathbf{x}) \wedge \beta_i(\mathbf{x}) \rightarrow Aux_i(\mathbf{x})$ follows from the induction hypothesis. Note that termination of this inductive transformation $\cdot^*$ is guaranteed by observing that in each step, we reduce the size of new rule bodies, replacing negative non-literal conditions by smaller expressions. □

Proof (Theorem 2). First, observe that all the rules generated in the translation either repeat one of the original rule heads in Π or just contain one atom $Aux(\mathbf{x})$. Thus, if the original program was disjunctive (resp. normal) then Π^* will be disjunctive (resp. normal). Second, just notice that the translation is recursively repeated until rule bodies exclusively contain literal conditions, so the final program will be a regular logic program in the usual sense. □

Proof (Theorem 3). It suffices to observe that the rules generated in each translation step preserves safety with respect to Definition 7. Assume we start from a safe rule and obtain its translation following the steps in Definition 5. In Step i) of that definition, each time we remove $\exists x\varphi$ and replace it by $\varphi[x/y]$ we are introducing a new free variable y in the rule that must satisfy condition *a)* in Definition 7 to maintain safety. But this is guaranteed because the original rule was safe and so, x occurred free and outside the scope of negation in φ. Therefore, y will occur free and outside the scope of negation in $\varphi[x/y]$, which is part of the resulting rule body. This means that the resulting rule satisfies *a)* in Definition 7 for variable y while the status of the rest of variables in the rule has not changed.

Now, take the rule $r : B \rightarrow Hd$ that results from iterating Step i) which, as we have seen, preserves safety. Notice that r does not contain quantified expressions outside the scope of negation, so that B^+ is just a conjunction of atoms. It can be easily observed that each rule $r' : B^+ \wedge (B^-)^\bullet \rightarrow Hd$ does not introduce new free

variables with respect to $B \rightarrow Hd$ (it just replaced any negative condition like $\neg\alpha(\mathbf{x})$ in B^- by a new atom $\neg Aux(\mathbf{x})$) while it maintains the original positive body B^+. So, as the original rule r was safe, all free variables in r' also satisfy condition *a)* in Definition 7, while *b)* is not applicable because r' is regular (its body exclusively consists of literals). Similarly, rules like $r'' : B^+ \wedge \alpha(\mathbf{x}) \rightarrow Aux(\mathbf{x})$ in $\Pi(B^-, B^+)$ do not introduce new free variables with respect to r either, while they maintain the same positive body B^+, so they will satisfy *a)* in Definition 7. On the other hand, any quantified condition like $\exists y\ \varphi$ that occurs in $\alpha(\mathbf{x})$ also occurred in a condition $\neg\alpha(\mathbf{x})$ in r. As r was safe, $\exists y\ \varphi$ will satisfy *b)* in Definition 7, so that rule r'' is safe too. □

Kato: A Plagiarism-Detection Tool for Answer-Set Programs*

Johannes Oetsch, Martin Schwengerer, and Hans Tompits

Institut für Informationssysteme 184/3, Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,schwengerer,tompits}@kr.tuwien.ac.at

Abstract. We present the tool Kato which is, to the best of our knowledge, the first tool for plagiarism detection that is directly tailored for answer-set programming (ASP). Kato aims at finding similarities between (segments of) logic programs to help detecting cases of plagiarism. Currently, the tool is realised for DLV programs but it is designed to handle various logic-programming syntax versions. We review basic features and the underlying methodology of the tool.

1 Background

With the rise of the Internet and its easy access of information, plagiarism is a growing problem not only in academia but also in science and technology in general. In software development, plagiarism involves copying (parts of) a program without revealing the source where it was copied from. The relevance of plagiarism detection for conventional program development is well acknowledged [1]—it is not only motivated by an academic setting to prevent students from violating good academic standards, but also by the urge to retain the control of program code in industrial software development projects.

We are concerned with plagiarism detection in the context of answer-set programming (ASP) [2]. In particular, we deal with disjunctive logic programs under the answer-set semantics [3]. Answer-set programming is characterised by the feature that problems are encoded in terms of theories such that the solutions of a problem instance correspond to certain models (the "answer sets") of the corresponding theory. It differs from imperative languages like C++ or Java (and also to some extent from Prolog) because of its genuine declarative nature: a logic program is a specification rather than an instruction of how to solve a problem; the order of the rules and the order of the literals within the heads and bodies of the rules have no effect on the semantics of a program. Hence, someone who copies code has other means to disguise the deed.

For conventional programming languages, sophisticated tools for plagiarism detection exist, like, e.g., YAP3 [4], Sim [5], JPlag [6], XPlag [7], and others [8]. However, most techniques are not adequate for ASP. The reason is the declarative nature of ASP as well as the lack of a control flow. Especially the fact that the order of statements (and of the literals of a statement) is not relevant for a program causes that existing

* This work was partially supported by the Austrian Science Fund (FWF) under project P21698.

techniques are unsuitable in general. As well, many commonly used tools work with a tokenisation: the source code is translated into a token string where code strings are replaced by generic tokens. For instance, a tokeniser could replace each concrete number by the abstract token <VALUE>. The resulting token strings are used for the further comparisons by searching for common substrings. However, the structure of a DLV program is rather homogeneous—there are not many built-in predicates—which makes this technique unsuitable for detecting copies.

The need for tools for plagiarism detection in ASP is clearly motivated by the growing application in academia and industry, but our primary interest to have such a tool is to apply it in the context of a laboratory course at our university. We thus developed the tool Kato which, to the best of our knowledge, is the first system for plagiarism detection that is directly tailored for ASP.[1] Kato aims at finding similarities between (segments of) logic programs to help detecting cases of plagiarism. Currently, the tool is realised for DLV programs but it is designed to handle various logic-programming syntax versions as well.[2] In what follows, we review the basic features of Kato and outline its underlying methodology.

2 Features and Basic Methodology of Kato

Kato was developed to find suspicious pairs of programs stemming from student assignments in the context of a course on logic programming at our university. Hence, the tool can perform pairwise similarity tests on a rather large set of relatively small programs. In what follows, we provide basic information concerning the implemented features of Kato and how they are realised.

Figure 1 shows the basic working steps needed to perform a test run, which can be divided into three major phases: First, the programs are parsed and normalised in a preprocessing step. Then, test specific preparations are applied. Finally, the programs are compared.

Following a hybrid approach, Kato performs four kinds of comparison tests, realising different layers of granularity: (i) a comparison of the program comments via the longest common subsequence (LCS) metric (see the work of Bergroth et al. [9] for an overview), (ii) an LCS test on the whole program, (iii) a fingerprint test, and (iv) a structure test. We recall that the LCS of two strings is the longest set of identical symbols in both strings with the same order. Hence, the LCS metric tolerates injected non-matching objects. Note that (i) and (ii) are language independent while (iii) and (iv) need to be adapted for different language dialects. All of these tests, outlined in more detail below, compare files pairwise and return a similarity value between 0 (no similarities) and 1 (perfect match).

LCS-Comment Tests. It is surprising what little effort some people spend to mask copied comments. This test reveals similarities between program comments when interpreted as simple strings via the LCS metric.

[1] The name of the tool derives, with all due acknowledgments, from Inspector Clouseau's loyal servant and side-kick, Kato.

[2] See http://www.dlvsystem.com/ for details about DLV.

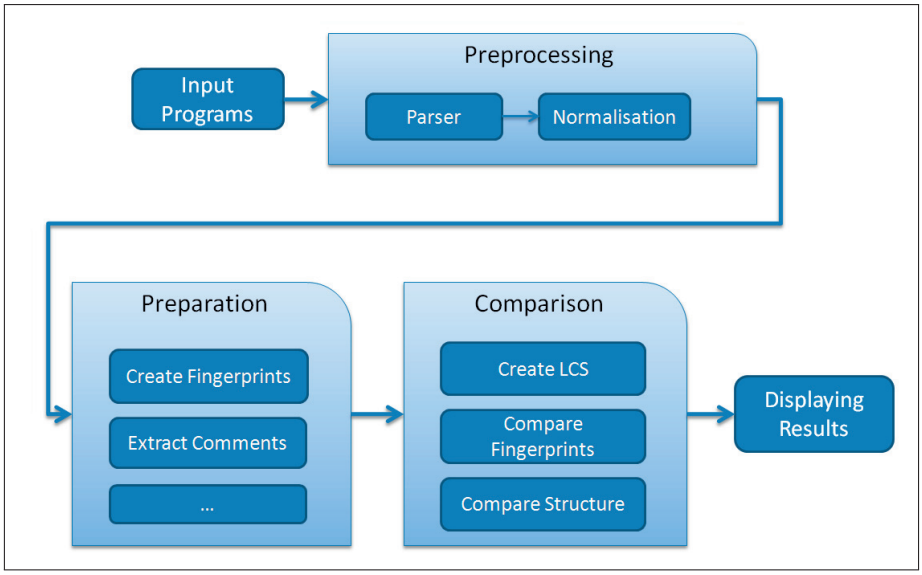


Fig. 1. Overview over a test run

LCS-Program Tests. Similar to the LCS-comment test, the whole programs are interpreted as two strings which are then tested for their longest common subsequence. This test represents an efficient method to detect cases of plagiarism where not much time has been spent to camouflage the plagiarism or parts of it.

Fingerprint Tests. A fingerprint of a program is a collection of relevant statistical data like hash-codes, the number of rules, the number of predicates, the number of constants, program size, and so on. After fingerprints of all programs are generated, they are compared pairwise. This gives a simple yet convenient way to collect further evidence for plagiarism.

Structure Tests. This kind of tests gives, by taking the structure of the programs into account, the most significant information in general. The central similarity function underlying the structure tests is defined as follows: Let $lit_H(r)$ be the multiset of all literals occurring in the head of a given rule r and $lit_B(r)$ the multiset of all literals occurring in the body of r. Then, for two rules r_1 and r_2, the *rule similarity*, $\sigma(r_1, r_2)$, is defined as

$$\sigma(r_1, r_2) = \frac{|lit_H(r_1) \cap lit_H(r_2)| + |lit_B(r_1) \cap lit_B(r_2)|}{\max(|lit_H(r_1)| + |lit_B(r_1)|, |lit_H(r_2)| + |lit_B(r_2)|)}.$$

Furthermore, for two programs P_1 and P_2 (interpreted as multisets of rules), the *similarity*, $\mathcal{S}(P_1, P_2)$, is given by

$$\mathcal{S}(P_1, P_2) = \frac{\sum_{r \in P_1} \max(\sigma(r, r') : r' \in P_2)}{|P_1|}.$$

Note that $\mathcal{S}$ is not symmetrical in its arguments. For any two programs P_1 and P_2, $\mathcal{S}(P_1, P_2)$ is mapped to a value between 0 and 1 which, roughly speaking, expresses to which extent P_1 is subsumed by P_2 by similar rules.

By definition, $\mathcal{S}$ thwarts disguising strategies like permuting rules or literals within rules. However, a more advanced plagiarist could also uniformly rename variables within rules or rename some auxiliary predicates. Therefore, our similarity test comes with different levels of abstraction to counter these malicious efforts. Such renaming is handled by finding and applying suitable substitution functions. Without going into details, the problem of finding such functions is closely related to the homomorphism problem for relational structures which is known to be NP-complete. To circumvent the high complexity, we use an efficient greedy heuristic to obtain our substitutions.

To make the similarity function sensitive to common rule patterns, we also implemented a context dependent extension: A *global occurrence table* gives additional information how specific two rules are. The main idea is that rare rules yield better evidence for a copy than common ones. Therefore, `Kato` collects and counts all rules in the considered corpus of programs and stores this information in an occurrence table. During the comparison, the rule similarity is then weighted depending on the frequency of the involved rules.

3 Further Information and Discussion

The tool is entirely developed in Java (version 6.0). The results of the program comparisons are displayed in tabular form with features like sorting and filtering. For the structure tests, the tool shows program pairs and highlights similar rules. Currently, `Kato` is designed for `DLV`'s language dialect but it can be easily extended to other dialects—it is planned to consider standard Prolog as well. `Kato` was successfully applied in a logic programming course at our university; all cases of plagiarism detected by the supervisors showed high similarities, and even further cases of plagiarism could be detected.

Detailed empirical analyses in terms of *precision* and *recall*, as well as comparisons of our approach with existing tools for plagiarism detection, are left for future work. A further topic for future work is to develop means to visualise the comparison results, e.g., to spot clusters of cooperating plagiarists more easily.

Another interesting aspect of `Kato` is a possible use as a software engineering tool: If a team is working on a program, different versions will emerge. Then, the question about the actual differences between two versions is immanent. `Kato` can be adapted to answer such questions.

Additional information about the tool, and how to obtain it, can also be found at

`http://www.kr.tuwien.ac.at/research/systems/kato.`

References

1. Clough, P.: Plagiarism in natural and programming languages: An overview of current tools and technologies. Technical Report CS-00-05, Department of Computer Science, University of Sheffield, UK (2000)

2. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge, England (2003)
3. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–385
4. Verco, K.L., Wise, M.J.: YAP3 : Improved detection of similarities in computer program and other texts. In Klee, K.J., ed.: Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education, New York, ACM Press (1996) 130–134
5. Gitchell, D., Tran, N.: Sim: A utility for detecting similarity in computer programs. In: Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education, ACM Press (1999) 266–270
6. Prechelt, L., Malpohl, G., Philippsen, M.: JPlag: Finding plagiarisms among a set of programs. Technical Report 2000-1, Fakultät für Informatik Universität Karlsruhe, Germany. (2000)
7. Arwin, C., Tahaghoghi, S.M.M.: Plagiarism detection across programming languages. In: Proceedings of the Twenty-Ninth Australasian Computer Science Conference (ACSC 2006). Volume 48 of CRPIT. Hobart, Australia, ACS (2006) 277–286
8. Jones, E.L.: Metrics based plagiarism monitoring. In: Proceedings of the Sixth Annual CCSC Northeastern Conference. (2001) 1–8
9. Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: SPIRE. (2000) 39–48

A Paraconsistent Semantics for Generalized Logic Programs

Heinrich Herre[1,2] and Axel Hummel[1,2]

[1] Department of Computer Science, Faculty of Mathematics and Computer Science, University of Leipzig, Johannisgasse 26, 04103 Leipzig, Germany, heinrich.herre@imise.uni-leipzig.de, hummel@informatik.uni-leipzig.de
[2] Research Group Ontologies in Medicine and Life Sciences, Institute of Medical Informatics, Statistics and Epidemiology, University of Leipzig, Härtelstrasse 16-18, 04107 Leipzig, Germany

Abstract. We propose a paraconsistent declarative semantics of possibly inconsistent generalized logic programs which allows for arbitrary formulas in the body and in the head of a rule (i.e. does not depend on the presence of any specific connective, such as negation(-as-failure), nor on any specific syntax of rules). For consistent generalized logic programs this semantics coincides with the stable generated models introduced in [HW97], and for normal logic programs it yields the stable models in the sense of [GL88].

Keywords: Paraconsistency, generalized logic programs, multi-valued logic

1 Introduction

Declarative semantics provides a mathematical precise definition of the meaning of a program in a way, which is independent of procedural considerations, easy to manipulate, and reason about. Logic programs and deductive databases should be as easy to write and comprehend as possible and as close to natural discourse as possible. Standard logic programs, in particular definite programs, seems to not be sufficiently expressive for a comprehensible representation of large knowledge bases and of informal descriptions. In recent years there has been an increasing interest in extensions of the classical logic programming paradigm beyond the class of normal logic programs. Generalized logic programs, introduced and studied in [HW97], admit more complex formulas in the rules and thus are more expressive and natural to use since they permit in many cases easier translation from natural language expressions and from informal specifications. The expressive power of generalized logic programs also simplifies the problem of translation of non-monotonic formalisms

into logic programs, [EH97], [EH99]. In many of the traditional logic programming semantics local inconsistency might spoil the whole program, because a contradictory statement $F \wedge \neg F$ implies every formula, i.e. the whole program would be trivialized. This is also the case for the semantics of stable generated models introduced and studied in [HW97].

In this paper we represent a declarative semantics of possibly inconsistent generalized logic programs. This paraconsistent semantics is an extension of the semantics of stable generated models, which agree on normal logic programs with the stable models of Gelfond and Lifschitz [GL88]. We assume the following leading principles for a well-behaved paraconsistent extension of logic programs.

1. The proposed syntax of rules in such programs resembles the syntax of normal logic programs but it applies to a significantly broader class of programs;
2. The proposed semantics of such programs constitutes an intuitively natural extension of the stable semantics of normal logic programs.
3. For consistent generalized logic programs the semantics coincides with the semantics of stable generated models.

The paper has the following structure. After introducing some basic notation in section 2, we introduce in section 3 several kinds of minimal models. From this we derive an adequate paraconsistent semantics for arbitrary theories in predicate logic. In section 4 we discuss the notion of a generalized program. In section 5, we define the concept of a paraconsistent stable generated model and show that the semantics satisfies the before mentioned principles 1.- 3. Section 6 contains the conclusion and a discussion of the related work.

2 Preliminaries

A *signature* $\sigma = \langle Rel, Const, Fun \rangle$ consists of a set of relation symbols, a set of constant symbols, and a set of function symbols. U_σ denotes the set of all ground terms of σ. For a tuple $t_1, \ldots, t_n$ we will also write $\overline{t}$ when its length is of no relevance. The logical functors are $\neg, \wedge, \vee, \rightarrow, \forall, \exists$. $L(\sigma)$ is the smallest set containing the atomic formulas of σ, and being closed with respect to the following conditions: if $F, G \in L(\sigma)$, then $\{\neg F,\ F \wedge G,\ F \vee G,\ F \rightarrow G,\ \exists x F,\ \forall x F\} \subseteq L(\sigma)$.

$L^0(\sigma)$ denotes the corresponding set of *sentences* (closed formulas). For sublanguages of $L(\sigma)$ formed by means of a subset $\mathcal{F}$ of the logical functors, we write $L(\sigma; \mathcal{F})$. Formulas from $L(\sigma; \{\neg, \wedge, \vee, \rightarrow\})$ are called

quantifier-free and a *quantifier-free theory* is a set of quantifier-free formulas. With respect to a signature σ we define $\mathrm{At}(\sigma) = L(\sigma;\emptyset)$, the set of all atomic formulas (also called *atoms*). For a set X of formulas let $\overline{X} = \{\neg F \mid F \in X\}$. Then the set of all *literals* is defined as $\mathrm{Lit}(\sigma) = \mathrm{At}(\sigma) \cup \overline{\mathrm{At}(\sigma)}$. We introduce the following conventions. When $L \subseteq L(\sigma)$ is some sublanguage, L^0 denotes the corresponding set of sentences. If the signature σ does not matter, we omit it and write, e.g., L instead of $L(\sigma)$. ω denotes the least infinite ordinal number, and $Pow(X)$ or 2^X denotes the set of all subsets of X.

A logic $\mathcal{L} = (L, C)$ over the language L can be understood as an operator $C : Pow(L) \to Pow(L)$ determining the consequences of a set $X \subseteq L$ of formulas. Cn denotes the closure operator of classical logic, i.e. $Cn(X)$ is the set of all classical logical consequences of X. Obviously, if X is classically inconsistent then $Cn(X) = L$. A logic (L, C) is said to be a *paraconsistent approximation* of (L, Cn) if the following conditions are satisfied.

1. $C(\{F, \neg F\}) \neq L$ for every formula $F \in L$ (Paraconsistency).
2. If $Cn(X) \neq L$ then $Cn(X) = C(X)$ (Conservativity).
3. $C(X) = C(C(X))$.

Definition 1 (Interpretation) *Let $\sigma = \langle Rel, Const, Fun\rangle$ be a signature. A Herbrand σ-interpretation is a set of literals $I \subseteq \mathrm{Lit}^0(\sigma)$ satisfying the condition $\{a, \neg a\} \cap I \neq \emptyset$ for every ground atom $a \in \mathrm{At}^0(\sigma)$ (interpretations with this property are also called total). Its* universe *is equal to the set of all ground terms U_σ; its canonical interpretation of ground terms is the identity mapping.*

The class of all Herbrand σ-interpretations is denoted by $\boldsymbol{I}(\sigma)$. In the sequel we shall also simply say 'interpretation' instead of 'Herbrand interpretation'. An interpretation I can be represented as a truth-value function from $At^0(\sigma)$ to $\{t, f, \top\}$ by the following stipulation: $I(a) = \top$ if $\{a, \neg a\} \subseteq I$, $I(a) = t$ if $a \in I \wedge \neg a \notin I$, and $I(a) = f$ if $\neg a \in I \wedge a \notin I$. Conversely, every truth-value function $I : At^0 \to \{t, f, \top\}$ can be understood as an interpretation. In the sequel we use the notion of an interpretation simultaneously as a set of literals and as a truth-value function.

A *valuation* over an interpretation I is a function ν from the set of all variables *Var* into the Herbrand universe U_σ, which can be naturally extended to arbitrary terms by $\nu(f(t_1, \ldots, t_n)) = f(\nu(t_1), \ldots, \nu(t_n))$. Analogously, a valuation ν can be canonically extended to arbitrary formulas F, where we write $F\nu$ instead of $\nu(F)$. Furthermore the truth-value function I can be extended to a function $\overline{I}$ which is defined for every formula

$F \in L$. In order to be in a position to give a formal definition of $\overline{I}$ we fix a linear ordering $f < \top < t$ and a unary function *neg* defined by $neg(t) = f, neg(f) = t, neg(\top) = \top$.

Definition 2 (Model Relation) *Let I be an interpretation of signature σ. The extension $\overline{I}$ of I is a function from the set of all sentences F from $L(\sigma)$ into the set $\{t, f, \top\}$, and it is defined inductively by the following conditions.*

1. $\overline{I}(F) = I(F)$ *for every* $F \in At^0(\sigma)$
2. $\overline{I}(\neg F) = neg(\overline{I}(F))$
3. $\overline{I}(F \wedge G) = min\{\overline{I}(F), \overline{I}(G)\}$
4. $\overline{I}(F \vee G) = max\{\overline{I}(F), \overline{I}(G)\}$
5. $\overline{I}(F \rightarrow G) = \overline{I}(\neg F \vee G)$
6. $\overline{I}(\exists x F(x)) = sup\{\overline{I}(F(x/t)) \ : \ t \in U(\sigma)\}$
7. $\overline{I}(\forall x F(x)) = inf\{\overline{I}(F(t)) \ : \ t \in U(\sigma)\}$

Let be $\{t, \top\}$ the set of designated truth values. We say that an interpretation I is a model of a set X of sentences, denoted by $I \models X$, if for every sentence $F \in X$ holds: $\overline{I}(F) \in \{t, \top\}$. The model relation between an interpretation $I \in \boldsymbol{I}(\sigma)$ and a formula $F \in L(\sigma)$ is defined by $I \models F$ iff $I \models F\nu$ for every valuation $\nu : Var \rightarrow U_\sigma$. $\mathrm{Mod}(X) = \{I \in \boldsymbol{I} : I \models X\}$ denotes the Herbrand model operator, and $\models$ denotes the corresponding consequence relation, i.e. $X \models F$ iff $\mathrm{Mod}(X) \subseteq \mathrm{Mod}(F)$. For a set $\mathbf{K} \subseteq \boldsymbol{I}(\sigma)$ and $F \in L(\sigma)$ define $\mathbf{K} \models F$ iff for all $I \in \mathbf{K}$ holds $I \models F$. The set $Th(\mathbf{K}) = \{F \mid \mathbf{K} \models F\}$ is called the theory of $\mathbf{K}$.

Proposition 1 *[We97] Let $C(X)$ be the operator defined by $C(X) = \{F \mid X \models F\}$. Then C satisfies paraconsistency, inclusion, idempotence and compactness, but not conservativity.*

An example which illustrates that the operator C violates the conservativity can be found in [We97, page 14].

Example 1 *Consider the set $X = \{a, a \rightarrow b\}$. Then the following interpretations are models of X: $I_1 = \{a, b\}$, $I_2 = \{a, b, \neg b\}$, $I_3 = \{a, \neg a, b\}$, $I_4 = \{a, \neg a, \neg b\}$, $I_5 = \{a, \neg a, b, \neg b\}$. Because of I_4 we obtain $b \notin C(X)$.*

3 Minimal Models

Our aim is to define a semantics for logic programs which defines for classical consistent programs the (two-valued) stable generated models,

but in case of inconsistent programs yields suitable three-valued models assuring paraconsistency. The class of models $\mathrm{Mod}(X)$ is not suitable because conservativity does not hold, i.e. there are consistent theories T such that $C(T) \neq Cn(T)$. An adequate solution of this problem uses different types of minimal models, one of them, minimizing inconsistency, was introduced in [Pr91] and studied in [We97]. Let $\leq$ be a transitive and reflexive relation on the set $\mathcal{K}$. An element $M \in \mathcal{K}$ is said to be $\leq$-minimal if and only if there is no $N \in \mathcal{K}$ such that $N \leq M$ and $N \neq M$. Let $\mathrm{Min}_{\leq}(\mathcal{K})$ be denote the set of $\leq$-minimal elements of $\mathcal{K}$. In this section we analyze several forms of minimal models of a paraconsistent theory. Let I be an interpretation, then $Pos(I) = I \cap At^0$ and $Neg(I) = I \cap \{\neg a : a \in At^0\}$. Let be $inc(I) = \{a : \{a, \neg a\} \subseteq I\}$.

Definition 3 *Let I, J be interpretations. Then*

1. $I \preceq J$ *if and only if* $Pos(I) \subseteq Pos(J)$ *and* $Neg(J) \subseteq Neg(I)$;
2. $I \sqsubseteq J$ *if and only if* $inc(I) \subseteq inc(J)$.

Using the relations $\preceq, \sqsubseteq, \subseteq$ we introduce the following forms of minimality.

Definition 4 *Let X be a set of formulas and I be an interpretation.*

1. *I is a t-minimal model of X iff* $I \in \mathrm{Min}_{\preceq}(\mathrm{Mod}(X))$.
2. *I is an inc-minimal model of X iff* $I \in \mathrm{Min}_{\sqsubseteq}(\mathrm{Mod}(X))$.
3. *I is an i-minimal model of X iff* $I \in \mathrm{Min}_{\subseteq}(\mathrm{Mod}(X))$.

We introduce following model operators: $\mathrm{Mod}_{incm}(X) = \{I \mid I$ is an inc-minimal model of $X\}$, $\mathrm{Mod}_{im}(X) = \{I \mid I$ is an i-minimal model of $X\}$, and $\mathrm{Mod}_{tm}(X) = \{I \mid I$ is a t-minimal model of $X\}$. Using these different notions of minimality we get following consequence operations, for $* \in \{incm, tm, im\}$: $X \models_* F$ iff $\mathrm{Mod}_*(X) \subseteq \mathrm{Mod}(\{F\})$.

Proposition 2 *Every inc-minimal model of a set X of formulas is i-minimal.*

Proof: Let I be an inc-minimal model of X and assume that I is not i-minimal. Then there is a model $J \subseteq I$, $J \models X$, and $J \neq I$. This implies the existence of a literal $l \in I - J$, assume that $l = a$ is a ground atom. Since J is a total interpretation this yields $\neg a \in J$, but then $\{a, \neg a\} \subseteq I$, which contradicts the inc-minimality of I. $\square$

There are t-minimal models not being inc-minimal and inc-minimal models not being t-minimal.

Example 2 *For clarification, we discuss the following examples:*

1. *Consider the program* $P_1 = \{\neg b \rightarrow a\}$. *Obviously,* $I = \{a, \neg a, \neg b\}$ *is a t-minimal model of* P, *but* I *is not inc-minimal because* $I = \{a, \neg b\}$ *is a model of* P *satisfying* $inc(J) = \emptyset$. *There are, trivially inc-minimal models not being t-minimal (note that every two-valued model is inc-minimal).*
2. *Let* $P_2 = \{b \rightarrow \neg a; \rightarrow a\}$. *Then* $I = \{\neg a, a, b\}$ *is an i-minimal model not being inc-minimal, since* $\{a, \neg b\}$ *is a model of* P.
3. *Every two-valued model is i-minimal (among the total models), but not, in general t-minimal. There are also t-minimal models not being i-minimal: the interpretation* $\{\neg a, a, \neg b\}$ *is a t-minimal model of* P_1, *but it is not i-minimal.*

We recall the following result from [We97].

Proposition 3 *1. Let* T *be a quantifier-free theory. Then the theory* T *has an inc-minimal model.*
2. *The consequence operation* $C_{incm}(X) = \{F \mid X \models_{incm} F\}$ *is a paraconsistent, non-monotonic approximation of* Cn.

The main result of this section is the following.

Proposition 4 *Let* T *be a quantifier-free theory and* I *an inc-minimal model of* T. *Then there exists a model* J *of* T *such that*

1. $inc(I) = inc(J)$
2. $J \preceq I$
3. *for all* $J_0 \preceq J$ *such that* $J_0 \neq J$ *either* $inc(J_0) \neq inc(I)$ *or* $J_0 \not\models T$.

Proof: We may assume that T is a set of clauses, these are formulas of the form $a_1 \vee \ldots \vee a_m \vee \neg b_1 \vee \ldots \vee \neg b_n$, where a_i, b_j are atomic formulas. Let be $\Delta(I) = \{J \mid J \preceq I$ and $J \models T$ and $inc(I) = inc(J)\}$. We consider decreasing sequences within the system $(\Delta(I), \preceq)$. Let be $J_0 \succeq \ldots \succeq J_n \succeq \ldots$ a decreasing sequence, $J_n \in \Delta(I), n < \omega$. Obviously, the sequence $\{J_n \mid n < \omega\}$ has a lower bound J^*, defined by $Pos(J^*) = \bigcap\{Pos(J_n) | n < \omega\}$, and $Neg(J^*) = \bigcup\{Neg(J_n) | n < \omega\}$. We show that $J^* \in \Delta(I)$. By *Zorn's lemma* this implies the result of the theorem. Since $inc(J^*) = inc(I)$, it remains to show that $J^* \models T$. Assume, this is not the case; then there is a formula $F(\overline{x}) = a_1(\overline{x}) \vee \ldots \vee a_m(\overline{x}) \vee \neg b_1(\overline{x}) \vee \ldots \vee \neg b_n(\overline{x})$ from T such that $J^* \not\models \forall \overline{x} F(\overline{x})$, which implies $J^*(\forall \overline{x} F(\overline{x})) \notin \{\top, t\}$, hence $J^*(\forall \overline{x} F(\overline{x})) = f$ which yields $J^*(\neg \forall \overline{x} F(\overline{x})) = J^*(\exists \overline{x} \neg F(\overline{x})) = t$. Hence, there are variable-free terms $\bar{t}$ such that $\{\neg a_1(\bar{t}), \ldots, \neg a_m(\bar{t}), b_1(\bar{t}), \ldots, b_n(\bar{t})\} \subseteq J^*$. This

condition implies $\{b_1(\bar{t}), \ldots, b_n(\bar{t})\} \subseteq J_n$ for every $n < \omega$. But, there must be also a number $k < \omega$ such that $\{\neg a_1(\bar{t}), \ldots, \neg a_m(\bar{t})\} \subseteq J_k$, and this implies $J_k(\bigwedge_{i \leq m} \neg a_i(\bar{t}) \wedge \bigwedge_{j \leq n} b_j(\bar{t})) = t$ (the value $\top$ is not possible, otherwise this would imply $inc(J_k) \neq inc(J^*)$). This gives a contradiction. □

Proposition 4 shows that for every model I of a quantifier-free theory T the set $\mathrm{Min}_{\preceq}(\{J \mid inc(J) = inc(I),\ J \in \mathrm{Mod}(T)\})$ is non-empty.

Corollary 5 *Let T be a universal theory, I a model of T and let J be a set of literals such that $J \preceq I$ and $inc(J) = inc(I)$. Then the set $\{K \mid J \preceq K \preceq I$ and $K \models T$ and $inc(K) = inc(I)\}$ contains a $\preceq$-minimal element.*

Proof: Follows immediately from proposition 4.

4 Sequents and Logic Programs

In the sequel we use Gentzen-sequents to represent rule knowledge as proposed in [HW97]. A sequent, then, is not a schematic but a concrete expression representing some piece of knowledge.[3]

Definition 5 (Sequent) *A* sequent *s is an expression of the form*

$$F_1, \ldots, F_m \ \Rightarrow\ G_1, \ldots, G_n$$

where $F_i, G_j \in L(\sigma)$ for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. The body *of s, denoted by $B(s)$, is given by $\{F_1, \ldots, F_m\}$, and the* head *of s, denoted by $H(s)$, is given by $\{G_1, \ldots, G_n\}$.* $\mathrm{Seq}(\sigma)$ *denotes the class of all sequents s such that $H(s), B(s) \subseteq L(\sigma)$, and for a given set $S \subseteq \mathrm{Seq}(\sigma)$, $[S]$ denotes the set of all ground instances of sequences from S.*

The satisfaction set of a formula $F \in L(\sigma)$ with respect to an interpretation $I \in \boldsymbol{I}(\sigma)$ is defined as $\mathrm{Sat}_I(F) = \{\nu \in {U_I}^{Var} : I \models F\nu\}$

Definition 6 (Model of a Sequent) *Let $I \in \boldsymbol{I}$. Then,*

$$I \models F_1, \ldots, F_m \ \Rightarrow\ G_1, \ldots, G_n \quad \textit{iff} \quad \bigcap_{i \leq m} Sat_I(F_i) \subseteq \bigcup_{j \leq n} Sat_I(G_j)$$

[3] The use of sequents is mainly technico-methodological, the sequent-arrow $\Rightarrow$ should be distinguished from the implication connective $\rightarrow$.

Obviously for every sequent $B \Rightarrow H$ and $I \in \boldsymbol{I}$ we have $I \models B \Rightarrow H$ iff $I \models \bigwedge B \rightarrow \bigvee H$. Sometimes, we represent the latter formula by the expression $\bigvee H \leftarrow \bigwedge B$. We define the following classes of sequents corresponding normal, normal disjunctive and generalized logic programs, respectively.

1. $\mathrm{NLP}(\sigma) = \{s \in \mathrm{Seq}(\sigma) : H(s) \in \mathrm{At}(\sigma),\ B(s) \subseteq \mathrm{Lit}(\sigma)\}$.
2. $\mathrm{NDLP}(\sigma) = \{s \in \mathrm{Seq}(\sigma) : H(s) \subseteq \mathrm{At}(\sigma),\ B(s) \subseteq \mathrm{Lit}(\sigma), H(s) \neq \emptyset\}$.
3. $\mathrm{GLP}(\sigma) = \{s \in \mathrm{Seq}(\sigma) : H(s), B(s) \subseteq L(\sigma; \neg, \wedge, \vee, \rightarrow)\}$.

For $P \subseteq \mathrm{GLP}(\sigma)$, the model operators $\mathrm{Mod}_*(P), * \in \{inc, i, t\}$ are defined as in section 3. The associated entailment relations are defined by $P \models_* F$ iff $\mathrm{Mod}_*(P) \subseteq \mathrm{Mod}(F)$, where $* = inc, i, t$, and $F \in L(\sigma)$. The *ground instantiation* of a generalized logic program P is denoted by $[P]$, and defined by $[P] = \{\theta(r) \mid r \in P, \theta \text{ is a ground substitution}\}$. Obviously, if $P \subseteq \mathrm{GLP}$, then $\mathrm{Mod}(P) = \mathrm{Mod}([P])$.

A preferential semantics for sequents is given by a preferred model operator $\Phi : 2^{\mathrm{Seq}} \rightarrow 2^{\mathbf{I}}$, satisfying the condition $\Phi(P) \subseteq \mathrm{Mod}(P)$ and selecting suitable preferred models. Our intuitive understanding of rules suggests a meaning which interprets a sequent as a rule for generating information. We may consider a model I of a set P of sequents as intended if I can be generated bottom-up starting from a suitable least interpretation by an iterated application of the sequents $r \in [P]$. A model of P which can be generated in this way is said to be *grounded* in P. The following examples show that even the following strong form of minimality is not sufficient to satisfy this condition. A model I of P is said to be *inc-t-minimal* if I is *inc-minimal* and there is no model J of P satisfying the conditions $inc(J) = inc(I)$, $J \preceq I$, and $J \neq I$.

Example 3 *1. Let $P = \{\neg p(a) \Rightarrow q(a)\}$. Every intended model of P should contain $q(a)$. But $M = \{p(a), \neg q(a)\}$ is also an inc-t-minimal model of P.*

2. This observation is also valid if P has no two-valued model.
Let $P = \{\Rightarrow r(c);\ \Rightarrow \neg p(a);\ \Rightarrow \neg p(b);\ \Rightarrow p(a), p(b);\ \neg p(x) \Rightarrow q(x)\}$. Every intended model of P should contain $q(c)$. Assume $I \models P$ is inc-minimal, but $q(c) \notin I$. Then, $\neg q(c) \in I$, which implies $p(c) \in I$ (note that I is total). But $p(c)$ cannot be generated by applying the sequents from $[P]$ because $p(c)$ does not appear in the head of any rule $s \in [P]$. [4]

[4] Application of a rule r means, roughly, to make the body $B(r)$ true and then to detach the head $H(r)$.

But $M_1 = \{\neg p(a), \neg p(b), p(a), q(a), q(b), \neg q(c), p(c), r(c), \neg r(a), \neg r(b)\}$ *is an inc-t-minimal model of* P.

5 Paraconsistent Stable Generated Models

Definition 7 (Interpretation Interval) *Let* $I_1, I_2 \in \boldsymbol{I}$ *and* $inc(I_1) = inc(I_2)$. *Then,* $[I_1, I_2] = \{I \in \boldsymbol{I} : I_1 \preceq I \preceq I_2$ *and* $inc(I) = inc(I_1)\}$. *For a program* $P \subseteq$ GLP *let be* $P_{[I_1,I_2]} = \{r \mid r \in [P], [I_1, I_2] \models B(r)\}$.

The following definition of a *paraconsistent stable generated model* combines the construction of a generated model in [HW97] with the notion of inc-minimality.

