CHR 2012 — Proceedings of the 9th International Workshop on Constraint Handling Rules

Jon Sneyers *Thom Frühwirth*

Report CW 624, September 2012

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

CHR 2012 — Proceedings of the 9th International Workshop on Constraint Handling Rules

Jon Sneyers *Thom Frühwirth*

Report CW 624, September 2012

Department of Computer Science, KU Leuven

Abstract

This volume contains the papers presented at CHR 2012, the 9th International Workshop on Constraint Handling Rules held on September 4th, 2012 in Budapest, at the occasion of ICLP 2012.

This workshop was the ninth in a series of annual CHR workshops. It means to bring together in an informal setting, people involved in research on all matters involving CHR, in order to promote the exchange of ideas and feedback on recent developments. Previous workshops on Constraint Handling Rules were organized in 2004 in Ulm (Germany), in 2005 in Sitges (Spain) at ICLP, in 2006 in Venice (Italy) at ICALP, in 2007 in Porto (Portgual) at ICLP, in 2008 in Hagenberg (Austria) at RTA, in 2009 in Pasadena (California, USA) at ICLP, in 2010 in Edinburgh (Scotland) at ICLP, and in 2011 in Cairo (Egypt) at the second CHR summer school.

Keywords : Constraint Handling Rules
CR Subject Classification : D.3.2, D.3.3, F.4.1

Proceedings
of the
Ninth International Workshop
on
Constraint Handling Rules

September 4th, 2012
Budapest, Hungary

Preface

This volume contains the papers presented at CHR 2012, the 9th International Workshop on Constraint Handling Rules held on September 4th, 2012 in Budapest, at the occasion of ICLP 2012.

There were 8 submissions. Each submission was reviewed by at least 3, and on the average 3.8, program committee members. The committee decided to accept 7 papers.

This workshop was the ninth in a series of annual CHR workshops. It means to bring together in an informal setting, people involved in research on all matters involving CHR, in order to promote the exchange of ideas and feedback on recent developments. Previous workshops on Constraint Handling Rules were organized in 2004 in Ulm (Germany), in 2005 in Sitges (Spain) at ICLP, in 2006 in Venice (Italy) at ICALP, in 2007 in Porto (Portgual) at ICLP, in 2008 in Hagenberg (Austria) at RTA, in 2009 in Pasadena (California, USA) at ICLP, in 2010 in Edinburgh (Scotland) at ICLP, and in 2011 in Cairo (Egypt) at the second CHR summer school.

More information about CHR is available on the CHR website[1]. The papers from all previous editions of the CHR workshop (as well as many other CHR-related papers) are also available for download from the CHR website[2].

We are grateful to all the authors of the submitted papers, the program committee members, and the reviewers for their time and efforts. We would also like to thank the ICLP general chair, Péter Szeredi, and the ICLP workshop chair, Mats Carlsson, for the excellent organization.

August 2012

Jon Sneyers and Thom Frühwirth

[1] CHR website: `http://dtai.cs.kuleuven.be/CHR`
[2] CHR bibliography: `http://dtai.cs.kuleuven.be/CHR/biblio.shtml`

Table of Contents

Program Committee

Additional Reviewers

An adaptation of Constraint Handling Rules for Interactive and Intelligent Installations

Henning Christiansen

Research group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: `henning@ruc.dk`. Home page: `www.ruc.dk/~henning`

Abstract. Constraint Handling Rules, CHR, have proved to be effective for a large range of reasoning tasks, which makes it interesting in different sorts of interactive installations. Typically, such an installation involves a large number of cooperating software components that need to refer to a common knowledge. Using CHR's constraint stores as a knowledge representation may be appealing from a theoretical point of view, but suffers from the inherent limitation of CHR, that a constraint store disappears immediately after a query has been evaluated.

An extension to CHR is proposed, which allows different processes to reason over and maintain a common knowledge base represented as text files containing constraints. Constraints are automatically read from and written to the files before and after a query has been executed, which means that the intended style of programming deviates only very little from traditional CHR programming.

1 Introduction

Constraint Handling Rules [1, 2], CHR for short, were created as a declarative language for defining constraint solvers to be used from Prolog, typically for standard domains such as real, integer or rational numbers. However, it was soon realized that CHR is a powerful language for knowledge representation and reasoning, as documented among others by [2, 3]. In such applications, the constraint store may be seen as a knowledge base, but in that respect, CHR has a deficiency as the constraint store disappears immediately after a query has been evaluated. In order to qualify fully as a knowledge representation formalism, it should be possible also to run several queries in the same constraint store as well as using CHR rules for updating the knowledge base represented as a constraint store.

We are here interested in extending the range of applications for CHR to include interactive installations involving some sort of knowledge assimilation or adaptive properties over time. This may be relevant, e.g., for collecting knowledge about different users, analyzing their behaviour and so on. We also want to promote experiments using CHR for analyzing streams of sensor data, that also can be considered as constraints, constantly updated by an external process. We also need to take into account the fact that any nontrivial interactive installation typically includes many different software components written in many different languages and incorporates a variety of prefabricated drivers for special hardware. Thus we need a common knowledge representation format, which is easily accessible from CHR (that we intend be dominating for reasoning tasks) as well as other languages used. Our aim is to provide facilities that allow programmers to write their CHR programs in a way as close as possible to the way they do for standard one-query-at-a-time applications.

We present a suggestion for such a framework, which forms an extended version of CHR, dubbed *ii*CHR to hint a relationship to *i*nteractive *i*nstallations. As a common format for knowledge storage and exchange, we use text files containing ground constraints written as Prolog facts instead of involving additional technologies such as database management systems. This provides a straightforward and transparent format, that is easily read and written also by external processes written in other programming languages.

Synchronization has been of less concern in the present work and, at least until we have gained more practical experience with *ii*CHR, we expect the developers to handle this using facilities of the operating system and hosting Prolog implementation. This paper is not about the semantics of CHR and its derivatives, and we leave any such considerations to the developer; here we provide a new environment that adapts an existing technology to a new range of applications. Application programs such as those we have in mind, may have clean parts that rely on a first-order semantics and good properties such as confluence. For the knowledge base management parts, we need to consider CHR as a nonmonotonic language in which procedural properties such as the order of rule applications do matter.

In the following section 2, we discuss our motivating applications and the design goals that we have emphasized, and section 3 describes the details of our *ii*CHR system. Heuristics and coding principles for using *ii*CHR are discussed in section 4, and section 5 demonstrates a small application that may serve as a first prototype for an art museum with four different programs cooperating around a common knowledge base. Finally, we review related work in section 7 and provide a concluding discussion, including a first evaluation and suggestions for future work, in section 8.

2 Motivation and design considerations

This work is motivated by a project developed at Roskilde's University's Department of Communication, Business and Information Technologies called the *Experience Cylinder* [4]. It is an installation with a 360° circular screen equipped with a number of synchronized projectors plus a Kinect device mounted above the installation to trace the visitors' movements. The first application has been developed together with the Viking Ship Museum of Roskilde, Denmark, to exhibit information about a journey made with a full size copy of a viking longship from Roskilde to Dublin and back. During the travel, all sorts of physical data were logged and lots of photos, videos and written accounts – all timestamped – have been recorded.[1] Here the circle represents a time line as well as a geographical circuit (the trip to Dublin went North of Scotland and the trip back south of Wales and England). When a visitor moves closer to the screen, appetizers will pop up corresponding to the indicated time point (alias geographical position), and he or she may require further details by simple gestures such as pointing.

The software platform used in the Experience Cylinder is currently being developed into a more generic form so that active components can be added without interface programming, and this is where our CHR-based components may come in. The current installation does not recognize the visitors nor take into account their current focus of interest (e.g., one visitor may be interested in the food onboard and another in the weather conditions), and here relevant "intelligent" components written in CHR may become interesting. Adding "intelligent", animated characters that interact with users, is also under consideration.

In our previous work, we have investigated and developed methods using CHR for especially abductive reasoning [5–7], including for language analysis [8–10]. This, together with other work, see, e.g., [2, 11], has demonstrated that CHR is highly suited for a large variety of reasoning task. A main goal with the present work is to make reasoning with CHR available for developers of interactive and intelligent installations. By mentioning "intelligent" here, we mean that an installation should appear as a knowledgeable, cooperative and perhaps even sympathetic partner, that helps to improve the experience for the user. Abductive reasoning and language processing are very concise metaphors in this context, as the task for the reasoner involves coming up with best explanations of what is actually going on inside or around the installation, e.g., figuring out users' intentions or focus of interest. These judgements need to be made from the ongoing discourse of sensor signals (or higher level signals extracted from sensor data by other software components).

In our choice of facilities, we have in mind developers who are motivated for trying out CHR for the applications and reasons mentioned. We hope to address both designers or artist, who care mainly about interaction and contents, and experienced technicians who can program, adapt and tie together the variety of software and hardware components involved. This calls for preserving the (relative) simplicity and transparency of CHR, adding as little additional complexity as possible, and reducing the need for learning yet another hybrid programming language. Our choice of plain text files of Prolog facts as the interlingua between software components avoids mixing in other conceptual frameworks, as would the use of relational databases or XML-based technology. Furthermore, this format is easily accessible also for developers with no experience in CHR and/or no interest in the knowledge intensive parts of an installation. We discuss possible consequences of this decision in section 8.

While we take for granted the qualities of CHR with respect to reasoning, it is less obvious how it works for updating of knowledge bases. This is evaluated in our conclusions, based on our test application developed in section 5.

[1] See `http://www.vikingeskibsmuseet.dk/en/the-sea-stallion-past-and-present/`.

3 The proposal: *ii*CHR

In the following, we describe all facilities of the *ii*CHR language, where it differs from or extends standard CHR. It is implemented on top of SWI Prolog [12] and its CHR library and inherits their facilities. The implementation is available at on the internet [13].

3.1 Declaration of constraints

Constraint predicates may be shared between different programs via a common file, which requires a certain agreement between the declarations in each program. Each program should declare the constraints it is using as described here. Instead of using CHR's standard way of declaring constraints, directives of the following form are used.

`:- iiCHR_constraint` *Constraint-Decl* `,` ... `,` *Constraint-Decl* `.`

Each constraint declaration takes the following form.

Constraint-Predicate`/`*Arity*`*[`*Options*`].`

Predicates and arities have the usual meaning as in CHR, with a small deviation when option `time_stamped` is in use (explained below). The options and the preceding asterisk may be left out, in which case the declaration is synonymous with a standard CHR constraint declaration. The following options are currently available.

`file(`*Path*`)`
: The *Path* determines a file relative to the current working directory; the file extension "`.con`" is automatically added. This file stores a collection of constraints for the given constraint predicate in a textual format, and is normally read into the initial constraint store before a query is executed. Depending on how the query is posed and other options, the constraints in the final constraint store for this predicate may or may not be written back to the file. A given file must only be used for one constraint predicate.
Constraint predicates that are declared with an associated file are called *shared*,[2] all other constraint predicates *private*.

`read_only`, `write_only`, `append`
: Only relevant for shared predicates, and at most one of them may be used for the same constraint predicate. `read_only` means that the constraints are never written back to the file after a query has been executed (unless option is overridden by the query predicate; below). `write_only` is defined analogously. `append` means that the file is not read in, and any constraints produced for this constraint predicate are appended to file.

[2] The term "shared" indicates that the given constraint may be *potentially* shared between different programs that access the same file.

`active`
: Only relevant for shared predicates. Normally, shared are made passive in each rule head where they occur (the rationale for this is discussed in sections 3.2 below). This option overrides this convention, but exceptions can be made in other ways as indicated below.

`time_stamped`
: An additional argument is added to each constraint but is invisibly in the normal rule syntax. When a constraint of the given predicate is created, a current time stamp is generated, corresponding to the current physical time. This facility is intended, among other things, for (virtual) sensor signals.

`locking`, `nowait`
: Only relevant for shared predicates. It affects the query predicates' use of the file; described below.

When more than one program refer to the same file via their *ii*CHR constraint declarations, these are expected to agree on the associated predicate name, arity and the `time_stamped` option. Constraints of shared predicates must be ground when attempted to be written or appended to a file; otherwise an exception is generated from the query predicates explained in section 3.3 below.

3.2 Rules

The rules of *ii*CHR are similar to those of CHR, including the usual propagation, simplification and simpagation rules, that we expect our reader to be familiar with.