Definition 8 (Paraconsistent Stable Generated Model) *Let* $P \subseteq$ GLP(σ). *An inc-minimal model* M *of* P *is called* paraconsistent stable generated, *symbolically* $M \in \mathrm{Mod}_{ps}(P)$, *if there is a chain of Herbrand interpretations* $I_0 \preceq \ldots \preceq I_\kappa$ *such that* $M = I_\kappa$, *and*

1. M *is inc-minimal*
2. $I_0 = inc(M) \cup \{\neg a \mid a \in At^0(\sigma)\}$.
3. *For successor ordinals* α *with* $0 < \alpha \leq \kappa$, I_α *is a* $\preceq$*-minimal extension of* $I_{\alpha-1}$ *satisfying the heads of all sequents whose bodies hold in* $[I_{\alpha-1}, M]$, *i.e.* $I_\alpha \in \mathrm{Min}_{\preceq}\{I \in \boldsymbol{I}(\sigma) : M \succeq I \succeq I_{\alpha-1}, inc(M) = inc(I), I \models \bigvee H(s)$, *for all* $s \in P_{[I_{\alpha-1},M]}\}$
4. *For limit ordinals* $\lambda \leq \kappa$, $I_\lambda = \sup_{\alpha<\lambda} I_\alpha$.

We also say that M *is* generated *by the* P-stable chain $I_0 \preceq \ldots \preceq I_\kappa$.

Intuitive, we define that an inc-minimal model M of a generalized logic program P is a paraconsistent stable generated model of P if M is created bottom-up by an iterative application of the rules of P starting with the state of no information (that means every atom is negated) and the inconsistency of M. In every step of the construction the model is extended in that way that the inconsistency is preserved and the head of every applicable sequent is satisfied.

Example 4 *Let* P *be the second logic program of Example 3. Because of the rules* $\{\Rightarrow \neg p(a); \Rightarrow \neg p(b); \Rightarrow p(a), p(b)\}$ *it is easy to see that* P *has no two-valued model. But there are two paraconsistent stable generated models:*
$M_2 = \{\neg r(a), \neg r(b), r(c), \neg p(a), \neg p(b), p(a), \neg p(c), q(a), q(b), q(c)\}$ *and*
$M_3 = \{\neg r(a), \neg r(b), r(c), \neg p(a), \neg p(b), p(b), \neg p(c), q(a), q(b), q(c)\}$.

The model M_2 is constructed by the chain $I_0^2 \preceq I_1^2 = M_2$.
In detail we obtain:
$I_0^2 = \{p(a)\} \cup \{\neg r(a), \neg r(b), \neg r(c), \neg p(a), \neg p(b), \neg p(c), \neg q(a), \neg q(b), \neg q(c)\}$.
So $P_{[I_0^2, M_2]} = \{\Rightarrow r(c);\ \Rightarrow \neg p(a);\ \Rightarrow \neg p(b);\ \Rightarrow p(a), p(b);\ \neg p(a) \Rightarrow q(a);\ \neg p(b) \Rightarrow q(b);\ \neg p(c) \Rightarrow q(c)\}$. *Therefore* $I_1^2 = M_2$ *with*
$I_1^2 = \{\neg r(a), \neg r(b), r(c), \neg p(a), \neg p(b), p(a), \neg p(c), q(a), q(b), q(c)\}$.
For M_3 we obtain:
$I_0^3 = \{p(b)\} \cup \{\neg r(a), \neg r(b), \neg r(c), \neg p(a), \neg p(b), \neg p(c), \neg q(a), \neg q(b), \neg q(c)\}$.
So $P_{[I_0^3, M_3]} = \{\Rightarrow r(c);\ \Rightarrow \neg p(a);\ \Rightarrow \neg p(b);\ \Rightarrow p(a), p(b);\ \neg p(a) \Rightarrow q(a);\ \neg p(b) \Rightarrow q(b);\ \neg p(c) \Rightarrow q(c)\}$ *and therefore* $I_1^3 = M_3$ *with*
$I_1^3 = \{\neg r(a), \neg r(b), r(c), \neg p(a), \neg p(b), p(b), \neg p(c), q(a), q(b), q(c)\}$.
Hence it follows: $P \models_{ps} q(c)$.

Remark: If we assume in definition 8 that the set $inc(M)$ is empty then we get the notion of a *stable generated model* as introduced and studied in [HW97].

Notice that the notion of stable generated models applies to programs admitting negation(-as-failure) in the head of a rule and nested negations, such as in $p(x) \wedge \neg(q(x) \wedge \neg r(x)) \Rightarrow s(x)$ which would be the result of folding $p(x) \wedge \neg ab(x) \Rightarrow s(x)$ and $q(x) \wedge \neg r(x) \Rightarrow ab(x)$.
It turns out that the length of the generated sequence of a stable generated model can be restricted by ω.

Proposition 6 *Let $P \subseteq$ GLP, and let M be a paraconsistent stable generated model of P generated by the sequence $M_0 \preceq \ldots \preceq M_\kappa$. Then there is an ordinal $\beta \leq \omega$ such that $M_\beta = M_\kappa$.*

Proof: We show that every sequence stabilizes at an ordinal $\beta \leq \omega$. Obviously, if $M_\alpha = M_{\alpha+1}$ then $M_\alpha = M_\gamma$ for all $\alpha < \gamma \leq \kappa$. It is sufficient to prove $M_\omega = M_{\omega+1}$. Analogously to [HW97], we proceed in two steps:

(1) First we show that if $s \in [P]$ and $[M_\omega, M] \models B(s)$ then there is a number $n < \omega$ such that $[M_n, M] \models B(s)$. Without loss of generality, we may assume that $B(s)$ is a set of clauses (disjunction of ground literals), i.e. $B(s) = \{C_1, \ldots, C_k\}$, $C_i = a_1^i \vee \ldots \vee a_{m_i}^i \vee \neg b_1^i \vee \ldots \vee \neg b_{n_i}^i$, $i \in \{1, 2, ..., k\}$. A clause C_i is said to be positive if the set $P(i) := M_\omega \cap \{a_1^i, \ldots, a_{m_i}^i\}$ is nonempty, otherwise it is called negative. Let $\{C_1, \ldots, C_s\}$ be the set of positive and $\{C_{s+1}, \ldots, C_k\}$ the set of negative clauses. Because the set $P := \bigcup_{1 \leq i \leq s} P(i)$ is finite, there is a number $j < \omega$ such that $P \subseteq M_j$. Then, trivially, $[M_j, M] \models C_1, \ldots, C_s$. It remains to show: if $M_j \preceq J \preceq M$ then $J \models C_{s+1}, \ldots, C_k$. We proof this

fact indirectly. Assume $J \not\models C_{s+1}, \ldots, C_k$. Then, there exists a number j, $s+1 \leq j \leq k$, such that $J \not\models C_j$ with the following form $C_j = a_1^j \vee \ldots \vee a_{m_j}^j \vee \neg b_1^j \vee \ldots \vee \neg b_{n_j}^j$. Because of $J(C_j) = f$, we obtain $neg(J(C_j)) = t$ and therefore $J \models \neg C_j$. So $J \models \neg a_1^j \wedge \ldots \wedge \neg a_{m_j}^j \wedge b_1^j \wedge \ldots \wedge b_{n_j}^j$ since De Morgan's laws are valid in our paraconsistent semantics. We may assume that the elements in the set $\{a_1^j, \ldots, a_{m_j}^j, b_1^j, \ldots, b_{n_j}^j\}$ are pairwise distinct. Now, we define $M_\omega^* = (M_\omega \setminus \{\neg b_1^j, \ldots, \neg b_{n_j}^j\}) \cup \{b_1^j, \ldots, b_{n_j}^j\}$. Then $Pos(M_\omega) \subseteq Pos(M_\omega^*)$ and $Neg(M_\omega^*) \subseteq Neg(M_\omega)$. So $M_\omega \preceq M_\omega^*$. Furthermore it holds $M_\omega^* \preceq M$ and therefore we obtain $M_\omega^* \in [M_\omega, M]$. Because of $M_\omega^* \cap \{a_1^j, \ldots, a_{m_j}^j, \neg b_1^j, \ldots, \neg b_{n_j}^j\} = \emptyset$ and $\{b_1^j, \ldots, b_{n_j}^j\} \subseteq M_\omega^*$ it follows $M_\omega^* \not\models C_j$. This is a contradiction to $[M_\omega, M] \models C_j$.

(2) Now, we show that $M_\omega = M_{\omega+1}$. It is sufficient to prove: if $s \in P_{[M_\omega, M]}$, then $M_\omega \models \bigvee H(s)$. By (1), the condition $s \in P_{[M_\omega, M]}$ implies that $s \in P_{[M_n, M]}$ for a certain number $n < \omega$, and hence for every $j > n$: $s \in P_{[M_j, M]}$. Hence, $M_j \models \bigvee H(s)$ for every j, $n < j < \omega$. Again, we may assume that $\bigvee H(s)$ is given as a set of clauses $\{C_1, \ldots, C_n\}$. We have to check that $M_\omega \models C_1, \ldots, C_n$. Assume, there is a j, $1 \leq j \leq n$, such that $M_\omega \not\models C_j$, then $M_\omega \models \neg C_j$, $C_j = a_1^j \vee \ldots \vee a_{m_j}^j \vee \neg b_1^j \vee \ldots \vee \neg b_{n_j}^j$, and $M_\omega \models \neg a_1^j \wedge \ldots \wedge \neg a_{m_j}^j \wedge b_1^j \wedge \ldots \wedge b_{n_j}^j$. It is easy to show that there exists a number $m < \omega$ such that $\{b_1^j, \ldots, b_{n_j}^j\} \subseteq M_m$, and from this follows $M_m \not\models C_j$, which is a contradiction. □

Corollary 7 *If M is a paraconsistent stable generated model of $P \subseteq$ GLP, then there is either a finite P-stable chain, or a P-stable chain of length ω, generating M.*

The following example shows that stable generated entailment is not cumulative, i.e. adding derivable formulas to programs may change their consequence set.

Example 5 (Observation 18, [HJW99]) *Let P be the following logic program: $P = \{\neg r(a) \Rightarrow q(a); \neg q(a) \Rightarrow r(a); \neg p(a) \Rightarrow p(a); \neg r(a) \Rightarrow p(a);\}$. Then $Mod_{ps}(P) = \{\{p(a), q(a)\}\}$. Therefore $P \models_{ps} p(a), q(a)$. But $Mod_{ps}(P \cup \{p(a)\}) = \{\{p(a), q(a)\}, \{p(a), r(a)\}\}$ and hence $P \cup p(a) \not\models q(a)$.*

The relation to consistent generalized programs is captured by the following proposition.

Proposition 8 *Let P be a generalized logic program, and assume P is consistent, i.e. has a two-valued classical interpretation. Then a model I of P is paraconsistent stable generated if and only if it is stable generated.*

Proof: By proposition 3 every inc-minimal model is two-valued. □

Corollary 9 *Let P be a normal logic program. Then a model I of P is paraconsistent stable generated if and only if it is stable (in the sense of [GL88]).*

Proof: P is always a two-valued model, since the negation $\neg$ does not appear in the heads of the rules. Now we may apply the preceding proposition and the result in [HW97] (stating that the stable generated models coincide with the stable models for normal logic programs). □

6 Conclusion and Related Work

A framework of paraconsistent logic programs was firstly developed by Blair and Subrahmanian in [BS89]; they employ Belnap's four-valued logic [Be77] as a theoretical basis, but this framework does not treat default negation in a program. Kifer and Lozinskii in [KL92] extend Blair's framework to theories possibly containing default negation. Sakama and Inoue are studying programs in [SI95] whose rules admit disjunction in the head and default negation in the bodies of the rules. Our approach gives a declarative semantics to logic programs whose rules admit arbitrary quantifier-free formulas in the heads and bodies containing negation that can be interpreted as default negation. This semantics coincides on normal logic programs with the stable models in [GL88]. Note, that stable models I satisfies the condition $I \cap \{a, \neg a\} \neq \emptyset$ for every ground atom, and that any adequate generalization of this notion to paraconsistent models should preserve this property. This is the reason, why we assume that the considered interpretations to be total. Our semantics uses the concept of minimal inconsistent interpretations as introduced by Priest in [Pr91], the results in [We97] and the notion of a stable generated model introduced and studied in [HW97].

By introducing a general definition of paraconsistent stable generated models, we have continued the foundation of a *stable model theory* for possibly inconsistent logic programs. It seems to be possible to analyze further extensions of normal logic programs within a similar framework, such as admitting quantifiers in the bodies and the heads of rules. As a consequence of the rapid growth of the Semantic Web, powerful ontology languages like Extended RDF [AADW08] were developed which use the logic programming paradigm. Therefore the application of the stable model theory to that family of languages is a beneficial challenge

for the near future. In [Hu09] a paraconsistent stable generated semantics for a four-valued logic is proposed which depends on the minimally inconsistent models too. Another interesting step is to develop a paraconsistent declarative semantics for generalized logic programs with two kinds of negation satisfying the coherency condition, i.e. $\sim F$ implies $\neg F$, where $\sim$ represents strong negation, and $\neg$ means weak negation which is assumed to be total.

Since the stable models in the sense of [GL88] correspond to the stable models of [FLL07] for normal logic programs, the stable models in the sense of [FLL07] agree also with the two-valued stable generated models if normal logic programs are considered. Because of the fact that the semantics of [FLL07] is also defined for generalized logic programs, a detailed characterization of the relationship between this semantics and the stable generated models belongs to our future plans.

Acknowledgment
Thanks due to the anonymous referees for their criticism and useful comments.

References

[AADW08] Analyti, A, Antoniou, G, Damsio, C. V. and Wagner, G.: Extended RDF as a Semantic Foundation of Rule Markup Languages, *Journal of Artificial Intelligence Research*, 32: 37-94, 2008

[Be77] Belnap, D.N.: A useful four-valued logic. In G. Epstein and J. M. Dunn, editors, *Modern Uses of Many-valued Logic*, 8-37, Reidel, 1977

[BS89] Blair, H.A. and Subrahamanian, V.S.: Paraconsistent logic programming, *Theoretical Computer Science*. 68: 135-154, 1989

[EH97] Engelfriet, J. and Herre, H.: Generated Preferred Models and Extensions of Nonmonotonic Systems, *Logic Programming*, 85-99, Proc. of the 1997 International Symposium, ed. J. Maluszynski, The MIT Press, 1997

[EH99] Engelfriet, J. and Herre, H.: Stable Generated Models, Partial Temporal Logic and Disjunctive Defaults, *Journal of Logic and Algebraic Programming*, 41 (1): 1-25, 1999

[FLL07] Ferraris, P., Lee, J. and Lifschitz, V.: A New Perspective on Stable Models, 372-379, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007, 2007

[GL88] Gelfond, M. and Lifschitz, V.: The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Proc. of ICLP*, 1070–1080. MIT Press, 1988

[Hu09] Hummel, A.: Untersuchungen von Ontologiesprachen mit modelltheoretischer Semantik, Diploma Thesis, University of Leipzig, 2009

[HW97] Herre, H. and Wagner, G.: Stable Models Are Generated by a Stable Chain, *Journal of Logic Programming*, 30 (2): 166-177, 1997

[HJW99] Herre, H., Jaspars, J. and Wagner, G.: Partial logics with two kinds of negation as a foundation for knowledge-based reasoning, in D. Gabbay and H. Wansing (Eds.), *What is negation ?*, pages 121-159. Kluwer Academic Publishers, 1999

[KL92] Kifer, M. and Lozinskii, E.: A logic for reasoning with inconsistency. *Journal of Automated Reasoning*, 8: 179-215, 1992

[Pr91] Priest, G: Minimally inconsistent LP. *Studia Logica*, 50 (2): 321-331, 1991

[SI95] Sakama, C. and Inoue, K.: Paraconsistent Stable Semantics for extended disjunctive programs; *Journal of Logic and Computation* 5: 265-285, 1995

[We97] Weber, S.: Investigations in Belnap's Logic of Inconsistent and Unknown Information, Dissertation, University of Leipzig, 1998

Stationary Generated Models of Generalized Logic Programs

Heinrich Herre[1,2] and Axel Hummel[1,2]

[1] Department of Computer Science, Faculty of Mathematics and Computer Science, University of Leipzig, Johannisgasse 26, 04103 Leipzig, Germany, heinrich.herre@imise.uni-leipzig.de, hummel@informatik.uni-leipzig.de
[2] Institute of Medical Informatics, Statistics and Epidemiology, University of Leipzig, Härtelstrasse 16-18, 04107 Leipzig, Germany

Abstract. The interest in extensions of the logic programming paradigm beyond the class of normal logic programs is motivated by the need of an adequate representation and processing of knowledge. One of the most difficult problems in this area is to find an adequate declarative semantics for logic programs. In the present paper a general preference criterion is proposed that selects the 'intended' partial models of generalized logic programs which is a conservative extension of the stationary semantics for normal logic programs of [Prz91]. The presented preference criterion defines a partial model of a generalized logic program as intended if it is generated by a stationary chain. It turns out that the stationary generated models coincide with the stationary models on the class of normal logic programs. The general wellfounded semantics of such a program is defined as the set-theoretical intersection of its stationary generated models. For normal logic programs the general wellfounded semantics equals the wellfounded semantics.

Keywords: Extensions of logic programs, semantics, knowledge representation

1 Introduction

Declarative semantics provides a mathematical precise definition of the meaning of a program in a manner, which is independent of procedural considerations, context-free, and easy to manipulate, exchange and reason about. Finding a suitable declarative or intended semantics is an important and difficult problem in logic programming and deductive data bases. Logic programs and deductive data bases must be as easy to write and comprehend as possible and as close to natural discourse as possible. Research in the area of logic programming and non-monotonic reasoning made a significant contribution towards the better understanding of

relations existing between various formalizations of non-monotonic reasoning and the discovery of deeper underlying principles of non-monotonic reasoning and logic programming. Standard logic programs are not sufficiently expressive for the representation of large classes of knowledge bases. In particular, the inability of logic programs to deal with arbitrary open formulas is an obstacle to use logic programming as a declarative specification language for software engineering and knowledge representation. Formalisms admitting more complex formulas are more expressive and natural to use since they permit in many cases easier translation from natural language expressions and from informal specifications. The additional expressive power of generalized logic programs significantly simplifies the problem of translation of non-monotonic formalisms into logic programs, and, consequently facilitates using logic programming as an inference engine for non-monotonic reasoning.

A set of facts can be viewed as a database whose semantics is determined by its minimal models. In the case of logic programs, minimal models are not adequate because they are not able to capture the directedness of rules, i.e. they do not satisfy the *groundedness* requirement. Therefore, *stable* models in the form of certain fixpoints have been proposed by Gelfond and Lifschitz [GL88] as the intended models of normal logic programs. We generalize this notion by presenting a definition which is neither fixpoint-based nor dependent on any specific rule syntax. We call our preferred models *stationary generated* because they are generated by a *stationary chain*, i.e. a stratified sequence of rule applications where all applied rules remain (in a certain sense) applicable throughout the model computation. The notion of a stationary model of a normal logic program was introduced in [Prz91] and further elaborated in [Prz94]. Stationary generated models - as expounded in the current paper - are defined in a different way. This notion can be easily extended to generalized logic programs which include several types of programs as special cases, among them disjunctive programs [Prz91] and super-logic programs [Prz96]. Lifschitz, Tang and Turner propose in [LTT99] a semantics for logic programs allowing for nested expressions in the heads and the body of the rules. The syntax is similar to our generalized logic programs, but the semantics differs.

In [AHP00] the notion of stationary generated AP-models was introduced. This notion differs from the stationary generated models as defined in the present paper. Stationary generated AP-models and stationary generated models are based on different truth-relations for three-valued

partial models. Hence, the current paper closes a gap that remained open in [AHP00].

The paper has the following structure. After introducing some basic notation in section 2, we recall some facts about Herbrand model theory and sequents in section 3. In section 4, we define the general concept of a stationary generated model, and then, in section 5 we investigate the relationship of this general concept to the original fixpoint-based definitions for normal programs as in [Prz91]. It turns out that for normal programs the stationary generated models coincide with the stationary models in the sense of [Prz91]. This fact motivates the introduction of the notion of a general well-founded semantics for a generalized logic program which is defined as the set-theoretical intersection of its stationary generated models. We believe that the notion of general well-founded semantic is the most natural generalization of well-founded semantics to generalized logic programs.

2 Preliminaries

A *signature* $\sigma = \langle Rel, Const, Fun\rangle$ consists of a set *Rel* of relation symbols, a set *Const* of constant symbols, and a set *Fun* of function symbols. U_σ denotes the set of all ground terms of σ. For a tuple $t_1, \ldots, t_n$ we will also write $\bar{t}$ when its length is of no relevance. The logical functors are $not, \wedge, \vee, \rightarrow, \forall, \exists$. $L(\sigma)$ is the smallest set containing the atomic formulas of σ, and being closed with respect to the following conditions: if $F, G \in L(\sigma)$, then $\{not\, F,\ F \wedge G,\ F \vee G,\ F \rightarrow G,\ \exists x F,\ \forall x F\} \subseteq L(\sigma)$.

$L^0(\sigma)$ denotes the corresponding set of sentences (closed formulas). For sublanguages of $L(\sigma)$ formed by means of a subset $\mathcal{F}$ of the logical functors, we write $L(\sigma; \mathcal{F})$. With respect to a signature σ we define the following sublanguages: $\mathrm{At}(\sigma) = L(\sigma; \emptyset)$, the set of all atomic formulas (also called *atoms*). The set $\mathrm{GAt}(\sigma)$ of all ground atoms over σ is defined as $\mathrm{GAt}(\sigma) = \mathrm{At}(\sigma) \cap \mathrm{L}^0(\sigma)$. $\mathrm{Lit}(\sigma) = L(\sigma;\ not)$, the set of all *literals*; for a set X of formulas let $\overline{X} = \{not\, F \mid F \in X\}$. Then, the set of all ground literals over σ is defined as $\mathrm{GAt}(\sigma) \cup \overline{\mathrm{GAt}(\sigma)}$. $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ exhibit particular ground atoms with the meaning true (value 1), false (value 0), undetermined (value $\frac{1}{2}$).

We introduce the following conventions. When $L \subseteq L(\sigma)$ is some sublanguage, L^0 denotes the corresponding set of sentences. If the signature σ does not matter, we omit it and write, e.g., L instead of $L(\sigma)$. Let $L^*(\sigma) = L(\sigma;\ not, \wedge, \vee, \forall, \exists)$, and $PL(\sigma) = L^*(\sigma) \cup \{F \rightarrow G | F, G \in L^*(\sigma)\}$. $PL(\sigma)$ is called the set of program formulas of signature σ. If Y is a partially

ordered set, then $\mathrm{Min}(Y)$ denotes the set of all minimal elements of Y, i.e. $\mathrm{Min}(Y) = \{a \in Y \mid \neg\exists a' \in Y : a' < a\}$. A *Herbrand interpretation* of the language $L(\sigma)$ is one for which the universe equals U_σ, and the function symbols are interpreted canonically.

Definition 1 (Partial Herbrand Interpretation) *Let be $\sigma = \langle Rel, Const, Fun\rangle$ a signature. A partial Herbrand interpretation I of σ is defined as follows $I = (U(\sigma), (f^I)_{f\in Fun}, (r^I)_{r\in Rel})$. Its* universe *$U(\sigma)$ is equal to the set of all ground terms U_σ; its canonical interpretation of ground terms is the identity mapping. The relation symbols $r \in Rel(\sigma)$ are interpreted by functions r^I defined by $r^I : U^{a(r)} \longrightarrow \{0, \frac{1}{2}, 1\}$ for every relation symbol $r \in Rel$, where $a(r)$ denotes the arity of r. Obviously, every Herbrand interpretation is determined by a function $i_I : \mathrm{At}(\sigma) \longrightarrow \{0, \frac{1}{2}, 1\}$.*

A partial Herbrand σ-interpretation I can be represented as a set of ground literals $I \subseteq \mathrm{GAt}(\sigma) \cup \overline{\mathrm{GAt}(\sigma)}$ such that there is no ground atom $a \in \mathrm{GAt}$ satisfying $\{a, not\, a\} \subseteq I$. For a partial Herbrand σ-interpretation I let $Pos(I) = I \cap \mathrm{GAt}$ and $Neg(I) = I \cap \overline{\mathrm{GAt}}$. A partial Herbrand interpretation I is two-valued (or total) if for every $a \in GAt$ holds $\{a, not\, a\} \cap I \neq \emptyset$. I is a partial interpretation over $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ if $\{\mathbf{u}, not\, \mathbf{u}\} \cap I = \emptyset$, and $\{\mathbf{t}, not\, \mathbf{f}\} \subseteq I$.

The class of all partial Herbrand σ-interpretations is denoted by $\boldsymbol{I}_H(\sigma)$. In the sequel we shall also simply say 'interpretation' instead of 'partial Herbrand interpretation'. A *valuation* over an interpretation I is a function ν from the set of all variables *Var* into the Herbrand universe U_σ, which can be naturally extended to arbitrary terms by $\nu(f(t_1, \ldots, t_n)) = f(\nu(t_1), \ldots, \nu(t_n))$. Analogously, a valuation ν can be canonically extended to arbitrary formulas F, where we write $F\nu$ instead of $\nu(F)$. Note that for a constant c, being a 0-ary function, we have $\nu(c) = c$. The model relation $\models \subseteq \boldsymbol{I}_H(\sigma) \times L^0(\sigma)$ between an interpretation and a sentence is defined inductively as follows.

Definition 2 (Model Relation) *Let $I \in \boldsymbol{I}_H(\sigma)$. Then the mapping i_I can be extended to a function $\widetilde{I}$ from the set of all sentences from $PL(\sigma)$ into $\{0, \frac{1}{2}, 1\}$.*

1. $\widetilde{I}\,(a) = i_I(a)$ *for atomic sentences* a.
2. $\widetilde{I}\,(not\, F) = 1 - \widetilde{I}\,(F)$.
3. $\widetilde{I}\,(F \wedge G) = min\{\widetilde{I}\,(F), \widetilde{I}\,(G)\}$.
4. $\widetilde{I}\,(F \vee G) = max\{\widetilde{I}\,(F), \widetilde{I}\,(G)\}$.

5. $\widetilde{I}\,(F \to G) = 1$ *if* $\widetilde{I}\,(F) \leq \widetilde{I}\,(G)$.
6. $\widetilde{I}\,(F \to G) = 0$ *if* $\widetilde{I}\,(F) \not\leq \widetilde{I}\,(G)$.
7. $\widetilde{I}\,(\exists x F(x)) = sup\{\widetilde{I}\,(F(x/t)) \mid t \in U(\sigma)\}$.
8. $\widetilde{I}\,(\forall x F(x)) = inf\{\widetilde{I}\,(F(x/t)) \mid t \in U(\sigma)\}$.

We write $I \models F \iff \widetilde{I}\,(F) = 1$ *for sentences* F *and for arbitrary formulas* F*:*
$I \models F \iff I \models F\nu$ *for all* $\nu : Var \to U_\sigma$. I *is called a model of* F*, and for sets* X *of formulas* $I \models X$ *if and only if for all* $F \in X$ *it holds* $I \models F$. *To simplify the notation we don't distinguish between* I *and* $\widetilde{I}$ *in the following. Two formulas* $F, G \in L(\sigma)$ *are said to be logical equivalent iff for every instantiation* ν *and every partial interpretation* I *the condition* $I(F\nu) = I(G\nu)$ *is satisfied.*

$\mathrm{Mod}_H(X) = \{I \in \boldsymbol{I}_H : I \models X\}$ denotes the Herbrand model operator, and $\models_H$ denotes the corresponding consequence relation, i.e. $X \models_H F$ iff $\mathrm{Mod}_H(X) \subseteq \mathrm{Mod}_H(F)$. In the following we omit the subscript H.

Let $L^{\wedge,\vee}(\mathrm{Lit}(\sigma))$ the smallest subset of $L(\sigma)$ containing the set $\mathrm{Lit}(\sigma)$ and closed with respect to the connectives $\wedge, \vee$.

Proposition 1 *[HJW95] For every formula* $F \in L_1(\sigma)$ *there is a formula* $G \in L^{\wedge,\vee}(\mathrm{Lit}(\sigma))$ *such that* F *and* G *are logical equivalent.*

Definition 3 (Partial Orderings between Interpretations) *Let be* $I, I_1 \in \boldsymbol{I}_H$ *two interpretations. We define the following orderings between* I *and* I_1.

1. *Let* $I \preceq I_1$ *if and only if* $Pos(I) \subseteq Pos(I_1)$ *and* $Neg(I_1) \subseteq Neg(I)$. $\preceq$ *is called the truth-ordering between interpretations, and* I_1 *is said to be a truth-extension (briefly t-extension) of* I.
2. I_1 *is informationally greater or equal to* I *iff* $I \subseteq I_1$. *The partial ordering* $\subseteq$ *between Herbrand interpretations is called information-ordering.* I_1 *is said to be an information-extension (briefly i-extension) of* I.
3. *Let* I, I_1 *be two-valued Herbrand interpretations. Define* $I \leq I_1$ *if and only if* $Pos(I) \subseteq Pos(I_1)$.

Obviously, if I, I_1 *are two-valued models then* $I \preceq I_1$ *iff* $Pos(I) \subseteq Pos(I_1)$.

Proposition 2 *The system* $\mathcal{C} = (\boldsymbol{I}_H, \preceq)$ *of consistent partial interpretations is a complete lattice.*

Proof: Let $\Omega \subseteq \boldsymbol{I}_H$ be an arbitrary non-empty subset. Define $I =_{df} \bigcup\{Pos(K) \mid K \in \Omega\} \cup \bigcap\{Neg(K) \mid K \in \Omega\}$, then I is the least upper bound of Ω, i.e. $I = sup\Omega$. Analogously, the infimum of Ω, denoted by $inf\Omega$ is defined by $inf\Omega = \bigcap\{Pos(K) \mid K \in \Omega\} \cup \bigcup\{Neg(K) \mid K \in \Omega\}$. $\square$

3 Sequents and Programs

Here, we propose to use sequents for the purpose of representing rule knowledge. A sequent, then, is a concrete expression representing some piece of knowledge.

Definition 4 (Sequent) *A* sequent $s = F_1, \ldots, F_m \Rightarrow G_1, \ldots, G_n$ *is an expression where* $F_i, G_j \in L(\sigma, \{\wedge, \vee, not\})$ *for* $i = 1, \ldots, m$ *and* $j = 1, \ldots, n$. *The* body *of* s, *denoted by* $B(s)$, *is given by* $\{F_1, \ldots, F_m\}$, *and the* head *of* s, *denoted by* $H(s)$, *is given by* $\{G_1, \ldots, G_n\}$. $\mathrm{Seq}(\sigma)$ *denotes the class of all sequents* s *such that* $H(s), B(s) \subseteq L(\sigma)$, *and for a given set* $S \subseteq \mathrm{Seq}(\sigma)$, $[S]$ *denotes the set of all ground instances of sequences from* S.

Definition 5 (Model of a Sequent) *Let* $I \in \boldsymbol{I}_H$. *Then we define,* $I \models F_1, \ldots, F_m \Rightarrow G_1, \ldots, G_n$ *iff for all ground substitutions the following condition is satisfied:* $I \models \bigwedge_{i \leq m} F_i\nu \rightarrow \bigvee_{j \leq n} G_j\nu$. I *is said to be a model of* $F_1, \ldots, F_m \Rightarrow G_1, \ldots, G_n$.

We define the following classes of sequents corresponding to non-negative, positive disjunctive, normal, normal disjunctive, and generalized logic programs, respectively.

1. $\mathrm{PLP}(\sigma) = \{s \in \mathrm{Seq}(\sigma) : H(s) \in \mathrm{At}(\sigma),\ B(s) \subseteq \mathrm{At}(\sigma) \cup \{\mathbf{u}, \mathbf{t}, \mathbf{f}\}$.
2. $\mathrm{PDLP}(\sigma) = \{s \in \mathrm{Seq}(\sigma) : B(s), H(s) \subseteq \mathrm{At}(\sigma), H(s) \neq \emptyset\}$.
3. $\mathrm{NLP}(\sigma) = \{s \in \mathrm{Seq}(\sigma) : H(s) \in \mathrm{At}(\sigma),\ B(s) \subseteq \mathrm{Lit}(\sigma)\}$.
4. $\mathrm{NDLP}(\sigma) = \{s \in \mathrm{Seq}(\sigma) : H(s) \subseteq \mathrm{At}(\sigma),\ B(s) \subseteq \mathrm{Lit}(\sigma), H(s) \neq \emptyset\}$.
5. $\mathrm{GLP}(\sigma) = \{s \in \mathrm{Seq}(\sigma) : H(s), B(s) \subseteq L(\sigma;\ not, \wedge, \vee)\}$.

Subsets of PLP are called *non-negative* logic programs, programs associated to PDLP are called *positive disjunctive* logic programs. NLP relates to *normal* logic programs, NDLP to *normal disjunctive* logic programs, and GLP to *generalized* logic programs.

Lemma 3 *1. Let* $J_0 \succeq J_1 \succeq \ldots J_n \succeq \ldots$ *be an infinite t-decreasing sequence of partial interpretations and* $J = inf\{J_n \mid n < \omega\}$. *Let*

$F \in L(\wedge, \vee, not) \cup \{G \rightarrow H \mid F, G \in L(\wedge, \vee, not)\}$. *Then there exists a number* k *such that for all* $s > k$ *the condition* $J(F) = J_s(F)$ *is satisfied.*

2. *Let* $J_0 \preceq J_1 \preceq \ldots J_n \preceq \ldots$ *be an infinite t-increasing sequence of partial interpretations and* $J = sup\{J_n \mid n < \omega\}$. *Let* $F \in L(\wedge, \vee, not) \cup \{G \rightarrow H \mid F, G \in L(\wedge, \vee, not)\}$. *Then there exists a number* k *such that for all* $s > k$ *the condition* $J(F) = J_s(F)$ *is satisfied.*

Let X be an interpretation and $P \subseteq$ GLP. X is said to be upward-consistent with respect to P if there is a model $I \models P$ such that $X \preceq I$.

Proposition 4 *Let* $P \subseteq$ GLP *and* K *an interpretation being upward-consistent with respect to* P. *Let* I *be a model of* P *such that* $K \preceq I$. *Then there exists a model* $J \models P$ *satisfying the following conditions:*

1. $K \preceq J \preceq I$;
2. *for every* $J_1 \in \boldsymbol{I}_H$ *the conditions* $K \preceq J_1 \preceq J$ *and* $J_1 \models P$ *imply* $J = J_1$.

Corollary 5 *Let* $P \subseteq$ GLP. *Every partial model of* P *is an t-extension of a t-minimal partial model and can be t-extended to a t-maximal partial model of* P.

Proposition 6 *Every non-negative logic program has a t-least partial model.*

4 Stationary Generated Models

Definition 6 (Truth Interval of Interpretations) *Let* $I_1, I_2 \in \boldsymbol{I}_H$. *Then,* $[I_1, I_2] = \{I \in \boldsymbol{I}_H : I_1 \preceq I \preceq I_2\}$. *Let* $P \subseteq$ GLP *and let* F *be a sentence. We introduce the following notions.*

- $[I, J](F) = inf\{K(F) | K \in [I, J]\}$
- $P_{[I,J]} = \{r \mid r \in [P]$ *and* $[I, J](B(r)) \geq \frac{1}{2}\}$
- $\overline{P}_{[I,J]} = \{r \mid r \in [P]$ *and* $[I, J](B(r)) = 1\}$

The following notion of a *stationary generated* or *stable generated partial model* is a refinement of the notion of a stable generated (two-valued) model which was introduced in [HW97].

Definition 7 (Stationary Generated Model) *Let be* $P \subseteq$ GLP. *A model* I *of* P *is called* stationary generated *or partial stable generated if there is a sequence* $\{I_\alpha : \alpha < \kappa\}$ *of interpretations satisfying the following conditions:*

1. $I_0 = \overline{GAt}$ *(is the t-least interpretation)*
2. $\alpha < \beta < \kappa$ *implies* $I_\alpha \preceq I_\beta$
3. $sup_{\alpha<\kappa} I_\alpha = I$
4. *For all* $\alpha < \kappa$*:* $I_{\alpha+1} \in Min_{tm}\{J \mid I_\alpha \preceq J \preceq I$ *and (a) for all* $r \in \overline{P}_{[I_\alpha, I]}$ *it holds* $I_{\alpha+1}(H(r)) = 1$ *and (b) for all* $r \in P_{[I_\alpha, I]} : I_{\alpha+1}(H(r)) \geq \frac{1}{2}\}$.
5. $I_\lambda = sup_{\beta<\lambda} I_\beta$ *for every limit ordinal* $\lambda < \kappa$.

We also say that I *is* generated *by the* P-stationary chain $\{I_\alpha \mid \alpha < \kappa\}$.

The set of all stationary generated models of P is denoted by $\mathrm{Mod}_{statg}(P)$. The resp. stationary generated entailment relations are defined as follows: $P \models_{statg} F$ iff $\mathrm{Mod}_{statg}(P) \subseteq \mathrm{Mod}(F)$.

Notice that our definition of stationary generated models also accommodates negation in the head of a rule and nested negations, such as in $p(x) \wedge not\,(q(x) \wedge not\, r(x)) \Rightarrow s(x)$ which would be the result of folding $p(x) \wedge not\, ab(x) \Rightarrow s(x)$ and $q(x) \wedge not\, r(x) \Rightarrow ab(x)$.

We continue this section with the investigations of some fundamental properties of the introduced concepts.

Lemma 7 *Let* $\{I_n \mid n < \omega\}$ *a t-increasing sequence of partial interpretations, i.e.* $I_n \preceq I_{n+1}$ *for all* $n < \omega$*, and let be* $sup\{I_n \mid n < \omega\} = I_\omega$*, and* $I_\omega \preceq I$*. Let* F *be a quantifier free sentence.*

1. *If* $[I_\omega, I](F) \geq \frac{1}{2}$*, then there is a number* $n < \omega$ *such that* $[I_n, I](F) \geq \frac{1}{2}$.
2. *If* $[I_\omega, I](F) = 1$*, then there is a number* $n < \omega$ *such that* $[I_n, I](F) = 1$.

Proposition 8 *Let* $P \subseteq$ GLP *and let* $I \in \mathrm{Mod}_{statg}(P)$ *which is generated by the sequence* $\{I_\alpha : \alpha < \kappa\}$*. Then there is an ordinal* $\beta \leq \omega$ *such that* $I_\beta = I$.

Corollary 9 *If* $P \subseteq$ GLP *and* $I \in \mathrm{Mod}_{statg}(P)$*, then there is either a finite* P*-stationary chain, or a* P*-stationary chain of length* ω*, generating* I.