As indicated above, shared constraints are made passive in rules by default. This prevents repeated applications of propagation and simpagation rules, that must be expected to have been applied already, to a set of constraints read in from a file. This way, we can expect linear time for loading of constraints from a file and it anticipates an incremental processing of constraints accumulated over time. The consequences for the programmer are discussed in section 4 below. Consider as an example the following *ii*CHR rule appearing in a source file, with `s/1` being a shared and `p/1` a private constraint.

```
p(X), s(Y) ==> Z is X+Y, s(Z).
```

It is compiled into the following CHR rule, using SWI Prolog's syntax for passive constraints.

```
p(X), s(Y)#passive ==> Z is X+Y, s(Z).
```

If needed, this can be overridden, either throughout the source file using the option `active` in the constraint declaration, or specifically for a given occurrence in the head of an *ii*CHR rule as follows.

```
p(X), s(Y)#active ==> Z is X+Y, s(Z).
```

The latter rule compiles as expected into the following CHR rule.

```
p(X), s(Y) ==> Z is X+Y, s(Z).
```

Both `#active` and `#passive` annotations can be used in the head of rules, however `#active` only for shared constraint predicates.

Constraints declared with the option `time_stamped` and arity n can be used in any rule as a constraint with n arguments, disregarding a hidden $n+1$'th argument holding a time stamp. If needed, the time stamp of a given constraint can be accessed in the head of a rule using an annotation of the form `#time(`T`)` where T is a Prolog variable. Whenever a call to a time stamped constraint predicate is made in the body of a rule, a timestamp is added automatically and invisibly, reflecting the actual physical time. We illustrate this by an example of a constraint declaration with two rules, showing how they are compiled into CHR and an auxiliary Prolog predicate. The *ii*CHR constraint `bind/2` declared below represents bindings from identifier to values. The first *ii*CHR rules implements a preference to new bindings over old ones, and the second one applies for bindings to a specific identifier, disregarding the time stamp.

```
:- iiCHR_constraint bind/2*[time_stamped].
bind(Key,_)#time(T1) \ bind(Key,V0)#time(T0) <=> T1 > T0 | true.
bind(id,V) ==> write('id bound to '), write(V), nl.
```

This code fragment is compiled into the following CHR code, that includes an auxiliary Prolog predicates that simulates to be the `bind` constraint within rule bodies.

```
:- chr_constraint bind/3.
bind(Key,_,T1) \ bind(Key,_,T0) <=> T1 > T0 | true.
bind(id,V,_) ==> write('id bound to '), write(V), nl.
bind(X,Y):- get_time(T), bind(X,Y,T).
```

The `get_time` predicate is an SWI Prolog built-in that produces the relevant time stamp.

Finally we introduce a notation for a simple variant of so-called negation as absence [14], which has turned out to be useful for the intended applications. The notation `not_exists` *Pattern* can be used in a guard or body of a rule to test that there is no constraint matching *Pattern* in the current constraint store. It obviously requires the programmer to have in mind which variables are instantiated through the head of the rule, and it does not conform with a first-order semantics; it is shown at work in section 5.3 below.

3.3 Queries

Queries can be posed using two alternative predicates called `executeQuery` and `executeUpdate`. The names are inspired by the JDBC interface [15] and are used for queries that either only read values or are expected to both read and write values from/to files. In the following, the *Query* argument can be any query referring to *ii*CHR constraints and additional Prolog predicates if needed. We introduce these predicates in an overall way and explain later the conventions used for accessing and locking files.

`executeQuery(`*Query*`)`
: The constraints for any shared predicates declared in this source file are read in from the relevant files (unless declared as `write_only` or `append`) and entered into the constraint store by calling them. Then the *Query* is executed in the usual way.

`executeUpdate(`*Query*`)`
: It proceeds as `executeQuery(`*Query*`)`, and – if this did not fail – extracts from the final constraint store all constraints of shared predicates (except those declared as `read_only`). If these predicates are declared with the `append` options, the constraints are appended to the file, otherwise written to the file and replacing any previous content of the file.

The query predicates can take an optional argument which is a list of options of the form

Constraint-Predicate`/`*Arity*`*`*Modifier*

where the *Modifier* is one of `ignore`, `read_only`, `write_only`, `append`, `locking` and `nowait`. The first one, `ignore`, indicates that the files associated with the given constraint is neither read in nor written to the associated file. The remaining options make the query execute as if they had been given in the constraint declarations. Alternatively, the modifiers can be given without a constraint predicate, meaning that they go for all predicates. Modifiers `locking`, `write_only` and `append` are only relevant for `executeUpdate`.

When `executeQuery` or `executeUpdate` wants to read in a number of files, it normally waits until it can access all the files. It does so in a way that it only holds files when all files can be accessed, except from tiny moments used for testing availability. When all files are available, they are locked for writing by any other processes, the files are read in, and then released. However, in case of `executeUpdate`, a lock is kept on any file covered by a `locking` option. The `nowait` option will lead to an exception in case the first attempt to access all files for reading does not succeed.

When `executeUpdate` has finished a query evaluation successfully, it will wait until it can access all relevant files for writing (with the same precautions as above), locks them temporarily, writes all files and finally releases all locks.

If it is not possible to get access to a required set of files for reading or writing within a system defined time limit, the given query predicate generates an exception. This time limit can be changed by the programmer for each process running *ii*CHR. A similar time limit is defined for the actual execution of a query, i.e., in between the reading and writing of files.

4 Programming patterns

We expect programs written using *ii*CHR to have two main functions, namely reasoning about knowledge and maintenance of knowledge. While CHR, and thus *ii*CHR, is well-suited for a large variety of reasoning tasks, its use for maintenance of knowledge bases may be less obvious and should be critically evaluated.

In the simplest case of adding a new piece of data to a knowledge base, this amounts to calling a shared constraint, which thus is added to the constraint store and in turn to the corresponding file. In case we do not want an unconditional addition of the new data, but have it evaluated and perhaps transformed in some way, we need to pay special attention, as shared constraints by default are passive in rule heads, as explained above. Assuming such a predicate, say `c/1`, it is often useful to introduce a private counterpart `newc/1`, to trigger the desired analysis. One useful programming pattern is to write a series of rules with `newc/1` in their head, ended by an insertion rule as follows.

```
newc(X) <=> c(X).
```

Replacement of a data value, analogous to an UPDATE statement in SQL, can be obtained by a rule based on the following pattern.

newc(*New-Value*) \ c(*Old-Value*) <=> c(*New-Value*).

Here the passive nature of `c/1` combined with the active `newc/1` is actually an advantage, as it eliminates the danger of loops that otherwise requires attention for CHR programs with rules of the form `c(···),... <=> c(···),...`.

As it appears, the *ii*CHR system gives only very rudimentary tools for synchronization of processes depending on shared constraint predicates. This is a design choice motivated by having the programmer spend as little effort on these issues as possible. However, a classical problem of data loss may arise in case two processes, say A and B, are triggered by the same event, so they start almost simultaneously, both read the file for the same shared constraint, say c/n, process for a while, then A re-writes the file and then B. In this case, the data produced by A is lost. Under such circumstances, and when preservation of all data is essential, the problem can be avoided using the `locking` option. In some cases, the `append` option may also do the job.

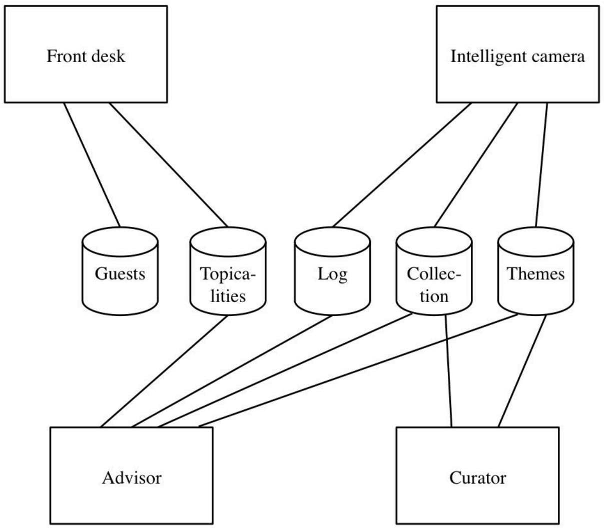
The other options related to reading and writing files serve mainly to suppress unnecessary file operations and can be used with very little intellectual effort.

Deadlocks are effectively prevented by the built-in time limits explained above. We expect these limits to be reached only in case of programming errors or when other external processes interfere. In extremely rare cases, it is possible to have two processes fighting to access the same file without finding a winner, until one or both of them run into a time limit. If this (very unlikely) turns out to be a problem, it can be avoided using pseudo-random numbers to determine small waiting times when file access is requested.

5 Example: An interactive art museum

We consider an art museum that works as a large interactive installation keeping track of the guests' movements. The museum houses a collection of paintings, each identified by painter and title. The museum's curator may dynamically remove paintings, include new ones and tag them with one or more themes. Such themes may reflect periods in the history of art, they can refer to a genre or elements depicted in the painting, or something completely different. There is a front desk that receives and says farewell to guest, and an automatic advisor which, based on the themes of the paintings that each guest has seen recently, may suggest him or her a possible next painting to look at. Such advice may be given via a wireless headset or a smartphone.

To support the different tasks, the installation includes four different software components written in *ii*CHR using five different data files. The following figure shows these parts with lines to indicate which software components use which files.



The `guests.con` file maintains a list of the guests currently in the museum. Example:

```
guest(peter). guest(mary).
```

The `collection.con` file represents the exhibited paintings together with their thematic tags. Examples:

```
painting(leonardo,monalisa,portrait).
painting(leonardo,monalisa,renaissance).
painting(leonardo,ladyWithEmine,animal).
painting(leonardo,ladyWithEmine,portrait).
painting(leonardo,ladyWithEmine,renaissance).
```

The `themes.con` file keeps a list of the themes currently in use in terms of a constraint `theme/1`. The `log.con` file is a record of which guests saw which paintings when, and is intended for statistic purposes that we do not consider here, and to avoid recommending a guest to see a painting that he or she has seen already. Example:

```
watchedPainting(peter,leonardo,ladyWithEmine,1341913905.404352).
```

Finally, the `topicalities.con` file holds measurements of the guests' interests in different themes, by adding a contribution for each theme of the recently watched painting with a geometric decay to emphasize the recent ones. Examples:

```
topicalityMeasure(peter,animal,0.10289999999999999).
topicalityMeasure(peter,portrait,0.8319300000000001).
topicalityMeasure(peter,renaissance,0.17492999999999997).
```

In the following, we go through the software components written in *ii*CHR. So far, they have been tested by posing queries manually into terminal windows, one such window for each component. However, the queries that we show below could equally well have been triggered by the operating system or a waiting loop implemented in Prolog.

5.1 The front desk program

Constraint declarations:

```
:- iiCHR_constraint guest/1*[file(guests)].
:- iiCHR_constraint topicalityMeasure/3*[file(topicalities)].
:- iiCHR_constraint enter/1, exit/1.
```

The relevant queries for the front desk are of the following forms.

```
?- executeUpdate(enter(guest))      ?- executeUpdate(exit(guest))
```

The `enter` constraint inserts a `guest` constraint, `exit` removes it together with all related `topicalityMeasure` constraints. The rules are as follows.

```
guest(Guest) \ enter(Guest) <=> write('Guest already in museum').
enter(Guest) <=> guest(Guest).
exit(Guest) \ topicalityMeasure(Guest,_,_) <=> true.
exit(Guest), guest(Guest) <=> true.
```

Referring to the discussion of programming heuristics above, it appears that the `enter` constraints serves as "`new`" version of `guest`, and `exit` is a trigger to start the two clean-up rules.

5.2 The curator's program

Constraint declarations:

```
:- iiCHR_constraint painting/3*[file(collection)].
:- iiCHR_constraint theme/1*[file(themes)].
:- iiCHR_constraint newPainting/3, removePainting/2, removeIfNotUsed/1.
```

The last constraint removes themes no longer in use. Queries:

```
?- executeUpdate(newPainting(Painter, Title, theme))  and
?- executeUpdate(removePainting(Painter, Title))
```

Notice that `newPainting` can be used for including entirely new paintings as well as adding new themes to an existing ones. The following rules serve to update the two shared constraint predicates.

```
painting(Painter,Title,Theme) \ newPainting(Painter,Title,Theme) <=> true.
```

```
theme(Theme) \ newPainting(Painter,Title,Theme)
  <=> painting(Painter,Title,Theme).
newPainting(Painter,Title,Theme)
  <=> theme(Theme), painting(Painter,Title,Theme).
removePainting(Painter,Title) \ painting(Painter,Title,Theme)
  <=> removeIfNotUsed(Theme).
painting(_,_,Theme) \ removeIfNotUsed(Theme) <=> true
removeIfNotUsed(Theme) \ topicalityMeasure(_,Theme,_) <=> true.
removeIfNotUsed(Theme), theme(Theme) <=> true.
```

The rules for `removeIfNotUsed` may be require a bit of explanation. One instance of it is generated for each theme of a painting being removed. However, if the theme is still in use, this instance is eliminated by a simpagation rule, otherwise it triggers the clean-up rules for `topicalityMeasure` and `theme` constraints.