We now relate the stationary generated models to the stable generated two-valued models as introduced in [HW97]. We recall the definition of [HW97].

Definition 8 (Stable Generated Model) *[HW97] Let* $P \subseteq$ GLP*. A two-valued model* M *of* P *is called* stable generated*, symbolically* $M \in \mathrm{Mod}_{sg}(P)$*, if there is a chain* $\{I_\alpha : \alpha < \omega\}$ *of two-valued Herbrand interpretations such that*

1. $m \leq n$ *implies* $I_m \subseteq I_n$ *and* $I_0 = \emptyset$.
2. I_{n+1} *is a minimal two-valued extension of* I_n *which is contained in* M *and which satisfies all sequents whose body is true in every two-valued interpretation from the set* $\{J \mid I_n \subseteq M\}$.
3. $M = \bigcup\{I_n \mid n < \omega\}$.

We also say that M *is* generated *by the* P-stable chain $\{I_n \mid n < \omega\}$.

Proposition 10 *Let* $P \subseteq$ GLP. *A two-valued model* I *of* P *is stable generated if and only if it is a stationary generated model of* P.

Corollary 11 *Let* $P \subseteq$ GLP. *Then* $\mathrm{Mod}_{sg}(P) \subseteq \mathrm{Mod}_{statg}(P)$.

Example 1 *Let* $S = \{\Rightarrow a, b;\ a \Rightarrow b\}$. *Then* $M = \{a, b\}$ *is not minimal since* $\{b\}$ *is a model of* S. *But* $\{a, b\}$ *is stable:* $I_0 = \emptyset$, $S_{[\emptyset,\{a,b\}]} = \{\Rightarrow a, b\}$; *and since* $\{a\} \in \mathrm{Min}\{I \mid \emptyset \leq I \leq M, I \models a \vee b\}$, *we obtain* $S_{[\{a\},\{a,b\}]} = \{\Rightarrow a, b; a \Rightarrow b\}$. *Obviously,* $\{a, b\}$ *is a minimal extension of* $\{a\}$ *satisfying* $a \vee b$ *and* b.

5 Stationary Generated Models of Normal Logic Programs

The aim of this section is to prove that for normal logic programs the stationary models introduced in [Prz91] coincides with our stationary generated models. To make the paper self-contained we recall the main notions. Let P be a normal logic program, i.e. the rules r have the form: $r := a_1, \ldots, a_m, not\, b_1, \ldots, not\, b_n \Rightarrow c$, where a_i, b_j, c are atomic. Let $I \subseteq B \cup \overline{B}$ be a (consistent) partial interpretation. The transformation $tr_I(r)$ is defined as follows.

- $tr_I(B^+(r)) = B^+(r)$ (positive literals are not changed);
- $tr_I(not\, b_i) = \mathbf{f}$, if $b_i \in I$; $tr_I(not\, b_i) = \mathbf{t}$, if $not\, b_i \in I$; $tr_I(not\, b_i) = \mathbf{u}$, if $\{b_i,\ not\, b_i\} \cap I = \emptyset$. Then $tr_I(B^-(r)) = tr_I(not\, b_1), \ldots, tr_I(not\, b_n)$;
- $tr_I(r) = B^+(r), tr_I(B^-(r)) \Rightarrow H(r)$.

The resulting program P/I which is called the $I-$*reduction* of P is defined by $P/I := \{tr_I(r) \mid r \in [P]\}$. P/I is an example of a so-called non-negative program [Prz91]. A normal logic program P is said to be non-negative, symbolically $P \subseteq$ PLP, if for every rule $r \in P$ the body $B(r)$ of r satisfies the condition $B(r) \subseteq At(\sigma) \cup \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. Every interpretation I contains $\mathbf{t}$ and $not\, \mathbf{f}$, and satisfied $\{\mathbf{u},\ not\, \mathbf{u}\} \cap I = \emptyset$. Every non-negative logic program has a t-least partial model that can be constructed as follows [Prz91].

Definition 9 *Let $P \subseteq$ PLP. The operator $T_P : 2^{\boldsymbol{I}} \to 2^{\boldsymbol{I}}$ is defined as follows.*
$T_P(I) = \{a \mid$ *there is a rule* $B(r) \Rightarrow a \in [P]$ *such that* $I(\bigwedge B(r)) = \mathbf{t}\} \cup$
$\{\, not\, a \mid$ *for every rule* $r \in [P]$ *satisfying* $H(r) = a$ *it is* $I(\bigwedge B(r)) = \mathbf{f}\}$.

The operator T_P is monotonic with respect to the truth-ordering $\preceq$. Since $(2^{\boldsymbol{I}}, \preceq)$ is a complete partial ordering the operator T_P has a least fixpoint I being a model of P. I is defined as follows. Let $I_0 = \{\, not\, a \mid a \in GAt\}$, i.e. I_0 is the t-least interpretation. We define an t-increasing sequence of partial interpretations: $I_0 \preceq I_1 \preceq \ldots \preceq I_n \preceq \ldots$ by $I_{n+1} = T_P(I_n)$. Obviously, $I_n \preceq I_{n+1}$, for $n < \omega$. Then $sup\{I_n : n < \omega\}$ is the least partial model of P; we denote it by $lpm(P)$.

Definition 10 *Let $P \subseteq$ NLP and I a partial interpretation. I is said to be a stationary model of P if and only if $lpm(P/I) = I$.*

Lemma 12 *Let $P \subseteq$ NLP and let I be a stationary model of P with the generating sequence $\{I_n \mid n < \omega\}$, $sup\{I_n : n < \omega\} = I$. Then for every $r \in [P]$, and every $n < \omega$, the following conditions are equivalent:*
1) $I_n(\bigwedge B(tr_I(r))) \geq \mathbf{u}$;
2) for all J satisfying the condition $I_n \preceq J \preceq I$ it holds $J(\bigwedge B(r)) \geq \mathbf{u}$.

Proof: 1) $\to$ 2). Let $r := a_1, \ldots, a_m, not\, b_1, \ldots, not\, b_n \Rightarrow c$, and $tr_I(r) = a_1, \ldots, a_m, v_1, \ldots, v_m \Rightarrow c$, where $v_i \in \{f, u, t\}$. Now we assume, that $I_n(\bigwedge B(tr_I(r))) \geq \mathbf{u}$, then $I_n(a_1 \wedge \ldots \wedge a_m \wedge v_1 \wedge \ldots \wedge v_n) \geq \mathbf{u}$. Then, for every $J : I_n \preceq J \preceq I$ we have $\{a_1, \ldots, a_m\} \cap J = \emptyset$. Assume this is not the case. Then there is a $not\, a_j \in J$, and since $Neg(J) \subseteq Neg(I_n)$ this implies $not\, a_j \in I_n$, hence $I_n(a_j) = f$ which yields $I_n(\bigwedge B(tr_I(r)) = \mathbf{f}$, which is a contradiction. This implies $J(a_i) \geq \mathbf{u}$ for every J: $I_n \preceq J \preceq I$. Furthermore, it holds $J(\, not\, b_j) \geq \mathbf{u}$ for every J: $I_n \preceq J \preceq I$. Assume, there is such an interpretation J satisfying $J(\, not\, b_j) = \mathbf{f}$. Then $b_j \in Pos(J)$ and this implies by $J \preceq I$ the condition $b_j \in Pos(I)$. By definition of the translation tr_I this would imply $tr_I(\, not\, b_j) = \mathbf{f}$, a contradiction to $I_n(\bigwedge B(tr_I(r))) \geq \mathbf{u}$.
2) $\to$ 1). Now we assume, that for all J: $I_n \preceq J \preceq I$: $J(\bigwedge B(r)) \geq \mathbf{u}$ We show that then $I_n(\bigwedge B(tr_I(r))) \geq \mathbf{u}$. Obviously, $I_n(a_1 \wedge \ldots \wedge a_n) \geq u$. It remains to show that $I_n(v_1 \wedge \ldots \wedge v_n) \geq u$. Assume this is not the case, then there is a number $j \leq n$ such that $I_n(v_j) = \mathbf{f}$. This implies $b_j \in I$. But then there is an extension J: $I_n \preceq J \preceq I$ such that $b_j \in Pos(J)$, hence $J(\, not\, b_j) = \mathbf{f}$, and this yields $J(\bigwedge B(r)) = \mathbf{f}$, which is a contradiction. □

We shall show below that the stationary generated models of a normal logic program S agree with the fixpoints of Γ_S, i.e. with stationary models as defined in [Prz91]. Since the definition of the extended Gelfond-Lifschitz transformation requires a specific rule syntax, the definition of stationary models based on it is not very general; as a consequence, Gelfond and Lifschitz are not able to treat negation-as-failure as a standard connective, and to allow for arbitrary formulas in the body of a rule. The interpretation of negation-as-failure according the stationary (generated) semantics seems to be the first general standard logical treatment of non-monotonic logic programs.

Proposition 13 *Let P be a normal logic program and I a stationary model of P. Then I is a stationary generated model of P.*

Proof: By assumption we have $I = lpm(P/I)$, and let $I_0 \preceq I_1 \preceq \ldots \preceq I_n$ the defining t-increasing sequence for I. Then $I = sup\{I_n \mid n < \omega\}$. We show that $\{I_n : n < \omega\}$ is a stationary chain generating I. By definition is $I_0 = \overline{GAt(\sigma)}$. We show that for every $n < \omega$ the interpretation I_{n+1} is a t-minimal extension of I_n satisfying the set $P_{[I_n,I]} = \{r \in [P] \mid$ for all $J : I_n \preceq J \preceq I$ it is $J(\bigwedge B(r)) \geq u\}$. Firstly, we prove for all $r \in P_{[I_n,I]}$ the condition $I_{n+1}(r) = t$. Then we show: if K is a partial interpretation satisfying the condition $I_n \preceq K \preceq I_{n+1}$, and if $K(r) = t$ for all $r \in P_{[I_n,I]}$, then $K = I_{n+1}$.
By definition it is $I_{n+1} = \{a \mid B(r) \Rightarrow a \in tr_I([P])$ and $I_n(\bigwedge B(r)) = t\} \cup \{\, not\, a$: for all $B(r) \Rightarrow a \in tr_I([P])$ it is $I_n(\bigwedge B(r)) = f\}$. Let $r \in P_{[I_n,I]}$, we show that $I_{n+1}(\bigwedge B(r)) \leq I_{n+1}(H(r))$. By lemma 12 it is $I_n(\bigwedge B(r)) \geq u$. If $I_n(\bigwedge B(r) = t$, then $I_{n+1}(H(r)) = t$ (by definition of I_{n+1} and we are ready. Now assume $I_n(\bigwedge B(tr_I(r))) = u$. It is sufficient to show that $I_{n+1}(H(r)) \geq u$. Assume this is not the case, then $I_{n+1}(H(r)) = I_{n+1}(a) = f$, hence $not\, a \in I_{n+1}$. But then $not\, a \in I_n$, hence $I_n(a) = f$. By definition of I_{n+1} for all $B(s) \Rightarrow a \in tr_I([P])$ is $I_n(\bigwedge B(s)) = f$, in particular $I_n(\bigwedge B(tr_I(r))) = f$, this is a contradiction. Hence $I_{n+1} \models P_{[I_n,I]}$. Now let K be satisfy the condition $I_n \preceq K \preceq I_{n+1}$. Obviously, if $K \models P_{[I_n,I]}$, then $Pos(K) \subseteq Pos(I_{n+1})$. It remains to show that $Neg(K) = Neg(I_{n+1})$. Assume this is not the case, then there is an element $not\, a \in Neg(K) - Neg(I_{n+1})$. Then a does not satisfy the condition for $Neg(I_{n+1})$, i.e. there is a rule $B(s) \Rightarrow a \in tr_I([P])$ such that $I_n(\bigwedge B(s)) \geq u$ (o.w. $not\, a \in Neg(I_{n+1})$. Let be $B(s) = tr(B(r))$. Then, by lemma 12 for all $J : I_n \preceq J \preceq I$ we have $J(\bigwedge B(r)) \geq u$, in particular $K(\bigwedge B(r)) \geq u$. Since $K(a) = f$ it follows $K \not\models B(r) \Rightarrow a$. From this follows that $Neg(K) - Neg(I_{n+1}) = \emptyset$, hence $Neg(K) = Neg(I_{n+1})$,

then I_{n+1} satisfies the conditions according to the definition of stationary generated model. □

Proposition 14 *Let I be a stationary generated model of the normal logic program P. Then I is a stationary model of P.*

Proof: Let $\{I_n : n < \omega\}$ be a stationary chain generating I. We show that this sequence coincides with the sequence associated to the least model of $tr_I(P) = P/I$. Let $Q = P/I$. We show that $T_Q(I_n) = I_{n+1}$ for every $n < \omega$, and we have to prove the following conditions:
a) $Pos(T_Q(I_n)) = Pos(I_{n+1})$, and b) $Neg(T_Q(I_n)) = Neg(I_{n+1})$.
a) To show: $Pos(T_Q(I_n)) \subseteq Pos(I_{n+1})$. Let be $a \in Pos(T_Q(I_n))$, then there is a rule $B(r) \Rightarrow a \in tr_I([P])$ such that $I_n(\bigwedge B(r)) = t$. Let $s \in [P]$ the rule satisfying $tr_I(s) = r$, and $B(s) = a_1, \ldots, a_m, \mathit{not}\, b_1, \ldots, \mathit{not}\, b_n$. Then $\{a_1, \ldots, a_m\} \subseteq I_n$. Furthermore, $tr_I(\mathit{not}\, b_j) = t$ for all $j \leq n$. That means $\mathit{not}\, b_j \in I$, and this implies the condition $s \in P_{[I_n,I]}$. Since $I_{n+1} \models s$ and $I_{n+1}(\bigwedge B(s)) = t$ this yields $a \in I_{n+1}$, hence finally $Pos(T_Q(I_n)) \subseteq I_{n+1}$. By induction hypothesis we assume $T_Q(I_{n-1}) = I_n$. Let $a \in Pos(I_{n+1}) - Pos(I_n)$, then there is a rule $B(r) \Rightarrow a \in P_{[I_n,I]}$, i.e. $[I_n, I](\bigwedge B(r)) \geq u$. But then there must be a rule of this kind satisfying $I_n(\bigwedge B(r)) = t$ (otherwise $I_{n+1} - \{a\}$ would be a model $P_{[I_n,I]}$). This shows that $a \in Pos(T_Q(I_n))$.
b) This condition follows immediately from the following claim:
$(*)$: $\mathit{not}\, a \in Neg(I_{n+1})$ iff for all $B(r) \Rightarrow a \in [P]$ it holds $I_n(\bigwedge B(tr_I(r))) = \mathbf{f}$. To prove $(*)$, let $\mathit{not}\, a \in Neg(I_{n+1})$, and assume there is a rule $B(r) \Rightarrow a \in [P]$ such that $I_n(\bigwedge B(tr_I(r))) \geq u$. Then by lemma 12 we have $K(\bigwedge B(r)) \geq u$ for every $K : I_n \preceq K \preceq I$, hence $B(r) \Rightarrow a \in P_{[I_n,I]}$. But then $I_{n+1} \not\models B(r) \Rightarrow a$, because $I_{n+1}(\bigwedge B(r)) \geq u$ and $I_{n+1}(a) = f$. Hence $I_{n+1} \not\models P[I_n, I]$, a contradiction.

Assume for all $r \in tr_I([P])$ with $B(r) \Rightarrow a$ the condition $I_n(\bigwedge B(r)) = f$. We have to show that $\mathit{not}\, a \in Neg(I_{n+1})$. Assume, this is not the case, then $\mathit{not}\, a \notin Neg(I_{n+1})$, then $\mathit{not}\, a \in Neg(I_n) - Neg(I_{n+1})$. From this follows that $I_{n+1} \cup \{\mathit{not}\, a\} \models P_{[I_n,I]}$, which gives a contradiction, because I_{n+1} is a minimal extension of I_n satisfying $P_{[I_n,I]}$. Let $I' = I_{n+1} \cup \{a\}$. We show: for all $r \in P_{[I_n,I]}$ the condition $I'(\bigwedge B(r)) \leq I'(H(r))$. If $H(r) \neq a$, then this is clear. Now let be $B(r) \Rightarrow a \in [P]$ and $tr_I(r) = B(s) \Rightarrow a$. By assumption is $I_n(\bigwedge B(s)) = f$. We show that $I_n(\bigwedge B(r)) = f$ (this is indeed sufficient). Let be $B(s) = a_1, \ldots, a_m, v_1, \ldots, v_n$, $tr_I(\mathit{not}\, b_j) = v_j$. If $I_n(a_i) = f$, then $I_n(\bigwedge B(r)) = f$ and we are ready. Assume $I_n(a_i) \geq u$ for every a_i, $i \leq m$. Then there is a $\mathit{not}\, b_j$ such that $tr_I(\mathit{not}\, b_j) = f$,

which means $b_j \in I$. Then there is an $J : I_n \preceq J \preceq I$ such that $b_j \in J$, hence $J(\bigwedge B(r)) = f$, and this means $r \in P_{[I_n,I]}$. $\square$

6 Conclusion and Future Research

By introducing a new general definition of stationary generated models, we have established the foundation of a theory of *partial models* for generalized logic programs. As a special case we get a model-theoretic interpretation of the well-founded semantics for normal logic programs. The consequence operator for generalized logic programs P - based on stationary generated models - exhibits a form of non-monotonic reasoning which is determined by the following definition: $P \models_{statg} \phi$ iff $Mod_{statg}(P) \subseteq Mod_H(\phi)$, ϕ a quantifier-free sentence. The corresponding closure operator of $\models_{statg}$ is defined by: $C_{statg}(P) = \{\phi | \phi$ quantifier-free and $P \models_{statg} \phi\}$. We believe that only cumulative consequence relations allow the development of a reasonable proof theory. Hence, it is an interesting task to find natural cumulative approximations of C_{statg}. In [HL07] non-monotonic reasoning was successfully applied to the integration problem for ontologies. We will explore the expressive power of generalized logic programs with stationary generated semantics for the representation and processing of knowledge in the field of clinical medicine.

Acknowledgment
Thanks due to the anonymous referees for their criticism and useful comments.

References

[AHP00] J. Alferes, H. Herre, L. M. Pereira. Partial Models of Extended Generalized Logic Programs. Int. Conference Computational Logic 2000, Springer LNAI 1861, 2000, pages 149-163

[AP1996] J. Alferes, L. M. Pereira. Reasoning with Logic Programming. Springer LNAI vol. 111, 1996

[EH99] J. Engelfriet, H. Herre. Stable Generated Models, Partial Temporal Logic and Disjunctive Defaults, *Journal of Logic Programming* 41 (1): 1-25, 1999

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Proc. of ICLP*, pages 1070-1080. MIT Press, 1988

[HJW95] H. Herre, J. Jaspars and G. Wagner. Partial logics with two kinds of negation as a foundation for knowledge-based reasoning, in D. Gabbay and H. Wansing (Eds.), *What is negation ?*, pages 121-159. Kluwer Academic Publishers, 1999

[HW97] H. Herre and G. Wagner. Stable Models Are Generated by a Stable Chain, *Journal of Logic Programming*, 30 (2): 166-177, 1997

[HL07] R. Hoehndorf, F. Loebe, J. Kelso and H. Herre. Representing default knowledge in biomedical ontologies: Application to the integration of anatomy and phenotype ontologies. *BMC Bioinformatics* Vol. 8, pp. 377.1-377.12, 2007

[LTT99] V. Lifschitz, L. R. Tang, H. Turner. Nested Expressions in Logic Programs *Annals of Mathematics and Artificial Intelligence* 1999

[Prz90] T.C. Przymusinski. Well-founded semantics coincides with three-valued stable-semantics. *Fundamenta Informaticae 13* (1990), 445-463

[Prz91] T.C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9:401-424, 1991

[Prz94] T.C. Przymusinsksi. Well-founded and stationary models of logic programs. *Annals of Mathematics and Artificial Intelligence* 12 (1994) 141-187

[Prz96] T.C. Przymusinski. Super Logic Programs and Negation as Belief. In: R. Dyckhoff, H. Herre, P. Schroeder-Heister, editors, *Proc. of the 5th Int. Workshop on Extensions of Logic Programming*, Springer LNAI 1050, 229–236

Constraint-Based Abstraction of a Model Checker for Infinite State Systems*

Gourinath Banda[1] and John P. Gallagher[1,2]

[1] Roskilde University, Denmark
[2] IMDEA Software, Madrid
Email: {gnbanda,jpg}@ruc.dk

Abstract. Abstract interpretation-based model checking provides an approach to verifying properties of infinite-state systems. In practice, most previous work on abstract model checking is either restricted to verifying universal properties, or develops special techniques for temporal logics such as modal transition systems or other dual transition systems. By contrast we apply completely standard techniques for constructing abstract interpretations to the abstraction of a CTL semantic function, without restricting the kind of properties that can be verified. Furthermore we show that this leads directly to implementation of abstract model checking algorithms for abstract domains based on constraints, making use of an SMT solver.

1 Introduction

Model Checking is a widely used technique for verifying properties of reactive systems expressed as formulas in a temporal logic, but is restricted to finite-state systems. Abstraction is an effective technique for handling infinite state spaces, where a finite or infinite number of *original states* are collectively represented with a single *abstract state*.

The theory of *abstract interpretation* formalises abstraction techniques. In abstract interpretation-based analyses, an *abstract domain* is first constructed and then a *Galois connection* between the original (or *concrete*) domain and the abstract domain is defined. Computations over the concrete domain are abstractly interpreted over the abstract domain. Due to the Galois connection, the result from abstract computation will always be an *over approximation* of the actual result, had the actual analysis been possible.

The present work is part of an attempt to develop a uniform CLP-based formal modelling and verification framework for verifying infinite state reactive systems. The modelling part of this framework was covered in [2] where it is shown (i) how to model linear hybrid automata (LHA) specifications in CLP; (ii) how standard program transformation tools of CLP can be applied to extract the underlying state transition system semantics; and (iii) how to abstract the (infinite) reachable state space with the domain of linear constraints. In this extended abstract, we show the verification part of the framework in which abstract interpretation and model checking are integrated.

* Work partly supported by the Danish Natural Science Research Council project *SAFT: Static Analysis Using Finite Tree Automata*.

The contributions of this work are threefold. First, we present the definition of a CTL-semantics function in a suitable form, using only monotonic functions and fixpoint operators. Second, we apply the standard abstract interpretation framework to get a precise abstraction of the CTL-semantics function. We do not construct an abstract transition system, which turns out to be an unnecessary restriction. Finally, we show how a constraint-based abstraction can be directly implemented from the abstract semantic function and show how *satisfiability modulo theories* (SMT)-technology can be exploited to improve the performance of our abstract model checker.

The structure of this paper is as follows. Section 2 introduces the syntax and semantics of CTL, and outlines the theory of abstract interpretation. Section 3 describes abstract model checking, that is, abstract interpretation of the CTL semantics function. Section 4 shows how the framework can be applied in practice to an abstraction based on linear constraints. Section 6 gives some experimental results, and we conclude in Section 8.

2 CTL, Model Checking and Abstract Interpretation

CTL is a formal language used to specify temporal properties. A well formed formula in CTL is constructed from one or more atomic propositions and eight CTL-operators. The syntax of CTL is given below.

Definition 1 (CTL Syntax). *The set of CTL formulas* ϕ *in* negation normal form *is inductively defined by the following grammar:*

$$\begin{aligned}\phi ::= & \mathit{true} \mid p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \phi_1 \leftrightarrow \phi_2 \mid AX\phi \mid EX\phi \mid AF\phi \\ & \mid EF\phi \mid AG\phi \mid EG\phi \mid AU[\phi_1,\phi_2] \mid EU[\phi_1,\phi_2] \mid AR[\phi_1,\phi_2] \mid ER[\phi_1,\phi_2]\end{aligned}$$

where p ranges over a set of atomic formulas $\mathcal{P}$.

A CTL-formula is in negation normal form (NNF) if and only if the negations are placed in front of the atomic propositions. Any formula not in NNF can be transformed into NNF by moving negations inwards using equivalence-preserving transformations.. The *unary operators* AF, AG, AX and the binary operators AU, AR are called *universal CTL operators*; while the unary EF, EG, EX and binary ER, EU operators are called *existential CTL operators*.

2.1 CTL Semantics

The semantics of CTL formulas is defined with respect to a Kripke structure, which is a state transition system whose states are labelled with atomic propositions that are true in that state.

Definition 2 (Kripke structure). *A Kripke structure is a tuple* $\langle S, \Delta, I, L, \mathcal{P}\rangle$ *where S is the set of states,* $\Delta \subseteq S \times S$ *is the transition relation,* $I \subseteq S$ *is the set of initial states,* $\mathcal{P}$ *is the set of propositions and* $L : S \rightarrow 2^{\mathcal{P}}$ *is the labelling function which returns the set of propositions that are true in each state. The set of atomic propositions is closed under negation.*

Given the *Kripke structure* $\langle S, \Delta, I, L, \mathcal{P} \rangle$, the meaning of a formula is the set of states in S where the formula holds; this is itself an abstraction of a more detailed trace-based semantics [9]. We define a function $[\![\cdot]\!] : CTL \to 2^S$ that returns the set of states where the formula holds. This function is called the*CTL-semantics function*.

Definition 3 (CTL-semantics function). *Given a Kripke structure $K = \langle S, \Delta, I, L, \mathcal{P} \rangle$, the semantic function $[\![\cdot]\!] : CTL \to 2^S$ is defined as follows.*

$$
\begin{array}{ll|ll}
[\![true]\!] & = S & [\![false]\!] & = \emptyset \\
[\![p]\!] & = \mathsf{states}(p) & [\![\neg p]\!] & = \mathsf{states}(\neg p) \\
[\![EX\phi]\!] & = \mathsf{pred}_\exists([\![\phi]\!]) & [\![\phi_1 \vee \phi_2]\!] & = [\![\phi_1]\!] \cup [\![\phi_2]\!] \\
[\![AX\phi]\!] & = \mathsf{pred}_\forall([\![\phi]\!]) & [\![\phi_1 \wedge \phi_2]\!] & = [\![\phi_1]\!] \cap [\![\phi_2]\!] \\
[\![EF\phi]\!] & = \mu Z.([\![\phi]\!] \cup \mathsf{pred}_\exists(Z)) & [\![ER[\phi_1, \phi_2]]\!] & = \nu Z.([\![\phi_2]\!] \cap ([\![\phi_1]\!] \cup \mathsf{pred}_\exists(Z)) \\
[\![AF\phi]\!] & = \mu Z.([\![\phi]\!] \cup \mathsf{pred}_\forall(Z)) & [\![AU[\phi_1, \phi_2]]\!] & = \mu Z.([\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathsf{pred}_\forall(Z)) \\
[\![AG\phi]\!] & = \nu Z.([\![\phi]\!] \cap \mathsf{pred}_\forall(Z)) & [\![EU[\phi_1, \phi_2]]\!] & = \mu Z.([\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathsf{pred}_\exists(Z)) \\
[\![EG\phi]\!] & = \nu Z.([\![\phi]\!] \cap \mathsf{pred}_\exists(Z)) & [\![AR[\phi_1, \phi_2]]\!] & = \nu Z.([\![\phi_2]\!] \cap ([\![\phi_1]\!] \cup \mathsf{pred}_\forall(Z))
\end{array}
$$

This semantics function makes use of three subsidiary functions $\mathsf{pred}_\exists : 2^S \to 2^S$, $\mathsf{pred}_\forall : 2^S \to 2^S$ and $\mathsf{states} : \mathcal{P} \to 2^S$ called the *existential predecessor function*, the *universal predecessor function* and *allocating function* respectively. These three functions are specific for a given Kripke structure K and are monotonic.

Definition 4. *Given a Kripke structure $K = \langle S, \Delta, I, L, \mathcal{P} \rangle$ we define functions $\mathsf{pred}_\exists : 2^S \to 2^S$, $\mathsf{pred}_\forall : 2^S \to 2^S$ and $\mathsf{states} : \mathcal{P} \to 2^S$ as follows.*

- $\mathsf{pred}_\exists(S') = \{s \mid \exists s' \in S' : (s, s') \in \Delta\}$ *returns the set of states having at least one of their successors in the set $S' \subseteq S$;*
- $\mathsf{pred}_\forall(S') = \mathsf{pred}_\exists(S') \backslash \mathsf{pred}_\exists(\mathsf{compl}(S'))$ *returns the set of states all of whose successors are in the set $S' \subseteq S$; the function $\mathsf{compl}(X) = S \setminus X$.*
- $\mathsf{states}(p) = \{s \in S \mid p \in L(s)\}$ *returns the set of states where $p \in \mathcal{P}$ holds.*

In the CTL semantic definition, $\mu Z.(F(Z))$ (resp. $\nu Z.(F(Z))$) stands for the least fixed point (resp. greatest fixed point) of the function $\lambda Z.F(Z)$. The Knaster-Tarski fixed point theorem [28] guarantees the existence of least and greatest fixed points for a monotonic function on a complete lattice. All the expressions $F(Z)$ occurring in $\mu Z.(F(Z))$ and $\nu Z.(F(Z))$ in the CTL semantic functions are functions $2^S \to 2^S$ on the complete lattice $\langle 2^S, \subseteq, \cup, \cap, S, \emptyset \rangle$. They are constructed with *monotonic operators* ($\cup$,$\cap$) and monotonic functions ($\mathsf{pred}_\exists, \mathsf{pred}_\forall$). Thus the CTL semantics function is well-defined.

2.2 Model Checking

Model checking is based on checking that the Kripke structure $K = \langle S, \Delta, I, L, \mathcal{P} \rangle$ possesses a property ϕ, written $K \models \phi$. This is defined to be true iff $I \subseteq [\![\phi]\!]$, or equivalently, that $I \cap [\![\neg\phi]\!] = \emptyset$. (Note that $\neg\phi$ should be converted to negation normal form). Thus model-checking requires implementing the CTL-semantics function which in essence is a *fixed point computation*. When the state-space S is finite the greatest and least fixed

point expressions can be evaluated as the limits of Kleene sequences. But when S is infinite, the fixed point computations might not terminate and hence the model checking of infinite state systems becomes undecidable. In this case we try to approximate $[\![.]\!]$ using the theory of *abstract interpretation*.

2.3 Abstract Interpretation

In abstract interpretation we replace the "concrete" semantic function by an abstract semantic function, developed systematically from the concrete semantics with respect to a Galois connection. We present the formal framework briefly.

Definition 5 (Galois Connection). $\langle L, \sqsubseteq_L \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M, \sqsubseteq_M \rangle$ *is a* Galois Connection *between the lattices* $\langle L, \sqsubseteq_L \rangle$ *and* $\langle M, \sqsubseteq_M \rangle$ *if and only if* $\alpha : L \to M$ *and* $\gamma : M \to L$ *are monotonic and* $\forall l \in L, m \in M, \alpha(l) \sqsubseteq_M m \leftrightarrow l \sqsubseteq_L \gamma(m)$.

In abstract interpretation, $\langle L, \sqsubseteq_L \rangle$ and $\langle M, \sqsubseteq_M \rangle$ are the concrete and abstract semantic domains respectively. Given a Galois connection $\langle L, \sqsubseteq_L \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M, \sqsubseteq_M \rangle$ and a monotonic concrete semantics function $f : L \to L$, then we define an abstract semantic function $f^\sharp : M \to M$ such that for all $m \in M$, $(\alpha \circ f \circ \gamma)(m) \sqsubseteq_M f^\sharp(m)$. Furthermore it can be shown that $\mathsf{lfp}(f) \sqsubseteq_L \gamma(\mathsf{lfp}(f^\sharp))$ and that $\mathsf{gfp}(f) \sqsubseteq_L \gamma(\mathsf{gfp}(f^\sharp))$.

Thus the abstract function $f^\sharp$ can be used to compute over-approximations of f, which can be interpreted using the γ function. The case where the abstract semantic function is defined as $f^\sharp = (\alpha \circ f \circ \gamma)$ gives the most precise approximation.

If M is a finite-height lattice then the non-terminating fixed point computations of $\mathsf{lfp}(f)$ and $\mathsf{gfp}(f)$ over L are approximated with a terminating fixed point computation over the finite lattice M.

We next apply this general framework to abstraction of the CTL semantic function, and illustrate with a specific abstraction in Section 4.

3 Abstract Interpretation of the CTL-Semantic function

In this section we consider abstractions based on Galois connections of the form $\langle 2^S, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^A, \subseteq \rangle$, where the abstract domain 2^A consists of sets of abstract states. In fact the abstract domain could be any lattice but for the purposes of this paper we consider such state-based abstractions, which will be further discussed in Section 4.

Definition 6. *Let* $\mathsf{pred}_\exists : 2^S \to 2^S$, $\mathsf{pred}_\forall : 2^S \to 2^S$, *and* $\mathsf{states} : \mathcal{P} \to 2^S$ *be the functions defined in Definition 4 and used in the CTL semantic function. Given a Galois connection* $\langle 2^S, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^A, \subseteq \rangle$, *we define* $\mathsf{apred}_\exists : 2^A \to 2^A$, $\mathsf{apred}_\forall : 2^A \to 2^A$ *and* $\mathsf{astates} : \mathcal{P} \to 2^A$ *as*

$$\mathsf{apred}_\exists = \alpha \circ \mathsf{pred}_\exists \circ \gamma \qquad \mathsf{apred}_\forall = \alpha \circ \mathsf{pred}_\forall \circ \gamma \qquad \mathsf{astates} = \alpha \circ \mathsf{states}$$

It follows directly from the properties of Galois connections that for all $S' \subseteq S$, $\alpha(\mathsf{pred}_\exists(S')) \subseteq \mathsf{apred}_\exists(\alpha(S'))$ and $\alpha(\mathsf{pred}_\forall(S')) \subseteq \mathsf{apred}_\forall(\alpha(S'))$.

Definition 7 (Abstract *CTL* semantics function). *Given a Galois connection* $\langle 2^S, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^A, \subseteq \rangle$*, the abstract CTL semantic function* $[\![\cdot]\!]^a : CTL \to 2^A$ *is defined as follows.*

$$
\begin{array}{llll}
[\![true]\!]^a & = A & [\![false]\!]^a & = \emptyset \\
[\![p]\!]^a & = \mathsf{astates}(p) & [\![\neg p]\!]^a & = \mathsf{astates}(\neg p) \\
[\![EX\phi]\!]^a & = \mathsf{apred}_\exists([\![\phi]\!]^a) & [\![\phi_1 \vee \phi_2]\!]^a & = [\![\phi_1]\!]^a \cup [\![\phi_2]\!]^a \\
[\![AX\phi]\!]^a & = \mathsf{apred}_\forall([\![\phi]\!]^a) & [\![\phi_1 \wedge \phi_2]\!]^a & = [\![\phi_1]\!]^a \cap [\![\phi_2]\!]^a \\
[\![EF\phi]\!]^a & = \mu Z.([\![\phi]\!]^a \cup \mathsf{apred}_\exists(Z)) & [\![ER[\phi_1,\phi_2]]\!]^a & = \nu Z.([\![\phi_2]\!]^a \cap ([\![\phi_1]\!]^a \cup \mathsf{apred}_\exists(Z)) \\
[\![AF\phi]\!]^a & = \mu Z.([\![\phi]\!]^a \cup \mathsf{apred}_\forall(Z)) & [\![AU[\phi_1,\phi_2]]\!]^a & = \mu Z.([\![\phi_2]\!]^a \cup ([\![\phi_1]\!]^a \cap \mathsf{apred}_\forall(Z)) \\
[\![AG\phi]\!]^a & = \nu Z.([\![\phi]\!]^a \cap \mathsf{apred}_\forall(Z)) & [\![EU[\phi_1,\phi_2]]\!]^a & = \mu Z.([\![\phi_2]\!]^a \cup ([\![\phi_1]\!]^a \cap \mathsf{apred}_\exists(Z)) \\
[\![EG\phi]\!]^a & = \nu Z.([\![\phi]\!]^a \cap \mathsf{apred}_\exists(Z)) & [\![AR[\phi_1,\phi_2]]\!]^a & = \nu Z.([\![\phi_2]\!]^a \cap ([\![\phi_1]\!]^a \cup \mathsf{apred}_\forall(Z))
\end{array}
$$

Since all the operators appearing in the abstract CTL-semantic are monotonic, the fixpoint expressions and hence the abstract semantic function is well defined. The following soundness theorem is the basis of our abstract model checking approach.

Theorem 1 (Safety of Abstract CTL Semantics). *Let* $K = \langle S, \Delta, I, L, \mathcal{P} \rangle$ *be a Kripke structure,* $\langle 2^S, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^A, \subseteq \rangle$ *be a Galois connection and* ϕ *any CTL-formula in negation normal form. Then* $\alpha([\![\phi]\!]) \subseteq [\![\phi]\!]^a$ *and* $\gamma([\![\phi]\!]^a) \supseteq [\![\phi]\!]$.

The proof follows from the fact that α is a join-morphism: that is, that $\alpha(S_1 \cup S_2) = \alpha(S_1) \cup \alpha(S_2)$ and the fact that $\alpha(S_1 \cap S_2) \subseteq \alpha(S_1) \cap \alpha(S_2)$.

This theorem provides us with a sound abstract model checking procedure for any CTL formula ϕ. As noted previously, $K \models \phi$ iff $[\![\neg\phi]\!] \cap I = \emptyset$ (where $\neg\phi$ is converted to negation normal form). It follows from Theorem 1 that this follows if $\gamma([\![\neg\phi]\!]^a) \cap I = \emptyset$. Of course, if $\gamma([\![\neg\phi]\!]^a) \cap I \supseteq \emptyset$ nothing can be concluded.

4 Abstract Model Checking in Constraint-based Domains

The abstract semantics given in Section 3 is not always implementable in practice for a given Galois connection $\langle 2^S, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^A, \subseteq \rangle$. In particular, the function γ yields a value in the concrete domain, which is typically an infinite object such as an infinite set. Thus evaluating the functions $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)$ and $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)$ might not be feasible.

In this section we show that the construction is implementable for transition systems and abstract domains expressed using linear constraints.

4.1 Constraint Representation of Transition Systems

We consider the set of linear arithmetic constraints (hereafter simply called constraints) over the real numbers.

$$c ::= t_1 \leq t_2 \mid t_1 < t_2 \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \mid \neg c$$

where t_1, t_2 are linear arithmetic terms built from real constants, variables and the operators $+$, $*$ and $-$. The constraint $t_1 = t_2$ is an abbreviation for $t_1 \leq t_2 \wedge t_2 \leq t_1$. Note

that $\neg(t_1 \leq t_2) \equiv t_2 < t_2$ and $\neg(t_1 < t_2) \equiv t_2 \leq t_2$, and so the negation symbol $\neg$ can be eliminated from constraints if desired by moving negations inwards by Boolean transformations and then applying this equivalence.

A constraint is *satisfied* by an assignment of real numbers to its variables if the constraint evaluates to *true* under this assignment, and is *satisfiable* if there exists some assignment that satisfies it. A constraint can be identified with the set of assignments that satisfy it. Thus a constraint over n real variable represents a set of points in R^n.

A constraint can be projected onto a subset of its variables. Denote by $\mathsf{proj}_V(c)$ the projection of c onto the set of variables X.

Let us consider a transition system defined over the state-space R^n. Let $\bar{x}, \bar{x}_1, \bar{x}_2$ etc. represent n-tuples of distinct variables, and $\bar{r}, \bar{r}_1, \bar{r}_2$ etc. represent tuples of real numbers. Let $\bar{x}/\bar{r}$ represent the assignment of values $\bar{r}$ to the respective variables $\bar{x}$. We consider transition systems in which the transitions can be represented as a finite set of *transition rules* of the form $\bar{x}_1 \xrightarrow{c(\bar{x}_1,\bar{x}_2)} \bar{x}_2$. This represents the set of all transitions from state $\bar{r}_1$ to state $\bar{r}_2$ in which the constraint $c(\bar{x}_1,\bar{x}_2)$ is satisfied by the assignment $\bar{x}_1/\bar{r}_1, \bar{x}_2/\bar{r}_2$. Such transition systems can be used to model real-time control systems [18, 2].

4.2 Computation of the CTL semantic function using constraints

A constraint representation of a transition system allows a constraint solver to be used to compute the functions $\mathsf{pred}_\exists$, $\mathsf{pred}_\forall$ and states in the CTL semantics. Let T be a finite set of transition rules. Let $c'(\bar{y})$ be a constraint over variables $\bar{y}$. It is assumed that the set of propositions in the Kripke structure used in the semantics is the set of linear constraints.

$$\mathsf{pred}_\exists(c'(\bar{y})) = \bigvee\{\mathsf{proj}_{\bar{x}}(c'(\bar{y}) \wedge c(\bar{x},\bar{y})) \mid \bar{x} \xrightarrow{c(\bar{x},\bar{x})} \bar{y} \in T\}$$
$$\mathsf{pred}_\forall(c'(\bar{y})) = \mathsf{pred}_\exists(c'(\bar{y})) \wedge \neg(\mathsf{pred}_\exists(\neg c'(\bar{y})))$$
$$\mathsf{states}(p) = p$$

In the definition of states we use p both as the proposition (the argument of states) and as a set of points (the result).

4.3 Abstract Domains Based on a Disjoint State-Space Partition

Suppose we have a transition system with n state variables; we take as the *concrete domain* the complete lattice $\langle 2^C, \subseteq\rangle$ where $C \subseteq 2^{R^n}$ is some nonempty, possibly infinite set of n-tuples including all the reachable states of the system.

We build an abstraction of the state space based on a disjoint partition of C say $A = \{d_1, \ldots, d_k\}$ such that $\bigcup A = C$. Such a partition can itself be constructed by an abstract interpretation of the transition relation [2]. Define a representation function $\beta : C \to 2^A$, such that $\beta(\bar{x}) = \{d \in A \mid x \in d\}$. We extend the representation function to sets of points, obtaining the abstraction function $\alpha; 2^C \to 2^A$ given by $\alpha(S) = \bigcup\{\beta(x) \mid x \in S\}$. Define the concretisation function $\gamma : 2^A \to 2^C$, as $\gamma(V) = \{x \in C \mid \beta(x) \subseteq V\}$. As shown in [24, 7], $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \subseteq)$ is a Galois connection. Because the partition A is disjoint the value of $\beta(x)$ is a singleton for all x, and the γ function can be written as $\gamma(V) = \bigcup\{\gamma(\{d\}) \mid d \in V\}$.

4.4 Representation of Abstraction Using Constraints

A constraint can be identified with the set of points that satisfies it. Suppose that each element d of the partition A is representable as a linear constraint c_d over the variables $x_1, \ldots x_n$. The β function can be rewritten as $\beta(x) = \{d \mid x \text{ satisfies } c_d\}$. Assuming that we apply α to sets of points represented by a linear constraint over $x_1, \ldots x_n$, we can rewrite the α and γ functions as follows.

$$\alpha(c) = \{d \mid \mathsf{SAT}(c_d \wedge c)\} \qquad \gamma(V) = \bigvee\{c_d \mid d \in V\}$$

4.5 Computation of α and γ functions using constraint solvers

The constraint formulations of the α and γ functions allows them to be effectively computed. The expression $\mathsf{SAT}(c_d \wedge c)$ occurring in the α function means "$(c_d \wedge c)$ is satisfiable" and can be checked by an SMT solver. In our experiments we use the SMT solver Yices [13]. The γ function simply collects a disjunction of the constraints associated with the given set of partitions; no solver is required.

4.6 Implementation of constraint-based abstract semantics

Combining the constraint-based evaluation of the functions $\mathsf{pred}_\exists$ and $\mathsf{pred}_\forall$ with the constraint-based evaluation of the α and γ functions gives us (in principle) a method of computing the abstract semantic counterparts of $\mathsf{pred}_\exists$ and $\mathsf{pred}_\forall$, namely $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)$ and $(\alpha \circ \mathsf{pred}_\forall \circ \gamma)$. This gives us a sound abstract semantics for CTL as discussed previously. The question we now address is the feasibility of this approach. Taken naively, the evaluation of these constraint-based functions (in particular $\mathsf{pred}_\forall$) does not scale up. We now show how we can transform these definitions to a form which can be computed much more efficiently, with the help of an SMT solver.

Consider the evaluation of $(\alpha \circ \mathsf{pred}_\forall \circ \gamma)(V)$ where $V \in 2^A$ is a set of disjoint partitions represented by constraints.

$$\begin{aligned}(\alpha \circ \mathsf{pred}_\forall \circ \gamma)(V) &= (\alpha \circ \mathsf{pred}_\forall)(\bigvee\{c_d \mid d \in V\}) \\ &= \alpha(\mathsf{pred}_\exists(\bigvee\{c_d \mid d \in V\}) \wedge \neg(\mathsf{pred}_\exists(\neg(\bigvee\{c_d \mid d \in V\})))) \\ &= \alpha(\mathsf{pred}_\exists(\bigvee\{c_d \mid d \in V\}) \wedge \neg(\mathsf{pred}_\exists(\bigvee\{c_d \in A \setminus V\}))\end{aligned}$$

In the last step, we use the equivalence $\neg(\bigvee\{c_d \mid d \in V\}) \leftrightarrow \bigvee\{c_d \in A \setminus V\})$, which is justified since the abstract domain A is a disjoint partition of the concrete domain; thus $A \setminus V$ represents the negation of V restricted to the state space of the system. The computation of $\mathsf{pred}_\exists(\bigvee\{c_d \in A \setminus V\})$ is much easier to compute (with available tools) than $\mathsf{pred}_\exists(\neg(\bigvee\{c_d \mid d \in V\}))$. The latter requires the projection operations proj to be applied to complex expressions of the form $\mathsf{proj}_{\bar{x}}(\neg(c_1(\bar{y}) \vee \cdots \vee c_k(\bar{y})) \wedge c(\bar{x}, \bar{y}))$, which involves expanding the expression (to d.n.f. for example); by contrast the former requires evaluation of simpler expressions of the form $\mathsf{proj}_{\bar{x}}(c_d(\bar{y}) \wedge c(\bar{x}, \bar{y}))$.

4.7 Further Optimisation by Pre-Computing Predecessor Constraints

We now show that we can improve the computation of the abstract function $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)$. Let $\{c_i\}$ be a set of constraints, each of which represents a set of points. It can easily seen that $\mathsf{pred}_\exists(\bigvee\{c_i\}) = \bigvee\{\mathsf{pred}_\exists(c_i)\}$. Consider the evaluation of $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)(V)$ where $V \in 2^A$ is a set of disjoint partitions represented by constraints.

$$\begin{aligned}(\alpha \circ \mathsf{pred}_\exists \circ \gamma)(V) &= (\alpha \circ \mathsf{pred}_\exists)(\bigvee\{c_d \mid d \in V\}) \\ &= \alpha(\bigvee\{\mathsf{pred}_\exists(c_d) \mid d \in V\})\end{aligned}$$

Give a finite partition A, we pre-compute the constraint $\mathsf{pred}_\exists(c_d)$ for all $d \in A$. Let $Pre(d)$ be the stored predecessor constraint for partition element d. The results can be stored as a table, and whenever it is required to compute $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)(V)$ where $V \in 2^A$, we simply evaluate $\alpha(\bigvee\{Pre(d) \mid d \in V\})$. The abstraction function α is evaluated efficiently using the SMT solver, as already discussed.

Note that expressions of the form $\alpha(\mathsf{pred}_\exists(\bigvee\{\cdots\}))$ occur in the transformed expression for $(\alpha \circ \mathsf{pred}_\forall \circ \gamma)(V)$ above. The same optimisation can be applied here too. Our experiments show that this usually yields a considerable speedup (2-3 times faster) compared to dynamically computing the $\mathsf{pred}_\exists$ function during model checking.

5 Implementation

The abstract CTL semantic function was implemented directly in Prolog without serious attempt at optimisation of the algorithm. The function $[\![\phi]\!]^a$ yielding a set S is represented by the predicate `absCtl(Phi,S)`, where `Phi` is a suitable representation of a CTL formula. Thus for example, the rule for evaluating a formula $AG\phi$, namely

$$[\![AG\phi]\!]^a = \nu Z.([\![\phi]\!]^a \cap \mathsf{apred}_\forall(Z))$$

is rendered in Prolog by the clauses

```
absCtl(ag(F),States) :-
  absCtl(F,FStates),
  gfp_ag(FStates,States).

gfp_ag(F,S) :-
  gfp('$VAR'('Z'), intersect(F,predForall('$VAR'('Z'))), S).
```

The predicate `gfp(Z,F,S)` computes the greatest fixed point of the function $\lambda Z.(F(Z))$, and is implemented naively as shown below by computing successive iterations $A, F(A)$, $F(F(A)), \ldots$ until $F^j(A) = F^{j+1}(A)$ for some j. Here, A is the set of all abstract regions for the system under consideration. There are improved fixpoint algorithms in the literature which could be applied, e.g. [4].

```
gfp(Z,E, S1) :-
  allStates(S),
  gfp_iteration(S, Z, E, S1).

gfp_iteration(Prev, Z, E, Fix) :-
  apply_arg(Z,Prev,E,E1),
  evalExpr(E1,Next),
  gfp_checkfix(Next,Prev,Z,E,Fix).

gfp_checkfix(E1,Prev,_,Fix) :-
  subset(Prev,E1),
  !,
  returnfixpoint(E1,Fix).
gfp_checkfix(E1,_,Z,E,Fix) :-
  gfp_iteration(E1, Z, E, Fix).

returnfixpoint(X,X).
```

The most relevant aspect of the prototype implementation is the interface to external libraries to perform constraint-solving functions. In implementing the $\mathsf{pred}_\exists$ operation we make use of a Ciao-Prolog interface to the PPL library [1]. In particular, this is used to compute the proj function. The α function is implemented using the SMT solver Yices [13]. We implemented an interface predicate `yicessat(C,Xs)`, where `C` is a constraint and `Xs` is the set of variables in `C`. This predicate simply translates `C` to the syntax of Yices, and succeeds if and only if Yices finds that the constraint is satisfiable. Using this predicate the definition of α, that is $\alpha(c) = \{d \mid \mathsf{SAT}(c_d \wedge c)\}$ can be implemented directly as defined.

6 Experiments Using an SMT Constraint Solver

Figure 1 shows the transitions of a water-level controller taken from [18]. The transitions are represented as constraint logic program clauses generated automatically from a Linear Hybrid Automaton specification of the controller, as explained in detail in [2]. The state variables in an atomic formula of form `rState(X,W,T,L)` represent the rate of flow (`X`), the water-level (`W`), the elapsed time (`T`) and the location identifier (`L`). The meaning of a clause of form

```
rState(X,W,T,L) :- c(X,W,T,L,X1,W1,T1,L1), rState(X1,W1,T1,L1)
```

is a transition rule $(\mathtt{X1},\mathtt{W1},\mathtt{T1},\mathtt{L1}) \xrightarrow{c(\mathtt{X},\mathtt{W},\mathtt{T},\mathtt{L},\mathtt{X1},\mathtt{W1},\mathtt{T1},\mathtt{L1})} (\mathtt{X},\mathtt{W},\mathtt{T},\mathtt{L})$. The initial state is given by the clause `rState(0,1,1)`. Note that there are transitions both from one location to another, and also from a location to itself, since the controller can remain in a location so long as an invariant is satisfied.

Figure 2 shows the result of an analysis of the reachable states of the system, based on computing an approximation of the minimal model of the constraint program in Figure 1. There are 8 *regions*, which cover the reachable states of the controller starting in the initial state (which is region 1). The term `v(N, rState(A,B,C,D),[])` means

```
rState(0,1,1).
rState(A,B,C,1) :-A<D, B = E+A-D, C = F+A-D, 10<
                 rState(D,E,F,1).
rState(0,A,B,1) :-D<E, E = F-2*(D-C), G = H+D-C, G=2, B = G, A=E,
                 rState(C,F,H,4).
rState(A,B,C,2) :-A<D, B = E+A-D, C = F+A-D, 2<
                 rState(D,E,F,2).
rState(0,A,B,2) :-D<E, E = F+D-C, = G+D-C, E=10, B = 0, A=E,
                 rState(C,F,G,1).
rState(A,B,C,3) :-A<D, B = E-2*(A-D), C = F+A-D,5<B>
                 rState(D,E,F,3).
rState(0,A,B,3) :-D<E, E = F+D-C, G = H+D-C, G=2, B = G, A=E,
                 rState(C,F,H,2).
rState(A,B,C,4) :-A<D, B = E-2*(A-D), C = F+A-D,2<C<
                 rState(D,E,F,4).
rState(0,A,B,4) :-D<E, E = F-2*(D-C), = G+D-C, E=5, B = 0, A=E,
                 rState(C,F,G,3).
```

Fig. 1. The Water-Level Controller

```
v(1, rState(A,B,C,D), [1*A=0,1*B=1,D=1]).
v(2, rState(A,B,C,D), [-1*B> -10,1*B>1,1*A+ -1*B= -1,D=1]).
v(3, rState(A,B,C,D), [1*B=10,1*A=0,1*C=0,D=2]).
v(4, rState(A,B,C,D), [-1*C> -2,1*C>0,1*A+ -1*C=0,1*B+ -1*C=10,D=2]).
v(5, rState(A,B,C,D), [1*B=12,1*A=0,1*C=2,D=3]).
v(6, rState(A,B,C,D), [-2*C> -11,1*C>2,1*A+ -1*C= -2,1*B+2*C=16,D=3]).
v(7, rState(A,B,C,D), [1*B=5,1*A=0,1*C=0,D=4]).
v(8, rState(A,B,C,D), [-1*C> -2,1*C>0,1*A+ -1*C=0,1*B+2*C=5,D=4]).
```

Fig. 2. Disjoint Regions of the Water-Level Controller

that the region labelled `N` is defined by the constraint in the third argument, with constraint variables `A,B,C,D` corresponding to the given state variables. The 8 regions are disjoint. We use this partition to construct the abstract domain as described in Section 4.3.

Our implementation of the abstract semantics function is in Ciao-Prolog with external interfaces to the Parma Polyhedra Library [1] and the Yices SMT solver [13]. Our prototype implementation of the fixpoint computations is very naive. Nonetheless we successfully checked many CTL formulas including those with CTL operators nested in various ways, which in general is not allowed in either UPPAAL [3] or HyTech [19].

Table 1 gives the results of abstract model checking two systems, namely, a water level monitor and a task scheduler. Both of these systems are taken from [18]. In the table: (i) the columns *System* and *Formula* indicate the system and the formula being checked; (ii) the columns A and Δ, respectively, indicate the number of abstract regions and original transitions in a system and (iii) the column *time* indicates the computation

time to check a formula on the computer with an Intel XEON CPU running at 2.66GHz and with 4GB RAM.

6.1 Water level controller

For the water level system, which has 4 state variables, no formula that we have tried to evaluate takes longer than 0.2 seconds to check. Since verifying certain properties requires finer abstractions (as discussed shortly in Section 6.3), we consider two variants, a coarse one with eight and a more refined abstraction with twelve abstract regions.

The formula $EF(W = 10)$ means "there exists a path along which eventually the water level (W) reaches 10", while $AG(W = 10 \rightarrow EF(W \neq 10))$ means "on every path it is possible for the water level not to get stuck at 10". The formula $EF(AG(min \leq W \leq max))$ where $min, max \in R$ states "possibly the water level stabilises and fluctuates between a minimum *min* and maximum *max*". Using this formula, we can check whether the system reaches a stable region with the given bounds on the water level.

6.2 Scheduler

For the scheduler system that has 8 state variables, 42 abstract regions and 12 transitions, the checking time increases. Here, formulas can take up to 5-6 seconds to check in our prototype implementation. We proved a number of safety and liveness properties, again successfully checking properties of a form beyond the capability of other model checkers. For example the formula $AG(K2 > 0 \rightarrow EF(K2 = 0))$, containing an EF nested within an AG, means that the tasks of high priority (whose presence is indicated by a strictly positive value of $K2$) do not get starved (that is, the value of $K2$ eventually returns to zero).

6.3 Increasing precision by property-specific refinements

The property $EF(W = 3)$ in the water level controller should hold on the system. But this formula cannot be verified when the state space is abstracted with 8 abstract regions. Because of the coarse abstraction, we cannot distinguish $W = 3$ from $W \neq 3$. The negation of the formula, namely $AG(W > 3 \vee W < 3)$, holds in the abstract initial state since there are infinite paths from region 1 which always stay in regions that are consistent with $W \neq 3$.

One approach to solving such cases is to make a property-specific refinement to the abstraction. Each region is split into three regions by adding $W = 3$, $W > 3$ and $W < 3$ respectively to each region. Consequently, since there are 8 regions in the current abstraction (shown in 2), we get a new abstraction with 24 abstract regions, of which only 12 are satisfiable. Only the satisfiable regions need to be retained, giving a total of 12 regions in this example. With this refined abstraction, the property $EF(W = 3)$ can then successfully be checked.

System	*Formula*	A	Δ	*Time (secs.)*
Waterlevel Monitor	$EF(W = 10)$	8	8	0.08
	$AG(W = 10 \rightarrow EF(W \neq 10))$	8	8	0.06
	$EF(AG(1 \leq W \leq 12))$	8	8	0.04
	$AG(W \geq 1))$	8	8	0.01
	$EF(W = 3)$	12	8	0.16
Task Scheduler	$EF(K2 = 1)$	42	14	5.51
	$AG(K2 > 0 \rightarrow EF(K2 = 0))$	42	14	3.49
	$AG(K2 \leq 1)$	42	14	3.49

Table 1. Experimental Results

6.4 Limitations implied by our modelling technique

We cannot always successfully check formulas of the form $AF\phi$, due to an abstraction introduced into our model of continuous behaviour (rather than the abstraction induced by the Galois connection). The reason for this is that the transitions of the system in general include additional self-transitions that were not intended in the original system. Such transitions with the *same location* for the successor state as well as predecessor state are those which do not respect the continuity of the physical system. For example, the transition rules of the water level controller allow a transition within location 1 directly from $W = 1$ to $W = 5$ without passing through $W = 3$. When trying to prove a formula of the form $AF\phi$, we need to refute the formula $\neg(AF\phi)$ i.e. $EG\neg\phi$. Because of the extra self transitions, there might exist a path from the initial state on which $\neg\phi$ holds forever. Thus refutation might not be possible. Other modelling techniques are needed to capture continuity.

7 Related Work

The topic of model-checking infinite state systems using some form of abstraction has been already widely studied. Abstract model checking is described by Clarke *et al.* [6]. In this approach a state-based abstraction is defined where an abstract state is a set of concrete states. A state abstraction together with a concrete transition relation Δ induces an *abstract transition relation* Δ_{abs}. Specifically, if X_1, X_2 are abstract states, $(X_1, X_2) \in \Delta_{abs}$ iff $\exists x_1 \in X_1, x_2 \in X_2$ such that $(x_1, x_2) \in \Delta$. From this basis an abstract Kripke structure can be built; the initial states of the abstract Kripke structure are the abstract states that contain a concrete initial state, and the property labelling function of the abstract Kripke structure is induced straightforwardly as well. Model checking over the abstract Kripke structure is correct for *universal* temporal formulas (ACTL), that is, formulas that do not contain operators EX, EF, EG or EU. Intuitively, the set of paths in the abstract Kripke structure represents a superset of the paths of the concrete Kripke structure. Hence, any property that holds for all paths of the abstract Kripke structure also holds in the concrete structure. If there is a finite number of abstract states, then the abstract transition relation is also finite and thus a standard (finite-state) model checker can be used to perform model-checking of ACTL properties. However,

if an ACTL property does not hold in the abstract structure, nothing can be concluded about the concrete structure, and furthermore checking properties containing existential path quantifiers is not sound in such an approach.

This technique for abstract model checking can be reproduced in our approach, although we do not explicitly use an abstract Kripke structure. Checking an ACTL formula is done by negating the formula and transforming it to negation normal form, yielding an *existential* temporal formula (ECTL formula). Checking such a formula using our semantic function makes use of the $\mathsf{pred}_\exists$ function but not the $\mathsf{pred}_\forall$ function. It can be shown that the composition $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)$ gives the $\mathsf{pred}_\exists$ function for the abstract transition relation defined by Clarke *et al.* Note that whereas abstract model checking the ACTL formula with an abstract Kripke structure yields an under-approximation of the set of states where the formula holds, our approach yields the complement, namely an over-approximation of the set of states where the negation of the formula holds.

There have been different techniques proposed in order to overcome the restriction to ACTL formulas. Dams *et al.* [10] present a framework for constructing abstract interpretations for transition systems. This involves constructing a *mixed transition system* containing two kinds of transition relations, the so-called free and constrained transitions. Godefroid *et al.* [16] proposed the use of *modal transition systems* [22] which consist of two components, namely *must*-transitions and *may*-transitions. In both [10] and [16], given a state abstraction together with a concrete transition system, a mixed transition system, or an (abstract) modal transition system respectively, is automatically generated. Following this, a modified model-checking algorithm is defined in which any formula can be checked with respect to the dual transition relations. There are certainly similarities between these approaches and ours, though more study of the precise relationship is needed. The *may*-transitions are captured by the abstract transitions defined by Clarke *et al.* [6] and hence by our abstract function $(\alpha \circ \mathsf{pred}_\exists \circ \gamma)$, as discussed above. We conjecture that the *must*-transitions are closely related to the abstract function $(\alpha \circ \mathsf{pred}_\forall \circ \gamma)$. We argue that the construction of abstract transition systems, and the consequent need to define different transitions preserving universal and existential properties, is an avoidable complication, and that our approach is conceptually simpler. Probably the main motivation for the definition of abstract transition systems is to re-use existing model checkers, as remarked by Cousot and Cousot [9].

The application of the theory of abstract interpretation to temporal logic, including abstract model checking, is thoroughly discussed by Cousot and Cousot [8, 9]. Our abstract semantics is inspired by these works, in that we also proceed by direct abstraction of a concrete semantic function using a Galois connection, without constructing any abstract transition relations. The technique of constructing abstract functions based on the pattern $(\alpha \circ f \circ \gamma)$, while completely standard in abstract interpretation [7], is not discussed explicitly in the temporal logic context. We focus only on state-based abstractions (Section 9 of [9]) and we ignore abstraction of traces. Our contribution compared to these works is to work out the abstract semantics for a specific class of constraint-based abstractions, and point the way to effective abstract model checking implementations using SMT solvers. Kelb [21] develops a related abstract model checking algo-

rithm based on abstraction of universal and existential predecessor functions which are essentially the same as our $\mathsf{pred}_\forall$ and $\mathsf{pred}_\exists$ functions.

Giacobazzi and Quintarelli [15] discuss abstraction of temporal logic and their refinement, but deal only with checking universal properties.

Our technique for modelling and verifying real time and concurrent systems using constraint logic programs builds on the work of a number of other authors, including Gupta and Pontelli [17], Jaffar *et al.* [20] and Delzanno and Podelski [11]. However we take a different direction from them in our approach to abstraction and checking of CTL formulas, in that we use abstract CLP program semantics when abstracting the state space (only briefly covered in the present work), but then apply this abstraction in a temporal logic framework, which is the topic of this work. Other authors have encoded both the transition systems and CTL semantics as constraint logic programs [5, 23, 25, 12, 14, 26, 27]. However none of these develops a comprehensive approach to abstract semantics when dealing with infinite-state systems. Perhaps a unified CLP-based approach to abstract CTL semantics could be constructed based on these works.

8 Conclusion

We have demonstrated a practical approach to abstract model checking, by constructing an abstract semantic function for CTL based on a Galois connection. Most previous work on abstract model checking is restricted to verifying *universal properties* and requires the construction of an abstract transition system. In other approaches in which arbitrary properties can be checked [16, 10], a dual abstract transition system is constructed. Like Cousot and Cousot [9] we do not find it necessary to construct any abstract transition system, but abstract the concrete semantic function systematically. Using abstract domains based on constraints we are able to implement the semantics directly. The use of an SMT solver adds greatly to the effectiveness of the approach.

Acknowledgements. We gratefully acknowledge discussions with Dennis Dams and César Sánchez.

References

1. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In M. V. Hermenegildo and G. Puebla, editors, *SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2002.
2. G. Banda and J. P. Gallagher. Analysis of Linear Hybrid Systems in CLP. In M. Hanus, editor, *LOPSTR 2008*, volume 5438 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2009.
3. G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *SFM-RT 2004*, number 3185 in Lecture Notes in Computer Science, pages 200–236. Springer, September 2004.
4. A. Browne, E. M. Clarke, S. Jha, D. E. Long, and W. R. Marrero. An improved algorithm for the evaluation of fixpoint expressions. *Theor. Comput. Sci.*, 178(1-2):237–255, 1997.
5. C. Brzoska. Temporal logic programming in dense time. In *ILPS*, pages 303–317. MIT Press, 1995.

6. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
7. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL'79*, pages 269–282. ACM Press, New York, U.S.A., 1979.
8. P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Autom. Softw. Eng.*, 6(1):69–95, 1999.
9. P. Cousot and R. Cousot. Temporal abstract interpretation. In *POPL'2000*, pages 12–25, 2000.
10. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.
11. G. Delzanno and A. Podelski. Model checking in CLP. In *TACAS*, pages 223–239, 1999.
12. X. Du, C. R. Ramakrishnan, and S. A. Smolka. Real-time verification techniques for untimed systems. *Electr. Notes Theor. Comput. Sci.*, 39(3), 2000.
13. B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In T. Ball and R. B. Jones, editors, *CAV 2006*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.
14. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite-state systems by specializing constraint logic programs. In M. Leuschel, A. Podelski, C. Ramakrishnan, and U. Ultes-Nitsche, editors, *Proceedings of the Second International Workshop on Verification and Computational Logic (VCL'2001)*, pages 85–96. Tech. Report DSSE-TR-2001-3, University of Southampton, 2001.
15. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples, and refinements in abstract model-checking. In P. Cousot, editor, *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*, volume 2126 of *Lecture Notes in Computer Science*, pages 356–373. Springer, 2001.
16. P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In K. G. Larsen and M. Nielsen, editors, *CONCUR 2001*, volume 2154 of *Lecture Notes in Computer Science*, pages 426–440. Springer, 2001.
17. G. Gupta and E. Pontelli. A constraint-based approach for specification and verification of real-time systems. In *IEEE Real-Time Systems Symposium*, pages 230–239, 1997.
18. N. Halbwachs, Y. E. Proy, and P. Raymound. Verification of linear hybrid systems by means of convex approximations. In *SAS'94*, volume 864 of *Lecture Notes in Computer Science*, pages 223–237, 1994.
19. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 460–463. Springer, 1997.
20. J. Jaffar, A. E. Santosa, and R. Voicu. A CLP proof method for timed automata. In J. Anderson and J. Sztipanovits, editors, *The 25th IEEE International Real-Time Systems Symposium*, pages 175–186. IEEE Computer Society, 2004.
21. P. Kelb. Model checking and abstraction: A framework preserving both truth and failure information. Technical report, Carl yon Ossietzky Univ. of Oldenburg, Oldenburg, Germany, 1994.
22. K. G. Larsen and B. Thomsen. A modal process logic. In *Proceedings, Third Annual Symposium on Logic in Computer Science, 5-8 July 1988, Edinburgh, Scotland, UK*, pages 203–210. IEEE Computer Society, 1988.
23. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'99)*, volume 1817 of *Springer-Verlag Lecture Notes in Computer Science*, pages 63–82, April 2000.
24. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

25. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In *Computational Logic*, volume 1861 of *LNCS*, pages 384–398, 2000.
26. G. Pemmasani, C. R. Ramakrishnan, and I. V. Ramakrishnan. Efficient real-time model checking using tabled logic programming and constraints. In *ICLP*, volume 2401 of *Lecture Notes in Computer Science*, pages 100–114, 2002.
27. J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Tranformation, 12th International Workshop, LOPSTR 2002, Madrid, Spain, September 17-20,2002, Revised Selected Papers*, pages 90–108, 2002.
28. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

Range Restriction for General Formulas

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
brass@informatik.uni-halle.de

Abstract. Deductive databases need general formulas in rule bodies, not only conjuctions of literals. This is well known since the work of Lloyd and Topor about extended logic programming. Of course, formulas must be restricted in such a way that they can be effectively evaluated in finite time, and produce only a finite number of new tuples (in each iteration of the T_P-operator: the fixpoint can still be infinite). It is also necessary to respect binding restrictions of built-in predicates: many of these predicates can be executed only when certain arguments are ground. Whereas for standard logic programming rules, questions of safety, allowedness, and range-restriction are relatively easy and well understood, the situation for general formulas is a bit more complicated. We give a syntactic analysis of formulas that guarantees the necessary properties.

1 Introduction

Deductive databases have not yet been very successful in practice (at least in terms of market share), although their basic idea is practically very important: Deductive databases aim at an integrated system of database and programming language that is based on the declarative paradigm which was so successful in database languages. Currently, database programming is typically done in languages like PHP or Java. The programs construct SQL statements, send them to the database server, fetch the results, and process them. The interface is not very smooth, and although the situation can be improved with specific database languages like PL/SQL and server-side procedures / user-defined functions within the DBMS, the language paradigms remain different. Object-oriented databases were one approach to develop an integrated system based on a single paradigm, but there the declarativity of the database query part was sacrificed, and they did not get a significant market share, too. Nevertheless, there is an obvious demand for integrated database/programming systems, and this demand has even grown because of object-relational features that need programming inside the database server, and because of web and XML applications.

As far as we know, the deductive database prototypes developed so far support only a Datalog variant, and do not support SQL. But SQL is a database standard, and many practical programmers are trained in the SQL language. It would certainly be helpful for the migration of people and projects to deductive technology, if a deductive system can first be used like a standard SQL DBMS,

and only when one wants to use advanced features like recursive views, powerful constraints, or stored procedures, one has to learn some form of Datalog.

However, supporting SQL in a deductive DBMS is not simply a matter of hiring a good programmer — it still needs research. A requirement is of course that everything that can be done in SQL can also be done in the deductive language (so that e.g., SQL queries can be translated internally to the deductive language, and then executed). But standard Datalog lacks some SQL features that are important in practice.

One obvious difference between SQL and Datalog is that SQL permits general formulas. Already Lloyd and Topor recommended that general formulas should be allowed in rule bodies [LT84,LT85,Llo87], and developed a transformation from these extended logic programming rules to standard rules (see also [LC05]). Although this transformation is an important yardstick to which newer solutions must be compared, it does not lead to a very efficient query evaluation. In case of disjunctive conditions, rules are split, which might lead in the worst case to exponentially many rules, and even in normal cases computation is duplicated. Therefore, it is worth to consider direct support for general formulas in rules. If a deductive database system should be successful, it must have performance for SQL queries that is more or less comparable to standard DBMS. Splitting complex conditions into many rules is not advantageous for that purpose.

In another paper we investigated how duplicates as in SQL can be supported in extended Datalog rules [Bra09]. There the duplicates that Prolog would generate after the Lloyd/Topor transformation from a rule did not agree with the duplicates from a very similar SQL query. This, too, shows that the Lloyd/Topor transfomation does not solve all problems with regard to general formulas in rule bodies.

In this paper we investigate a basic problem that every concrete deductive database system must solve: At least each iteration of the T_P-operator used in bottom-up evaluation must be effectively computable. It might be that the iteration does not terminate, but this is a quite different problem.

Furthermore, a concrete system has a lot of built-in predicates, for instance $<$ is essential for many database queries. In standard database systems (not applying a constraint solver), we must ensure that in a call to $<$ both arguments are ground. Since derived predicates can be called with different binding patterns, the system must be able to automatically find a possible evaluation sequence. This is not difficult for standard rules, where only the body literals must be reordered, but it is technically a bit more complicated for general rules.

2 Standard Rules

Let us first quickly repeat the usual solution to range-restriction in the case of standard rules with conjunctions of literals (positive and negative) in the body. The predicates are classified into

– EDB-predicates ("extensional database"), the given database relations: Of course, these predicates have finite extensions, i.e. they are a finite set of tuples.
– IDB-predicates ("intensional database"), the predicates defined by rules: Only these predicates can appear in the rule heads. They have finite extensions in each step of the fixpoint iteration with the T_P-operator (computing immediate consequences of rules). If the iteration does not terminate, the extension might be infinite in the minimal model, but that is a different problem (not subject of this paper).
– Built-in predicates like "<", which are defined by program code inside the DBMS. These predicates can have infinite extensions, i.e. they might be true for an infinite number of argument values.

In bottom-up evaluation, which is the basis of deductive databases, all rules must be range-restricted. In the most basic case, this means that every variable that is used in a rule must appear at least once in a positive body literal with an EDB or IDB predicate (not a built-in predicate with an infinite extension). In this way, every variable is first bound to a finite set of possible values, and then negative body literals and literals with built-in predicates like "<" can be evaluated. This also ensures that every single rule application produces only finitely many result tuples, containing only values that appeared in the finite database relations.

However, these restrictions are very severe. For instance, they do not permit to give names to arbitrary subformulas, e.g. the following would not be range-restricted:

```
lt(X, Y) :- X < Y.
```

Note that such a view also cannot be defined in SQL databases. However, deductive databases must offer features that go beyond SQL, and since derived predicates are such an important construct in deductive databases, a much stronger support can be expected.

The next step in the development was to adapt the definition of range-restriction to the magic-set transformation. There, predicates are assigned binding patterns, which define which arguments are bound (given inputs), and which are free (searched outputs) when a predicate is called. For instance, the above rule would be legal when `lt` has only the binding pattern `bb` (both arguments are bound).

Definition 1 (Binding Pattern). *A binding pattern β for a predicate of arity n is a string over the alphabet* $\{\mathtt{b},\mathtt{f}\}$ *of length n.*

A predicate with more interesting binding patterns is `sum`, where `sum(X,Y,Z)` means `X+Y=Z`. This predicate supports binding patterns `bbf`, `bfb`, `fbb` and `bbb`. I.e. given two arguments, the third can be computed.

In the following, we assume that terms are constants or variables. To some degree, function symbols (term constructors) can be replaced by built-in predicates. Consider for example the standard list append predicate:

```
append([], L, L).
append([F|R], L, [F|RL]) :- append(R, L, RL).
```

By introducing a new variable for each composed term, and using a predicate `cons(X,Y,Z)` $\equiv$ `Z=[X|Y]`, one gets the following definition of `append` without structured terms:

```
append([], L, L).
append(X, L, Y) :- cons(F,R,X), append(R,L,RL), cons(F,RL,Y).
```

The predicate `cons` supports the binding patterns `bbf` (list construction), and `ffb` (splitting a list). Of course, structured terms are a useful and compact notational convenience on the user level. However, internally, there terms can be flattened as shown in the example. The advantage is that terms with evaluable functions like `+` and terms with record constructors like `[_|_]` can be handled in the same framework. For deductive databases this is important since SQL programmers are used to terms with the standard arithmetic operators. The disadvantage of this solution is that data structures with variables in them cannot be handled conveniently. While there are nice applications of such terms in logic programming, they are very uncommon in database applications (and anyway cannot be stored in classical databases).

Definition 2 (Binding Pattern Specification, Valid Binding Patterns). *A binding pattern specification is a mapping* bp *which defines for each predicate* p*, a set of binding patterns* $\mathsf{bp}(\mathsf{p}) \neq \emptyset$*, called the valid binding patterns for this predicate. If* A *is an atomic formula with predicate* p*, we permit to write* $\mathsf{bp}(\mathsf{A})$ *for* $\mathsf{bp}(\mathsf{p})$*.*

Definition 3 (Allowed Interpretation). *Given a binding pattern specification* bp*, an interpretation* $\mathcal{I}$ *is called allowed if it satisfies the binding pattern restrictions of* bp *in the following sense:*

- *Let* n *be the arity of* p *and* $1 \leq i_1 < \cdots < i_k \leq n$ *be the index positions with* $\beta(i_j) = \mathsf{b}$*.*
- *Then for all values* $c_1, \ldots, c_k$ *from the domain of* $\mathcal{I}$*, the following set is finite:*

$$\{(d_1, \ldots, d_n) \in \mathcal{I}[\![\mathsf{p}]\!] \mid d_{i_1} = c_1,\ \ldots,\ d_{i_k} = c_k\}$$

- *Furthermore, it is possible to effectively compute this set for given* $c_1, \ldots, c_k$*.*

For built-in predicates, the valid binding patterns correspond to the implemented variants of a predicate, e.g. $\mathsf{bp}(\texttt{sum}) = \{\mathsf{bbf}, \mathsf{bfb}, \mathsf{fbb}\}$. It is possible, but not necessary to add `bbb`. E.g. if one has an implementation for `bbf`, one can execute `sum(1,2,3)` like `sum(1,2,X)` $\wedge$ `X=3`. Thus, a binding pattern β is more general than a binding pattern β' iff they have the same length n, and $\beta_i = \mathsf{b}$ implies $\beta'_i = \mathsf{b}$ for $i = 1, \ldots, n$.