Obviously this program can only be understood by a procedural reading and is not meaningfully considered in a first-order semantics.

5.3 The intelligent camera's program

Our imaginary museum has equipment that tracks the different guests and generates a signal each time a guest has been watching a painting for more than, say, one minute. With today's technologies, such a facility can made relatively easily and for small money, and there are available driver software that can be adapted for this purposes. Which detailed technology is used is not of importance here, and to give it a name, we refer to it as the intelligent camera, although the best solution in practice may not need to involve cameras. The following program executes the necessary updates whenever such a signal is reported.

Constraint declarations:

```
:- iiCHR_constraint watchedPainting/3*[file('log'),time_stamped,append].
:- iiCHR_constraint topicalityMeasure/3*[file('topicalities')].
:- iiCHR_constraint painting/3*[file(collection),read_only].
:- iiCHR_constraint newWatchedPainting/3.
```

When a guest is reported to have watched a painting, his or her topicality measures are updated. We use (arbitrarily) a decay factor of 0.7 and an increment of 0.3. When the sum of these two numbers is 1, the measure will be between 0 and 1, converging to 1 for a theme that is repeated over and over.

Queries: `?- executeUpdate(newWatchedPainting(`*Guest*`,`*Painter*`,`*Title*`)).`
Rules:

```
newWatchedPainting(Guest,_,_) \ topicalityMeasure(Guest,Theme,W)
    <=> W1 is W * 0.7, topicalityMeasure(Guest,Theme,W1).
newWatchedPainting(Guest,Painter,Title), painting(Painter,Title,Theme)
        \ topicalityMeasure(Guest,Theme,W)
  <=> W1 is W + 0.3, topicalityMeasure(Guest,Theme,W1).
```

```
newWatchedPainting(Guest,Painter,Title), painting(Painter,Title,Theme)
  ==> not_exists topicalityMeasure(Guest,Theme,_)
  | topicalityMeasure(Guest,Theme,0.3).

newWatchedPainting(Guest,Painter,Title)
  <=> watchedPainting(Guest,Painter,Title).
```

To understand these rules, recall the code pattern for constraint replacement explained in section 4 above and which is applied here in an extended form. The first rule decays every measure for the given user, the second one increments it for the themes of the current painting, with the exception that the third rule applies in case there is no previous measure constraint for that user and theme. The last rule inserts a new `watchedPainting` constraint into the log.

5.4 The advisor's program

Constraints declarations:

```
:- iiCHR_constraint watchedPainting/3*[file('log'),time_stamped,read_only].
:- iiCHR_constraint topicalityMeasure/3*[file('topicalities'),read_only].
:- iiCHR_constraint painting/3*[file(collection),read_only].
:- iiCHR_constraint suggestAdvice/0.
:- iiCHR_constraint advice/4.
```

The `suggestAdvise` constraint is a trigger that initiates a search for possible relevant advice to give to the current guests, and a constraint `advice(`*Guest*`,` *Painter*`,` *Title*`,` *Theme*`,` *Weight*`)` represents an advice to be issued to a given *Guest* to see a given painting; the *Theme* and *Weight* represents the topicality measure that gave rise to the selection of this advice. We leave it unspecified how the advices produced in this way are communicated to the guests.

Queries: `?- executeQuery(newWatchedPainting(`*Guest*`,`*Painter*`,` *Title*`)).`

Rules:

```
suggestAdvice, topicalityMeasure(Guest,Theme,M),
                   painting(Painter,Title,Theme)
  ==> M > 0.63, not_exists watchedPainting(Guest,Painter,Title)
  | advice(Guest,Painter,Title,Theme,M).

advice(Guest,_,_,_,W1) \ advice(Guest,_,_,_,W2) <=> W1 >= W2 | true.
```

The first rule generates an advice for every guest and painting having a theme with a topicality measure greater that a certain threshold, however only if the guest has not seen that picture already. The threshold 0.63 indicates that either the 3 most recent paintings or 4 out of 5 most recent are tagged with the given theme. The last rule removes all but one best advice for each guest.

6 Evaluation of the example

In examining the suitability of *ii*CHR for intelligent and interactive installations, we consider it already passed with a high mark with respect to the "intelligent" dimension, i.e., reasoning, based on the comprehensive literature on this issue referenced above. For the interaction and coordination aspects, we may assess the collection of programs developed in the previous section, which concerns mainly knowledge base maintenance and no advanced reasoning. As it appeared, it was natural to program in a procedural style in which propagation and simpagation rules were interesting for their side effects on the constraint store.

This sort of programming may – at first glance – appear rather awkward to many developers, those who normally think in terms of explicit loops, tests and database statements, as well as those familiar with declarative programming in CHR in isolated contexts. However, we also observe that our programs were based on few and described design patterns that can be learned.

We included all the necessary code in the example section above, nothing was left out, and the number of code lines is actually quite small. As a thought experiment, we may consider an implementation using Java and JDBC, that would likely result in much longer code with at least as many strange maneuvers to get everything right. To this comparison, we can add that *ii*CHR is a powerful reasoning system, so we can continue to develop our application adding more and more advanced analyses of user behaviour within the same framework, which is rather hypothetical to consider in a 100% imperative setting such as Java.

7 Related work

Potential concurrency for CHR has attracted attention, especially for confluent programs, for which the order of rule applications is immaterial. We can imagine several processors working in parallel applying their own sets of rules to the same constraint store; see [2] for overview and references to primary literature. A notion of transactions for CHR has been proposed by [16] and which will be interesting to investigate for our purpose, so that each query executed in an *ii*CHR program is made into such a transaction.

Where our system somehow fakes that all processes work on the same constraint store, by constantly reading and writing files, the mentioned ideas could be used to implement a system in which the different programs actually accessed the same constraint store. We are not aware of any efficient and workable implementation of such systems that are ready to be applied in the sort of installations that we have in mind. Notice also that the `locking` option of *ii*CHR's constraint declarations provides a simple and effective way of defining transactions.

A mechanism has been suggested by [17] for adaptation of CHR proofs once produced to changing input constraints, i.e., for small changes in the environment, a small adjustment of the proof may lead to an updated conclusion. Such methods may be used for tracking the users of an installation, but it is not clear to us whether it can be used to model accumulation and refinement of knowledge over time. The cost of maintaining a data structure for proof trees may also outbalance the gain of reusing previous proof steps.

Efficient compilation of CHR into imperative languages such as C or Java are described in [18] that also reports efficient implementations. Such combined paradigms may also be interesting candidates for adding reasoning components to interactive installations. We have not studied this in detail, but we may expect some difficulties due to impedance mismatch between the two worlds in one.

We are not aware of other work on rule-based systems in interactive installations, that goes beyond having reasoning as a separate subsystem with no real integration. The idea of processes communication through a common constraint store is present in the paradigm of Concurrent Constraint Programming (CCP) [19]; it is not directly comparable as CCP does not include its own rule-based language for defining constraints. An application of CCP to interactive systems is described in [20]. There are also some similarities with the Linda system [21] from the 1980s in which parallel processes communicate through a common tuple store.

8 Conclusion

We have presented an adaptation of CHR, in the shape of the *ii*CHR framework, to be used for interactive installations, and which allows a fairly straightforward cooperation between different processes collaborating around the same body of accumulated knowledge. We have used a rather unsophisticated, but effective common representation of knowledge in terms of text files, which is also easily accessible from system components written in other programming languages. However, seen from the *ii*CHR developer's point of view the actual representation is more or less immaterial as reading and writing of these files are integrated in a natural way into the querying facilities.

Another possible way to use CHR to maintain a shared, developing knowledge base may be to have a central "knowledge server" as a perpetually running CHR process, that waits for external requests and executes them one by one in the developing constraint store. However, this approach may by vulnerable due to potential memory leaks, failures and runtime errors.

We have considered using a database system for storing constraints instead of using text files, but the conversion of the output from the database into constraints would be at least as time consuming as reading a plain text file.

However, it may be interesting to integrate into *ii*CHR an additional sort of database resident constraints intended for very large constraint sets with limited operations. Only small portions determined by a search key should to be loaded into CHR, and updating could be restricted forms that are easily translated into INSERT and UPDATE statements of SQL. In forthcoming work, we plan to make experiments of developing real applications running in the Experience Cylinder framework [4] mentioned in section 2 to obtain more experience in order to test and refine the *ii*CHR design.

References

1. Frühwirth, T.W.: Theory and practice of Constraint Handling Rules. Journal of Logic Programming **37**(1-3) (1998) 95–138
2. Frühwirth, T.W.: Constraint Handling Rules. Cambridge University Press (2009)

3. Christiansen, H.: Executable specifications for hypothesis-based reasoning with Prolog and Constraint Handling Rules. J. Applied Logic **7**(3) (2009) 341–362
4. Andreasen, T., Gallagher, J.P., Møbius, N., Padfield, N.: The Experience Cylinder, an immersive interactive platform. In Emonet, R., Florea, A.M., eds.: AMBIENT 2011, The First International Conference on Ambient Computing, Applications, Services and Technologies, ThinkMind (2011) 25–31
5. Abdennadher, S., Christiansen, H.: An experimental CLP platform for integrity constraints and abduction. In: Proceedings of FQAS2000, Flexible Query Answering Systems: Advances in Soft Computing series, Physica-Verlag (Springer) (2000) 141–152
6. Christiansen, H., Dahl, V.: HYPROLOG: A new logic programming language with assumptions and abduction. In Gabbrielli, M., Gupta, G., eds.: ICLP. Volume 3668 of Lecture Notes in Computer Science., Springer (2005) 159–173
7. Christiansen, H.: On the implementation of global abduction. In Inoue, K., Satoh, K., Toni, F., eds.: CLIMA VII. Volume 4371 of Lecture Notes in Computer Science., Springer (2006) 226–245
8. Christiansen, H.: CHR Grammars. Int'l Journal on Theory and Practice of Logic Programming **5**(4-5) (2005) 467–501
9. Christiansen, H., Dahl, V.: Meaning in Context. In Dey, A., Kokinov, B., Leake, D., Turner, R., eds.: Proceedings of Fifth International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-05). Volume 3554 of Lecture Notes in Artificial Intelligence. (2005) 97–111
10. Christiansen, H., Dahl, V.: Abductive logic grammars. In Ono, H., Kanazawa, M., de Queiroz, R.J.G.B., eds.: WoLLIC. Volume 5514 of Lecture Notes in Computer Science., Springer (2009) 170–181
11. Sneyers, J.: Constraint Handling Rules. Website (checked 2012) `http://dtai.cs.kuleuven.be/projects/CHR/`.
12. SWI development team: SWI Prolog Website (checked 2012) `http://www.swi-prolog.org/`.
13. Christiansen, H.: *ii*CHR. Website (available autumn 2012) `http://www.ruc.dk/~henning/iiCHR/`.
14. Van Weert, P., Sneyers, J., Schrijvers, T., Demoen, B.: Extending CHR with negation as absence. In: Proceedings of the Third Workshop on Constraint Handling Rules. (2006) 125–139
15. Andersen, L.: JDBC 4.0 Specification. Sun Microsystems, Inc., JSR 221 (2006) `http://jcp.org/aboutJava/communityprocess/final/jsr221/`.
16. Schrijvers, T., Sulzmann, M.: Transactions in Constraint Handling Rules. In de la Banda, M.G., Pontelli, E., eds.: ICLP. Volume 5366 of Lecture Notes in Computer Science., Springer (2008) 516–530
17. Wolf, A., Gruenhagen, T., Geske, U.: On the incremental adaptation of CHR derivations. Applied Artificial Intelligence **14**(4) (2000) 389–416
18. Van Weert, P., Wuille, P., Schrijvers, T., Demoen, B.: CHR for imperative host languages. In Schrijvers, T., Frühwirth, T.W., eds.: Constraint Handling Rules. Volume 5388 of Lecture Notes in Computer Science. Springer (2008) 161–212
19. Saraswat, V.A., Rinard, M.C., Panangaden, P.: Semantic foundations of concurrent constraint programming. In Wise, D.S., ed.: POPL, ACM Press (1991) 333–352
20. Toro-Bermudez, M., Rueda, C., Agon, C., Assayag, G.: NTCCRT: A concurrent constraint framework for real-time interaction. In: Proc. International Computer Music Conference, ICMC 2009. (2009) 93–96
21. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7**(1) (1985) 80–112

Pros and Cons of Using CHR for Type Inference

János Csorba, Zsolt Zombori, and Péter Szeredi

Department of Computer Science and Information Theory
Budapest University of Technology and Economics
{csorba,zombori,szeredi}@cs.bme.hu

Abstract. We report on using logic programming and in particular the Constraint Handling Rules extension of Prolog to provide static type analysis for the Q functional language. We discuss some of the merits and difficulties of CHR that we came across during implementation of a type inference tool.