For standard EDB predicates, it suffices to have one binding pattern $\mathsf{ff}\ldots\mathsf{f}$. This corresponds to the "full table scan". If there are indexes, other binding patterns might become interesting.

For user-defined predicates, mode declarations or a program analysis defines the valid binding patterns.

Now let $\mathsf{vars}(\mathsf{t})$ be the set of variables that appears in term t. Since terms are only constants or variables, the set is a singleton or empty. For a formula φ, we write $\mathsf{vars}(\varphi)$ for the free variables in that formula.

The following notion of "input variables" for a literal is helpful to define range-restriction:

Definition 4 (Input Variables). *Given an atomic formula* $\mathsf{A} = \mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$ *and a binding pattern* $\beta = \beta_1 \ldots \beta_n$ *for* p*, the set of input variables of* A *with respect to* β *is*

$$\mathsf{input}(\mathsf{A}, \beta) := \bigcup \{\mathsf{vars}(\mathsf{t}_i) \mid 1 \leq i \leq n,\ \beta_i = \mathsf{b}\}$$

(i.e. all variables that appear in bound arguments).

Input variables in body literals must be bound before the literal can be called. Input variables in head literals are bound when the rule is executed. Now range-restriction for standard rules can be defined as follows:

Definition 5 (Range-Restricted Standard Rule). *A rule*

$$\mathsf{A} \leftarrow \mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n \wedge \neg \mathsf{B}_{n+1} \wedge \cdots \wedge \neg \mathsf{B}_m$$

is range-restricted given a binding pattern β *for the head literal, iff there is a permutation* π *of* $\{1, \ldots, m\}$ *such that*

- *for every* $i \in \{1, \ldots, m\}$ *with* $\pi(i) \leq n$ *there is* $\beta_i \in \mathsf{bp}(\mathsf{B}_{\pi(i)})$ *such that*

$$\mathsf{input}(\mathsf{B}_{\pi(i)}, \beta_i) \subseteq \mathsf{input}(\mathsf{A}, \beta) \cup \mathsf{vars}(\mathsf{B}_{\pi(1)} \wedge \cdots \wedge \mathsf{B}_{\pi(i-1)}),$$

- *for every* $i \in \{1, \ldots, m\}$ *with* $\pi(i) > n$ *it holds that*

$$\mathsf{vars}(\mathsf{B}_{\pi(i)}) \subseteq \mathsf{input}(\mathsf{A}, \beta) \cup \mathsf{vars}(\mathsf{B}_{\pi(1)} \wedge \cdots \wedge \mathsf{B}_{\pi(i-1)}),$$

- *and furthermore it holds that*

$$\mathsf{vars}(\mathsf{A}) \subseteq \mathsf{vars}(\mathsf{B}_1 \wedge \cdots \wedge \mathsf{B}_n \wedge \neg \mathsf{B}_{n+1} \wedge \cdots \wedge \neg \mathsf{B}_m) \cup \mathsf{input}(\mathsf{A}, \beta).$$

The permutation π determines a possible evaluation sequence for the body literals. Note that different binding patterns for the head literal might need different evaluation sequences of the body literals. E.g., when `append` is called with binding pattern `bbf`, the given order of body literals works fine:

```
append(X, L, Y) :- cons(F,R,X), append(R,L,RL), cons(F,RL,Y).
```

If, however, `append` is called with binding pattern `ffb`, the following evaluation sequence is needed:

```
append(X, L, Y) :- cons(F,RL,Y), append(R,L,RL), cons(F,R,X).
```

In deductive databases, possible queries cannot be anticipated, therefore there is a strong need to support different binding patterns for derived predicates.

Interestingly, when the magic set transformation is applied to the rules, the result is a program in which each rule is range-restricted for the binding pattern $\mathsf{ff}\ldots\mathsf{f}$, thus bottom-up evaluation can be easily applied afterwards.

3 Extended Rules

In extended logic programming, the rule bodies can be arbitrary first order formulas. Since a formula is a complex tree structure, we can no longer use a simple permutation in order to define an evaluation sequence. Consider

$$\mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$$

and suppose that the following binding restrictions are given: $\mathsf{p}\colon\mathsf{bf}$, $\mathsf{q}\colon\mathsf{bf}$, $\mathsf{r}\colon\mathsf{f}$. Then the only possible evaluation sequence is $\mathsf{r},\mathsf{p},\mathsf{q}$. Of course, one could require that the user writes the formula in a way that left-to-right evaluation is possible. That would simplify the definition a bit, but it would contradict the declarative paradigm. Furthermore, it would not be practical if the derived predicate supports several binding patterns.

The first task is now to generalize the notion of binding patterns from predicates to formulas:

Definition 6 (Generalized Binding Pattern). *A generalized binding pattern for a formula φ is a pair of sets of variables, written $\mathsf{X}_1,\ldots,\mathsf{X}_n \longrightarrow \mathsf{Y}_1,\ldots,\mathsf{Y}_m$, such that $\{\mathsf{X}_1,\ldots,\mathsf{X}_n,\mathsf{Y}_1,\ldots,\mathsf{Y}_m\} \subseteq \mathsf{vars}(\varphi)$.*

This should mean that given values for $\mathsf{X}_1,\ldots,\mathsf{X}_n$, we can compute a finite set of candidate values for $\mathsf{Y}_1,\ldots,\mathsf{Y}_m$. The final decision, whether the formula is true or false in a given interpretation can usually be done only when we have values for all free variables in the formula.

Generalized binding patterns are related to finiteness dependencies [RBS87]. Finiteness dependencies have first been studied for infinite relations (with attributes instead of variables). The definition of when a finiteness dependency is satisfied for a given relation in [RBS87] is a bit unclear: "If $r(\mathsf{X}_1,\ldots,\mathsf{X}_n)$ is finite, then $r(\mathsf{Y}_1,\ldots,\mathsf{Y}_m)$ is finite." Finiteness dependencies have been used for general formulas in [EHJ93], but there the definition of satisfaction is based on the number of function applications that lead from X-values to Y-values.

Our own definition of the meaning of $\mathsf{X}_1,\ldots,\mathsf{X}_n \longrightarrow \mathsf{Y}_1,\ldots,\mathsf{Y}_m$ adds to the basic finiteness requirement an important computability property (and links it to the given binding patterns for the predicates). In order not to overload the semantics of "finiteness dependency" further, we used a different name.

Definition 7 (Valid Generalized Binding Pattern). *A generalized binding pattern $\mathsf{X}_1,\ldots,\mathsf{X}_n \longrightarrow \mathsf{Y}_1,\ldots,\mathsf{Y}_m$ for a formula φ is valid iff for every allowed interpretation $\mathcal{I}$ and all values $d_1,\ldots,d_n$ from the domain of $\mathcal{I}$*

- *the set*

$$\{(\mathcal{A}(\mathsf{Y}_1),\ldots,\mathcal{A}(\mathsf{Y}_m)) \mid (\mathcal{I},\mathcal{A}) \models \varphi,\ \mathcal{A}(\mathsf{X}_1) = d_1,\ldots,\mathcal{A}(\mathsf{X}_n) = d_n\}$$

  is finite (i.e. there are only finitely many possible assignments to the Y_i that make the formula true, given values for the X_i), and
- *a finite superset of this set is effectively computable.*

Consider again the case $\mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$. The binding restrictions of the subformula $\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X})$ can be described with the generalized binding patterns:

- $\emptyset \longrightarrow \mathsf{X}$ (because r supports binding pattern f)
- $\mathsf{Y} \longrightarrow \mathsf{Z}$ (because q supports binding pattern bf)

When we have computed a set $\mathcal{D} = \{d_1, \ldots, d_n\}$ of values for X according to the first binding pattern, we cannot say yet whether the entire formula will be true or false. But what has to be guaranteed is that the formula will be certainly false if X has a value outside the set $\mathcal{D}$, no matter what values the other variables will have. The second binding pattern $\mathsf{Y} \longrightarrow \mathsf{Z}$ means that when we already have a single value (or finite set of values) for Y, then there are only finitely many variable assignments for Z such that the formula is true, and Y has the given value (or one from the finite set).

Since negation can be used everywhere inside a formula, not only before atomic formulas, we also need to clarify the meaning of a generalized binding pattern in negated context: In this case we are interested to get finitely many values such that the formula is false. Again, computing a superset is possible. It must only be guaranteed that the formula is true for every value outside the computed set.

Given a set of generalized binding patterns, the following closure operation computes immediate consequences. This closure is for instance used after the union of sets of binding patterns done for a conjunction. Consider again the formula $\mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$. The left subformula $\mathsf{p}(\mathsf{X},\mathsf{Y})$ gives the generalized binding pattern $\mathsf{X} \longrightarrow \mathsf{Y}$ corresponding to the binding pattern bf for p (actually, it also gives many more implied generalized binding patterns, see below). The right subformula discussed above yields (among others) $\emptyset \longrightarrow \mathsf{X}$ and $\mathsf{Y} \longrightarrow \mathsf{Z}$. Given these three generalized binding patterns, we can compose them to get $\emptyset \longrightarrow \mathsf{X},\mathsf{Y},\mathsf{Z}$ which means that the complete formula is evaluable with a finite result. This composition is done by the closure operator defined in a minute. An additional purpose of the closure operator is to add trivially implied generalized binding patterns. E.g., when we have $\mathsf{X} \longrightarrow \mathsf{Y}$, this implies $\mathsf{X} \longrightarrow \mathsf{X},\mathsf{Y}$ and $\mathsf{X},\mathsf{Y} \longrightarrow \mathsf{X}$ and $\mathsf{X},\mathsf{Z} \longrightarrow \mathsf{Y},\mathsf{Z}$. Such implied generalized binding patterns are important e.g. for intersections done for disjunctive conditions.

Definition 8 (Closure of Sets of Generalized Binding Patterns). *Let $\mathcal{B}$ be a set of generalized binding patterns for a formula φ. Then*

$$\begin{aligned}\mathsf{cl}_\varphi(\mathcal{B}) := \{\mathcal{X} \longrightarrow \mathcal{Y} \mid\ & \mathcal{X} \subseteq \mathsf{vars}(\varphi), \mathcal{Y} \subseteq \mathsf{vars}(\varphi),\\ & \textit{there are } n \in \mathbb{N}_0,\ \mathcal{X}_1 \longrightarrow \mathcal{Y}_1\ , \ldots,\ \mathcal{X}_n \longrightarrow \mathcal{Y}_n \in \mathcal{B}\\ & \textit{such that for } i = 1, \ldots, n\textit{:}\\ & \mathcal{X}_i \subseteq \mathcal{X} \cup \textstyle\bigcup_{j=1}^{i-1} \mathcal{Y}_j,\\ & \mathcal{Y} \subseteq \mathcal{X} \cup \textstyle\bigcup_{j=1}^{n} \mathcal{Y}_j\}.\end{aligned}$$

Theorem 1. *If every generalized binding pattern in $\mathcal{B}$ is valid, then also every binding pattern in $\mathsf{cl}_\varphi(\mathcal{B})$ is valid.*

It is known that the Armstrong axioms for functional dependencies are sound and complete also for finiteness dependencies [RBS87]. Since generalized binding patterns have a somewhat different semantics, this result does not automatically carry over, but at least the soundness is obvious:

- If $\mathcal{X} \subset \mathcal{Y}$, then $\mathcal{X} \longrightarrow \mathcal{Y}$ is valid (Reflexivity).
- If $\mathcal{X} \longrightarrow \mathcal{Y}$ is valid, then $\mathcal{X} \cup \mathcal{Z} \longrightarrow \mathcal{Y} \cup \mathcal{Z}$ is valid (Augmentation).
- If $\mathcal{X} \longrightarrow \mathcal{Y}$ and $\mathcal{Y} \longrightarrow \mathcal{Z}$ are valid, then $\mathcal{X} \longrightarrow \mathcal{Z}$ is valid.

Definition 9 (Computation of Generalized Binding Patterns).
We need the following auxillary operation:

$$\begin{aligned}\mathsf{intersect}_\varphi(\mathcal{B}_1, \mathcal{B}_2) := \{\mathcal{X} \longrightarrow \mathcal{Y} \mid\ & \mathcal{X} \in \mathsf{vars}(\varphi),\ \mathcal{Y} \in \mathsf{vars}(\varphi),\\ & \textit{there are } \mathcal{X}_1 \longrightarrow \mathcal{Y}_1 \in \mathcal{B}_1,\ \mathcal{X}_2 \longrightarrow \mathcal{Y}_2 \in \mathcal{B}_2\\ & \textit{with } \mathcal{X}_1 \cup \mathcal{X}_2 \subseteq \mathcal{X},\ \textit{and } \mathcal{Y} \subseteq \mathcal{Y}_1 \cap \mathcal{Y}_2\}.\end{aligned}$$

The functions gbp^+ *and* gbp^- *define sets of generalized binding patterns for arbitrary formulas (in positive/negated context):*

- *If* φ *is an atomic formula* $\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$ *(where* p *is not* $=$*):*

$$\begin{aligned}\mathsf{gbp}^+(\varphi) &:= \mathsf{cl}_\varphi\big(\{\mathcal{X} \longrightarrow \mathsf{vars}(\varphi) \mid \textit{there is } \beta \in \mathsf{bp}(\mathsf{p}) \textit{ with } \mathcal{X} = \textstyle\bigcup_{\beta(i)=\mathsf{b}} \mathsf{t}_i\}\big)\\ \mathsf{gbp}^-(\varphi) &:= \mathsf{cl}_\varphi(\{\mathsf{vars}(\varphi) \longrightarrow \mathsf{vars}(\varphi)\})\end{aligned}$$

- *If* φ *is an atomic formula* $\mathsf{t}_1 = \mathsf{t}_2$*:*

$$\begin{aligned}\mathsf{gbp}^+(\varphi) &:= \mathsf{cl}_\varphi\big(\{\mathcal{X} \longrightarrow \mathsf{vars}(\varphi) \mid \mathcal{X} = \mathsf{vars}(\mathsf{t}_1) \textit{ or } \mathcal{X} = \mathsf{vars}(\mathsf{t}_2)\}\big)\\ \mathsf{gbp}^-(\varphi) &:= \mathsf{cl}_\varphi(\{\mathsf{vars}(\varphi) \longrightarrow \mathsf{vars}(\varphi)\})\end{aligned}$$

- *If* φ *is a negated formula* $\neg\varphi_0$*:*

$$\begin{aligned}\mathsf{gbp}^+(\varphi) &:= \mathsf{gbp}^-(\varphi_0)\\ \mathsf{gbp}^-(\varphi) &:= \mathsf{gbp}^+(\varphi_0)\end{aligned}$$

- *If* φ *is a conjunction* $\varphi_1 \wedge \varphi_2$*:*

$$\begin{aligned}\mathsf{gbp}^+(\varphi) &:= \mathsf{cl}_\varphi\big(\mathsf{gbp}^+(\varphi_1) \cup \mathsf{gbp}^+(\varphi_2)\big)\\ \mathsf{gbp}^-(\varphi) &:= \mathsf{cl}_\varphi\big(\mathsf{intersect}(\mathsf{gbp}^-(\varphi_1),\ \mathsf{gbp}^-(\varphi_2))\big)\end{aligned}$$

- *If* φ *is a disjunction* $\varphi_1 \vee \varphi_2$*:*

$$\begin{aligned}\mathsf{gbp}^+(\varphi) &:= \mathsf{cl}_\varphi\big(\mathsf{intersect}(\mathsf{gbp}^+(\varphi_1),\ \mathsf{gbp}^+(\varphi_2))\big)\\ \mathsf{gbp}^-(\varphi) &:= \mathsf{cl}_\varphi\big(\mathsf{gbp}^-(\varphi_1) \cup \mathsf{gbp}^-(\varphi_2)\big)\end{aligned}$$

- *If* φ *has the form* $\exists X\colon \varphi_0$ *or* $\forall X\colon \varphi_0$*:*

$$\begin{aligned}\mathsf{gbp}^+(\varphi) &:= \mathsf{cl}_\varphi(\{\mathcal{X} \longrightarrow (\mathcal{Y} - \{X\}) \mid \mathcal{X} \longrightarrow \mathcal{Y} \in \mathsf{gbp}^+(\varphi_0),\ X \notin \mathcal{X}\})\\ \mathsf{gbp}^-(\varphi) &:= \mathsf{cl}_\varphi(\{\mathcal{X} \longrightarrow (\mathcal{Y} - \{X\}) \mid \mathcal{X} \longrightarrow \mathcal{Y} \in \mathsf{gbp}^-(\varphi_0),\ X \notin \mathcal{X}\})\end{aligned}$$

Theorem 2.

- *Every* $\mathsf{X}_1, \ldots, \mathsf{X}_n \longrightarrow \mathsf{Y}_1, \ldots, \mathsf{Y}_m \in \mathsf{gbp}^+(\varphi)$ *is valid.*
- *In the same way, for* $\mathsf{X}_1, \ldots, \mathsf{X}_n \longrightarrow \mathsf{Y}_1, \ldots, \mathsf{Y}_m \in \mathsf{gbp}^-(\varphi)$ *there are only finitely many assignments for the* Y_i *that make the formula false (given values for the* X_i*), and finite supersets of these sets are effectively computable.*

Up to now, we have computed only an upper bound for the values that make a formula true. As explained above for the example $\mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$, the reason was that when we consider subformulas, we might not know values for all variables yet that appear in the subformula. Of course, in the end, we want to know the exact set of variable assignments that make the formula true.

This is possible when we have computed candidate assignments for all variables that occur in the formula. Then we can recursively step down the formula and check for every given variable assignment whether the corresponding subformula is true or false. For the atomic formulas this is obvious (our definition of allowed interpretation implies that we can test whether a given tuple of values is contained in the extension of a predicate). For $\wedge$, $\vee$, $\neg$ it is also clear how the truth values computed for the subformulas can be combined. The interesting case are the quantifiers. Let us consider the example

$$\mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge \exists\, \mathsf{Z}\colon (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X})).$$

Suppose that the valid binding patterns for the predicates are $\mathsf{p}\colon \mathsf{bf}$, $\mathsf{q}\colon \mathsf{bf}$, $\mathsf{r}\colon \mathsf{f}$. Consider the subformula $\exists\, \mathsf{Z}\colon (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$. In the first phase, we can compute a set of possible values for X, i.e. we get the generalized binding pattern $\emptyset \longrightarrow \mathsf{X}$. But we have no chance to check whether the existential condition is indeed true without having a value for Y (which can only be computed after we have a value for X). But from the finite set of candidate values for X we can compute a finite number of assignments for (X,Y) which must be checked.

In the second phase, we can assume that we have values for all free variables in a subformula. In the example, we must check whether $\exists\, \mathsf{Z}\colon (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$ is indeed true, given values for X and Y. This is possible if there are only a finite number of candidate values for the quantified variable Z which must be tried. So we need that the generalized binding pattern $\mathsf{X},\mathsf{Y} \longrightarrow \mathsf{Z}$ holds for the quantified subformula $\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X})$. This is indeed the case because q supports the binding pattern bf.

Note how different the situation would be if q permitted only the binding pattern bb. We could still compute a finite set of candidate assignments for X and Y, i.e. be sure that the formula $\mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge \exists\, \mathsf{Z}\colon (\mathsf{q}(\mathsf{Y},\mathsf{Z}) \wedge \mathsf{r}(\mathsf{X}))$ is false outside this set. But we had no possibility to check whether the existential condition is indeed satisfied without "guessing" values for the quantified variable Z.

For universally quantified variables, we need that the quantified formula can be false only for a finite set of values, so that it suffices to explicitly check this set. E.g. consider

$$\mathsf{r}(\mathsf{X}) \wedge \forall\, \mathsf{Y}\colon \mathsf{p}(\mathsf{X},\mathsf{Y}) \rightarrow \mathsf{r}(\mathsf{Y}).$$

(again with the binding patterns $\mathsf{p}\colon \mathsf{bf}$ and $\mathsf{r}\colon \mathsf{f}$). For the universal quantifier to be evaluable, the condition is $\mathsf{X} \longrightarrow \mathsf{Y} \in \mathsf{gbp}^-(\mathsf{p}(\mathsf{X},\mathsf{Y}) \rightarrow \mathsf{r}(\mathsf{Y}))$.

Definition 10 (Range-Restriction). *A rule $\mathsf{A} \leftarrow \varphi$ is range-restricted given a binding pattern β for A iff*

1. $\mathcal{X} \longrightarrow \mathcal{Y} \in \mathsf{gbp}^+(\varphi)$ *where*
   - $\mathcal{X} := \mathsf{input}(\mathsf{A}, \beta)$ *(variables occurring in bound arguments in the head)*
   - $\mathcal{Y} := \mathsf{vars}(\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \leftarrow \varphi)$ *(all variables in the rule except quantified ones).*
2. *for every subformula $\exists\, \mathsf{Z}\colon \varphi_0$ in positive (unnegated) context, or subformula $\forall\, \mathsf{Z}\colon \varphi_0$ in negated context:*

$$(\mathsf{vars}(\varphi_0) - \{\mathsf{Z}\}) \longrightarrow \mathsf{Z} \in \mathsf{gbp}^+(\varphi_0),$$

3. *for every subformula $\forall\, \mathsf{Z}\colon \varphi_0$ in positive context, or subformula $\exists\, \mathsf{Z}\colon \varphi_0$ in negated context:*

$$(\mathsf{vars}(\varphi_0) - \{\mathsf{Z}\}) \longrightarrow \mathsf{Z} \in \mathsf{gbp}^-(\varphi_0).$$

Theorem 3. *The immediate consequences of a range-restricted rule (according to the T_P-operator) can be effectively computed, given values for the input arguments of the head literal.*

As mentioned above, the magic set transformation turns a rule that is range-restricted for a binding pattern β into a rule that is range-restricted for the binding pattern $\mathsf{ff}\ldots\mathsf{f}$ (by adding a condition to the body that binds the input arguments). Then the immediate consequences of the rule can be computed without further restrictions on input arguments.

4 A Possible Extension

It is also possible to define a slightly more liberal version of range-restriction that requires only that variables are bound in the subformula in which they are used. E.g., $\mathsf{p}(\mathsf{X}) \leftarrow \mathsf{q}(\mathsf{X}) \vee \mathsf{r}(\mathsf{X},\mathsf{Y})$ would not be range-restricted according to Definition 10. That is no real problem, since one can write $\mathsf{p}(\mathsf{X}) \leftarrow \mathsf{q}(\mathsf{X}) \vee \exists\, \mathsf{Y}\colon \mathsf{r}(\mathsf{X},\mathsf{Y})$, or alternatively, $\mathsf{p}(\mathsf{X}) \leftarrow (\mathsf{q}(\mathsf{X}) \wedge \mathsf{Y} = \mathsf{nil}) \vee \mathsf{r}(\mathsf{X},\mathsf{Y})$. Nevertheless, it would also be possible (and an improvement for the user) to accept the original version of the rule.

The important insight here is that for an existentially quantified variable (including variables that are free in the rule, but appear only in the body) it is not necessary that the quantified formula is true only for a finite set of values. It is only required that we have to check only a finite set of values. In the example, $\mathsf{q}(\mathsf{X}) \vee \mathsf{r}(\mathsf{X},\mathsf{Y})$ might be true for an infinite set of Y-values (when $\mathsf{q}(\mathsf{X})$ is already true). However, values outside the extension of r will all behave in the same way, therefore it suffices to check a single such value.

In [Bra92] (page 21), we have solved the problem by computing bottom-up not only sets of bound variables (in positive/negated context), but also unbound

variables (in positive/negated context). In the critical condition $\mathsf{q}(\mathsf{X}) \vee \mathsf{r}(\mathsf{X},\mathsf{Y})$ the variable Y is neither bound nor unbound (while X is bound). A generalization of this idea to the case with built-in predicates is subject of our further work.

Another idea is to have a weaker version of generalized binding patterns, where $\mathcal{X} \longrightarrow \mathcal{Y}$ means that given values for the $\mathcal{X}$, it suffices to check a finite set of values for the $\mathcal{Y}$: If the formula is not true for any of these values, there cannot be any assignment that makes it true (with the given values for the $\mathcal{X}$).

5 Related Work

Of course, questions of domain independence and safety (finite answers) have been studied for quite a long time, [Dem92] gives a good overview over earlier work. [Dem92] generalizes this to arbitrary formulas, but does not consider built-in predicates.

[RBS87] have finiteness dependencies, which are similar to our generalized binding patterns, but consider only standard rules. As explained above, the main difference between finiteness dependencies and generalized binding patterns is our additional computability requirement.

[EHJ93] have finiteness dependencies and arbitrary formulae, but their semantics of finiteness dependencies is again different: Their goal is to prove domain independence of a formula and a finiteness dependency $\mathcal{X} \longrightarrow \mathcal{Y}$ means that values for the $\mathcal{Y}$ can be only a bounded number of function applications farther away from the active domain than values for the $\mathcal{X}$. So they consider computable functions, but the built-in predicates discussed here are more general because they can support several binding patterns. The paper also aims at computing the result of a formula, but the method is very different than ours. They investigate the translation of formulas into relational algebra. The last step is explained only by an example, and it seems that sometimes it might be necessary to enumerate the entire active domain.

[Mah97] studies finiteness constraints, which have the form $\varphi \Rightarrow \mathcal{X} \rightarrow_{fin} \mathcal{Y}$ and mean that for each fixed assignment for the variables in $\mathcal{X}$, there are only finitely many values of the variables in $\mathcal{Y}$ in the tuples of p satisfying φ with the given values of the variables in $\mathcal{X}$. The paper mainly investigates the implication problem for these dependencies and for constrained functional dependencies. One might think that when p simply contains the free variables of φ as attributes, then $\varphi \Rightarrow \mathcal{X} \rightarrow_{fin} \mathcal{Y}$ basically means the same as $\mathcal{X} \longrightarrow \mathcal{Y} \in \mathsf{gbp}^+(\varphi)$. However, the purpose is very different. For instance, in Maher's approach, φ is restricted by a constraint domain, with a typical case being linear arithmetic constraints over integers. In our approach, φ is a more or less arbitrary first order formula. It is also not clear how knowledge about binding patterns for used predicates can be specified in Maher's approach: He considers only a single relation besides the very special predicates in the condition φ. This is no fault of the approach, the goals are simply different. Furthermore, we do not use a constraint solver as Maher, but do a simple syntactic bottom-up computation. Of course, this also gives different results. For instance, from $5 \leq \mathsf{X} \wedge \mathsf{X} \leq 5$, Maher would

conclude that X has only a single possible value. In our approach, this formula is evaluable only for a given value of X, since otherwise the binding restrictions for the built-in predicate $\leq$ are not satisfied.

6 Conclusions

I am convinced that deductive databases can still become a serious competitor to standard relational and object-relational databases for many practical projects. Declarative programming has many advantages, and for single queries this is already standard in the database field (SQL is a declarative language). Deductive databases would lift the declarativity to the level of programs, but this is not as easy as it was expected in the times when deductive databases were a hype topic. More research is still needed.

In this paper, we attacked a very basic problem: Which formulas should be allowed in rule bodies? Of course, we need that they have finite solutions, and that the solutions are effectively computable. In a realistic setting, a deductive database will have many built-in predicates with different binding restrictions. The necessary definition is technically not very easy, but still natural and understandable.

Questions of domain independence and safety for general formulas have been investigated before, and finiteness dependencies studied in the literature behave quite similar to our generalized binding patterns. However, the coupling of finiteness conditions with the computability of an upper bound, and the two-step approach to the evaluation of a formula are unique features of the current paper.

Our long-term goal is to develop a deductive database system that supports stepwise migration from classical SQL.

References

[Bra92] S. Brass: Defaults in Deductive Databases (in German). Doctoral Thesis, University of Hannover, 1992.

[Bra09] S. Brass: A Logic with Duplicates for an SQL-compatible Deductive Database. Submitted for publication.

[Dem92] R. Demolombe: Syntactical characterization of a subset of domain-independent formulas. *Journal of the ACM (JACM) 39:1*, 71–94, 1992.

[EHJ93] M. Escobar-Molano, R. Hull, D. Jacobs: Safety and translation of calculus queries with scalar functions. In *Proc. of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS'93)*, 253–264, ACM, 1993.

[LC05] M. Leuschel, S. Craig: A reconstruction of the Lloyd-Topor transformation using partial evaluation. In P. Hill (ed.), *Pre-Proceedings of LOPSTR'05*, Imperial College, London, UK, 2005.
`http://eprints.ecs.soton.ac.uk/11198/`.

[Llo87] J. W. Lloyd: *Foundations of Logic Programming*, 2nd edition. Springer-Verlag, Berlin, 1987.

[LT84] J. W. Lloyd, R. W. Topor: Making Prolog more expressive. *The Journal of Logic Programming 1 (1984)*, 225–240.

[LT85] J. W. Lloyd, R. W. Topor: A basis for deductive database systems. *The Journal of Logic Programming 2 (1985)*, 93–109.

[LTT99] V. Lifschitz, L. R. Tang, H. Turner: Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence 25:3–4 (1999)*, 369–389. `http://citeseer.ist.psu.edu/lifschitz99nested.html`.

[Mah97] M. J. Maher: Constrained Dependencies. *Theoretical Computer Science 173 (1997)*, 113–149. `http://www.cse.unsw.edu.au/~mmaher/pubs/cdb/condep.ps`.

[RBS87] R. Ramakrishnan, F. Bancilhon, A. Silberschatz: Safety of recursive Horn clauses with infinite relations. In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS'87)*, 328–339, ACM, 1987.

Transforming Imperative Algorithms to Constraint Handling Rules

Slim Abdennadher, Haythem Ismail, and Frederick Khoury

Department of Computer Science, German University in Cairo
[slim.abdennadher, haythem.ismail, frederick.nabil]@guc.edu.eg
http://met.guc.edu.eg

Abstract. Different properties of programs, implemented in Constraint Handling Rules (CHR), have already been investigated. Proving these properties in CHR is fairly simpler than proving them in any type of imperative programming language, which triggered the proposal of a methodology to map imperative programs into equivalent CHR. The equivalence of both programs implies that if a property is satisfied for one, then it is satisfied for the other.
The mapping methodology could be put to other beneficial uses. One such use is the automatic generation of global constraints, at an attempt to demonstrate the benefits of having a rule-based implementation for constraint solvers.

1 Introduction

Algorithms have properties that define their operation and their results, such as correctness and confluence which can be illustrated and proven in programs of various languages. Due to the differences between languages, proofs differ from one to another, and could therefore be easier in some than in others. The aim of the paper is to present an approach to map imperative programs to equivalent rule-based ones. Thus, a technique to prove a property in an imperative program would be to prove this same property in the corresponding rule-based version.
In this paper, a mapping from an imperative programming language to Constraint Handling Rules (CHR) is presented. CHR is a concurrent, committed-choice constraint logic programming language especially designed to implement constraint solvers. CHR, which was developed as an enhancement to the constraint programming paradigm, is declarative and rule-based, and is the language of choice in this paper due to the already existing results to prove several properties, such as correctness, confluence, and termination.
Other areas of research could also benefit from the previously mentioned transformation, such as the development of rule-based solvers for global constraints. The mapping schematic in this work could be applied in the translation of imperative constraint solvers to declarative rule-based solvers, with a purpose of analyzing these constraints through CHR. With the flexibility and expressivity of CHR, such a translation could have benefits on the functionality of the involved constraint-handlers.

This paper is structured as follows. In Section 2 we briefly present the syntax and semantics of a subset of CHR. In Section 3, we give a formal presentation of the mapping and prove the equivalence of the imperative algorithm and the corresponding generated CHR program. In Section 4, we present using examples the methodology for transforming imperative algorithms into CHR. Finally, we conclude in Section 5 with a summary and a discussion of future work.

2 Constraint Handling Rules

This section presents the syntax and the operational semantics of a subset of CHR, namely simpagation rules. We use two disjoint sorts of predicate symbols for two different classes of constraints: *built-in constraint symbols* and *user-defined constraint symbols (CHR symbols)*. Built-in constraints are those handled by a predefined constraint solver that already exists. User-defined constraints are those defined by a CHR program. Simpagation rules are of the form

$$Rulename \ @ \ H_1 \backslash H_2 \ \texttt{<=>} \ C \ \texttt{|} \ B,$$

where *Rulename* is an optional unique identifier of a rule. The *head* $H_1 \backslash H_2$ consists of two parts H_1 and H_2. Both parts consist of a conjunction of user-defined constraints. The *guard* C is a conjunction of built-in constraints, and the *body* B is a conjunction of built-in and user-defined constraints. If H_1 is an empty conjunction, then we omit the symbol "\" and the rule is called a simplification rule.
The operational semantics of a simpagation rule is based on an underlying theory CT for the built-in constraints and a state G which is a conjunction of built-in and user-defined constraints. A simpagation rule of the form $H_1 \backslash H_2 \Leftrightarrow C \mid B$ is applicable to a state $H_1' \wedge H_2' \wedge G$ if $CT \models G_B \rightarrow \exists \bar{x}((H_1 = H_1' \wedge H_2 = H_2') \wedge C)$, where $\bar{x}$ are the variables in H_1 and H_2, and G_B is a conjunction of the built-in constraints in G. The state transition is defined as follows:

$$H_1' \wedge H_2' \wedge G \mapsto H_1' \wedge G \wedge B \wedge C \wedge (H_1 = H_1' \wedge H_2 = H_2')$$

3 Operational Equivalence

In Section 4, we gave a quasi formal description of the mapping from $\mathcal{I}$ to CHR programs. We now turn to a more formal presentation of the mapping, ending this section with a proof of equivalence between $\mathcal{I}$ and corresponding CHR programs.

3.1 The Language $\mathcal{I}$

To simplify the exposition, we impose two simple restrictions on $\mathcal{I}$ programs. It should be clear that the following restrictions are only syntactic; the expressive power of $\mathcal{I}$ is preserved.

1. *The identifier on the left-hand side of an assignment statement does not occur on the right-hand side and is not declared in the same statement.*
   This can easily be enforced by the careful use of temporary variables and the separation of declaration from initialization.
2. *No array variables are used.*
   Assuming that all arrays have fixed sizes, an $\mathcal{I}$ program with an array A indexed from 0 to n may be replaced by $n+1$ variables $A_0, \ldots A_n$. Naturally, several other changes will need to be made. In particular, some loops will need to be unwrapped.

The set of thus restricted $\mathcal{I}$ programs may be defined recursively as follows, where we assume the standard imperative syntax of identifiers and expressions.

Definition 1. The set $\mathcal{I}$ is the smallest set containing all of the following forms.

1. $dt\ x$`;`, where dt is a data type and x an identifier
2. x `=` e`;`, where x is an $\mathcal{I}$ identifier and e is an $\mathcal{I}$ expression
3. `if` e $\{P_1\}$ `else` $\{P_2\}$, where e is a Boolean expression and $P_1, P_2 \in \mathcal{I}$
4. `while` e $\{P\}$, where e is a Boolean expression and $P \in \mathcal{I}$
5. $P_1\ P_2$, where $P_1, P_2 \in \mathcal{I}$.

We can provide standard operational semantics for $\mathcal{I}$ in the spirit of [2]. A store σ is a partial function from $\mathcal{I}$ identifiers to $\mathcal{I}$ values. We denote by Σ the set of all possible $\mathcal{I}$ stores. Usually, the semantics is given by a transition system on the set Γ of program configurations, where $\Gamma \subseteq (\mathcal{I} \times \Sigma) \cup \Sigma$. A configuration γ is *terminal* if $\gamma \in \Sigma$. Given that $\mathcal{I}$ programs are deterministic, every terminating program P and initial store σ_i have a unique terminal configuration $[P](\sigma)$. To define the semantics of $\mathcal{I}$, it thus suffices to define the function $[P]$. A recursive definition (on the structure of $\mathcal{I}$ programs) of $[P]$ is given in Figure 1.[1]

In what follows, x is an $\mathcal{I}$ identifier, e is an $\mathcal{I}$ expression, and $\sigma^{x \mapsto v}$ is identical to σ except that $\sigma(x) = v$.

1. $[dt\ x;](\sigma) = \sigma^{x \mapsto Default(dt)}$
   where $Default(dt)$ is the default value for the $\mathcal{I}$ data type dt.
2. $[x\texttt{=}e\texttt{;}](\sigma) = \sigma^{x \mapsto [\![e]\!]^\sigma}$
   where $[\![e]\!]^\sigma$ is the value of the $\mathcal{I}$ expression e with respect to the store σ.
3. $[\texttt{if}\ e\ \{P_1\}\ \texttt{else}\ \{P_2\}](\sigma) = [P](\sigma)$
   where $P = P_1$ if $[\![e]\!]^\sigma$ is true and $P = P_2$ otherwise.
4. $[\texttt{while}\ e\ \{P\}](\sigma) = \gamma$
   where $\gamma = [\texttt{while}\ e\ \{P\}]([P](\sigma))$ if $[\![e]\!]^\sigma$ is true, and $\gamma = \sigma$ otherwise.
5. $[P_1 P_2](\sigma) = [P_2]([P_1](\sigma))$

Fig. 1. Operational semantics of $\mathcal{I}$.

[1] Note that $[P](\sigma)$ is undefined for nonterminating configurations.

3.2 The CHR Fragment

In Section 3.3, we present a mapping from $\mathcal{I}$ to CHR. Naturally, the mapping is not onto, and the image thereof is comprised of CHR programs with only two constraint symbols: `var/2` and `state/2`. The constraint `var` was satisfactorily discussed in Section 4. In this section, we examine the constraint `state/2` and some features of CHR programs employing it. For the purpose of the formal construction, we take `state` to be a binary constraint.

A constraint `state`(b, n) intuitively indicates that the current CHR state corresponds to the state of the $\mathcal{I}$ program following the execution of a statement uniquely identified by the pair (b, n). As per Definition 4, b is a nonempty string over the alphabet $\Sigma_{012} = \{$`0`, `1`, `2`$\}$ starting with a `0`, and n is a non-empty string over the alphabet Σ_D composed of the set of decimal digits and the separator `#`. According to Definition 1, an $\mathcal{I}$ program is a sequence of statements. Each of these statements corresponds to a pair (b, n), where $b =$ `0` and n is a nonempty string that does not contain the `#` (i.e., a numeral); the number represented by n indicates the order of the statement in the $\mathcal{I}$ program. The special pair (`0`, `0`) corresponds to the state before any statement has been executed. If the statement corresponding to a pair (b, n) is a `while` loop, then a statement in the body of the loop will correspond to the pair $(b, n\texttt{\#}m)$, where m is a numeral denoting the order of the statement within the body of the loop. Similarly, if an `if-then-else` statement corresponds to the pair (b, n), then a statement within the `then` block corresponds to the pair (b`1`, n`#`m). A statement within the `else` block corresponds to the pair (b`2`, n`#`m). In both cases, m is a numeral denoting the order of the statement within the block.

In order to facilitate the definition of the transformation from $\mathcal{I}$ to CHR, we need some terminology to succinctly talk about CHR programs with `state` constraints. We start with two properties of these constraints.

In the sequel, if m and n are numerals, then m_{+n} is the numeral denoting the number $[\![m]\!] + [\![n]\!]$, where $[\![x]\!]$ is the number denoted by the numeral x.

Definition 2. Let P be a CHR program and let $s =$ `state`(b, n) be a constraint in P.

1. s is *terminal* if there is a rule $r = H_1$ `\` s, H_2 `<=>` C `|` B in P, such that no `state` constraints appear in B. Such a rule r is a *terminal rule*.
2. s is *maximal* if $n = uv$, where v is the longest numeral suffix of n, and for every other constraint `state`$(b', u'v')$ in P, with v' the longest numeral suffix of $u'v'$, b is a substring of b' and either u is a proper substring of u' or $u = u'$ and $[\![v]\!] > [\![v']\!]$.

It is easy to show that if a CHR program has a maximal constraint, then it is unique.

Definition 3. In what follows $P, P1$ and $P2$ are CHR programs, $b \in \Sigma^+_{012}$, and $n \in \Sigma^+_D$.

1. The *n-translation of P* is the CHR program P_{+n} which is identical to P with every constraint `state`$(b', n'\texttt{\#}m)$ replaced by a constraint `state`$(b', n'\texttt{\#}m_{+n})$.

2. The (b,n)-*nesting of* P is the CHR program $(b,n)\triangleright P$ which is identical to P with every constraint $\texttt{state(0}b'\texttt{,}n'\texttt{)}$ replaced by a constraint $\texttt{state(}bb'\texttt{,}n\texttt{\#}n'\texttt{)}$.
3. The (b,n)-*termination of* P is the CHR program $(b,n)\triangledown P$ which is identical to P with every terminal rule

$$H_1 \ \texttt{\textbackslash} \ \texttt{state(}b'\texttt{,}n'\texttt{),}H_2\texttt{<=>}\ C\texttt{|}B$$

replaced by the rule

$$H_1 \ \texttt{\textbackslash} \ \texttt{state(}b'\texttt{,}n'\texttt{),}H_2\texttt{<=>}\ C\texttt{|}B\texttt{,state(}b\texttt{,}n\texttt{)}$$

4. Let $\texttt{state(0,}n'\texttt{)}$ be a maximal constraint in P_1. The *concatenation* of P_1 and P_2 is the CHR program

$$P1 \circ P2 = (0,n)\triangledown P1 \cup P2_{+n}$$

where $n = n'_{+1}$.

3.3 The $\mathcal{I}$-CHR Transformation

We can now give the mapping from $\mathcal{I}$ to CHR programs a more formal guise, defining it as a system $\mathcal{T}$ of functions from syntactic $\mathcal{I}$ constructs to syntactic CHR constructs.

Definition 4. An $\mathcal{I}$-CHR transformation is a quadruple $\mathcal{T} = \langle \mathcal{N}, \mathcal{V}, \mathcal{E}, \mathcal{F} \rangle$, where

- $\mathcal{N}$ is an injection from the set of $\mathcal{I}$ identifiers into the set of CHR constants.
- $\mathcal{V}$ is an injection from the set of $\mathcal{I}$ identifiers into the set of CHR variables.
- $\mathcal{E}$ is an injection from the set of $\mathcal{I}$ expressions into the set of CHR expressions, such that $\mathcal{E}(e)$ is similar to e with every identifier x replaced by $\mathcal{V}(x)$, every constant replaced by the equivalent CHR constant, and every operator replaced by the equivalent CHR operator.[2]
- $\mathcal{F} : \mathcal{I} \longrightarrow$ CHR is an injection defined recursively as shown in Figure 2.

The following proposition states some syntactic properties of CHR programs resulting from the above transformation.

Proposition 1. *In what follows* P *is an* $\mathcal{I}$ *program and* $(b,n) \in \Sigma^+_{012} \times \Sigma^+_D$.

1. *Every rule in* $\mathcal{F}(P)$ *has exactly one* `state` *constraint in the head and at most one different* `state` *constraint in the body.*
2. *Every* `state` *constraint occurring in* $\mathcal{F}(P)$ *occurs in the head of at least one rule.*
3. $\mathcal{F}(P)$ *has a unique maximal constraint.*

[2] Note the implicit, yet crucial, assumption here. We are assuming that there are constant- and operator- bijections between $\mathcal{I}$ and CHR.

In what follows, x is an $\mathcal{I}$ identifier, e is an $\mathcal{I}$ expression, and V is a (possibly empty) conjunction of CHR constraints of the form $\texttt{var}(\mathcal{N}(y), \mathcal{V}(y))$, one for each identifier y occurring in e.

1. $\mathcal{F}(dt\ x\texttt{;}) = \{\texttt{state(0,0) <=> var(}\mathcal{N}(x)\texttt{, }Default(dt)\texttt{)}\}$
   where $Default(dt)$ is the default value for the $\mathcal{I}$ data type dt.

2. $\mathcal{F}(x\texttt{ = }e\texttt{;}) = \{V\ \texttt{\textbackslash state(0,0), var(}\mathcal{N}(x)\texttt{,\_) <=> }\mathcal{V}(x)\texttt{=}\mathcal{E}(e)\texttt{, var(}\mathcal{N}(x)\texttt{,}\mathcal{V}(x)\texttt{)}\}$

3. $\mathcal{F}(\texttt{if }e\ \{P_1\}\ \texttt{else}\ \{P_2\}) = \texttt{(01,0)} \triangleright \mathcal{F}(P_1) \cup \texttt{(02,0)} \triangleright \mathcal{F}(P_2) \cup S$

   where $S = \{V\ \texttt{\textbackslash state(0,0) <=>}\ \mathcal{E}(e) \mid \texttt{state(01,0\#0)},$
   $V\ \texttt{\textbackslash state(0,0) <=> \textbackslash+}\mathcal{E}(e) \mid \texttt{state(02,0\#0)}\}$

4. $\mathcal{F}(\texttt{while }e\ \{P\}) = \texttt{(0,0)} \triangledown (\texttt{(0,0)} \triangleright \mathcal{F}(P)) \cup S$

   where $S = \{V\ \texttt{\textbackslash state(0,0) <=>}\ \mathcal{E}(e) \mid \texttt{state(0,0\#0)},$
   $V\ \texttt{\textbackslash state(0,0) <=> \textbackslash+}\mathcal{E}(e)\ \texttt{| true}\}$

5. $\mathcal{F}(P_1P_2) = \mathcal{F}(P_1) \circ \mathcal{F}(P_2)$

Fig. 2. Definition of the function $\mathcal{F}$ from $\mathcal{I}$ to CHR programs

Note that, had the last statement of the proposition been false, case 5 in Figure 2 would not have made sense. The following important result follows from Definition 4 and Proposition 1.

Theorem 1. *Let P be an $\mathcal{I}$ program. If G is a state containing a single* `state` *constraint that occurs in $\mathcal{F}(P)$, then exactly one rule in $\mathcal{F}(P)$ is applicable to G.*

Corollary 1. *If P is an $\mathcal{I}$ program, then $\mathcal{F}(P)$ is confluent.*

Corollary 2. *If P is an $\mathcal{I}$ program and* `state(0,0)` $\mapsto^*_{\mathcal{F}(P)} G$*, then G contains at most one* `state` *constraint.*

Given Corollary 1, we will henceforth denote the unique final state of $\mathcal{F}(P)$ when started in state G by $[\mathcal{F}(P)](G)$. Note that, given Proposition 1, $[\mathcal{F}(P)](G)$ contains no `state` constraints.

Proposition 2. *In what follows P is an $\mathcal{I}$ program, $(b, n) \in \Sigma^+_{012} \times \Sigma^+_D$, and G is a state containing no* `state` *constraints.*

1. $[\mathcal{F}(P)_{+n}](G \wedge \texttt{state(0,}n\texttt{)}) = [\mathcal{F}(P)](G \wedge \texttt{state(0,0)})$*, for any numeral n.*
2. $[(b, n) \triangleright \mathcal{F}(P)](G \wedge \texttt{state(}b\texttt{, }n\texttt{\#0)}) = [\mathcal{F}(P)](G \wedge \texttt{state(0,0)})$.
3. $[(b, n) \triangledown \mathcal{F}(P)](G \wedge s) = [\mathcal{F}(P)](G \wedge s) \wedge \texttt{state(}b\texttt{,}n\texttt{)}$*, where s is a* `state` *constraint that occurs in $\mathcal{F}(P)$.*

Intuitively, P is equivalent to $\mathcal{F}(P)$ if they have the same effect; that is, if they map equivalent states to equivalent states.

Definition 5. Let $\mathcal{T} = \langle \mathcal{N}, \mathcal{V}, \mathcal{E}, \mathcal{F} \rangle$ be an $\mathcal{I}$-CHR transformation.

1. An $\mathcal{I}$ store σ is equivalent to a CHR state G, denoted $\sigma \equiv G$, whenever $\sigma(x) = v$ if and only if $G = G' \wedge \texttt{var}(\mathcal{N}(x), v)$, where G' is a state that contains no `state` constraints.
2. An $\mathcal{I}$ configuration γ is equivalent to a CHR state G, denoted $\gamma \equiv G$, if either $\gamma = \langle P, \sigma \rangle$ and $G = G' \wedge \texttt{state(0,0)}$ where $\sigma \equiv G'$, or $\gamma = \sigma \equiv G$.
3. An $\mathcal{I}$ program P_1 is equivalent to a CHR program P_2, denoted $P_1 \equiv P_2$, if for every σ and G where $\langle P_1, \sigma \rangle \equiv G$, $[P_1](\sigma) \equiv [P_2](G)$.

Theorem 2. *For every $\mathcal{I}$ program P and every $\mathcal{I}$-CHR transformation $\mathcal{T} = \langle \mathcal{N}, \mathcal{V}, \mathcal{E}, \mathcal{F} \rangle$, $P \equiv \mathcal{F}(P)$.*

Proof. See the appendix.

4 Methodology for the Conversion of Imperative Algorithms to CHR

In this section, we will informally discuss the methodology to convert an algorithm written in a mini imperative programming language, called $\mathcal{I}$, to an equivalent CHR program.
The basic features of the language $\mathcal{I}$ are:

- Variable declaration and assignment
- Alternation using the `if-then-else` commands
- Iteration using the `while-do` command
- Fixed-size arrays

In the following, we present the implementation of each of these features of imperative programming with the intent of implementing an equivalent program in CHR.

4.1 Variable Declaration

In order to create a storage location for a variable, whenever one is declared, a constraint is added to the constraint store and is given the initial value of this variable as a parameter.
The fragment of code

```
int x = 0;
int y = 7;
```

will be transformed into the following CHR rules:

```
r1 @ state(0, 0) <=> var(x,0), state(0, 1).
r2 @ state(0, 1) <=> var(y,7).
```

The constraint `var/2` is used to store the value of the variables. The head of rule `r1` describes the start of the execution of the program by using a constraint `state/1`. Rule `r1` replaces the first `state` constraint by a `var/2` constraint and a new `state` constraint that triggers the execution of the second rule `r2`.
In general, a *variable declaration* in an imperative programming language can be expressed in CHR using a simplification rule of the form:

$$C_{current} \texttt{ <=> } V, C_{next}$$

where $C_{current}$ and C_{next} are each a constraint `state/1` with a constant unique parameter. V is a constraint used for the purpose of storing the value of the variable being declared. A constraint V is of the form `var(`*variable,value*`)`.

4.2 Variable Assignment

Assigning a value to an already declared variable in CHR is quite similar to the declaration of the variable. However, instead of adding a constraint with the initial value of the variable, we replace the already existing constraint resulting from the last assignment of a value to the variable with a new `var` constraint with the new assignment.
The fragment of code

```
int x = 0;          // asg1
int y = x + 3;      // asg2
```

will be transformed into the following CHR rules:

```
asg1 @ state(0, 0) <=> var(x,0), state(0, 1).
asg2 @ var(x,V) \ state(0, 1) <=> Y = V + 3, var(y,Y).
```

Rule `asg1` performs a variable declaration with an initial value of `0`. Rule `asg2` uses the value of `x` to compute the value of `y` keeping the same information about `x` in the constraint store.
A *variable assignment* in an imperative programming language can be expressed in CHR using a simpagation rule of the form:

$$V \setminus C_{current}, V_{old} \texttt{ <=> } C, V_{new}, C_{next}$$

where V is a conjunction of `var` constraints needed to calculate the new value to be assigned. $C_{current}$ and C_{next} are the same as in the variable declaration rule. V_{old} is the constraint with the old value of the variable which is being assigned a new value, and V_{new} is the same constraint but passed the new value being assigned. C is a conjunction of built-in constraints calculating the new value which is to be assigned. In case the new value being assigned does not depend on values of other variables, both V and C are discarded from the rule and it becomes a simplification rule.

4.3 Alternation

For the fragment of code

```
int a = 10;     // declaration
if(a % 2 == 0)
    a = a * 2; // statement 1
else
    a = a / 2; // statement 2
```

the statements `declaration`, `statement 1`, and `statement 2` are transformed into the following CHR rules:

```
declaration @ state(0, 0) <=> var(a,10), state(0, 1).
statement1 @ state(01, 1#0), var(a,A) <=> NewA = A * 2,  var(a,NewA).
statement2 @ state(02, 1#0), var(a,A) <=> NewA = A // 2,  var(a,NewA).
```

To allow the CHR program to choose whether to execute `statement1` or `statement2` after the `declaration`, we add two rules that are responsible for this choice.

```
goto1 @ var(a,A) \ state(0, 1) <=>
               Tmp = A mod 2, Tmp = 0 |  state(01, 1#0).
goto2 @ var(a,A) \ state(0, 1) <=>
               Tmp = A mod 2, \+(Tmp = 0) |  state(02, 1#0).
```

Alternation in imperative programming, achieved using `if-then-else` expressions can be expressed in CHR using two simpagation rules of the form:

$$V \setminus C_{current} \texttt{ <=> } C \mid C_{ifbranch}$$
$$V \setminus C_{current} \texttt{ <=> } \neg C \mid C_{elsebranch}$$

where V is a conjunction of `var` constraints needed to evaluate the condition of the `if-then-else` expression. $C_{current}$ is the `state` constraint holding the current state, the state that an `if-then-else` expression is to be executed. $C_{ifbranch}$ is a `state` constraint indicating that the next state is the beginning of the body of the `if` block. $C_{elsebranch}$ is a `state` constraint indicating that the next state is the beginning of the body of the `else` block. C is a guard that evaluates the condition of the `if` statement and $\neg C$ is a guard that evaluates to the negation of C.

4.4 Iteration

Consider the following code fragment

```
int a = 0;      // declaration
while(a < 10)
    a = a + 1; // while block
```

The statements `declaration` and `while block` are transformed into the following CHR rules:

```
declaration @ state(0, 0) <=> var(a,0), state(0, 1).
while_block @ state(0, 1#0), var(a,A) <=>
                NewA = A + 1, var(a,NewA), state(0, 1).
```

To evaluate the `while-do` condition and add a repetition mechanism for the block of `while-do` as long as this condition holds and to terminate the iteration otherwise, we add the following rules:

```
continue @ var(a,A) \ state(0, 1) <=> A < 10 | state(0, 1#0).
terminate @ var(a,A) \ state(0, 1) <=> \+(A < 10) | true.
```

Iteration in imperative programming, achieved using `while-do` expressions, can be expressed in CHR using the following rules:

$$V \setminus C_{startwhile} \texttt{<=>} C \mid C_{executebody}$$
$$V \setminus C_{startwhile} \texttt{<=>} \neg C \mid C_{terminatewhile}$$

where V is a conjunction of `var` constraints needed to evaluate the condition of the `while-do` expression. $C_{startwhile}$ is a `state` constraint holding the current state, the state indicating that a `while-do` expression is to be executed. $C_{executebody}$ is a `state` constraint indicating that the next state is the beginning of the body of the `while` block. $C_{terminatewhile}$ is a `state` constraint indicating that the next state is the beginning of the code following the `while-do` expression, i.e. the termination of the `while-do` expression. $C_{endwhile}$ is a `state` constraint indicating that the block of the `while-do` has ended and that the condition of the loop needs to be checked again. C is a guard that evaluates the condition of `while-do` and $\neg C$ is a guard that evaluates to the negation of C.

4.5 Arrays

To simulate arrays in CHR, we represent them using lists and make use of built-in constraints to either access or modify an element of the list. We assume the existence of the predicate `nth0(N, List, Element)` that holds if `Element` is the `N`th value of the list `List`.
Given `nth0/3`, an access to an array element of the form `x = a[3]` is performed in CHR using a rule of the form:

```
arraysR1 @ var(a,A) \ state(B, N), var(x,_) <=>
               nth0(3, A, Element), var(x,Element), state(B, N+1).
```

where `A` is the list containing the values of the array. `arraysR1` is written according to the rule for variable assignment except that `nth0/3` is used to obtain the value of the element to be assigned to `x`.
We also add the following implementation of `replace0/4` to allow for array element assignment:

```
replace0(List, Index, Value, Result):-
   nth0(Index, List, _, Rest), nth0(Index, Result, Value, Rest).
```

`replace0/4` makes use of `nth0(N, List, Element, Rest)`, which behaves similarly to `nth0/3` except that `Rest` is all elements in `List` other than the `Nth` element. The resulting predicate `replace0/4` sets the element at index `Index` of `List` to the value `Value` and gives the list `Result` as the new list with the modified element.
We then represent an assignment of the form `a[3] = x` using a CHR rule of the form:

```
arraysR2 @ var_x(X) \ state(B, N), var(a,A) <=>
   replace0(A, 3, X, NewA),  var(a,NewA), state(B, N+1).
```

`arraysR2` is written according to the rule for variable assignment except that `replace0/4` is used to obtain the new list `NewA` which is the new status of the variable `a`.

Example 1. The following imperative code fragment finds the minimum value in an array `a` of length `n` and stores it in a variable `min`:

```
int temp; int min; int i;
min = a[0];
i = 1;
while(i<n){
   temp = a[i];
   if(temp<min){
      min = temp;
   }
   i = i+1;
}
```

Note that there are no declarations for both `a` and `n` as they are expected to be given as input to the program.
Using the conversion method represented above, the following CHR rules are generated.

```
min1 @ state(0, 0) <=> var(temp,0), state(0, 1).
min2 @ state(0, 1) <=> var(min,0), state(0, 2).
min3 @ state(0, 2) <=> var(i,0), state(0, 3).
min4 @ var(a,A) \ state(0, 3), var(min,MIN) <=>
            nth0(0,A,Newmin), var(min,Newmin), state(0, 4).
min5 @ state(0, 4), var(i,I) <=> NewI = 1, var(i,NewI), state(0, 5).
min6 @ var(i,I), var(n,N) \ state(0, 5) <=> (I < N) | state(0, 5#0).
min7 @ var(i,I), var(n,N) \ state(0, 5) <=> \+(I < N) | true.
min8 @ var(i,I), var(a,A) \ state(0, 5#0), var(temp,TEMP) <=>
          nth0(I,A,NewTemp), var(temp,NewTemp), state(0, 5#1).
min9 @ var(temp,TEMP), var(min,MIN) \ state(0, 5#1) <=>
          (TEMP < MIN) | state(01, 5#1#0).
min10 @ var(temp,TEMP), var(min,MIN) \ state(0, 5#1) <=>
          \+(TEMP < MIN) | state(0, 5#2).
```

```
min11 @ var(temp,TEMP) \ state(01, 5#1#0), var(min,MIN) <=>
          NewMIN = TEMP, var(min,NewMIN), state(0, 5#2).
min12 @ state(0, 5#2), var(i,I) <=> NewI = I + 1, var(i,NewI), state(0, 5).
```

To run the CHR program, the following goal is used to pass the necessary constraints and trigger the first rule:

```
var(a, A), length(A, N), var(n, N), state(0, 0).
```

5 Conclusion and Future Work

The context of this paper was a presentation of a conversion methodology to generate rule-based programs from imperative programs. Given a proof of equivalence between both programs, it can be implied that both programs will function alike. The purpose of this generation is to use the rule-based programs in proving properties such as correctness and confluence, which subsequently proves these properties for the imperative programs. We selected CHR as a rule-based language due to the existence of results for proving several properties of programs. There are several implementations of global constraint solvers which are of an imperative nature. An additional use for the implemented conversion methodology could be to automatically generate solvers for these global constraints instead of their manual implementation. The benefit of this conversion is to exploit the flexibility and expressivity of CHR.
An interesting direction for future work is to investigate how the proposed approach can be combined with previous approaches, e.g. [3, 4]. To improve the efficiency of the generated solvers the set of rules should be reduced. The operational equivalence results of CHR programs [1] can be applied to find out the redundant rules. However, in most of the cases, the rules are not redundant but they can be reduced by merging two or more rules in one.

References

1. S. Abdennadher and T. Frühwirth. Operational equivalence of CHR programs and constraints. In *5th International Conference on Principles and Practice of Constraint Programming, CP99*, LNCS 1713, 1999.
2. G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.
3. F. Raiser. Semi-automatic generation of chr solvers for global constraints. In *14th International Conference on Principles and Practice of Constraint Programming*, 2008.
4. I. Sobhi, S. Abdennadher, and H. Betz. Constructing rule-based solvers for intentionally-defined constraints. In *Special Issue on Recent Advances in Constraint Handling Rules*. 2008.

Appendix: Proof of Theorem 2

Let σ be an $\mathcal{I}$ store and let $G = G' \wedge \texttt{state(0,0)}$ be a CHR state, such that $\langle P, \sigma \rangle \equiv G$. Given Definition 5, it suffices to show that $[P](\sigma) \equiv [\mathcal{F}(P)](G)$. We shall prove this result by structural induction on the structure of P.

Basis. We have two base cases.

1. $P = dt\ x\texttt{;}$
   Given the semantics of $\mathcal{I}$, $[P](\sigma) = \sigma^{x \mapsto Default(dt)}$. By Definition 4,
   $$[\mathcal{F}(P)](G) = G' \wedge \texttt{var(}\mathcal{N}(x), Default(dt)\}$$
   It follows from Definition 5 that $[P](\sigma) \equiv [\mathcal{F}(P)](G)$.
2. $P = x \texttt{ = } e\texttt{;}$
   From the semantics of $\mathcal{I}$, $[P](\sigma) = \sigma^{x \mapsto [\![e]\!]^\sigma}$. By Definition 4, if $G' = G'' \wedge \texttt{var(}\mathcal{V}(x), \sigma(x)\texttt{)}$ then
   $$[\mathcal{F}(P)](G) = G'' \wedge \texttt{var(}\mathcal{N}(x), [\![\mathcal{E}(e)]\!]^G\texttt{)}$$
   Since $\langle P, \sigma \rangle \equiv G$, it follows that, for every identifier y in e, $\texttt{var(}\mathcal{N}(y), \sigma(y)\texttt{)}$ is a constraint in G''. Hence, given the conjunction V of constraints in the head of the only rule in $\mathcal{F}(P)$ (case 2 in Figure 2), the variable $\mathcal{V}(y)$ is bound to $\sigma(y)$, for every identifier y in e. Thus, from the definition of $\mathcal{E}$, it follows that $[\![e]\!]^\sigma = [\![\mathcal{E}(e)]\!]^G$. Consequently, $[P](\sigma) \equiv [\mathcal{F}(P)](G)$.

Induction hypothesis. P_1 and P_2 are $\mathcal{I}$ programs with $P_1 \equiv \mathcal{F}(P_1)$ and $P_2 \equiv \mathcal{F}(P_2)$.

Induction step. We have three recursive rules in the definition of P.

1. $P = \texttt{if } e\ \{P_1\}\ \texttt{else}\ \{P_2\}$
   Suppose that $[\![e]\!]^\sigma$ is true. In this case, $[P](\sigma) = [P_1](\sigma)$ as per the operational semantics of $\mathcal{I}$. Now, consider the rule
   $$V\ \texttt{\textbackslash state(0,0) <=>}\ \mathcal{E}(e)\ \texttt{| state(01,0\#0)}$$
   in $\mathcal{F}(P)$ (case 3 in Figure 2). Similar to case 2 in the proof of the basis, $[\![e]\!]^\sigma = [\![\mathcal{E}(e)]\!]^G$. Thus, the above rule is applicable to G. Furthermore, from Theorem 1, the above rule is the only rule applicable to G. Hence, $G \mapsto_{\mathcal{F}(P)} G_1$, where
   $$G_1 = G' \wedge \texttt{state(01,0\#0)}$$
   Since the set of `state` constraints occurring in $\texttt{(01,0)} \triangleright \mathcal{F}(P_1)$ is disjoint from the set of `state` constraints in the rest of $\mathcal{F}(p)$, and since `state(01,0#0)` occurs in $\texttt{(01,0)} \triangleright \mathcal{F}(P_1)$, then
   $$[\mathcal{F}(P)](G) = [\mathcal{F}(P)](G_1) = [\texttt{(01,0)} \triangleright \mathcal{F}(P_1)](G_1) \quad (1)$$

From Proposition 2 it follows that

$$[\texttt{(01,0)} \triangleright \mathcal{F}(P_1)](G_1) = [\mathcal{F}(P_1)](G' \wedge \texttt{state(0,0)})$$

But $G' \wedge \texttt{state(0,0)} = G$. Therefore, given (1), $[\mathcal{F}(P)](G) = [\mathcal{F}(P_1)](G)$. From the induction hypothesis it follows that

$$[\mathcal{F}(P)](G) \equiv [P_1](\sigma) = [P](\sigma)$$

The proof is similar, *mutatis mutandis*, in case $[\![e]\!]^\sigma$ is false.

2. $P = \texttt{while}\ e\ \{P_1\}$
We prove the equivalence by induction on the number i of iterations of the loop. If $i = 0$, then it must be that $[\![e]\!]^\sigma$ is false. According to the semantics of I, $[P](\sigma) = \sigma$. We can show (in a fashion similar to that of proving case 2 of the basis) that $[\![e]\!]^\sigma = [\![\mathcal{E}(e)]\!]^G$. Thus, the only rule in $\mathcal{F}(P)$ applicable to G is the rule

$$V\ \texttt{\textbackslash state(0,0) <=> \textbackslash +}\mathcal{E}(e)\ \texttt{| true}$$

Since this is a terminal rule, then $[\mathcal{F}(P)](G) = G'$. By Definition 5, $G' \equiv \sigma$. Thus, $[P](\sigma) = [\mathcal{F}(P)](G)$.
As an induction hypothesis, suppose that whenever σ is such that $i = k$, $[P](\sigma) \equiv [\mathcal{F}(P)](G)$. Now, let σ be a store, such that $i = k+1$. Clearly, $[\![e]\!]^\sigma$ is true. Thus, $[P](\sigma) = [P]([P_1](\sigma))$, where $[P_1](\sigma)$ is a store for which $i = k$. It could be shown that $[\![\mathcal{E}(e)]\!]^G = [\![e]\!]^\sigma$. Thus, the only rule in $\mathcal{F}(P)$ applicable to G is the rule

$$V\ \texttt{\textbackslash state(0,0) <=>}\ \mathcal{E}(e)\ \texttt{| state(0,0\#0)}$$

Thus, $G \mapsto_{\mathcal{F}(P)} G_1$, where

$$G_1 = G' \wedge \texttt{state(0,0\#0)}$$

Now, `state(0,0)` is the only `state` constraint occurring both in $\texttt{(0,0)} \triangledown (\texttt{(0,0)} \triangleright \mathcal{F}(P_1))$ and the rest of $\mathcal{F}(P)$. Moreover, according to the definition of $\triangledown$, `state(0,0)` occurs only in the bodies of rules in $\texttt{(0,0)} \triangledown (\texttt{(0,0)} \triangleright \mathcal{F}(P_1))$. Hence,

$$[\mathcal{F}(P)](G) = [\mathcal{F}(P)](G_1) = [\mathcal{F}(P)]([\texttt{(0,0)} \triangledown (\texttt{(0,0)} \triangleright \mathcal{F}(P_1))](G_1)) \qquad (2)$$

From Proposition 2 it follows that

$$[\texttt{(0,0)} \triangledown (\texttt{(0,0)} \triangleright \mathcal{F}(P_1))](G_1) = [\mathcal{F}(P_1)](G' \wedge \texttt{state(0,0)}) \wedge \texttt{state(0,0)}$$

But $G' \wedge \texttt{state(0,0)} = G$. Therefore, given (2), it follows that

$$[\mathcal{F}(P)](G) = [\mathcal{F}(P)]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,0)}) \qquad (3)$$

By the induction hypothesis, $[\mathcal{F}(P_1)](G) \equiv [P_1](\sigma)$. Thus, from Definition 5, $[\mathcal{F}(P_1)](G) \wedge \texttt{state(0,0)} \equiv \langle P, [P_1](\sigma)\rangle$. Since $[P_1](\sigma)$ is a store for which $i = k$, then $[P](\sigma) = [P]([P_1](\sigma)) = [\mathcal{F}(P)]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,0)})$. Consequently, given (3), $[P](\sigma) \equiv [\mathcal{F}(P)](G)$.

3. $P = P_1\ P_2$
Let $\texttt{state(0,}n'\texttt{)}$ be the unique maximal constraint in $\mathcal{F}(P_1)$. Given Definition 3, $\texttt{state(0,0)}$ occurs only in the head of a rule in $\texttt{(0,}n\texttt{)}\triangledown\mathcal{F}(P_1)$, where $n = n'_{+1}$. The only constraint occurring both in $\texttt{(0,}n\texttt{)}\triangledown\mathcal{F}(P_1)$ and $\mathcal{F}(P_2)_{+n}$ is $\texttt{state(0, }n\texttt{)}$. However, it only occurs in the bodies of rules of $\texttt{(0,}n\texttt{)}\triangledown\mathcal{F}(P_1)$. Hence,

$$[\mathcal{F}(P)](G) = [\mathcal{F}(P)]([\texttt{(0,}n\texttt{)}\triangledown\mathcal{F}(P_1)](G)) \tag{4}$$

By Proposition 2,

$$[\texttt{(0,}n\texttt{)}\triangledown\mathcal{F}(P_1)](G) = [\mathcal{F}(P_1)](G) \wedge \texttt{state(0,}n\texttt{)}$$

Hence,

$$[\mathcal{F}(P)](G) = [\mathcal{F}(P)]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,}n\texttt{)}) \tag{5}$$

Now, the constraint $\texttt{state(0,}n\texttt{)}$ occurs only in the head of rules in $\mathcal{F}(P_2)_{+n}$. In addition, other $\texttt{state}$ constraints in $\mathcal{F}(P_2)_{+n}$ do not occur elsewhere in $\mathcal{F}(P)$. Hence,

$$[\mathcal{F}(P)]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,}n\texttt{)}) = [\mathcal{F}(P_2)_{+n}]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,}n\texttt{)})$$

By Proposition 2,

$$[\mathcal{F}(P_2)_{+n}]([\mathcal{F}(P_1)](s) \wedge \texttt{state(0,}n\texttt{)}) = [\mathcal{F}(P_2)]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,0)})$$

From (5) it follows that

$$[\mathcal{F}(P)](G) = [\mathcal{F}(P_2)]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,0)})$$

But, given the induction hypothesis, $[\mathcal{F}(P_1)](G) \equiv [P_1](\sigma)$. Thus, from Definition 5, $[\mathcal{F}(P_1)](G) \wedge \texttt{state(0,0)} \equiv \langle P_2, [P_1](\sigma)\rangle$. It, thus, also follows from the induction hypothesis that

$$[\mathcal{F}(P)](G) \equiv [P_2]([P_1](\sigma))$$

Hence, given the semantics of $\mathcal{I}$,

$$[\mathcal{F}(P)](G) \equiv [P](\sigma)$$

□

Acknowledgments. We would like to thank Abdellatif Olama for preliminary work done in the same field.

Persistent Constraints in Constraint Handling Rules

Hariolf Betz, Frank Raiser, and Thom Frühwirth

Faculty of Engineering and Computer Sciences, Ulm University, Germany
firstname.lastname@uni-ulm.de

Abstract. In the most abstract definition of its operational semantics, the declarative and concurrent programming language CHR is trivially non-terminating for a significant class of programs. Common refinements of this definition, in closing the gap to real-world implementations, compromise on declarativity and/or concurrency. Building on recent work and the notion of persistent constraints, we introduce an operational semantics avoiding trivial non-termination without compromising on its essential features.

1 Introduction

Constraint Handling Rules [1] (CHR) is a declarative, multiset- and rule-based programming language suitable for concurrent execution and powerful program analyses. Several operational semantics have been proposed for CHR [1], situated between an abstract and an implementation level.

The most abstract operational semantics – constituting the basis for most other variants – is called the "very abstract" operational semantics and denoted as ω_{va}. It is firmly rooted in first-order logic, defining a state transition system but providing no execution model. Hence, it is oblivious to termination issues, unfavorably causing the class of rules known as *propagation rules* to induce *trivial non-termination*.

The de-facto standard in avoiding trivial non-termination is set by the operational semantics ω_t [1], providing the basis for most available CHR implementations. In ω_t, every propagation rule is applicable only once to a specific combination of constraints, thus avoiding trivial non-termination. This is realized by keeping a *propagation history* – also called *token store* – in the program state.

On the downside, token stores break with declarativity: Two states that differ only in their token stores may exhibit different operational behaviour while sharing the same logical reading. Therefore, we consider token stores as *non-declarative elements* in program states. The propagation history also hinders effective concurrent execution of CHR programs, as it has to be distributed adequately.

With concurrency in mind, [2] defines operational semantics based on sets rather than multisets, which effectively avoids trivial non-termination without

recurring to token stores. With nine transition rules, however, the resulting state transition system is unusually complex, thus reducing clarity and complicating formal proofs. Furthermore, abandoning multiset semantics is a severe break with existing approaches and the presence of non-declarative elements remains. Notably, the authors of [2] reckon that any "reasonable [multiset-based] semantics" for CHR requires a propagation history. This work is proof to the contrary.

Recent work on linear logical algorithms [3] and the close relation of CHR to linear logic [4] suggest a novel approach: we introduce the notion of *persistent constraints* to CHR, a concept reminiscent of "banged" propositions in linear logic. Persistent constraints provide a finite representation of the results of any number of propagation rule firings. Furthermore, we explicitly define our state transition system as irreflexive. It shows that, in combination, these ideas solve the problem of trivial non-termination.

Building on earlier work in [5], we thus develop the operational semantics $\omega_!$ for CHR in this work. As opposed to existing approaches, it achieves a high degree of declarativity whilst preserving the potential of ω_{va} for effective concurrent execution. Its state transition system requires only two rules, such that each transition step corresponds to a CHR rule application, thus facilitating formal reasoning over programs.

In Section 2 we introduce CHR and present its operational semantics ω_{va} and ω_t. We then introduce $\omega_!$ and discuss its properties in Section 3, before comparing it to other approaches in Section 4. Finally, in Section 5 we conclude and consider possible directions of future work.

2 Preliminaries

This section introduces CHR and its two most important operational semantics. A complete listing of available operational semantics is given in [6]. In this work, we concentrate on the so-called *very abstract* operational semantics ω_{va} and *theoretical* operational semantics ω_t. A *refined* variant of the latter – introduced in [7] and denoted as ω_r – reduces several sources of non-determinism and is the de-facto standard for CHR implementations.

The very abstract operational semantics ω_{va} is the semantics with the simplest state definition and fewest restrictions on rule applications. We introduce it in Section 2.2 before presenting its refinement into the theoretical operational semantics ω_t in Section 2.3.

2.1 The Syntax of CHR

Constraint Handling Rules distinguishes between two kinds of constraints: *user-defined* (or simply CHR) constraints and *built-in* constraints. The latter are processed by a predefined solver implementing a complete and decidable constraint theory $\mathcal{CT}$.

CHR itself is an advanced rule-based rewriting language. Its eponymous rules are of the form

$$r@\ H_1 \backslash H_2 \Leftrightarrow G \mid B_c, B_b$$

where H_1 and H_2 are multisets of user-defined constraints, called the *kept head* and *removed head*, respectively. The *guard* G is a conjunction of built-in constraints and the *body* consists of a conjunction of built-in constraints B_b and a multiset of user-defined constraints B_c. The *rule name* r is optional and may be omitted along with the @ symbol.

Intuitively speaking, a rule is applicable, if a part of the current state can be matched to all head constraints such that the guard is satisfied. Application of a rule removes from the state the constraints matched to H_2 and adds the guards and the body constraints to the state. In this work, we put special emphasis on the class of rules where $H_2 = \emptyset$, called *propagation rules*. Propagation rules can be written alternatively as $H_1 \Rightarrow G \mid B_c, B_b$.

2.2 Very Abstract Operational Semantics

The very abstract operational semantics ω_{va} [1] is the most general specification of an operational semantics for CHR. Its state definition only contains one component and the transition system is given by a single rule.

Definition 1 (ω_{va}-State).
A ω_{va}-state $\sigma_{va} = \langle C \rangle$ *is a conjunction* C *of built-in and CHR constraints.*

The only allowed state transition in ω_{va} is the application of a CHR rule.

Definition 2 (ω_{va}-Transition). *Let* $r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B$ *be an instance of a rule* $r \in P$ *with new local variables* $\bar{x}$ *and* $\mathcal{CT} \models \forall(\mathbb{G} \rightarrow \exists \bar{x}.G)$. *Then*

$$\langle H_1 \wedge H_2 \wedge \mathbb{G} \rangle \rightarrowtail_{\omega_{va}} \langle H_1 \wedge G \wedge B \wedge \mathbb{G} \rangle$$

Note that the above definition, based on instantiation of rules, requires all arguments of constraints in H_1 and H_2 to be variables. Ground terms can be realized by an equality constraint in the guard, and similarly, multiple occurrences of the same variable are not allowed, but have to be realized via guard constraints. This restriction simplifies the formulation of ω_{va}, but it also makes for less elegant programs. Most derived operational semantics – including ω_t, ω_{set}, and $\omega_!$ discussed herein – avoid this restriction.