Keywords: logic programming, static type inference, CSP, CHR, Q language

1 Introduction

Our paper presents an application of Constraint Handling Rules (CHR) for the type analysis of the Q functional language. We implemented the `qtchk` *type inference* tool, which has been developed in a collaborative project between Budapest University of Technology and Economics and Morgan Stanley Business and Technology Centre, Budapest.

The main goal of the type inference tool is to detect type errors and provide detailed error messages explaining the inconsistency. The `qtchk` program infers the possible types of all expressions in the program. Consequently, for any syntactically correct Q program the analyser will detect type inconsistencies, and will assign a type to each type-consistent expression of the Q program at hand.

We reported on the main issues of the type inference application in our ICLP2012 paper [9]. There we described type inference as a Constraint Satisfaction Problem (CSP) and presented how the task of type analysis can be mapped onto a CSP. In the present paper we focus on the implementation details: how we solved this problem using the Constraint Handling Rules extension of Prolog [4, 6].

In Section 2 we briefly introduce the Q language and provide some background information. In Section 3 we give an overview of the main issues and of the implementation details of the constraint based type inference tool for Q. Section 4 is devoted to the discussion of some difficulties in using CHR. In Section 5 we provide an evaluation of using CHR.

2 Preliminaries

In this section we give some background to our work following [9], where the reader can find more details. We first introduce the Q programming language, and then give an overview of the type language developed for Q. Finally we discuss the mapping of a type inference task into a Constraint Satisfaction Problem.

2.1 The Q Programming Language

Q is a highly efficient vector processing functional language, which is well suited to performing complex calculations quickly on large volumes of data.

Types Q is a strongly typed, dynamically checked language. This means that while each variable, at any point of time, is associated with a well defined type, the type of a variable is not declared explicitly, but stored along its value during execution. The most important types are as follows:

– **Atomic types** in Q correspond to those in SQL with some additional date and time related types that facilitate time series calculations.[1]
– **Lists** are built from Q expressions of arbitrary types, e.g. `(1;2.2;‘abc)` is a list comprising two numbers and a symbol.
– **Dictionaries** are a generalisation of lists and provide the foundation for tables. A dictionary is a mapping that is given by exhaustively enumerating all domain-range pairs.
– **Tables** are lists of special dictionaries that correspond to SQL records.
– **Functions** correspond to mathematical mappings specified by an algorithm.

Main Language Constructs Q being a functional language, functions form the basis of the language. A function is composed of an optional parameter list and a body comprising a sequence of expressions to be evaluated. Function application is the process of evaluating the sequence of expressions obtained after substituting actual arguments for formal parameters. For example, the expression `f: {[x] $[x>0;sqrt x;0]}` defines a function of a single argument x, returning $\sqrt{x}$, if $x > 0$, and 0 otherwise.

Some built-in functions (dominantly mathematical functions) with one or two arguments have a special behaviour called *item-wise extension*. Normally, the built-in functions take atomic arguments and return an atomic result of some numerical calculation. However, these functions extend to list arguments item-wise. If a unary function is given a list argument, the result is the list of results obtained by applying the function to each element of the input list. A binary function with an atom and a list argument evaluates the atom with each list element. When both arguments are lists, the function operates pair-wise on elements in corresponding positions. Item-wise extension applies recursively in case of deeper lists, e.g. `((1;2); (3;4)) + (0.1; 0.2) = ((1.1;2.1); (3.2;4.2))`

While being a functional language, Q also has imperative features, such as multiple assignment[2] of variables and loops.

[1] Q has the following 16 atomic types: `boolean`, `byte`, `short`, `int`, `long`, `real`, `float`, `char`, `symbol`, `date`, `datetime`, `minute`, `second`, `time`, `timespan`, `timestamp`.
[2] Assignment is denoted by a colon, e.g. `x:x*2` doubles the value of the variable `x`.

2.2 Type Language for Q

In this subsection we describe the type language developed for Q. We allow polymorphic type expressions, i.e., any part of a complex type expression can be replaced with a variable. Expressions are built from atomic types and variables using type constructors. The abstract syntax of the type language – which is at the same time the Prolog representation of types – is as follows:

```
TypeExpr =
      AtomicTypes     | TypeVar     | symbol(Name)     | any
    | list(TypeExpr) | tuple([TypeExpr,...,TypeExpr])
    | dict(TypeExpr, TypeExpr)     | func(TypeExpr, TypeExpr)
```

AtomicTypes This is shorthand for the 16 atomic types of Q. Furthermore, the `numeric` keyword is used to denote a type consisting of all numeric values.

TypeVar represents an arbitrary type expression, with the restriction that the same variables stand for the same type expression. Type variables make it possible to define polymorphic type expressions, such as `list(A) -> A` (a function mapping a list of a certain type to a value of the given type) and `tuple([A,A,B])`.

`symbol(`*Name*`)` The named symbol type is a degenerate type, as it has a single instance only, namely the provided symbol. Nevertheless, it is important because in order to support certain table operations, the type reasoner needs to know what exactly the involved symbols are.

`any` This is a generic type description, which denotes all data structures allowed by the Q language.

`list(`*TE*`)` The set of all lists with elements from the set represented by *TE*.

`tuple([`TE_1`, ..., `TE_k`])` The set of all lists of length k, such that the i^{th} element is from the set represented by TE_i.

`dict(`TE_1`,`TE_2`)` The set of all dictionaries, defined by an explicit association between domain list (TE_1) and range list(TE_2) via positional correspondence.

`func(`TE_1`, `TE_2`)` The set of all functions, such that the domain and range are from the sets represented by TE_1 and TE_2, respectively.[3]

2.3 Type Inference as a Constraint Satisfaction Problem

In this subsection we give an overview of our approach of transforming the problem of type reasoning onto a CSP. Type reasoning starts from program code that can be seen as a complex expression built from simpler expressions. Our aim is to assign a type to each expression appearing in the program in a coherent manner. The types of some expressions are known immediately, while other kinds of information are provided by the program syntax, which imposes restrictions between the types of certain expressions. The aim of the reasoner is to assign a type to each expression that satisfies all the restrictions.

[3] To help readability, we often use the notation `A -> B` instead of `func(A,B)`.

We associate a CSP variable with each sub-expression of the program. Each variable has a domain, which initially is the set of all possible types. Different type restrictions can be interpreted as constraints that restrict the domains of some variables. In this terminology, the task of the reasoner is to assign a value to each variable from the associated domain that satisfies all the constraints. However, our task is more difficult than a classical CSP, because there are infinitely many types (e.g. tuples can be of arbitrary length), which cannot be represented explicitly in a list. Representing infinite domains is a challenge for performing type reasoning.

Partial Ordering We say that type expression T_1 is a subtype of type expression T_2 ($T_1 \leq T_2$) if, and only if, all values that belong to T_1 also belong to T_2. The subtype relation determines a partial ordering over the type expressions.

For example, consider the `tuple([int,int])` type which represents all lists of length two, where both elements are integers. It is obvious that every value that belongs `tuple([int,int])` also belongs to `list(int)`, i.e., the type expression `tuple([int,int])` is a subtype of `list(int)`.

It is very easy to check whether the subtype relation holds between two type expressions. For atomic type expressions this is immediate. Complex type expressions can be checked using some simple recursive rules. For example, `list(A)` is a subtype of `list(B)` if, and only if, `A` is subtype of `B`.

Finite Representation of the Domain The domain of a variable is initially the set of all the types, which can be constrained with different upper and lower bounds, based on the partial ordering.

An upper bound restriction for variable X_i is a list $L_i = [T_{i1}, \ldots, T_{in_i}]$, meaning that the upper bound of X_i is $\bigcup_{j=1}^{n_i} T_{ij}$, i.e., the type of X_i is a subtype of some element of L_i. Disjunctive upper bounds are very common and natural in Q, for example, the type of an expression might have to be either `list` or `dict`. The conjunction of upper bounds is easily described by having multiple upper bounds. If variable X_u gets a new upper bound L_v (e.g. because it turns out that variable X_u is a subtype of X_v, and so X_u inherits the upper bound of X_v), this means that the value of X_u has to be in $\bigcup(T_{uj} \bigcap T_{vk})$, for all $1 \leq j \leq n_u$ and $1 \leq k \leq n_v$.

A lower bound restriction for variable X_i is a single type expression T_i, meaning that T_i is a subtype of X_i. For lower bounds, it is their union which is naturally represented by having multiple constraints: if X has two lower bounds T_1 and T_2, then $T_1 \cup T_2$ is a subtype of X. We do not use lists for lower bounds, so we cannot represent the intersection of lower bounds. We chose this representation because no language construct in Q yields a conjunctive lower bound.

3 Implementing Type Inference using CHR

We built a Prolog program called `qtchk` that implements the type reasoning as a Constraint Satisfaction Problem using the Constraint Handling Rules library.

It runs both in SICStus Prolog 4.1 [7] and SWI Prolog 5.10.5 [8]. It consists of over 8000 lines of code[4]. Q has many irregularities and lots of built-in functions (over 160), due to which a complex system of constraints had to be implemented using over 60 constraints. The detailed user manual for `qtchk` can be found in [3] that contains lots of examples along with the concrete syntax of the Q language.

The system has two main components: a parser and a type inference engine. The parser builds an abstract tree (AST) representation of the code, where each node represents a sub-expression. Afterwards, we traverse the AST and formulate CHR constraints on which type inference is performed. Both phases detect and store errors, which are presented to the user. In this section we focus on the implementation of the CHR based type inference component.

3.1 Representing variables

All subexpressions of the program are associated with CSP variables. If some constraint fails, we need to know which expression is erroneous in order to generate a useful error message. If the arguments of the constraints are Prolog variables, we do not have this information at hand. Hence, instead of variables we use identifiers `ID = id(N,Type,Error)`[5] which are Prolog terms with three arguments: an integer `N` which uniquely identifies the corresponding expression, the type proper `Type` (which is a Prolog variable before the type is known) and an error flag `Error` which is used for error propagation. We use the same representation for type variables in polymorphic types, e.g. the type `list(X)` may be represented by `list(id(2))`.

3.2 Constraint Reasoning

After parsing, the type analyser traverses the abstract syntax tree and imposes constraints on the types of the subexpressions of the program. The constraints describing the domain of a variable are particularly important, we call them *primary constraints.* These are the upper and lower bound constraints. We will refer to the rest of the constraints as *secondary constraints.* Secondary constraints eventually restrict domains by generating primary constraints, when their arguments are sufficiently instantiated (i.e., domains are sufficiently narrow).

Our aim is to eventually eliminate all secondary constraints. If we manage to do this, the domains described by the primary constraints constitute the set of possible type assignments to each expression. In case some domain is the empty set, we have a type error. Otherwise, the program is considered type correct.

If the upper and lower bounds on a variable determine a singleton set[6], then we know the type of the variable and we say that it is *instantiated.* If

[4] We are happy to share the code over e-mail with anyone interested in it.

[5] In order to make the examples easier to read, we will omit the two variable arguments of the `id/3` compound term, i.e. use `id(N)` instead of `id(N,Type,Error)`.
Also note that we will use the terms "variable" and "identifier" interchangeably.

[6] This is the case, e.g., when the lower and upper bounds are the same, or when there is an atomic upper bound.

all arguments of a secondary constraint are instantiated, then there are two possibilities. If the instantiation satisfies the constraint, then the latter can be removed from the store. Otherwise, the constraint fails.

Constraint reasoning is performed using the Constraint Handling Rules library of Prolog. In the next two paragraphs we describe how these constraints interact with each other.

Interaction of Primary Constraints Primary constraints represent variable domains. If a domain turns out to be empty, this indicates a type error and we expect the reasoner to detect this. Hence, it is very important for the constraint system to handle primary constraints as "cleverly" as possible. For this, we formulated rules to describe the following interactions on primary constraints:

– Two upper bounds on a variable should be replaced with their intersection.
– Two lower bounds on a variable should be replaced with their union.
– If a variable has an upper and a lower bound such that there is no type that satisfies both, then the clash should be made explicit by setting the upper bound to the empty set.
– Upper and lower bounds can be polymorphic, i.e., might contain other variables. Since lower bounds must be subtypes of upper bounds, we can propagate constraints to the variables appearing in the bounds.

We illustrate our use of CHR by presenting some rules that describe the interaction of primary constraints. Our two primary constraints are

– `subTypeOf(ID,L)`: The type of identifier `ID` is a subtype of some type in `L`, where `L` is a list of polymorphic type expressions.
– `superTypeOf(ID,T)`: The type of identifier `ID` is a supertype of type `T`, a polymorphic type expression.

With polymorphic types we can restrict the domain by a type expression containing the – not yet known – type of another identifier. If the type of such an identifier becomes known, the latter is replaced by the type in the constraint. For example, consider the following two constraints:

```
subTypeOf(id(1),[float,list(id(2))])
superTypeOf(id(1),tuple([id(3),int])
```

Suppose the types of `id(2)` and `id(3)` both turn out to be `int`. Then the above two constraints are automatically replaced with constraints:

```
subTypeOf(id(1),[float,list(int)])
superTypeOf(id(1),tuple([int,int])
```

Due to the lower bound, `float` can be eliminated from the upper bound. This is performed by the following CHR rule:

```
superTypeOf(X,A) \ subTypeOf(X,B0) <=> eliminate_sub(A, B0, B) |
      subTypeOf(X, B).
```

Here, the Prolog predicate: `eliminate_sub(A,B0,B)` means that the list of upper bounds `B0` can be reduced to a proper subset `B` based on lower bound `A`.

To conclude the above example, we obtain:

```
subTypeOf(id(1),[list(int)])
superTypeOf(id(1),tuple([int,int])
```

In another example, we show how two upper bounds on the same identifier are handled. Suppose we have the following constraints:

```
subTypeOf(id(1),[float,list(int)])
subTypeOf(id(1),[tuple([int,int]),func(int,float)])
```

The upper bounds trigger the following CHR rule:

```
subTypeOf(X,T1), subTypeOf(X,T2) <=> type_intersection(T1,T2,T) |
      create_log_entry(intersection(X,T1,T2, T)),
      subTypeOf(X,T).
```

The predicate `type_intersection(T1,T2,T)` posts a constraint stating `T` is the intersection of `T1` and `T2`. We obtain a single upper bound:

```
subTypeOf(id(1),[tuple([int,int])])
```

Interaction of the Secondary Constraints Unfortunately, it is not realistic to capture all interactions of secondary constraints as that would require exponentially many rules in the number of constraints. Hence, we only handle the interaction of secondary constraints with primary constraints. This means, we do not have any CHR rules with multiple secondary constraints in their heads. Secondary constraints restrict domains by generating the proper primary and secondary constraints, when the domains of their arguments are sufficiently narrow: if certain arguments of the constraints are within a certain domain, then some other argument can be restricted further.

We obtain most of our secondary constraints from the program syntax. In general, a syntactic construct imposes restrictions on the types of its subconstructs. E.g., the type of the left side of an assignment has to be at least as "broad" as the type of the right side. Similarly, for every built-in function, there is a well-defined relation between the types of its arguments and the type of the result. These relations can be expressed with corresponding CHR constraints.

For example, we use the secondary constraint `sum/3` to capture the relation between the types of arguments and that of the result of the built-in function '+'. Let us consider the Q expression `x+y`, and let the types associated with `x`, `y` and `x+y` be `id(1)`, `id(2)`, and `id(3)`, respectively. This Q expression gives rise to the secondary constraint `sum(id(1), id(2), id(3))`. If the first argument of `sum/3` turns out to be integer, then the type of the second argument and the type of the result must be the same (according to the behaviour of the function '+' in Q). Consequently, the `sum` constraint can be removed from the constraint store, and a new constraint `eq(id(2), id(3))` is added, expressing the equivalence of two types. This is performed by the following CHR simplification rules:

```
sum(X,Y,Z) <=> known_type(X,int) | eq(Y,Z).
sum(X,Y,Z) <=> known_type(Y,int) | eq(X,Z).
```

3.3 Error Handling

During constraint reasoning, a failure of Prolog execution indicates some type conflict. In such situations, before we roll back to the last choice point, we remember the details of the error. We maintain a log[7] that contains entries on how various domains change during the reasoning and what constraints were added to the store. Furthermore, to make error handling more uniform, whenever secondary constraints are found violated, they do not lead to failure, but they set some domain empty. Hence, we only need to handle errors for primary constraints. Whenever a domain gets empty, we mark the expression associated with the domain and we look up the log to find the domain restrictions that contributed to the clash. We create and assert an error message and let Prolog fail. For example, the following message

```
Expected to be broader than (int -> numeric) and
               narrower than (int -> int)
  file:samples/s1.q  line:13  character:4
   {[x] f[x]}
   ^^^^^^^^^^
```

indicates that the underlined function definition is erroneous: the return value is numeric or broader (inferred from the type of `f`), although it is supposed to be narrower than integer (inferred from a type declaration).

3.4 Labeling

After all constraints are added to the constraint store, we use labeling to find a type assignment to each program expression (i.e., to each identifier associated with a node of the abstract syntax tree) that satisfies the constraints. This involves another traversal of the abstract syntax tree to make sure no program expression is left without a type assignment. We select the next identifier `X` to be labelled and set its domain to a singleton set, based on its current domain. We implemented this by adding a new constraint `label(X)`. This constraint triggers the narrowing of the domain of `X` through the following CHR rules:

```
label(X) <=> id_known_type(X,_) | true.
label(X), superTypeOf(X,A), subTypeOf(X,L) <=>
        label_upwards(X,A,L,Type),
        hasType(X,Type).
label(X), superTypeOf(X,A) <=>
        label_upwards(X,A,[any],Type),
```

[7] We use the `create_log_entry` procedure in all CHR rules to facilitate creating error messages.

```
        hasType(X,Type).
label(X), subTypeOf(X,L) <=>
        label_downwards(X,L,Type),
        hasType(X,Type).
label(X) <=>
        label_downwards(X,[any],Type),
        hasType(X,Type).
```

First, we check if the type of `X` is already known. If so, we do nothing. Otherwise, we have four cases based on the presence or absence of a lower and upper bound:

– If we have a lower and an upper bound, we nondeterministically select a type from the domain. We start from the lower bound and successively try the broader types. This directionality is comfortable for implementation, because while a type might have many subtypes (e.g. any tuple of integers is a subtype of the type 'list of integers'), it has only few supertypes.
– If only a lower bound is present, we set the upper bound to `any` and proceed as in the previous case.
– If only an upper bound is present, we start from that type and go successively to its subtypes.
– If there is neither a lower, nor an upper bound, then we assume an implicit upper bound `any` and proceed as above.

Note that the `hasType/2` constraint, use above in the labeling code, translates to an upper and a lower bound:
`hasType(X,Y):- subTypeOf(X,[Y]), superTypeOf(X,Y).`

4 Difficulties

In this section, we discuss some difficulties that we had to overcome during the implementation of the type inference tool. These problems arose on the one hand from some special features of the Q language, and on the other hand from some limitations of the CHR library used. We hope that these experiences can be useful for the CHR community.

4.1 Handling Meta-Constraints

As we described earlier, several built-in functions of Q have a special behaviour, called item-wise extension. We discuss the implementation of this feature now.

Let us consider, for example, the constraint `sum` which captures the relation between the arguments and the result of the built-in function '+'. If some of the arguments turn out to be lists, then the relation should be applied to the types of the list elements. We could capture this by adding adequate rules to the `sum` constraint. However, the rules describing the list extension behaviour would have to be repeated for each built-in function, which is counter-productive. Instead, we introduced a meta-constraint `list_extension/3`.

Consider a binary built-in function f, which extends item-wise to lists in both arguments and which imposes constraints Cs on its atomic arguments and result. Suppose that f has argument types identified by X, Y and a result type identified by Z. We cannot add the constraints of Cs to the constraint store until we know that the arguments are all of atomic type. Instead, we use the meta-constraint list_extension(Dir,Args,Fun), where Dir specifies which arguments can be extended item-wise to lists, Args is the list of arguments on which the list of constraints[8] imposed by function Fun, will have to be formulated.

Hence, the constraint list_extension(both,[X,Y,Z],+) is added in our example. If later the input arguments are inferred to be atomic, then the meta-constraint list_extension/3 adds the atomic constraints Cs and removes itself:

```
subTypeOf(X,Ux), subTypeOf(Y,Uy) \
  list_extension(both,[X,Y,Z],Fun) <=> nonlist(Ux), nonlist(Uy) |
  list_ext_constraints(Fun,[X,Y,Z],Cs), ( foreach(C,Cs) do C ).
```

Here, the complicated part is to find the arguments of the proper constraints imposed by the given built-in function. We solved this by asserting the relevant information in the list_ext_constraints predicate. E.g. in the case of the Q function '+' we have the following fact:

```
list_ext_constraints(+, [A,B,C], [sum(A,B,C)]).
```

If, on the other hand, some argument turns out to be a list, the meta-constraint is replaced by another one. For example, if we know that the types of X and Y are list(A) and list(B), then the type of Z must be a list as well and we replace the list_extension constraint with the following two constraints: list_extension(both,[A,B,C],+) and hasType(Z,list(C)).

In fact, the difficulty of the implementation was caused by the following restriction of CHR: it is not possible to refer to a constraint in a rule head by supplying a variable holding its name and a list of its arguments (cf. the call/N built-in predicate group of Prolog).

To express item-wise extension, it would be more convenient to write rules where the name of a constraint can also be a variable. If such "meta-rules" were available the list_extension meta-constraint would become unnecessary.

For example, in the case of unary functions, where the corresponding constraint has two arguments (the input and the output types), item-wise extension could be implemented using the following, quite natural "meta-rule"[9]:

```
call(Cons,A,B) <=> is_list(A,X), is_list_extensible(Cons) |
      call(Cons,X,Y), hasType(B,list(Y)).
```

where is_list_extensible(Cons) succeeds exactly when Cons has the list-extension behaviour, is_list(A,X) means that the type of A is list(X).

[8] Note that there are several built-in functions, whose type is described using more than one constraint.

[9] Here we assume that CHR supports meta-constraints in rule heads using the call/N formalism of Prolog.

4.2 Copying Constraints over Variables

Local variables are made globally unique by the parser. This means, that variables with the same name have the same value, so we can constrain their types to be the same. However each occurrence of a variable that holds a polymorphic function can have a different type assigned. Let us show an example:

```
f:{[x] x+2}                     (1)
...
f [2]                           (2)
...
f [1.1f]                        (3)
```

In the first line, `f` is defined to be a function having a single argument `x` which returns `x+2`. This means that the type of `f` is a (polymorphic) function which maps `A` to `B` (`A -> B`), where a secondary constraint `sum(A, int, B)` holds between the argument and the result. In (2) and (3) there are two different applied occurrences of function `f`, which specialise this `sum` constraint in two independent ways. In these examples `f` is applied to an integer and to a float, therefore the types of the second and third occurrence of `f` are `int -> int` and `float -> float`.

The above example shows that if the type of a variable is a (polymorphic) function then we cannot assume that the type of an applied occurrence is the same as that of the defining occurrence. To capture the relationship between these types we introduced a relationship, called "specialisation", which holds if the type of the applied occurrence can be obtained from that of the defining occurrence by first copying it and then substituting zero or more type variables in it with (possibly polymorphic) types.

A straightforward natural implementation of the "specialisation" constraint would be the following:

- at the defining occurrence of a variable: post the relevant type constraints;
- at the applied occurrence of a variable: read the type constraints posted for a variable and apply the "specialisation" relationship.

This approach requires that the CHR library provides means for accessing the constraints that involve a specified argument, a feature similar to the `frozen(X. Goals)` built-in predicate of SICStus Prolog. Unfortunately, the CHR implementations we used do not have this feature. This means that a Q variable holding a polymorphic function has to be treated specially: the constraints involving its type have to be collected and remembered, so that they can be accessed at the applied occurrences of the given Q variable.