An inherent problem of ω_{va} is its behavior with respect to propagation rules: If a state can fire a propagation rule once, it can do so again and again, ad infinitum. In the literature, this problem is referred to as *trivial non-termination* of propagation rules.

2.3 Theoretical Operational Semantics

The theoretical operational semantics ω_t [1, 6] is based on the idea of using a token store to avoid trivial non-termination. Under ω_t, a propagation rule can only be applied once to each combination of constraints matching the head. Hence, the token store keeps a history of fired propagation rules, which is based on constraint identifiers.

Definition 3 (Identified CHR Constraints).
An identified CHR constraint $c\#i$ *is a CHR constraint* c *associated with a unique integer* i*, the* constraint identifier*. We introduce the functions* $\mathrm{chr}(c\#i) = c$ *and* $\mathrm{id}(c\#i) = i$*, and extend them to sequences and sets of identified CHR constraints in the obvious manner.*

The definition of a CHR state in ω_t is more complicated, because identified constraints are distinguished from unidentified constraints and the token store is added [1].

Definition 4 (ω_t-State).
A ω_t-state *is a tuple of the form* $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n^{\mathbb{V}}$ *where the* goal (store) $\mathbb{G}$ *is a multiset of constraints, the* CHR (constraint) store $\mathbb{S}$ *is a set of numbered CHR constraints, the* built-in (constraint) store $\mathbb{B}$ *is a conjunction of built-in constraints. The* propagation history *(or* token store*)* $\mathbb{T}$ *is a set of tuples* (r, I)*, where* r *is the name of a propagation rule and* I *is an ordered sequence of the identifiers of constraints that matched the head constraints of* r *in a previous application of* r*. Finally, the set* $\mathbb{V}$ *of* global variables *contains the variables that occur in the initial goal.*

This state definition entails a more complicated transition system, consisting of the following three types of transitions:

Definition 5 (ω_t-Transitions).

1. **Solve.** $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n^{\mathbb{V}} \rightarrowtail_{\omega_t} \langle \mathbb{G}, \mathbb{S}, \mathbb{B}', \mathbb{T} \rangle_n^{\mathbb{V}}$
   where c *is a built-in constraint and* $\mathcal{CT} \models \forall((c \wedge \mathbb{B}) \leftrightarrow B')$.
2. **Introduce.** $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n^{\mathbb{V}} \rightarrowtail_{\omega_t} \langle \mathbb{G}, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}^{\mathbb{V}}$
   where c *is a CHR constraint.*
3. **Apply.** $\langle \mathbb{G}, H_1 \cup H_2 \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n^{\mathbb{V}} \rightarrowtail_{\omega_t} \langle B \uplus \mathbb{G}, H_1 \cup \mathbb{S}, \mathrm{chr}(H_1) = H_1' \wedge \mathrm{chr}(H_2) = H_2' \wedge G \wedge \mathbb{B}, \mathbb{T} \cup \{(r, \mathrm{id}(H_1) + \mathrm{id}(H_2))\} \rangle_n^{\mathbb{V}}$
   where $r \ @ \ H_1' \setminus H_2' \Leftrightarrow G \mid B$ *is a fresh variant of a rule in* P *with fresh variables variables* $\bar{x}$ *such that* $\mathcal{CT} \models \exists(\mathbb{B}) \wedge \forall(\mathbb{B} \rightarrow \exists \bar{x}(\mathrm{chr}(H_1) = H_1' \wedge \mathrm{chr}(H_2) = H_2' \wedge G))$ *and* $(r, \mathrm{id}(H_1) + \mathrm{id}(H_2)) \notin \mathbb{T}$.

By construction, ω_t restricts the number of applications of a propagation rule for each given combination of head constraints to one. This stands in contrast to its declarative reading as well as its execution under ω_{va}, where a propagation rule may be applied any number of times. The ω_t-state also contains non-declarative elements: the set of identified CHR constraints, the propagation history, and the integer n used for identification.

3 Operational Semantics with Persistent Constraints

We now introduce our proposal for an operational semantics $\omega_!$ with persistent constraints. It is based on three important ideas:

1. In ω_{va}, the body of a propagation rule can be generated any number of times given that the corresponding head constraints are present in the store. In order to give consideration to this theoretical behavior while avoiding trivial non-termination, we introduce those body constraints as so-called *persistent constraints*. A persistent constraint can be regarded as a finite representation of a very large, though unspecified number of identical constraints. For a proper distinction, constraints that are non-persistent are henceforth called *linear* constraints.
2. Not only the bodies of propagation rules can be generated indefinitely many times in ω_{va}. Consider the following program:

$$\begin{aligned} &\text{r1} \ @ \ a(X) \Longrightarrow b(X) \\ &\text{r2} \ @ \ b(X) \ \Leftrightarrow \ c(X) \end{aligned}$$

   If executed with a goal $a(0)$, this program can generate an arbitrary number of constraints of the form $b(0)$. As a consequence of this, it can also generate arbitrarily many constraints $c(0)$. To take these indirect consequences of propagation rules into accout, we introduce a rule's body constraints as persistent, whenever its removed head can be matched completely with persistent constraints.
3. As a persistent constraint represents an arbitrary number of identical constraints, we consider several occurences of a persistent constraint as idempotent. We now adapt our execution model such that a transition takes place only if the post-transition state is not equivalent to the pre-transition state. By the thus irreflexive transition system, we avoid trivial non-termination of propagation rules.

To realize the first two ideas, we adapt the definion of states in $\omega_!$ with respect to ω_t: The goal store $\mathbb{G}$ of ω_t-states is split into a store $\mathbb{L}$ of linear constraints and a store $\mathbb{P}$ of persistent constraints. The components $\mathbb{B}$ and $\mathbb{V}$ of ω_t-states are retained, but the token-related components $\mathbb{S}, \mathbb{T}$, and n are eliminated.

Definition 6 ($\omega_!$-State).
A $\omega_!$-state *is a tuple of the form* $\langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle$*, where* $\mathbb{L}$ *and* $\mathbb{P}$ *are multisets of CHR constraints called the* linear (CHR) store *and* persistent (CHR) store*, respectively.* $\mathbb{B}$ *is a conjunction of built-in constraints and* $\mathbb{V}$ *is a set of variables.*

We define the notion of *strictly local variables* which we will apply below.

Definition 7 (Strictly local variables). *Let* $\sigma = \langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle$ *be an* $\omega_!$ *state. Then we call the variables occurring in* $\mathbb{B}$ *but not in* $\mathbb{L}$*,* $\mathbb{P}$*, or* $\mathbb{V}$ *the* strictly local variables *of* σ*.*

To realize the third idea, we adapt the equivalence relation between $\omega_!$-states. The following definition of state equivalence is based on [5], adding condition 5 to handle idempotence of persistent constraints.

Definition 8 (Equivalence of $\omega_!$-States).
Equivalence between $\omega_!$*-states is the smallest equivalence relation* $\equiv$ *over* $\omega_!$*-states that satisfies the following conditions:*

1. (Equality as Substitution) *Let X be a variable, t be a term and $\doteq$ the syntactical equality relation.*

$$\langle \mathbb{L}, \mathbb{P}, X \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle \equiv \langle \mathbb{L}\,[X/t]\,, \mathbb{P}\,[X/t]\,, X \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle$$

2. (Transformation of the Constraint Store) *If $\mathcal{CT} \models \exists \bar{s}.\mathbb{B} \leftrightarrow \exists \bar{s}'.\mathbb{B}'$ where $\bar{s}, \bar{s}'$ are the strictly local variables of $\mathbb{B}, \mathbb{B}'$, respectively, then:*

$$\langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle \equiv \langle \mathbb{L}, \mathbb{P}, \mathbb{B}', \mathbb{V} \rangle$$

3. (Omission of Non-Occurring Global Variables) *If X is a variable that does not occur in $\mathbb{L}$, $\mathbb{P}$ or $\mathbb{B}$ then:*

$$\langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \{X\} \cup \mathbb{V} \rangle \equiv \langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle$$

4. (Equivalence of Failed States)

$$\langle \mathbb{L}, \mathbb{P}, \bot, \mathbb{V} \rangle \equiv \langle \mathbb{L}', \mathbb{P}', \bot, \mathbb{V}' \rangle$$

5. (Contraction)

$$\langle \mathbb{L}, P \uplus P \uplus \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle \equiv \langle \mathbb{L}, P \uplus \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle$$

Based on this definition of state equivalence, we define CHR as a rewrite system over equivalence classes of states. Let Σ be the set of all $\omega_!$-states, then the transition relation $\rightarrowtail_{\omega_!}$ satisfies $\rightarrowtail_{\omega_!} \subseteq (\Sigma/\equiv) \times (\Sigma/\equiv)$. Note that we use the term *state* interchangably to denote $\omega_!$-states *per se*, as well as equivalence classes over such states. As discussed above, we require that a post-transition state τ needs to be different to the pre-transition state σ, thus making the transition relation irreflexive.

Definition 9 ($\omega_!$-Transitions).

$$\frac{\begin{array}{c} r \;@\; (H_1^l \uplus H_1^p) \backslash (H_2^l \uplus H_2^p) \Leftrightarrow G \mid B_c, B_b \quad H_2^l \neq \emptyset \quad \sigma \neq \tau \\ \sigma = [\langle H_1^l \uplus H_2^l \uplus \mathbb{L}, H_1^p \uplus H_2^p \uplus \mathbb{P}, G \wedge \mathbb{B}, \mathbb{V} \rangle] \end{array}}{\rightarrowtail_{\omega_!} [\langle H_1^l \uplus B_c \uplus \mathbb{L}, H_1^p \uplus H_2^p \uplus \mathbb{P}, G \wedge \mathbb{B} \wedge B_b, \mathbb{V} \rangle] = \tau}$$

$$\frac{\begin{array}{c} r \;@\; (H_1^l \uplus H_1^p) \backslash H_2^p \Leftrightarrow G \mid B_c, B_b \quad \sigma \neq \tau \\ \sigma = [\langle H_1^l \uplus \mathbb{L}, H_1^p \uplus H_2^p \uplus \mathbb{P}, G \wedge \mathbb{B}, \mathbb{V} \rangle] \end{array}}{\rightarrowtail_{\omega_!} [\langle H_1^l \uplus \mathbb{L}, H_1^p \uplus H_2^p \uplus B_c \uplus \mathbb{P}, G \wedge \mathbb{B} \wedge B_b, \mathbb{V} \rangle] = \tau}$$

Note that in a concurrent environment, the second inference rule can be executed without any restrictions: As persistent constraints cannot be removed by other rule applications every process can independently use them to fire rules. The first inference rule can be executed concurrently, if is is guaranteed, that rule applications do not interfere, in the manner described in [8].

3.1 Termination Behavior

Our proposed operational semantics $\omega_!$ results in a termination behavior different from ω_t and ω_{va}. Compared to ω_{va}, the problem of trivial non-termination is solved in $\omega_!$. In comparison with ω_t, we find that there exist programs that terminate under $\omega_!$ but not under ω_t, and vice versa.

Example 1. Consider the following straightforward CHR program for computing the transitive hull of a graph represented by edge constraints $e/2$:

$$\mathrm{t} \ @ \ e(X,Y), e(Y,Z) \Longrightarrow e(X,Z)$$

Due to the presence of propagation rules, this program is not terminating under ω_{va}. Under ω_t, termination depends on the initial goal: It is shown in [9] that this program terminates for acyclic graphs. However, goals containing graphs with cycles, like $\langle (e(1,2), e(2,1)), \emptyset, \top, \emptyset \rangle_0^{\emptyset}$, result in nontermination.

When executed under $\omega_!$, the previous goal terminates after computing the transitive hull.

$$\begin{aligned} & \langle \{e(1,2), e(2,1)\}, \emptyset, \top, \emptyset \rangle \\ \rightarrowtail_{\omega_!}^{t} \ & \langle \{e(1,2), e(2,1)\}, \{e(1,1)\}, \top, \emptyset \rangle \\ \rightarrowtail_{\omega_!}^{*} \ & \langle \{e(1,2), e(2,1)\}, \{e(1,1), e(1,2), e(2,1), e(2,2)\}, \top, \emptyset \rangle \not\rightarrowtail_{\omega_!} \end{aligned}$$

In fact, we can show that the above program terminates under $\omega_!$ for all possible inputs.

Proposition 1. *Under $\omega_!$, the transitive hull program terminates for every possible input.*

Proof. The only rule t propagates e constraints, which are necessarily persistent. The propagated constraints contain only the arguments X, Z, already recieved as parameters. Hence, no new arguments are introduced. Any given initial state contains only a finite number of arguments. Therefore, only finitely many different e constraints can be built from these arguments. As rule application is irreflexible, the computation therefore has to stop after a finite number of transition steps. □

Example 2. Consider the following exemplary CHR program:

$$\begin{aligned} \mathrm{r1} \ @ \ a & \Longrightarrow b \\ \mathrm{r2} \ @ \ c(X), b & \Leftrightarrow c(X+1) \end{aligned}$$

The above program terminates under ω_t and ω_r: There can only be a finite number of a-constraints in the initial goal, hence rule $r1$ only creates a finite number of b-constraints. This, in turn, allows only a finite number of increments being made by rule $r2$.

In contrast, our proposed semantics $\omega_!$ results in the above program being non-terminating, as the following infinite derivation shows:

$$\begin{aligned} & \langle \{a, c(X)\}, \emptyset, \top, \{X\} \rangle \\ \rightarrowtail_{\omega_!}^{r1} \ & \langle \{a, c(X)\}, \{b\}, \top, \{X\} \rangle \\ \rightarrowtail_{\omega_!}^{r2} \ & \langle \{a, c(X+1)\}, \{b\}, \top, \{X\} \rangle \\ \rightarrowtail_{\omega_!}^{r2} \ & \langle \{a, c(X+2)\}, \{b\}, \top, \{X\} \rangle \rightarrowtail_{\omega_!}^{r2} \ldots \end{aligned}$$

3.2 Limitations of the current approach

The approach specified in this work entails a significant discrepance w.r.t. ω_{va} when fresh variables are introduced in rule bodies. For example, consider the following program:

$$\begin{array}{ll} \text{r1} \ @ \ a & \Longrightarrow b(X) \\ \text{r2} \ @ \ b(X), b(X) & \Leftrightarrow \ c \end{array}$$

If executed with the initial goal a, this program would cause the following infinite derivation under ω_{va}:

$$\begin{array}{l} \quad \langle a \rangle \\ \rightarrowtail^{r1}_{\omega_{va}} \ \langle a \wedge b(X') \rangle \\ \rightarrowtail^{r1}_{\omega_{va}} \ \langle a \wedge b(X') \wedge b(X'') \rangle \rightarrowtail^{r1}_{\omega_{va}} \ \ldots \end{array}$$

The variables $X', X'', \ldots$ each are explicitly distinct from each other and from the variable X which occurs in the rule body. Thus, it is not possible to derive the constraint c from goal a under ω_{va}.

Under $\omega_!$, however, the following derivation is possible:

$$\begin{array}{l} \quad \langle \{a\}, \emptyset, \top, \emptyset \rangle \\ \rightarrowtail^{r1}_{\omega_!} \ \langle \{a\}, \{b(X')\}, \top, \emptyset \rangle \equiv \langle \{a\}, \{b(X'), b(X')\}, \top, \emptyset \rangle \\ \rightarrowtail^{r2}_{\omega_!} \ \langle \{a\}, \{b(X'), c\}, \top, \emptyset \rangle \equiv \langle \{a\}, \{b(X'), b(X'), c\}, \top, \emptyset \rangle \end{array}$$

Therefore, the current formulation of the operational semantics $\omega_!$ for CHR is only applicable to range-restricted programs, i.e. rules that do not introduce new variables in their bodies.

4 Related Work

In [2] the set-based semantics ω_{set} has been introduced. Its development was, amongst other considerations, driven by the intention to eliminate the propagation history. Besides addressing the problem of trivial non-termination in a novel manner, it reduces non-determinism in a way closely resembling ω_r.

Similarly to ω_t, a propagation rule is only fired once for a possible matching in ω_{set}. Unlike ω_t, however, additional single firings are possible in ω_{set}. These depend on the further development of the built-in store. Nonetheless, there remains a limit on the number of rule firings.

Our approach to eliminate trivial non-termination consists of the combination of two essential components: an irreflexive state transition system and persistent constraints. Using irreflexivity for termination is a straightforward consequence of adding persistent constraints. The separation of propositions, or constraints, into linear and persistent ones was inspired by the work on linear logical algorithms in [3]. CHR differs significantly from linear logical algorithms, because of its support for built-in constraints, their underlying constraint theory and interaction with user-defined constraints.

Figure 1 relates the different operational semantics for CHR in a hierarchical order. At the root, we have the abstract semantics ω_{va} from which the other

semantics are derived. The operational semantics ω_t introduces token stores to solve the trivial non-termination problem. Numerous extensions of it have been published [6], as indicated by a dotted elements in the figure. In particular, the operational semantics ω_r [7] is an important specialization of ω_t, as it is the foundation of most existing CHR implementations. Again, numerous extensions apply to ω_r.

In the right-hand column, we have placed ω_{set}, which, is another specialization of ω_{va}. Having identified shortcomings of the token store approach, the authors of [2] give a set-based operational semantics for CHR instead.

By placing our approach into the middle column, we emphasize that it is a distinct approach to the trivial non-termination problem. The remaining entries in Figure 1 under the category of persistent semantics indicate that $\omega_!$ – analogously to ω_t – can serve as the basis for a multitude of extensions and implementation-specific variants.

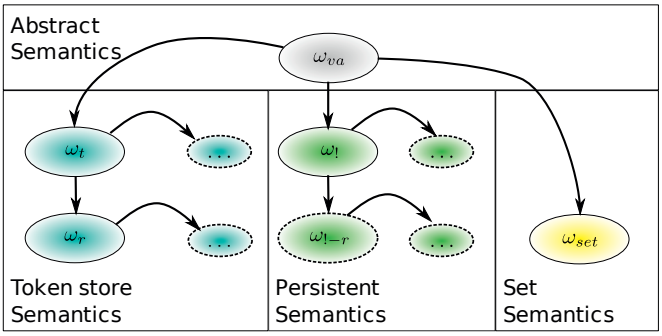


Fig. 1. Relations between Operational Semantics of CHR

The benefits of $\omega_!$ in comparison with the other cited approaches are summarized in Figure 2. In the following, we discuss the different evaluation criteria and the corresponding results given in Figure 2.

Termination on propagation rules: While forming the basis for all other semantics, ω_{va} itself is a theoretical construct, made impractical by its trivial non-termination on propagation rules. Derived semantics apply various strategies to avoid this problem, as outlined above.

Effective concurrency: In ω_t and ω_r, the necessity to distribute token stores constitutes an impediment to effective concurrent execution. We deem ω_{va}, ω_{set}, and $\omega_!$ effective bases for concurrent execution, as they do not introduce auxiliary elements causing inference between rule applications.

Declarative states: In ω_t, ω_r, and ω_{set}, program states contain elements that have no correspondence in the declarative reading. States in $\omega_!$ and ω_{va} avoid such non-declarative elements, thus simplifying proofs of program properties.

Number of transition rules: To varying degrees, the transition systems of the investigated operational semantics encompass concrete execution strategies. Especially in the cases of ω_r and ω_{set}, this makes for a large number

of transition rules at the expense of clarity and simplicity of proofs. Second only to ω_{va}, our system $\omega_!$ consists of only two inference rules. More importantly, each transition step corresponds to an application of a CHR rule, thus simplifying proofs of program properties.

Preservation of multiset semantics: It should be noted that the multiset semantics is a key feature of CHR, although strictly speaking it is in contrast with the paradigm of declaritivity w.r.t. first-order logic. It is already present in the constitutive semantics ω_{va} and is effectively made use of in many programs. In this respect, ω_{set} exerts a strong break with the tradition of CHR that $\omega_!$ avoids.

Reduced non-determinism: The refined semantics ω_r and the set-based semantics ω_{set} significantly reduce the inherent non-determinism of CHR: Firstly, they determine that rules are to be tried for applicability in textual order. Secondly, they fix the order in which CHR constraints are tried to match to the rules. Our semantics $\omega_!$, along with ω_{va} and ω_t, is distinctly non-deterministic. Nonetheless, it leaves open the possibility of restricting non-determinism, analogously to ω_r reducing the non-determinism of ω_t. However, this comes at the cost of additional transition rules and possibly introducing non-declarative elements into states.

	ω_{va}	ω_t	ω_r	ω_{set}	$\omega_!$
Termination on propagation rules:	-	+	+	+	+
Effective concurrency:	+	-	-	+	+
Declarative states:	+	-	-	-	+
Number of transition rules:	1	3	7	9	2
Preservation of multiset semantics:	+	+	+	-	+
Reduced non-determinism:	-	-	+	+	-

Fig. 2. Comparison of the different operational semantics

5 Conclusion and Future Work

For this work, we investigated the extent to which several desirable features are found in the most prominent operational semantics of CHR. As Figure 2 shows, each semantics displays certain limitations for specific fields of application. Inspired by linear logic, we introduced the concept of persistent constraints to CHR. Building on earlier work in [5], we proposed a novel operational semantics $\omega_!$ that provides a better trade-off between the most significant features.

The transition system of $\omega_!$ consists of two rules, such that each transition directly corresponds to a CHR rule application. Its irreflexive formulation straightforwardly solves the trivial non-termination problem. Furthermore, all elements

of $\omega_!$-states correspond to the declarative reading of states. Both properties facilitate formal proofs of program properties and hence are advantageous for program analysis.

Concerning concurrency, $\omega_!$ inherits the suitability for concurrent execution from ω_{va}. In this respect, persistent constraints have a clear advantage over token stores: As they do not hinder rule applications, their distribution is not critical and less synchronization is required.

Our proposed operational semantics $\omega_!$ displays a termination behavior different from the commonly used operational semantics ω_t. The classes of programs terminating under $\omega_!$ and ω_t do not contain each other. Hence, either semantics may be more favorable, depending on the application. Also, in its current formulation, $\omega_!$ is only applicable to range-restricted CHR programs – a limitation we plan to address in the future.

Furthermore, we intend to formulate and prove clear statements on the soundness and completeness of our semantics with respect to ω_{va} and to further investigate the differing termination behavior between $\omega_!$ and other semantics. Finally, as ω_t is the basis for numerous extensions to CHR [6], we plan to investigate the effect of building these extensions on $\omega_!$ instead.

References

1. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009)
2. Sarna-Starosta, B., Ramakrishnan, C.: Compiling Constraint Handling Rules for efficient tabled evaluation. In Hanus, M., ed.: 9th Intl. Symp. Practical Aspects of Declarative Languages, PADL. Volume 4354 of Lecture Notes in Computer Science., Nice, France, Springer-Verlag (jan 2007) 170–184
3. Simmons, R.J., Pfenning, F.: Linear logical algorithms. In Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I., eds.: Automata, Languages and Programming, 35th International Colloquium, ICALP 2008. Volume 5126 of Lecture Notes in Computer Science., Springer-Verlag (2008) 336–347
4. Betz, H., Frühwirth, T.: A linear-logic semantics for constraint handling rules. In van Beek, P., ed.: Principles and Practice of Constraint Programming, 11th International Conference, CP 2005. Volume 3709 of Lecture Notes in Computer Science., Sitges, Spain, Springer-Verlag (October 2005) 137–151
5. Raiser, F., Betz, H., Frühwirth, T.: Equivalence of CHR states revisited. In Raiser, F., Sneyers, J., eds.: 6th International Workshop on Constraint Handling Rules (CHR). (2009) 34–48
6. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. Accepted by *Journal of Theory and Practice of Logic Programming* (2008)
7. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In Demoen, B., Lifschitz, V., eds.: Logic Programming, 20th International Conference, ICLP 2004. Volume 3132 of Lecture Notes in Computer Science., Saint-Malo, France, Springer-Verlag (September 2004) 90–104
8. Sulzmann, M., Lam, E.S.L.: Parallel execution of multi-set constraint rewrite rules. In Antoy, S., Albert, E., eds.: Proceedings of the 10th International ACM SIG-

PLAN Conference on Principles and Practice of Declarative Programming (PPDP), Valencia, Spain, ACM (July 2008) 20–31
9. Pilozzi, P., Schreye, D.D.: Proving termination by invariance relations. In Hill, P.M., Warren, D.S., eds.: 25th International Conference Logic Programming, ICLP. Volume 5649 of Lecture Notes in Computer Science., Pasadena, CA, USA, Springer-Verlag (July 2009) 499–503

A Tool for Generating Partition Schedules of Multiprocessor Systems

Hans-Joachim Goltz and Norbert Pieth

Fraunhofer FIRST, Berlin, Germany
{hans-joachim.goltz,nobert.pieth}@first.fraunhofer.de

Abstract. A deterministic cycle scheduling of partitions at the operating system level is supposed for a multiprocessor system. In this paper, we propose a tool for generating such schedules. We use constraint based programming and develop methods and concepts for a combined interactive and automatic partition scheduling system. This paper is also devoted to basic methods and techniques for modeling and solving this partition scheduling problem. Initial application of our partition scheduling tool has proved successful and demonstrated the suitability of the methods used.

1 Introduction

Particularly in safety-critical areas such as medical applications and the aerospace and automotive industries, the behavior of both simple and highly complex embedded systems must be exactly known. This is achieved by defining the workflow patterns of the individual subtasks, so-called scheduling. In many computer applications a dynamic scheduling of the processes is used. By contrast, systems operating in safety-critical areas execute defined and sequenced work steps, with execution being continuously repeated (see also [8], [9]). Often, the execution of a workflow pattern takes only a few seconds.

In this paper, we suppose a deterministic cycle scheduling of partitions at the operating system level and a given scheduling method among tasks within each partition. The tasks in one partition can only be executed during the fixed time slices allocated to this partition. When constructing such a scheduling of partitions, the execution sequence of the individual work packages is often defined manually. Here, developers soon encounter problems, given the very large number of possible variations and constraints that have to be taken into account. For instance, a specific sequence of work steps must be taken into consideration in the scheduling process. At the same time, a component such as a processor should, if possible, be able to execute a work step in one piece to avoid unnecessary switching overhead.

We developed a scheduling tool that generates the partitions schedules for such a multiprocessor system using constraint based programming methods. Here, all the constraints of these complex scheduling tasks are taken into account even before the actual systems control program - the scheduler - is configured.

The basic idea is to avoid conflicts and optimize schedules beforehand rather than troubleshooting after the event.

Our research is concerned with the development of methods, techniques and concepts for a combination of interactive and automatic scheduling. The automated solution search will be implemented in such a way that it normally allows a solution to be found in a relatively short time, if such a solution exists. The scheduling tool will be flexible enough to take into account special user requirements and to allow constraints to be modified easily, if no basic conceptual change in the problem specification is necessary. An essential component is the automatic heuristic solution search with an interactive user-intervention facility. The user will, however, only be able to alter a schedule such that no hard constraints are violated.

An exemplary application of a combined interactive and automatic partition scheduling system was developed. The first test phase has been successfully completed. The scheduling tool is designed to be user-friendly. Its graphical interface and combined automatic and interactive solution search component allow the quick generation of schedules, which can be individually tweaked within the given ranges.

2 Problem Description

In this section, we explain the used notions and describe the problem. It is assumed that a multiprocessor system and a set of applications are given. Each application consists of a set of tasks. Each application will be located in one partition on one processor. The dedicated processor may be statically configured or one part of the scheduling process.

A *time slice* is the non-preemptive (i.e. uninterrupted) usage of processor time, solely allocated for one partition. A partition consists of a set of time slices, which have to follow the given constraints and which in total will be repeated periodically. Note that we do not consider the operation or inherent schedule of the tasks within a partition. This is outside of the scope of this paper. A *period* is specified by the time distance between beginning of cycle N until

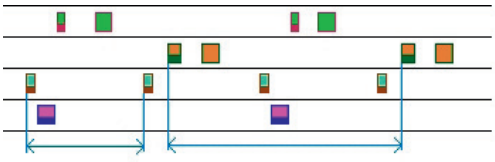


Fig. 1. periods of partitions

beginning of cycle N+1 of a partition allocation of the processor time. Since a period may have also other properties, we call this distance by *period length*,

too. The *duration per period* is the duration (sum of time slices) the processor is allocated to the partition per period. A partition may get one or several time slice(s) per period. In Figure 1 the periods of two partitions are marked. One partition consists of one time slice per period while the other partition consists of two time slices per period.

Furthermore an overall time interval is given for which the partition scheduling will be generated. In the following this time interval is called *scheduling period* (also known as hyper period). The goal is the generation of a deterministic partition scheduling for this given scheduling period such that the given constraints described below are satisfied. This generated scheduling will be repeated continuously on the discrete processors.

For each partition there are constraints on the period and the duration per period. There may exist partitions which have to follow strong defined periods, so called *fixed periods*. Variations are not allowed in these cases. We distinguish between those and *flexible periods*, which allow certain variations. These flexible periods are very useful during solution search, in particular when the processor load reaches the limit of the period capacity.

A special focus must be held on the situation at the end of each period and the entry situation for the following period. All constraints must still be met in this intersection. For a coupled system some synchronization activities will take place and have to be considered. It should be aimed to allow larger variants in period and duration for these overall scheduling periods from one to its follower. These special aspects are not presented in depth in this paper.

If one application should be spread over different processors than in our system there should be used one copy of this application for each processor involved and there must be specifications of the relations of the processes.

Between two partitions on different processors there may be definitions of relations of various kinds. These relations always reference to the beginning of one period or the end of the last time slice belonging to that period. The pattern of these relations may be e.g.:

$$\begin{aligned}
&\text{begin}(\ldots) + X \leq \text{begin}(\ldots)\\
&\text{end}(\ldots) + X = \text{begin}(\ldots)\\
&\text{begin}(\ldots) + X \leq \text{begin}(\ldots)\\
&\text{end}(\ldots) > \text{begin}(\ldots) + X\\
&\text{end}(\ldots) + X < \text{end}(\ldots)
\end{aligned}$$

A problem is specified by an amount of special definitions and constraints. The important components of a complete problem specification consist of:

1. a definition of the basic problem parameters:
   (a) the length of the scheduling period,
   (b) the basic time unit such that all time values are integers (for instance 1 ms or 1/10 ms),
   (c) the worst case waiting time at the end of a scheduling period (time, which may be necessary for the synchronization of loosely coupled processors);
2. a set of definitions for each processor:
   (a) the time for changing a partition on this processor,

(b) the general time for writing data after the end of a time slice (communication activity);

3. a set of definitions and constraints for each partition (application):
   (a) the processor allocation or constraints for that (e.g.: not processor X; another processor than the processor dealing with partition Y)
   (b) the period length of the partition,
   (c) for the period length, the allowed difference is specified by its minimum and its maximum,
   (d) the duration of a period,
   (e) for the duration of a period, the allowed difference is specified by its minimum and its maximum,
   (f) the minimal CPU load (e.g. per mill) within the scheduling period,
   (g) the maximal number of time slices within a period,
   (h) the minimal duration of a time slice,
   (i) special constraints on the end of the scheduling period;
4. a set of definitions for the relations between the partitions; the relations can be of different kind and related to the begin and/or the periods of two partitions belonging to different processors.

3 Problem Modeling

The problem of generating partition scheduling for multiprocessor systems can be suitably modeled in terms of a set of constraints and a constraint based programming language can be used for solution search. Constraint Logic Programming with constraints over finite integer domains, CLP(FD), has been established as a practical tool for solving discrete combinatorial problems (e.g. [4], [10], [11]). The success of the search often depends directly on the chosen model and the modeling options on the chosen constraint solver. Global constraints use domain-specific knowledge and specialized consistency methods to obtain better propagations results. They can be applied to large problem instances and in general improve the efficiency for solving real-life problems. The global constraints built into the Constraint Logic Programming language CHIP are particularly useful for problem modeling. Examples of global constraints are `cumulative` and `diffn` (see e.g. [1], [2]). Note that global constraints which are similar to `diffn` exist also in other constraint based programming languages with other names.

The basic method for the problem representation by constraint programming is described in the following and refers to the problem specification given in Section 2. Note that the special constraints related to the end of the scheduling are not considered in this paper. Concerning the definition of the basic problem parameters it is supposed that a scheduling horizon (hyper period) is given and that all values are integers (see (1) of the specification). The given processors are numbered by natural numbers $1, 2, \ldots$. Let $A_1, \ldots, A_{n_A}$ be the given applications (we identify these also with the partitions they are belonging to). For each A_k we define a domain variable $proc(A_k)$ for the allocation of the processor. The

domain of this variable is equal to the numbers of the allowed processors (see (3a) of the specification). If a processor allocation is given then $proc(A_k)$ is equal to the corresponding number of the processor.

A sequence of periods $p_1^k, p_2^k, \ldots, p_{m_k}^k$ is defined for each A_k . The length of the sequence depends on the scheduling period and the sum of the periods belonging to a partition. For each period p_i^k domain variables for the length of the period $l(p_i^k)$ and the duration of the period $d(p_i^k)$ are defined by the given values and the allowed differences (see (3b,c,d,e) of the specification). If differences of the period length and the duration per period are not allowed then $l(p_i^k)$ and $d(p_i^k)$ are integers.

Furthermore, a sequence of time slices $s_{i,1}^k, s_{i,2}^k, \ldots, s_{i,n_{k,i}}^k$ is defined for each period p_i^k . The length of such a sequence $n_{k,i}$ is given the maximal number of time within a period (see (3g) of the specification). For each time slice $s_{i,j}^k$ we define domain variables $start(s_{i,j}^k)$ for the starting time and $d(s_{i,j}^k)$ for duration of this slice. Firstly, the domain of $start(s_{i,j}^k)$ is given by the scheduling period (from 0 to Max, the scheduling period). Let min_s be the minimal duration of a time slice and max_p be the maximal duration of a period (see (3h,e) of the specification). The domain of $d(s_{i,j}^k)$ is defined by the union of $\{0\}$ and the interval $[min_s, max_p]$. A time slice $s_{i,j}^k$ is only relevant if $d(s_{i,j}^k)$ is different from 0 . Since one time slice of each period has to be relevant we can suppose that the first time slice of each period $d(s_{i,1}^k)$ is different from 0 . Furthermore, we can suppose that

$$start(s_{i,j}^k) \; + \; d(s_{i,j}^k) \; < \; start(s_{i,j+1}^k)$$
$$start(s_{i,1}^k) \; + \; d(p_i^k) \; \leq start(s_{i+1,1}^k)$$

for all the corresponding time slices and periods. Then the following equation is to be satisfied for the period length:

$$l(p_i^k) \; = \; start(s_{i+1,1}^k) \; - \; start(s_{i,1}^k)$$

The duration of a period $d(p_i^k)$ is equal to the sum of the durations of time slices within this period:

$$d(p_i^k) \; = \; d(s_{i,1}^k) \; + \; \ldots \; + \; d(s_{i,n_{k,i}}^k)$$

The minimal CPU load within the scheduling period (see (3f) of the specification) corresponds an integer $minD_k$. Then, for each partition A_k, this constraint can be modeled by the inequality

$$d(p_1^k) \; + \; \ldots \; + \; d(p_{m_k}^k) \; \geq \; minD_k$$

It is important to state by a constraint that all time slices must not overlap. Since the time for changing a partition has to be integrated into such a constraint we define the extended duration $d_1(s_{i,j}^k)$ of a relevant time slice by the sum of $d(s_{i,j}^k)$ and the time for changing a partition. If a processor allocation is not given and the time for changing is different on the processors then the symbolic `element`-constraint can used for computing this duration. In the case of a time slice with $d(s_{i,j}^k) \; = \; 0$, the extended duration $d_1(s_{i,j}^k)$ is also equal to 0 .

We consider a time slice as a "two-dimensional rectangle" with the dimensions *"time"* and *"processor"*. Such a rectangle can be represented by

$$[start(s^k_{i,j}), proc(A_k), d_1(s^k_{i,j}, 1]\,.$$

The use of the global `diffn`-constraint ensures that these rectangles must not overlap. For our problem we need only one `diffn`-constraint. Note that time slices with duration of 0 are not relevant for the `diffn`-constraint. If for each application the processor allocation is given and is fixed then a non-overlapping constraint can generated for each processor separately. In this case, the `diffn`-constraint with "one-dimensional rectangle" can be considered or the global constraint `cumulative`-constraint can be used with a resource limit of 1.

The relations between the periods of the partitions (see (4) of the specification) can be easily represented by arithmetic constraints (equalities, disequalities, inequalities). The time for writing data after the end of a time slice (see (2b) of the specification) has to be integrated into these constraints.

4 Solution Search

A solution of a constraint problem is an assignment of the domain variables to values of their domains such that all the constraints are satisfied. A constraint solver over finite domains is not complete because consistency is only proved locally. Thus, a search is generally necessary to find a solution. Often, this search is called "labeling". The basic idea behind this procedure is to select a variable from the set of problem variables considered, choose a value from the domain of this variable and then assign this value to the variable; if the constraint solver detects a contradiction backtracking is used to choose another value. This is repeat until values are assigned to all problem variables such that the constraints are satisfied or until it is proven that there is no solution. In our scheduling system, the domain-reducing strategy is also used for the search. This strategy is a generalization of the labeling method and was presented in [5]:

- The assignment of a value to the selected variable is replaced by a reduction of the domain of this variable.
- If backtracking occurs, the not yet considered part of the domain is taken as the new domain for a repeated application of this method.

Practical applications have shown that a reduced domain should be neither too small nor too large. A solution is narrowed down by this reduction procedure, but it does not normally generate a solution for the problem. Thus, after domain reduction, assignment of values to the variables must be performed, which may also include a search. The main part of the search, however, is carried out by the domain-reducing procedure. A conventional labeling algorithm can be used for the final value assignment. If a contradiction is detected during the final value assignment, the search backtracks into the reducing procedure.

Since a constraint solver is used for the partition scheduling tool the search space is reduced before the solution search begins. In each search step the search

space is further reduced by the constraint solver. Nevertheless, in most cases, the search spaces of relevant problems are too large and it is not possible to use a complete solution search within an acceptable time amount. Therefore, the complete search spaces cannot be investigated and heuristics are needed for a successful solution search.