We strongly believe that in order to support this and similar use cases, CHR libraries should provide access to the constraints that are in the store.

4.3 Handling Equivalence Classes of Variables

The constraint system yields lots of equalities. For example, two occurrences of the same (non-function-valued) variable give rise to an equality constraint.

One way to handle this is to propagate all primary constraints between equal variables, i.e. whenever $X = Y$, Y inherits all primary constraints of X and the other way round. For example, a simple implementation of propagating the upper bounds in the equality constraint (`eq/2`) would be the following:

```
eq(X,Y), subTypeOf(X, T) ==> subTypeOf(Y,T).            (1)
eq(X,Y), subTypeOf(Y, T) ==> subTypeOf(X,T).            (2)
```

Unfortunately, this solution is rather inefficient, since all reasoning is repeated for each variable made equal to some other one. Moreover, we have found cases which lead to an infinite propagation of CHR constraints. In the following paragraph we outline an example of this.

As we have seen in Section 3 two upper bounds on a variable are replaced with their intersection. Let us suppose that variable `A` has two upper bounds `list(X)` and `list(Y)`. There is an *intersection rule* which replaces these two with the upper bound `list(Z)`, where `Z` is a new variable and $\texttt{Z} \leq \texttt{X}$ and $\texttt{Z} \leq \texttt{Y}$ also have to be satisfied. Consider the following state of the constraint store:

```
eq(id(1), id(2)),
subTypeOf(id(1), [list(id(3))]),
subTypeOf(id(2), [list(id(4))]).
```

First the equality rule can fire, yielding two upper bounds on `id(1)` and `id(2)`. Now, the intersection rule can produce new upper bounds on these variables, which can be propagated to the other variable by the equality rule again.

It is easy to show that the above constraint store yields an infinite loop using these rules. Consider the following condition C: The two variables (`id(1)` and `id(2)`) have got at least two upper bounds in total, where each variable has at least one, and there exist two upper bounds, on which the intersection rule has not fired yet. It is easy to see that if C holds, then at least one rule can fire (intersection, or equality). On the other hand C is an invariant condition, as when any of these two rules fire. C remains true, if it was true before. Together with the initial state, where C also holds, this constraint store yields an infinite loop (regardless of the rule execution order).

The problem is caused by repeating the reasoning at each equal identifier. We solved this by introducing a directionality to the constraint propagation: we take a strict total order on identifiers and only propagate constraints towards the smaller identifier. The smallest in a set of equal identifiers thus represents the whole set in the sense that it accumulates all constraints.[10] Once the type of the smallest identifier becomes known, it gets propagated back to the other identifiers. Hence, instead of `eq(X,Y)` we introduced the constraint `represented_by(X,Y)`, where $\texttt{Y} \leq \texttt{X}$ holds. Furthermore, for all constraints C we have a new rule, which states that if `X` is represented by `Y` and `X` occurs in C, then it should be substituted with `Y`. As we could not formulate meta-constraints with CHR, we had to provide propagation rules for every single constraint. For example, in case of the constraint `sum` we needed the following code:

[10] This is similar to how Prolog handles the unification of two variables.

```
represented_by(A,B) \ sum(A,C,D) <=> sum(B,C,D).
represented_by(A,B) \ sum(C,A,D) <=> sum(C,B,D).
represented_by(A,B) \ sum(C,D,A) <=> sum(C,D,B).
```

This yielded lots of new rules, however, it was easy to generate them automatically, using a small Prolog program.

There are efficiency problems even with this solution. Suppose we have the following constraint: `c(...,id(2),...)` and a propagation rule R, whose head matches the above constraint (possibly involving other heads) and the body of the rule contains a new CHR constraint: `d(...,id(2),...)`. If `id(2)` later turns out to be equivalent to `id(1)`, then we substitute `id(2)` with `id(1)` in every constraint that contains `id(2)`. This yields a store with constraints:

```
c(...,id(1),...)
d(...,id(1),...)
```

The propagation rule (R) can fire now, which might infer the second constraint (`d`) again. In order to avoid further efficiency losses we added idempotency rules for every constraint, that is, we remove duplicate constraints.

However, this solution also has a negative consequence. It is possible that duplicate constraints yield redundant inferences, if these are fired before the idempontency rules. Consider the following example: let the constraint store contain constraints C_i for all $1 \leq i \leq n$, furthermore let us suppose we have propagation rules (R_j): C_j => C_{j+1} for all $1 \leq j \leq n-1$. Let us examine what happens when C_1 is a constraint inferred redundantly (twice), as described above. If the R_j rules are fired before the idempontency rules, then it is possible that we infer all C_i constraints twice, before eliminating the duplicates. This results in $2n$ inference steps instead of the optimal 1 (if the duplicate C_1 is eliminated before applying rules R_j). The problem occurs because we have no control over the firing order of CHR rules with different heads. We believe that a way to prescribe the order of such rules, e.g. using some priorities, would often help in improving the efficiency of CHR applications.

The solution of this problem of redundant inference is still an open question.

4.4 Labeling

The implementation of labeling posed several challenges. We noticed that the order in which identifiers are selected is crucial for efficiency. For example, it is important to label subexpressions first and then find the type of a complex expression. Another example is the function application, where labeling should first assign a type to the input and then the type of the output is typically automatically inferred by the constraints. Consequently, labeling involves a traversal of the abstract syntax tree, and at each node we decide the order in which expressions are labelled based on the syntactic construct involved. Often we had to rely on heuristics as it was hard to guess what order would work best in practice.

The next difficulty arises when we already know which identifier to label, and we have to choose a value. The set of all types is infinite, so we cannot try all values for a variable during labeling. hence we made some restrictions. First, we only allow a fixed increase in the term depth of types. This depth increase was experimentally set to two. E.g. if X is known to be a subtype of `list(any)`, then we replace `any` with terms of depth at most two. Hence, we will *not* replace `any` with `list(list(list(int)))`.

Second, we restrict using the `tuple` type. This is needed because tuples can have arbitrarily many arguments. If there is an identifier X and neither its lower nor its upper bound contains the `tuple` type, then we do not assign a tuple type to it. E.g. if X has the upper bound `list(int)`, then we only try `list(int)`. If, however, X also has a lower bound `tuple([int,int])`, then we try both `tuple([int,int])` and `list(int)`. We have found no Q programs where these restrictions led labeling astray: not finding an existing assignment. This is because nested types are not typical in Q and because our constraint system tends to recognise the need for a `tuple` type before labeling.

The main challenge of labeling comes from the fact that it aims to traverse a huge search space. The abstract syntax tree can have many nodes even for moderately long programs, hence we have many identifiers. Besides, Q programs are typically full of ambiguous expressions (in terms of type), so without labeling, very few types are known for sure. All this amounts to labeling being the bottleneck of type inference.

A solution to this problem would be to find a good partitioning of the program, such that not all the tree is labelled together, but in smaller portions. Consider, for example, two function definitions. The first expression contains an expression E_1 that allows many different types. Labeling assigns one possible type to E_1 and then starts labeling the second function definition. Suppose the second definition contains a type error at expression E_2 which leads labeling to failure. Hence, we backtrack to the choice point at E_1, and assign another possible type to E_1. However, this type has nothing to do with the type mismatch – since it occurs in a different function definition, – and we get failure again at E_2. This cycle is repeated until all possible types for E_1 are tried and only then do we conclude that the contains a type error. This procedure could be made more efficient by placing a cut after labeling the first function definition, thus eliminating the irrelevant choice point. Realizing that the types of expressions in one piece of code are independent from those of another can lead to much smaller fragments to be labelled, which has the potential to drastically reduce the time spent on labeling. Dependency analysis ([1]) could be used to find a code partitioning. Also, some kind of intelligent backtracking ([2]) algorithm could be used to avoid unnecessary choice points. However, adapting these techniques to the Q language requires further work.

5 Evaluation and Future Work

In this section we discuss our motivation for using CHR in the implementation of the type inference tool and summarise our experiences. We also mention topics that we intend to explore in the near future to improve our type reasoner.

5.1 Why Use CHR?

As we have seen in Subsection 2.2, our types are not necessarily disjoint (e.g. list and tuple). If the type of an expression becomes known to be a list of integers (`list(int)`), it is possible that later it is further narrowed down to a specific tuple (e.g. `tuple(int,int)`). Such a behaviour would be quite difficult to achieve in a unification based (purely Prolog) setup. This recognition led us to handle the problem of type inference as a CSP. Nevertheless, the number of possible types – even with some depth limit – is so large that it is hard to imagine an efficient implementation based on the CLP(FD) library. Furthermore, mapping the types to natural numbers (required by most CLP(FD) libraries) is also a non-trivial task. Choosing CHR for type reasoning seemed to be a good decision, as it is flexible enough to handle the above problems. These considerations motivated the use of the CHR library.

5.2 Our Experiences with Using CHR

CHR has proved to be a good choice as it is a very flexible tool for describing the behaviour of constraints. In CHR, arbitrary Prolog structures can be used as constraint arguments, therefore it was natural to handle the special domain defined by the type language.

However, we also had negative experiences with CHR. As described in Section 4, it often would be more convenient if we could write "meta-rules" in CHR. The need to access the constraint store also arose in some situations. For efficiency reasons, we believe it would often be useful to be able to influence the firing order of rules with different heads. Furthermore, the most of the debugging of our CHR programs was seriously hampered by the lack of a tracing tool.

5.3 Future Work

Lots of difficulties arose from our decision to represent CSP variables with identifiers, instead of using logical variables. This complicates handling the type equality of expressions. We introduced identifiers to facilitate error handling: whenever a constraint fails, its identifier arguments allows for immediately pointing to the erroneous expression in the program. We intend to explore the possibility of returning to logical variables, since it promises to be much more efficient.

There is an extension of CHR which allows for providing rule priorities, called CHRrp [5], which could help avoiding some efficiency problems that we mentioned in Section 4.

We would also like to examine our constraint propagation mechanism in terms of soundness and completeness, in order to be able to make more precise statements about the output of the reasoner.

Conclusion

This paper summarised our experiences using CHR for type analysis of programming languages. We found CHR to be a valuable tool, however, we believe there is still room for improvement: giving the programmer greater control over the constraint reasoning mechanism could further increase programmer productivity.

Acknowledgements

The work reported in the paper has been developed in the framework of the project “Talent care and cultivation in the scientific workshops of BME” project. This project is supported by the grant TÁMOP - 4.2.2.B-10/1-2010-0009

We are grateful to the referees of the first version of our paper for their detailed and insightful comments. These comments form the basis of our future work plan.

References

1. Austin, T.M., Sohi, G.S.: Dynamic dependency analysis of ordinary programs. SIGARCH Comput. Archit. News 20(2), 342–351 (Apr 1992), `http://doi.acm.org/10.1145/146628.140395`
2. Baker, A.B.: Intelligent backtracking on constraint satisfaction problems: Experimental and theoretical results (1995)
3. Csorba, J., Szeredi, P., Zombori, Z.: Static Type Checker for Q Programs (Reference Manual) (2011), http://www.cs.bme.hu/~zombori/q/qtchk_reference.pdf
4. Fruehwirth, T.: Theory and Practice of Constraint Handling Rules. In: Stuckey, P., Marriot, K. (eds.) Journal of Logic Programming. vol. 37(1–3), pp. 95–138 (October 1998)
5. Koninck, L.D., Schrijvers, T., Demoen, B.: Chr rp: Constraint handling rules with rule priorities (2007)
6. Schrijvers, T., Demoen, B.: The K.U.Leuven CHR system: implementation and application. In: First Workshop on Constraint Handling Rules: Selected Contributions. pp. 1–5 (2004)
7. SICS: SICStus Prolog Manual version 4.1.3. Swedish Institute of Computer Science (September 2010), `http://www.sics.se/sicstus/docs/latest4/html/sicstus.html`
8. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. TPLP 12(1-2), 67–96 (2012)
9. Zombori, Z., Csorba, J., Szeredi, P.: Static Type Inference for the Q language using Constraint Logic Programming. In: 28th International Conference on Logic Programming (ICLP 2012). Leibniz International Proceedings in Informatics (LIPIcs) (2012)

Womb Grammars: Constraint Solving for Grammar Induction

Veronica Dahl and J. Emilio Miralles

Simon Fraser University, Burnaby, BC V5A-1S6, Canada,
`veronica@cs.sfu.ca, emiralle@sfu.ca`

Abstract. We present Womb Grammars, a novel constraint-based framework implemented in CHRG and particularly useful for inducing, from known linguistic constraints that describe phrases in a language called the source, the linguistic constraints that describe phrases in another language, called the target. We present as well an application that uses as source an existing language fairly related to the target. Next we propose and motivate an intriguing research thread that uses as source language a (non-natural but coupled with our framework, generatively very powerful) universal language of our own device. Finally, we discuss further ramifications of our work.

Keywords: Womb Grammars, grammar induction, constraint acquisition, universal language, parsing, CHRG, Constraint Based Grammars, Property Grammars

1 Introduction

Language processing typically involves analyzing or generating sentences in a given language. In the case of analysis, there is an implicit aim of *sanctioning the input* as correct or incorrect with respect to the grammar, which itself is assumed to be correct. Incorrect sentences will either just fail to be analyzed or be explicitly identified as incorrect, with or without a rationale being given. In the case of synthesis, the aim is to produce correct sentences or phrases from some given representation of their meaning. Also in this case the (purportedly correct) grammar sanctions sentences as correct (by producing them as output).

A less typical, but also useful language processing activity is that of *grammar sanctioning*, where the input's correctness is known and the grammar must be either completed, corrected or fully inferred. For instance, in the area of grammar induction, a formal grammar (or a subset thereof) is inferred from corpora of correct and incorrect sentences with the aid of machine learning methods. A recent survey of this field can be found in D'Ulizia et al [9].

In this article we propose a use of grammar sanctioning which, to the best of our knowledge, is novel: inferring a language's syntax given its lexicon, a sufficiently representative set of correct phrases in it, and the property-based syntax of another language. Section 2 presents the background and motivation. Section 3 describes our Womb Grammar Model, first in its Hybrid Parsing incarnation and next as Universal Grammar Parsing, and then discusses existing and possible extensions. Section 4 presents implementation details, and Section 5 our concluding remarks.

2 Motivation, Background

Since the advent of CHR [10] and of its grammatical counterpart CHRG [6], constraint-based linguistic formalisms can materialize through fairly direct methodologies. During a recent visit to Universite de Nice, we proposed the use of our CHRG implementation of Property Grammars [8] to address efficiency concerns in Jacques Farré's work on acquiring context-free grammars by applying error-repair techniques to the grammar (as opposed to the input sentence). Farré's work currently uses an extended version of the CYK algorithm in order to repair tree-based grammar rules by either suppressing, inserting or replacing a symbol, or by permuting two adjacent symbols.[1]

Although in both method and scope our present work is very distant from that work, it motivated us to try a constraint-based approach to the extended problem of inducing grammar constraints. This led us to explore the possibility of using the known constraints of related languages, which resulted in the present work.

Grammar induction has met with reasonable success using different views of grammar: a) as a parametrized, generative process explaining the data [15,12], b) as a probability model, so that learning a grammar amounts to selecting a model from a prespecified model family [4,16,7], and c) as a Bayesian model of machine learning [11].

Using linguistic information from one language for the task of describing another language has also yielded good results, albeit for specific tasks—such as disambiguating the other language [3], or fixing morphological or syntactic differences by modifying tree-based rules [13]—rather than for syntax induction.

This usually requires parallel corpora, an interesting exception being [7], where information from the models of two languages is shared to train parsers for two languages at a time, jointly. This is accomplished by tying grammar weights in the two hidden grammars, and is useful for learning dependency structure in an unsupervised empirical Bayesian framework.

Most of these approaches have in common the target of inferring *syntactic trees*. As noted, for example, in [2], constraint-based formalisms that make it possible to evaluate each constraint separately are advantageous in comparison with classical, tree-based derivation methods. For instance the Property Grammar framework [1] defines phrase acceptability in terms of the properties or constraints that must be satisfied by groups of categories (e.g. English noun phrases can be described through a few constraints such as precedence (a determiner must precede a noun), uniqueness (there must be only one determiner), exclusion (an adjective phrase must not coexist with a superlative), and so on).

[1] http://deptinfo.unice.fr/~jf/Airelles/publis/iwpt09.pdf

Rather than resulting in either a parse tree or failure, such frameworks characterize a sentence through the list of the constraints a phrase satisfies and the list of constraints it violates, so that even incorrect or incomplete phrases will be parsed.

As well, simpler models can be arrived at in less costly fashion by giving up syntactic trees as a focus and focusing instead on *grammar constraints*, also called *properties*. For instance, if we were to work with tree-oriented rules such as:

```
  np --> det, adj, n.
```

their adaptation into a language where nouns must precede adjectives would require changing every rule where these two constituents are involved. In contrast, by expressing the same rule in terms of separate constraints, we only need to change the precedence constraint, and the modification carries over to the entire grammar without further ado.

We therefore propose a method for inferring constraint-based grammars such as Property Grammars (PG) [1] from a known lexicon and a representative sample of correct phrases. We already have a working prototype implementation in terms of Constraint Handling Rule Grammars (CHRG) [6], which we are optimizing. Our method does not require parallel corpora, does not a priori rely on probability (although it could be thus enhanced), and exploits the known grammar of a (manifest or latent) language to complete the grammar of another language of which we only know the lexicon. Like preceding work, we start by relying on some syntactic structure that is known but imperfect, and aim at completing and perfecting it. We then throw away even this starting point, and derive our constraints from a universal grammar of our own device. Our proof-of-concept is restricted so far to simple noun phrases and admittedly, our success within it may not as easily percolate to more encompassing subsets of language. However it is an elegant and concise solution, already useful in itself, with the potential to generalize further.

3 The Womb Grammar Model

3.1 A first approach: Hybrid Parsing

The intuitive idea Let L^S (the source language) be a human language that has been studied by linguists and for which we have a reliable parser that accepts correct sentences while pointing out, in the case of incorrect ones, what grammatical constraints are being violated. Its syntactic component will be noted L^S_{syntax}, and its lexical component, L^S_{lex}.

Now imagine we come across a dialect or language called the target language, or L^T, which is close to L^S but has not yet been studied, so that we can only have access to its lexicon (L^T_{lex}) but we know its syntactic constraints overlap significantly with those of L^S—perhaps being a dialect of it, or a close “cousin”. If we can get hold of a sufficiently representative corpus of sentences in L^T

that are known to be correct, we can feed these to a hybrid parser consisting of L^S_{syntax} and L^T_{lex}. This will result in some of the sentences being marked as incorrect by the parser. An analysis of the constraints these "incorrect" sentences violate can subsequently reveal how to transform L^S_{syntax} so it accepts as correct the sentences in the corpus of L_T—i.e., how to transform it into L^T_{syntax}. If we can automate this process, we can greatly aid the work of our world's linguists, the number of which is insufficient to allow the characterization of the myriads of languages and dialects in existence. Figures 1 and 2 respectively show our problem and our proposed solution through Hybrid Parsing in schematic form.

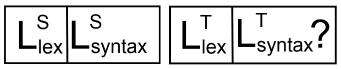


Fig. 1. The Problem

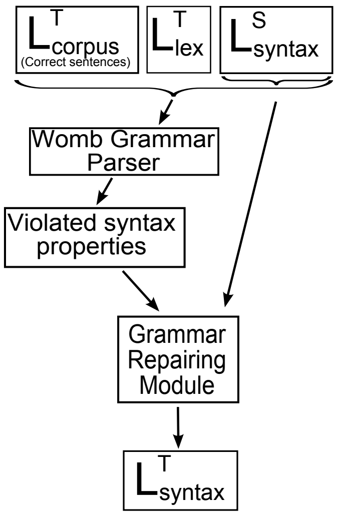


Fig. 2. The Solution

An Example Let $L^S = English$ and $L^T = French$, and let us assume that English adjectives always precede the noun they modify, while in French they always post-cede it (an oversimplification, just for illustration purposes). Thus "the blue book" is correct English, whereas in French we would more readily say "le livre bleu".

If we plug the French lexicon and the English syntax constraints into our Womb Grammar parser, and run a representative corpus of (correct) French noun phrases by the resulting hybrid parser, the said precedence property will be declared unsatisfied when hitting phrases such as "le livre bleu". The grammar repairing module can then look at the entire list of unsatisfied constraints, and produce the missing syntactic component of L^T's parser by modifying the constraints in L^S_{syntax} so that none are violated by the corpus sentences.

Some of the necessary modifications are easy to identify and to perform, e.g. for accepting "le livre bleu" we only need to delete the (English) precedence requirement of adjective over noun (noted $adj < n$). However, subtler modifications may be in order, perhaps requiring some statistical analysis in a second round of parsing: if in our L^T corpus, which we have assumed representative, *all* adjectives appear after the noun they modify, French is sure to include the reverse precedence property as in English: $n < adj$. So in this case, not only do we need to delete $adj < n$, but we also need to add $n < adj$.

3.2 An intriguing approach: Universal Grammar Parsing

Note that if the corpus of L^T's phrases addressed is a representative sample of correct phrases, and the lexicon is complete and also correct, our problem simplifies tremendously. Constituency (i.e. the property that sanctions some constituents, e.g. "determiner", as appropriate for noun phrases) no longer needs to be checked: we simply list as legal constituents all those that appear in the lexicon. Properties (such as precedence, above discussed) that need to be reversed no longer need a statistical analysis per se, nor a second round of parsing or a grammar repair module. In fact *we do not need a specific source language's syntax at all!* All we need to do is a) list all categories in the lexicon as legal constituents, b) postulate a hypothetical L^S (which we shall call L^U, or Universal Language) which lists all possible constraints in L^T with respect to the legal constituents, and c) run the corpus sentences through our parser. The desired grammar for L^T is obtained by deleting from L^U all those constraints that the parser detected as unsatisfied.

For instance, instead of listing the French precedence requirement $adj < n$ to be confirmed or modified by analysis from the corpus, we list the two (contradicting) L^U constraints: $adj < n$ and $n < adj$ (as we do in fact for any pair of legal constituents of the phrase). As soon as a sentence in the corpus violates one of these rules, our parser will delete it. If some phrase in the corpus violates both rules, both will be deleted, corresponding to no precedence requirement between these two symbols.

Of course, the assumption that we have a completely representative sample of correct phrases may be unrealistic, for example in the case of indigenous languages where the native speakers have fluency but little knowledge of grammar concepts. While it is reasonable to rely on the fact that a native speaker will only produce correct phrases, s/he may have forgotten to inform of some particular structure which is nevertheless correct. However, we can exploit the off-line stage of grammar corroboration by the native speaker, which is needed anyway, to complete the corpus. Any constraints arrived at by the parser which are not validated by the native informant indicate some flaw of the corpus and will prompt the linguist to engage in a clarification dialogue with the native speaker in order to complete the corpus until it is truly representative.

Figure 3 shows the schematic form of our solution in terms of Universal Grammar Parsing; here our Womb Grammar parser has been modified to not only detect violated constraints as before, but to delete them from the list of constraints of the source (universal) language, and thus produces the list of syntactic constraints characterizing the (addressed subset of our) target language.

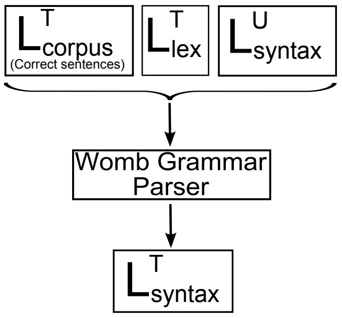


Fig. 3. Universal Grammar Parsing

3.3 Extensions

There could be interesting theoretical ramifications to our work too. More than three decades ago, Noam Chomsky put forward what is widely known as the innate theory of language: that “all children share the same internal constraints which characterize narrowly the grammar they are going to construct”, and that exposure to a specific language determines the actual specialization of these constraints into those of the grammar they will construct [5]. We might say that these internal constraints, if the theory is correct, characterize what may be seen as a (latent, perhaps even impossible to materialize by itself) universal language. Our Womb Grammar formalism might be useful as an experimental aid to uncovering a core of such universal language constraints phrase by phrase, perhaps relative to families of languages, or to help shed some light upon specific areas of linguistic research, e.g. phylogenetic classification.