The search includes two kinds of nondeterminism: *selection of a domain variable* and *choice of a reduced domain* concerning the selected variable. If labeling is used, the reduced domain consists of a single value. The success of the domain-reducing strategy depends on the chosen heuristics for the order of variable selection and for the determination of the reduced domain.

Our experience has shown that in many cases either a solution can be found within only a few backtracking steps, or a large number of backtracking steps are needed. We therefore use the following basic search method: *the number of backtracking steps is restricted, and different heuristics are tried out*. This means that backtracking is carried out on different heuristics. With regard to the problems discussed in this paper, the user can choose between different methods for the solution search. In particular, the user can control the following parameters: the number of attempts with different heuristics, the number of permitted backtracking steps for one attempt, and the priorities of the partitions.

In the recent partition scheduling tool, a static ordering is used for the heuristic of variable selection. This ordering is defined by the priorities of the partitions and the following ordering of the relevant domain variables related to the selected partition A_k:

$proc(A_k)$, $l(p_1^k)$, $d(p_1^k)$,
$start(s_{1,1}^k)$, $d(s_{1,1}^k)$, $start(s_{1,2}^k)$, $d(s_{1,2}^k)$, ...,
$l(p_2^k)$, $d(p_2^k)$, $start(s_{2,1}^k)$, $d(s_{2,1}^k)$, ...

The domain-reducing strategy is used for the following domain variables: the length of a period $l(p_i^k)$, the duration of a period $d(p_i^k)$, and the starting time of a time slice $start(s_{i,j}^k)$. The used search strategy prefers the allocation of the domain maximum to the domain variable $d(s_{i,j}^k)$ (duration of a time slice). The goal of this strategy is to minimize the number of time slices of a period. Thus the switching overhead can be reduced.

The following properties should be taken into consideration for the determination of priorities of the partitions:

- the allowed difference of the period length (partitions with fixed periods should be scheduled firstly),
- the number of allowed time slices per period (if this number is equal to 1 then this partition should be scheduled earlier);
- the durations per period;
- the desired period length;
- the relations between partitions.

If a domain variable of period length is selected then the heuristics for the choice of a reduced domain or a value can be controlled by a parameter such that one of the following heuristic is used:

- the given value of the period length is preferred,
- the minimum of the domain value is preferred,
- the maximum of the domain value is preferred.

Furthermore, there is a parameter such that the choice of a reduced domain (or a value) for the starting time variable of a time slice can be controlled by the parameter values: minimum, maximum, middle. Additionally, there is a heuristic for minimizing the number of time slices per period.

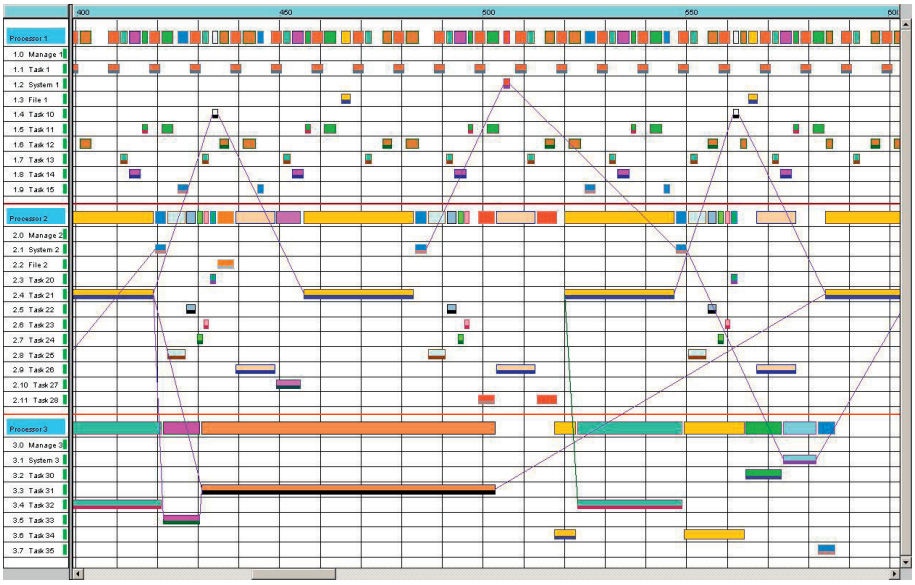


Fig. 2. Example of a partition scheduling, excerpt of approx. 200 ms

5 Graphical Interface

The scheduling tool is designed to be user-friendly by its graphical interface. The generated schedule can be graphically displayed in a clear form with a flexible layout and a user-defined level of detail. Figure 2 shows a part of an exemplary partition schedule for three processors. This partition schedule is generated by our tool and displayed by the graphical interface. For each application the partition schedule is represented in one row of this figure. Moreover, for each processor, all partitions of the processor are graphical represented in one row. The

relations between two partitions are marked by lines. The following interactive-scheduling actions are possible with the help of the graphical interface:

- scheduling an individual partition,
- scheduling marked partitions automatically,
- removing marked partitions from the schedule,
- moving time slices of a partition within the schedule,
- scheduling the remaining partitions automatically.

These actions can be performed in any order or combination, and no automatic backtracking is caused by such user actions. The user can, however, only alter a schedule in such as way that no hard constraints are violated.

The user interface and the combined automatic and interactive solution search component allow the quick generation of schedules, which can be individually tweaked. For instance, different scenarios can be easily tried out and changes swiftly implemented, consistency with respect to the specifications being guaranteed at all times. The targeted development of variants enables the solution process to be made far more flexible and efficient, even when adopting an iterative approach and when the initial tolerance limits are exceeded. With the help of the graphical interface the user can interactively generate and evaluate different variants, depending on the optimization criterion. This enables the user to incorporate his expertise.

6 Implementation and Results

The Constraint Logic Programming language CHIP ([3]) was selected as the implementation language. The global constraints and the object oriented component built into CHIP are particularly useful for problem modeling.

For the representation of the problem, we used three phases: the definition of the problem, the internal relational representation, and the internal object oriented representation. For the first phase, the definition of the problem, we developed a declarative language for problem descriptions. All components of a partition scheduling problem can be easily defined using this declarative language. Thus, the graphical user interface is only needed to set the parameters. In the second phase, the problem definition is transformed into an internal relational representation. In the third phase, the internal object-oriented representation is generated from the internal relational representation. The definition data are converted into a structure that is suitable for finite integer domains. The object-oriented representation is used for the solution search and the graphical user interface.

Our scheduling tool can generate in less than a minute a consistent schedule for complex multiprocessor systems with many thousands of time slices for an arbitrary interval. Additionally, the generated schedule is guaranteed to be error-free and executable. Even extreme optimizations are properly manageable because it is possible to generate schedules that allow over 90 per cent CPU load. In addition, the results of the scheduling process can easily be converted

into other formats, enabling them to be integrated into the overall system development process. It should be noted that the currently implemented version of our partition scheduling tool supposes that an allocation of the partitions to the processors is given.

7 Conclusions and Future Work

The initial application of our partition scheduling system has been proved successful and has demonstrated the suitability of the used methods. From this application, we were able to obtain useful information for our future work. Our future research on partition scheduling problems will include investigations of heuristics for variable and value selection and continued study of the influence of different modeling techniques on the solution search. Furthermore we will extend our implementation of a partition scheduling system such that scheduling process can also allocate partitions to processors. The development of special search methods is necessary for this goal. Moreover, a graphical interface for the problem specification will be implemented. The methods, techniques and concepts developed or under development will also be tested on other applications.

References

1. A. Aggoun and N. Beldiceanu,"Extending CHIP in order to solve complex scheduling and placement problems", *Math. Comput. Modelling*, 17(7):57–73, 1993.
2. E. Beldiceanu and E. Contejean, "Introducing global constraints in CHIP", *J. Mathematical and Computer Modelling*, 20(12):97–123, 1994.
3. M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier, "The constraint logic programming language CHIP", in *Int. Conf. Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, 1988.
4. M. Dincbas, H. Simonis, and P. van Hentenryck, "Solving large combinatorial problems in logic programming", *J. Logic Programming*, 8:75–93, 1990.
5. H.-J. Goltz, "Reducing domains for search in CLP(FD) and its application to job-shop scheduling", in U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming – CP'95*, volume 976 of *Lecture Notes in Computer Science*, pages 549–562, Springer-Verlag, 1995.
6. J. Jaffar and M. J. Maher, "Constraint logic programming: A survey", *J. Logic Programming*, 19/20:503–581, 1994.
7. K. Marriott and P. J. Stuckey, "Programming with Constraints: An Introduction", The MIT Press, Cambridge (MA), London, 1998.
8. Y. Lee, D. Kim, M. Younis, and J. Zhou, "Partition Scheduling in APEX Runtime Environment for Embedded Avionics Software", in *Proc. IEEE Real-Time Computing Systems and Applications*, pages 103109, Oct. 1998.
9. Y. Lee, D. Kim, M. Younis, and J. Zhou, "Scheduling Tool and Algorithm for Integrated Modular Avionics Systems", in *Proc. Digital Avonics Systems Conference (DASC)*, Oct. 2000.
10. P. J. Stuckey (editor), "Principles and Practice of Constraint Programming – CP 2008", volume 5202 of *Lecture Notes in Computer Science*, Springer-Verlag, 2008.
11. M. Wallace, "Practical Applications of Contraint Programming", *Constraints, An International Journal*, 1:139–168, 1996.

Efficiency of Difference-List Programming

Ulrich Geske[1], Hans-Joachim Goltz[2]

[1]University of Potsdam
ugeske@uni-potsdam.de

[2]Fraunhofer FIRST, Berlin
goltz@first.fraunhofer.de

Abstract. The difference-list technique is described in literature as effective method for extending lists to the right without using calls of append/3. There exist some proposals for automatic transformation of list programs into difference-list programs. However, we are interested in construction of difference-list programs by the programmer, avoiding the need of a transformation step. In [GG09] it was demonstrated, how left-recursive procedures with a dangling call of append/3 can be transformed into right-recursion using the unfolding technique. For simplification of writing difference-list programs using a new cons/2 procedure was introduced. In the present paper, we investigate how efficieny is influenced using cons/2. We measure the efficiency of procedures using accumulator technique, cons/2, DCG's, and difference lists and compute the resulting speedup in respect to the simple procedure definition using append/3. Four Prolog systems were investigated and we found different behaviour concerning the speedup by difference lists. A result of our investigations is, that an often advice given in the literature for avoiding calls append/3 could not be confirmed in this strong formulation.

1 Introduction

Appending an additional element E as last element to an existing list could be performed by copying all existing list elements and the additional element E into a new list (using the Prolog procedure append/3).. Instead of, this operation could be performed by an efficient (physical) concatenation using the difference list notation. Every list may be presented as a difference list. For example, the list `[1,2,3]` could be represented as difference of the lists `[1,2,3|X]` and `X`. If list X contains E as first element (e.g. X=[E|Y]), E is the next element after 3 without a copying operation). A term [E|Y] is called an incomplete list. The Prolog standard does not provide any special notation for difference lists. A possible notation of a difference list from two lists `L` and `R` may be given by a notation `L⊕R`, e.g. `L-R` or `L\R` (the symbol used must be defined as oparator in the concrete Prolog system). If ⊕ denotes a comma (,), L and R are two arguments in an argument list. The earliest extended description of difference lists was given by Clark and Tärnlund in [CT77]. A concatenation of the difference lists `U⊕V=[1,2,3|X]⊕X` and `V⊕W>=X⊕nil` results in the difference list `U⊕W=[1,2,3,4]⊕nil≡[1,2,3,4]` as soon as `X` is computed to

4⊕nil. The difference list notation is a syntactic variant of the accumulator technique (a comma is used for ⊕). While in the accumulator technique accumulator and result parameter are separated into two terms, which needs two variables for accessing them, in the difference list notation both information are accessible by one variable with the advantage of an easy to survey structure of procedures.

Our analysis of presentations of the use of difference list in Prolog textbooks showed that this technique is often not adequately explained [see also GG09]. Especially, a clear and convincing rule, where in procedures to specify the incomplete list (e.g. [E|Y]) is not supplied. Dependent from a concrete problem, the incomplete list has to be specified in the head of a rule, in one of the calls of the body or in the last (recursive) call of the body of a rule. A solution of this problem could be the use of the paradigm of grammar rules or its extension, Definite Clause Grammar (DCG), which was originally developed for language processing but may be used for list processing, too. Natural language sentences, coded as list of words, must be processed phrase by phrase from left to right, consuming some words for a phrase and leaving the rest for the following phrases. In the DCG formalism it is sufficient to specify the sequence of phrases. The argument pattern for traversing the list will be generated automatically in accumulator technique.

```
%DCG-Specification for copying a list
dcg_copy([]) --> [].
dcg_copy([X|RR]) --> [X], dcg_copy(RR).

%Generated program by automatic program transformation
dcg_copy([],L,L).
dcg_copy([X|Xs],Acc,Res):-
  'C'(Res,X,RR),
  dcg_copy(Xs,Acc,RR).
```

Fig. 1 Copying a list as DCG specification

An specification example for DCG's is copying a list (Fig. 1). The specification [X] means, taking the first element X from a list [X|Xs]. This specification is transformed into a call of the built-in procedure 'C'/3, which could be defined as Prolog procedure by 'C'([X|Xs],X,Xs).

For supporting teaching and application of difference list programming, we have proposed a procedure cons(X, [X|Xs]-Xs) in [GG09] which takes pattern from the 'C'/3 procedure. The advantages of cons/2 are both, its difference-list format, and that the definition of cons/2 could be added to each Prolog system while 'C'/3 and the DCG formalism are not part of the ISO-Prolog until now and are not available in each system. The use of cons/2 allows the application of a simple rule for writing explicit difference-programs and to find out simply the right place for the mentioned incomplete list [E|Y]. In this paper we investigate the efficiency applying cons/2 compared with other kinds of specification.

2 Top-down and Bottom-up Construction of Lists

The notions of top-down and bottom-up procedures for traversing structures like trees are well established. We will use the notions top-down construction of lists and bottom-up construction of lists in this paper to describe the result of the process building lists with a certain order of elements in relation to the order in which the elements are taken from the corresponding input list.

Top-down construction of lists

> The order el_1' – el_2' of two arbitrary elements el_1', el_2' in the constructed list corresponds to the order in which the two elements el_1, el_2 are taken from the input term (perhaps a list or a tree structure).

Bottom-up construction of lists

> The order el_2' – el_1' of two arbitrary elements el_1', el_2' in the constructed list corresponds to the reverse order in which the two elements el_1, el_2 are taken from the input term (perhaps a list or a tree structure).

An input list may be, e.g., [2, 4, 3, 1]. A top-down construction of the result list [2, 4, 3, 1] is given if the elements are taken from left to right from the input list and put into the constructed list in a left to right manner. If the elements are taken from the input list by their magnitude and put into the result list from left to right, the list [1 2 3 4] will be (top-down) constructed. A bottom-up construction of the result list [1, 3, 4, 2] is given if the elements of the list [2, 4,3,1] are taken from left to right from the input list and put into the constructed list in a right-to-left manner. If the elements are taken from the input list by their magnitude and put into the result list from right to left, the list [4 3 2 1] will be (bottom-up) constructed. Which programming techniques could be used for a top-down- respectively a bottom-up construction of lists? Accumulator technique is an often used technique, which allows both, top-down- and bottom-up construction of lists. Examples are the procedures for traversing in pre-order manner accapp_pre_td/3 and accapp_pre_bu/3 (Fig. 5).. These procedures use besides accumulators calls of append/3. But, also without use of accumulators, top-down and bottom-up list-constructions are possible. Examples are pre_order/2 and pre_order_bu/2 (see also Fig. 5). Again, calls of append/3 are needed in these definitions.

3 Construction Rules for difference list procedures

There are different possibilities to avoid the use of a call of append/3. A rather trivial improvement is given by unfolding an append/3 call which puts a single element X in front of a list A to give the result list RR, i.e RR=[X|A]. E.g., the append-free procedures acc_pre_td/3 result (Fig. 2), if in accapp_pre_td/3 the equivalent [X|L1] for Xs is inserted and the corresponding call of `append([X],L1,Xs)` is crossed.
The difference-list procedure dl_pre_td/2 (see Fig. 5) is a syntactic variant of the accumulator version of the corresponding procedure acc_pre_td/3, which results by substitution of the second and third argument, say ARG2 and ARG3 (ARG2 should be

the accumulator parameter, ARG3 the result parameter) with the difference list ARG3-ARG2.

```
acc_pre_td(tree(X,L,R),Rs, [X|L1]):-
       acc_pre_td(L, L2, L1),
       acc_pre_td(R, Rs, L2).
acc_pre_td([],L,L).
```

Fig. 2 Append-free pre-order tree-traversal by accumulator technique

The presented transformation steps for a procedure into difference-list notation has a serious disadvantage: the starting point is a procedure definition which uses an append/3-call (which should be avoided). Moreover, the precondition for the transformation, an admissible call of append/3, is not always given as examples pre_order_bu/3 (Fig. 5) shows. An admissible structure is given if a single element should be put in front of a list, i.e. this element is part of the first argument of the call of append/3. The fulfilment of this condition can not be always ensured. Therefore append/3 calls must be avoided at all. An alternative, we propose, is the use of the cons/2 procedure. The definition of cons/2 is choosen to suit the syntactic format of the difference list notation.

```
cons(Element,[Element|Rest]-Rest).
```

Fig. 3 Definition of the cons operation

The advantage of this format is its correspondence to the format which is needed for composing a resulting difference list from its parts. The position of the call of cons/2 in the body of a procedure is irrelevant, in general, but if this call occurs at its "natural" position a formal guideline for constructing difference list is possible.

Top-down construction of a difference list Res-Acc results from the "natural" order of sub-difference-lists **Res**-Temp1, ..., TempI-TempI+1, ..., TempN-**Acc**.

Bottom-up construction of a difference list Res-Acc results from the "natural" order of sub-difference-lists Temp1-**Acc**, ...TempI-1 - TempI, ..., **Res**-TempN

Rule 1 Informal rules for top-down- and bottom-up composition of difference lists

4 Benchmarks

4.1 Benchmark tests - processing trees

There exist, corresponding to [Sterling-Shapiro86], three different possibilities for the linear traversal of trees. Any node X in a binary tree, besides the leave nodes, has a *Left* and a *Right* successor tree. A pre-order traversal visits the tree in the following order: X, Left Right, which may be programmed using a call append([X|Left], Right, Tree) (pre_order/2 in Fig. 5). Correspondingly, an in-order traversal is given by the

call append(Left, [X|Right],Tree) and a post-order traversal by the sequence of calls append(Right,[X],T), append(Left,T,Tree). A concrete example is shown in Fig. 13.

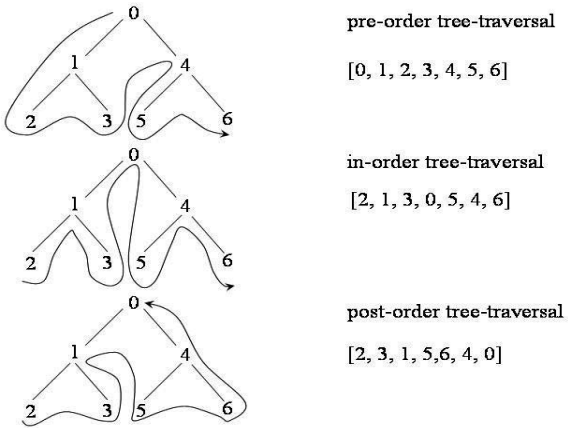


Fig. 4 Examples for linear traversals of binary trees

Fig. 4 presents four different algorithms for the pre-order-functions. A naive algorithm uses the append procedure to compose the result from the different parts. An extension of this algorithm is the additional use of an accumulator with the advantage that the first argument of the append/3-call is always a list of one element – there is no danger for looping forever. A further improvement is the avoidance of append/3-calls at all by substituting it by the new introduced cons/2 procedure. Finally, unfolding of the call of cons/2 leads to the known difference list format. A procedure in a difference-list format could be derived step-by-step, as explained or it may be specified in one step as described in the following for pre-order tree-traversal.
Specification of a pre-order traversal: The result of a pre-order traversal is the difference-list L-LN. In a top-down construction of the result, the node X of the structure tree(X,Left,Right) is visited first and supplies the difference-list L-L1, the traversal of the left subtree supplies the difference-list L1-L2, and the traversal of the right subtree supplies L2-LN. In a bottom-down construction of the result, the node X of the structure tree(X,Left,Right) is visited first and supplies the difference-list L2-LN, the traversal of the left subtree supplies the difference-list L1-L2, and the traversal of the right subtree supplies L-L1. The in-order and post-order traversals are specified analogous (see also Fig. 6, Fig. 7).

For processing the procedures a tree of a certain depth is automatically generated by a call of the procedure

```
binary_tree([],_,0).
binary_tree(tree(LR-T,Left,Right),LR,T) :-
   T1 is T-1,
   binary_tree(Left,LR-l,T1),
   binary_tree(Right,LR-r,T1).
```

Definitions for pre-order tree-traversal	
Top-down construction of result	Bottom-up construction of result

`%use of append/3; accumulator-free`
`pre_order(tree(X,L,R), Xs) :-`
`  pre_order(L,LN),`
`  pre_order(R,L0),`
`  append([X|LN],L0,Xs).`
`pre_order([],[]).`
	`%use of append/3; accumulator-free`
	`pre_order_bu(tree(X,L,R), Xs) :-`
	`  pre_order_bu(L,LN),`
	`  pre_order_bu(R,L0),`
	`  append(LN,[X],L1),`
	`  append(L0,L1,Xs).`
	`pre_order_bu([],[]).`

`%use of append/3;use of accumulator`
`accapp_pre_td(tree(X,L,R),L0,LN):-`
`  append([X],L1,LN),`
`  accapp_pre_td(L,L2,L1),`
`  accapp_pre_td(R,L0,L2).`
`accapp_pre_td([],L,L).`
	`%use of append/3;use of accumulator`
	`accapp_pre_bu(tree(X,L,R), L0,LN) :-`
	`  append([X],L0,L2).`
	`  accapp_pre_bu(L,L2, L1),`
	`  accapp_pre_bu(R,L1,LN).`
	`accapp_pre_bu([],L,L).`

`%use of cons/2`
`%use of accumulator (part of DL)`
`d_pre_td(tree(X,L,R),LN-L0):-`
`/*LN=[X|L1]*/  cons(X,LN-L1),`
`  d_pre_td(L, L1-L2),`
`  d_pre_td(R, L2-L0).`
`d_pre_td([],L-L).`
	`%use of cons/2`
	`%use of accumulator (part of DL)`
	`d_pre_bu(tree(X,L,R),LN-L0):-`
	`/*L2=[X|L0]*/ cons(X, L2-L0),`
	`  d_pre_bu(L, L1-L2),`
	`  d_pre_bu(R, LN-L1).`
	`d_pre_bu([],L-L). ),`

`%call of cons/2 unfolded`
`%use of accumulator (part of DL)`
`dl_pre_td(tree(X,L,R),[X|L1]-L0):-`
`  dl_pre_td(L,L1 -L2),`
`  dl_pre_td(R,L2 -L0).`
`dl_pre_td([],L-L).`
	`%call of cons/2 unfolded`
	`%use of accumulator (part of DL)`
	`dl_pre_bu(tree(X,L,R),LN-L0):-`
	`  dl_pre_bu(L,  L1-[X|L0]),`
	`  dl_pre_bu(R,  LN- L1).`
	`dl_pre_bu([],L-L).`

Fig. 5 Different procedure definitions for pre-order tree-traversal

Definitions for in-order tree-traversal	
Top-down construction of result	Bottom-up construction of result

`%use of append/3; no accumulator`
`in_order(tree(X,L,R), Xs) :-`
`  in_order(L,LN),`
`  append(LN,[X|L0],Xs),`
`  in_order(R,L0).`
`in_order([],[]).`
	`%use of append/3; no accumulator`
	`in_order_bu(tree(X,L,R), Xs) :-`
	`  in_order_bu(L,LN),`
	`  in_order_bu(R,L0),`
	`  append(L0,[X|LN],Xs).`
	`in_order_bu([],[]).`

`%use of append/3;use of accumulator`
`accapp_in_td(tree(X,L,R), L0,LN) :-`
`  accapp_in_td(L,L1,LN),`
`  append([X],L2,L1),`
`  accapp_in_td(R,L0,L2).`
`accapp_in_td([],L,L).`
	`%use of append/3;use of accumulator`
	`accapp_in_bu(tree(X,L,R), L0,LN) :-`
	`  accapp_in_bu(L,L0,L2),`
	`  append([X],L2,L1),`
	`  accapp_in_bu(R,L1,LN).`
	`accapp_in_bu([],L,L).`

`%use of cons/2;accumulator (in DL)`
`d_in_td(tree(X,L,R),LN-L0):-`
`  d_in_td(L, LN-L1),`
`/*L1=[X|L2]*/  cons(X,L1-L2),`
`  d_in_td(R, L2-L0).`
`d_in_td([],L-L).`
	`%use of cons/2`
	`%use of accumulator (part of DL)`
	`d_in_bu(tree(X,L,R),LN-L0) :-`
	`  d_in_bu(L, L2-L0),`
	`/*L1=[X|l2]*/ cons(X,L1-L2),`
	`  d_in_bu(R, LN-L1).`
	`d_in_bu([],L-L). ),`

`%cons/2 unfolded; accumulator(in DL)`
`dl_in_td(tree(X,L,R),LN-L0):-`
`  dl_in_td(L,LN-[X|L2]),`
`  dl_in_td(R,L2-L0).`
`dl_in_td([],L-L).`
	`% cons/2 unfolded;accumulator(in DL)`
	`dl_in_bu(tree(X,L,R),LN-L0):-`
	`  dl_in_bu(L, L2-L0),`
	`  dl_in_bu(R, LN-[X|L2]).`
	`dl_in_bu([],L-L).`

Fig. 6 Different procedure definitions for in-order tree-traversal

Definitions for post-order tree-traversal	
Top-down construction of result	Bottom-up construction of result

`%use of append/3; no accumulator`
`post_order(tree(X,L,R), Xs) :-`
`  post_order(L,LN),`
`  post_order(R,L0),`
`  append(L0,[X],L1),`
`  append(LN,L1,Xs).`
`post_order([],[]).`
	`%use of append/3; no accumulator`
	`post_order_bu(tree(X,L,R), Xs) :-`
	`  post_order_bu(L,LN),`
	`  post_order_bu(R,L0),`
	`  append([X],L0,L1),`
	`  append(L1,LN,Xs).`
	`post_order_bu([],[]).`

`%use of append/3 and accumulator`
`accapp_post_td(tree(X,L,R),L0,LN) :-`
`  accapp_post_td(L,L1,LN),`
`  accapp_post_td(R,L2,L1),`
`  append([X],L0,L2).`
`accapp_post_td([],L,L).`
	`%use of append/3 and accumulator`
	`accapp_post_bu(tree(X,L,R),L0,LN) :-`
	`  accapp_post_order_bu(L,L0,L2),`
	`  accapp_post_order_bu(R,L2,L1),`
	`  append([X],L1,LN).`
	`accapp_post_order_bu([],L,L).`

`%use of cons/2`
`%use of accumulator (part of DL)`
`d_post_td(tree(X,L,R),LN-L0):-`
`  d_post_td(L, LN-L1),`
`  d_post_td(R, L1-L2),`
`/*L2=[X|L0]*/  cons(X,L2-L0).`
`d_post_td([],L-L).`
	`%use of cons/2`
	`%use of accumulator (part of DL)`
	`d_post_bu(tree(X,L,R),  LN-L0):-`
	`  d_post_bu(L,  L2-L0),`
	`  d_post_bu(R,  L1-L2),`
	`/*LN=[X|L1]*/  cons(X, LN-L1).`
	`d_post_bu([],L-L). ),`

`%call of cons/2 unfolded`
`dl_post_td(tree(X,L,R),LN-L0):-`
`  dl_post_td(L, LN-L1),`
`  dl_post_td(R, L1-[X|L0]).`
`dl_post_td([],L-L).`
	`%call of cons/2 unfolded`
	`dl_post_bu(tree(X,L,R),[X|L1]-L0):-`
	`  dl_post_bu(L, L2-L0),`
	`  dl_post_bu(R, L1-L2).`
	`dl_post_bu([],L-L).`

Fig. 7 Different procedure definitions for post-order tree-traversal

and finally processed by an ordering processed by an ordering procedure, e.g. (measurement of cpu-time not included):

```
?- binary_tree(Tree,0,16),!,pre_order(Tree,L).
```

A depth of 16 for the tree was a good compromise for the used computer with 331 MHz processor takt rate and 192 MB memory concerning consumption of time and memory. Each benchmark test was repeated 10 times and the mean value was computed.
The investigated systems, in which the benchmark procedures were consulted (using `consult/1`), are CHIP version 5.8.0.0, ECLiPSe version 5.8 #95, SWI-Prolog version 5.6.64, and SICStus-Prolog version 4.07.

4.2 Speedup

The speedup is the relationship of the processing time of the (naive) accumulator-free version of a procedure by the processing time of the improved version for the same algorithm. In Table 1 the speedup is presented for each tree-order traversal (in-order, pre-order, post-order), each method (top-down (TD), bottom-up (BU)), and each algorithm (append/3+accu, cons/2, DCG's, difference lists) in each of the investigated Prolog system. For comparision purpose the naive programming method using calls

of the built-in append/3 without using an accumulator argument is used (procedures in_order/2, pre_order/2, post_order/2, in_order_bu/2, pre_order_bu/2, post_order_bu/2 – in Table 1 denoted by: append/3). The rows denoted by myappend/3 show the speedup with a user-defined append/3-procedure. The other classes of algorithms use an accumulator either as separate argument or as part of a difference list. The algorithm which complements the naive procedure by an accumulator is called append/3+accu in Table 1 (procedures accapp_in_td/2, accapp_pre_td/2, accapp_post_td/2, accapp_in_bu/2, accapp_pre_bu/2, accapp_post_bu/2). A substitution of append/3 by a call of the new procedure cons/2 leads to the class of algorithms which is called cons/2 in Table 1 (procedures d_in_td/2, d_pre_td/2, d_post_td/2, d_in_bu/2, d_pre_bu/2, d_post_bu/2).

Table 1: Speedup for TD/BU-in-/pre-/post-order traversal of a tree

Prolog system	CHIP 5.8		ECLiPSe 5.8		SWI 5.6.64		SICStus 4.07	
Construction	TD	BU	TD	BU	TD	BU	TD	BU
pre-order								
myappend/3	1.27	1.25	0.96	1.06	0.99	1.00	0.32	0.24
append/3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
append/3+accu	2.90	4.84	4.21	6.33	3.72	8.48	1.09	1.44
cons/2	3.90	6.10	3.24	5.15	4.00	8.70	0.96	1.23
DCG(‘C’/3)	4.21	6.74	5.35	8.77	4.74	10.58	0.93	1.22
DCG(-->/2)			5.30	8.51	6.24	11.73	1.22	1.58
difference lists	5.00	7.80	4.50	7.06	5.19	12.84	1.35	1.91
in-order								
myappend/3	1.22	1.22	0.97	1.06	1.01	0.99	0.36	0.34
append/3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
append/3+accu	2.51	2.56	2.90	3.85	2.41	2.45	1.08	0.99
cons/2	3.72	3.32	2.47	3.18	2.71	2.61	0.86	0.93
DCG(‘C’/3)	3.60	3.69	4.07	5.54	3.15	3.11	0.92	0.85
DCG(-->/2)			4.00	5.64	3.63	3.50	1.18	1.17
difference lists	4.29	4.25	3.61	4.66	3.98	3.73	1.48	1.42
post-order								
myappend/3	1.25	1.14	0.97	0.96	0.99	1.00	0.25	0.34
append/3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
append/3+accu	4.52	3.32	6.41	4.44	7.96	5.04	1.42	1.34
cons/2	5.78	4.25	5.20	3.57	8.16	5.37	1.23	1.14
DCG(‘C’/3)	6.57	4.50	8.43	5.65	10.48	6.45	1.21	1.16
DCG(-->/2)			8.68	5.69	12.05	8.57	1.47	1.53
difference lists	7.59	5.55	7.87	4.66	12.34	7.24	1.91	1.83

Finally, unfolding the call of cons/2 results in the procedures which form the class called “difference lists” in Table 1 (procedures dl_in_td/2, dl_pre_td/2, dl_post_td/2, dl_in_bu/2, dl_pre_bu/2, dl_post_bu/2). A possible class of procedures with an additional argument for the accumulator instead of combing it with the result parameter in a difference list is not considered here. It is a syntactic variant of the difference-list notation and is denoted in the literature as “accumulator” version.
Table 1 shows that each of the investigated Prolog systems behave in a different manner. But, roughly we may classify the systems into two groups. One group consists of the CHIP system and SWI-Prolog.. Each improvement in the algorithm (in the assumed order of improvement: append/3+accu, cons/2, difference list) mirrors in a partly significant speedup of the execution. Depending of the problem, the speedup of execution times of the method append/3+accu compared with the append/3 algorithm is between about 2.5 and 5 for CHIP and about 2.5 and 8.5 for SWI-Prolog. Application of the cons/2-algorithm gives a speedup of about 25% less the speedup of the difference list method (cf. also Table 2). The maximum speedup by the cons/2-method is about 8-fold compared with the execution time by the append/3-method. Both systems process obviously a built-in procedure and a user-defined procedures comparable fast (see also rows myappend/3 and append/3 in Table 1).
The second group of Prolog systems contains the systems ECLiPSe and SICStus-Prolog. The characteristic of this class is a missing strong connection between the assumed improvement given by the algorithms and the speedup of execution. The accumulator-procedures which contain a call of append/3 may have a greater speedup than the corresponding procedures which use a call of cons/2 instead of append/3. An explanation for this effect may be that calls of the built-in (compiled) procedure append/3 will be processed faster than calls of the user-procedure cons/2. The cons/2-algorithm supplies a speedup of about 30% less the speedup of the difference list method (see also Table 2). The maximum speedup by the cons/2 procedure in this group is 5.2. For SICStus-Prolog the maximum speedup of about 2 occurs for the difference list procedure.

Table 2: Average relative speedup for TD/BU-in-/pre-/post-order traversal of a tree

Prolog system	CHIP 5.8		ECLiPSe 6.0.82		SWI 5.6.64		SICStus 4.0/7	
Construction	TD	BU	TD	BU	TD	BU	TD	BU
append/3	0.33		0.51		0.36		0.69	
append/3+accu	0.60		0.88		0.66		0.74	
cons/2	0.77		0.72		0.72		0.65	
DCG(‘C’/3)	0.85		1.18		0.85		0.64	
DCG(-->/2)			1.19		1.02		0.82	
difference list	1.00		1.00		1.00		1.00	

In these tests the minimum measured speedup of the algorithms append/3+accu and cons/2 reaches 58% of the speedup by difference lists (TD-pre-order with CHIP and TD-in-order in SICStus-Prolog). This result may be important for programming practice. DCG’s algorithms are able to perform procedure execution more efficient than difference list algorithms (Table 2).

5 Summary and Future Work

We have proposed simple, schematic rules for using difference lists. Our rule generalizes both bottom-up construction of lists using accumulators and top-down construction of lists using calls to append/3 to the notion of difference list. The introduction of the cons/2 operation serves as a didactic means to facilitate and simplify the use of difference lists. This operation could easily be removed from the procedures by an unfolding operation.

The benchmark tests demonstrate that the gain concerning the speedup depends from the used Prolog system. Speedup factors of 1.35 minimum to 12.84 maximum could be found for the same traversal order (pre-order) in different systems by using difference lists instead of the naive algorithm with calls of append/3. SWI-Prolog supplies for the procedures of the benchmark tests a maximum speedup of about 12, for CHIP-system this figure is about 8, for ECLiPSe a maximum speedup of about 2 results, and the for SICStus system the maximum speedup is about 2. The highest possible speedup occur when difference lists or DCG's are used. A reasonable speedup occurs when a call of cons/2 is used, with the advantage that such a procedure is easier to read and to maintain. Because of considerable high speedup values for the append+accu algorithm the often given advice "calls of append/3 should be avoided" should be substituted by "try using accumulators as often as possible".

A comparison of the efficienies of the difference-list algorithm and the DCG algorithm leads to the assumption which is to verify yet that a compiled version of the proposed cons/2 procedure will improve the efficieny significant.

References

[AFSV00] Albert, E.; Ferri, C.; Steiner, F.; Vidal, G.: Improving Functional Logic-Programs by difference-lists. In He, J.; Sato, M.: Advances in Computing Sciece – ASIAN 2000. LNCS 1961. pp 237-254. 2000.

[CT77] Clark, K.L.; Tärnlund, S,Å: A First Order Theory of Data and Programs. In: Inf. Proc. (B. Gilchrist, ed.), North Holland, pp. 939-944, 1977.

[GG09] Geske, U.; Goltz, H.-J.: A guide for manual construction of difference-list procedures. In: Seipel, D.; Hanus, M.; Wolf, A. (eds): Applications of Declarative Programming and Knowledge Management, Springer-Verlag, LNAI 5437, pp 1-20, 2009.

[MS88] Marriott, K.; Søndergaard, H.: Prolog Transformation by Introduction of Difference-Lists. TR 88/14. Dept. CS, The Univ. of Melbourne, 1988.

[MS93] Marriott, K.; Søndergaard, H.: Prolog Difference-list transformation for Prolog. New Generation Computing, 11 (1993), pp. 125-157, 1993.

[SS86] Sterling, L; Shapiro, E.: The Art of Prolog. The MIT Press, 1986. Seventh printing, 1991.

[ZG88] Zhang, J.; Grant, P.W.: An automatic difference-list transformation algorithm for Prolog. In: Kodratoff, Y. (ed.): Proc. 1988 European Conf. Artificial Intelligence. pp. 320-325. Pittman, 1988.

The workshops on (constraint) logic programming (WLP) are the annual meeting of the Society of Logic Programming (GLP e.V.) and bring together researchers interested in logic programming, constraint programming, and related areas like databases, artificial intelligence and operations research. The 23rd WLP was held in Potsdam at September 15 – 16, 2009. The topics of the presentations of WLP2009 were grouped into the major areas: Databases, Answer Set Programming, Theory and Practice of Logic Programming as well as Constraints and Constraint Handling Rules.

ISBN 978-3-86956-026-7