4 Implementation Details

Our CHRG implementation enters the appropriate Womb Grammar constraints (i.e., either those of a similar language for our Hybrid Parsing as described in 3.1, or our Universal Language constraints described in 3.2) in terms of a constraint g/1, whose argument stores each possible grammar property. For instance, our English grammar hybrid parser includes the constraints: g(obligatority(n)), g(constituency(det)), g(precedence(det,adj)), g(unicity(det)), g(requirement(n,det)), g(dependence(det,n)), and so on. These properties are weeded out upon detection of a violation by CHRG rules that look for them, e.g. an input noun phrase where an adjective precedes a noun will provoke deletion of the constraint g(precedence(n,adj)) when the following CHRG rule applies:

```
!word(C2,_,_), ... , !word(C1,_,_), {g(precedence(C1,C2))} <:>
{update(precedence(C1,C2))}.
```

Each word is stored in a CHRG symbol word/3, along with its category and traits (i.e. word(n,[sing,masc],livre)). Since the CHRG parse predicate stores and abstracts the position of each word in the sentence, this simpagation rule is triggered when a word of category C2 comes before one of category C1, given the existence of the grammar constraint that C1 must precede C2. In CHRG syntax the symbols prefixed with exclamation points are kept, while the ones without are replaced by the body of the rule, in this case an update constraint that invokes some housekeeping procedures. Each of the properties dealt with has similar rules associated with it.

5 Concluding Remarks

With the modesty of AI workers—that is, without claiming linguistic plausibility—we have proposed the appealingly simple and elegant Womb Grammar framework as a constraint-based embodiment both of Chomsky's innate theory of language, and hybrid language parsing for grammar induction. CHR and CHRG have once more proved ideal implementation means, since they allow us to both express and test linguistic constraints in very modular fashion. We have also shown that this framework allows us to induce property-based grammars for simple noun phrases in a very direct fashion. Finally, we have shown that if the corpus of target language sentences is representative, we do not even need the syntax of a related language: simply by using our proposed universal syntax (i.e., all constraints that could in theory apply to the phrases targeted, e.g. noun phrases) we obtain the same good results as with hybrid parsing, while sparing ourselves the need for an explicit module for grammar repair. We postulate that the same methodology can be used for other types of simple phrases and probably for more complex phrases as well.

To the best of our knowledge, this is the first time the idea of grammar induction through weeding out constraints of a hybrid or a universal constraint-based grammar has been proposed. The most closely related work (in that it is also based on property grammars) we have been able to find in the literature addresses grammar development rather than grammar induction, and requires several parsers and the development of different grammar versions over different corpora [2].

It is interesting to note that if the input is not totally representative, the hybrid approach yields a shorter output than the Universal Language one. This is in line with the transfer hypothesis from the field of Second Language Acquisition, whereby knowledge of one language bleeds into another [14].

Admittedly, extending our work into more complex phrases and incorporating other language analysis dimensions such as semantics may not prove as elegantly concise as the restriction to simple phrases has allowed us; however as an aid to linguists at the very least, it may already be a big service to provide them with automatic characterizations of core phrases for languages or dialects that have not yet been studied. Given that the number of living languages in the world is estimated at 6909 (`http://www.ethnologue.com/`), the practical implications of our work stand to be mind-boggling. With this initial work we hope to stimulate further research along these lines.

Acknowledgement. We gratefully acknowledge fruitful discussions with Jacques Farré, Michel Rueher and Enrico Formenti during V. Dahl's visit to Université de Nice which prompted her to address the grammar acquisition problem through Property Grammars first, leading her eventually to the idea of Womb Grammars. We are also grateful for the anonymous referees' helpful comments and for support from NSERC Discovery grant 31611024. The name "Womb Grammars" was inspired by the recent banning by some American Congressmen of the use of certain feminine-referring words on the grounds that they violated "decorum" (`http://www.telegraph.co.uk/news/worldnews/northamerica/usa/9333256/Michigan-bans-women-representatives-for-using-the-V-word.html`). The contrast between such views and the flattering implication in established uses of masculine-referring expressions such as "a seminal contribution", or "a virile people" jumped to Dahl's mind. Just as human wombs have the power to generate all human races, Womb Grammars were designed to potentially generate the grammars for all human languages. May our present work contribute to the free evolution of language usage in ways that honor women and men equally.

References

1. Blache, P.: Property grammars: A fully constraint-based theory. In: Christiansen, H., Skadhauge, P.R., Villadsen, J. (eds.) CSLP. Lecture Notes in Computer Science, vol. 3438, pp. 1–16. Springer (2004)
2. Blache, P., Guenot, M.L., van Rullen, T.: A corpus-based technique for grammar development (2003)
3. Burkett, D., Klein, D.: Two languages are better than one (for syntactic parsing). In: EMNLP. pp. 877–886. ACL (2008)
4. Charniak, E., Johnson, M.: Coarse-to-fine n-best parsing and maxent discriminative reranking. In: Knight, K., Ng, H.T., Oflazer, K. (eds.) ACL. The Association for Computer Linguistics (2005)
5. Chomsky, N.: Deep structure, surface structure, and semantic interpretation. In: Steinberg, D., Jakobovits, L. (eds.) Semantics: An Interdisciplinary Reader in Philosophy, Linguistics and Psychology. Cambridge University Press (1978)
6. Christiansen, H.: CHR grammars. TPLP 5(4-5), 467–501 (2005)
7. Cohen, S.B., Smith, N.A.: Covariance in unsupervised learning of probabilistic grammars. Journal of Machine Learning Research 11, 3017–3051 (2010)
8. Dahl, V., Blache, P.: Directly executable constraint based grammars. Proc. Journees Francophones de Programmation en Logique avec Contraintes (2004)
9. D'Ulizia, A., Ferri, F., Grifoni, P.: A survey of grammatical inference methods for natural language learning. Artif. Intell. Rev. 36(1), 1–27 (2011)
10. Frühwirth, T.W.: Theory and practice of constraint handling rules. J. Log. Program. 37(1-3), 95–138 (1998)
11. Headden, W.P., Johnson, M., McClosky, D.: Improving unsupervised dependency parsing with richer contexts and smoothing. In: HLT-NAACL. pp. 101–109. The Association for Computational Linguistics (2009)
12. Klein, D., Manning, C.D.: Corpus-based induction of syntactic structure: Models of dependency and constituency. In: Scott, D., Daelemans, W., Walker, M.A. (eds.) ACL. pp. 478–485. ACL (2004)

13. Nicolas, L., Molinero, M.A., Sagot, B., Trigo, E.S., de La Clergerie, E., , Farré, J., Vergés, J.M.: Towards efficient production of linguistic resources: the Victoria Project (2009)
14. Odlin, T.: Language Transfer: Cross-Linguistic Influence in Language Learning. The Cambridge Applied Linguistics Series, Cambridge University Press (1989)
15. Pereira, F.C.N., Schabes, Y.: Inside-outside reestimation from partially bracketed corpora. In: Thompson, H.S. (ed.) ACL. pp. 128–135. ACL (1992)
16. Wang, M., Smith, N.A., Mitamura, T.: What is the jeopardy model? a quasi-synchronous grammar for qa. In: EMNLP-CoNLL. pp. 22–32. ACL (2007)

Substitution-based CHR Solver for Bivariate Binomial Equation Sets

Alia el Bolock, Amira Zaki, and Thom Frühwirth

Ulm University, Institute of Software Engineering and Compiler Construction, Germany
`{aliaelbolock}@gmail.com,{amira.zaki,thom.fruehwirth}@uni-ulm.de`

Abstract. Various methods for solving non-linear algebraic systems exist, as this question is amongst the most popular in both the realm of mathematics and computation. As most of these methods use approximations, this work focuses on finding and directly solving a tractable subset. Bivariate binomial systems of non-linear polynomial equations were chosen and solved by simulating the by hand method, using the declarative logic programming language Constraint Handling Rules. Substitution methods and different equation notations are used to extend the solvability of the subset.

Key words: Binomial, Bivariate, Constraint Handling Rules, Non-linear, Polynomial, Substitution

1 Introduction

The world we live in is to the most part non-linear. Thus, it is natural that non-linear systems reoccur everywhere around us. Diverse fields of science and life, such as mechanics, robotics, chemistry and economics require solving non-linear systems for their basic applications. The special case of polynomial systems occurs even more frequently in the real world and has the advantage of being simpler than random non-linear systems and easier to visualize.

Solving non-linear algebraic systems of equations, polynomial and non-polynomial, is a very important subfield of mathematics, as non-linear systems of equations can not be solved quantitatively but to the most part only through approximations. Over the years many different methods for solving non-linear polynomial and non-polynomial systems of equations have been developed. The most common approaches for dealing with non-linear equations are either numerical or symbolic [18, 11, 12], continuation [16], reduction [19, 20] or iterative and interval methods [17, 14, 5, 9, 13], and sometimes even a combination of them, for example in most computer algebra tools [26] and [4]. But only one of these algorithms is based on the logic programming paradigm using the rule based programming language Constraint Handling Rules, namely INCLP($\mathbb{R}$) [6], which is also based on approximating results of a non-linear system.

The aim of this work and the conducted research is finding a tractable subset of non-linear systems of equations, for which exact roots can be efficiently and non-aproximatively calculated and providing a Constraint Handling Rules (CHR) solver for said subset. The subset chosen is that of binomial bivariate equation sets and polynomial equations of degrees up to four. The method derived for solving this subset, is one that simulates one of the possible approaches of humans when met with such problem sets. The implemented solver artificially reanimates what humans would do by hand, always baring in mind the most efficient approach given the problem set and the advantages provided by CHR. It basically uses isolation and substitution methods for solving the bivariate system of non-linear equations.

The majority of the various pre-existing methods for solving non-linear polynomial equation sets, especially those in the context of constraint programming, are based on interval and approximation methods. This work focuses on trying to find the largest subset that can be solved *exactly* and thus having the highest precision possible.

The problem field is narrowed down to cover non-linear polynomial equations. Starting from bivariate systems, alongside univariate equations with degrees less than five, accuracy and solvability could be ensured. As proof of concept, the solver algorithm was tested for the binomial case.

2 Concepts

2.1 Algebra

Properties of Equations Univariate equations are ones with one variable, while bivariate equations have two variables. We distinguish between univariate and multivariate polynomials, meaning polynomials with only one variable and multiple variables respectively.

Non-linear System of Equations and its Roots A non-linear system of equations is a set of n equations, containing at least one non-linear equation; meaning an equation with degree not equal to one. Finding the roots of the system of equations, means finding a vector $x = (x_1, ..., x_n)$ that simultaneously solves all equations within the systems [25]. Non-linear systems of equations can either have a finite number of solutions, infinite solutions (consistent) or no solutions at all (inconsistent). Under-defined systems of equations are ones with more variables than equations, while an over-defined system has more equations than variables.

Polynomials A polynomial is a finite sum of terms with non-negative degrees. A polynomial consisting of one term is called monomial and that of two terms is called binomial. The standard form of polynomial equations is $P(x) = c_n x^n +$

$c_{n-1}x^{n-1} + ... + c_0$, where $c_i \in \mathbb{Q}$, $n \in \mathbb{Z}$, $c_n \neq 0$ and $n \neq 0$. According to the fundamental theorem of algebra, any polynomial equation can be expressed as $P(x) = c_n(x-r_1)(x-r_2)...(x-r_n)$, where $r_i \in \mathbb{C}$ are the roots of the polynomial. Such roots can be directly determined for univariate polynomial equations with degrees up to four, for which solution formulae exist [21, 15]. It was however proven that no such formulae could exist for polynomial equation with higher degrees [22–24].

2.2 Constraint Handling Rules

Constraint Handling Rules (CHR) is a high-level, constraint-based, declarative logic programming language, invented by Prof. Thom Frühwirth in 1991. CHR adapts the basic concepts of mathematical logic representation and is thus highly and easily applicable to various problems. CHR is a committed-choice, single-assignment language, with multi-headed rules and conditional rule application through guards. Having simplification, propagation and simpagation (a mixture of the afore mentioned rules) as the only operators that can deal with constraints, CHR is well suited for representing mathematical problems and solving them straightforwardly. The properties of CHR enable the user to design anytime, online, confluent and concurrent algorithms, depending on the semantics used. More detailed explanations of CHR, its properties and advanced examples, can be found in [7].

3 Solution Algorithm

The chosen methodology for solving an input system of equations, is based on the usual thought procedure most humans would follow. The implemented solver gives a numeric solution for a non-linear input system. Given a set of two equations, the solver basically first tries to turn one of them into a univariate equation, by isolating one of the system variables. Then, this univariate equation is solved and its solutions renders the second equation univariate, in which case it can in turn be solved. This can be achieved for equations in one of the two solvable cases: an already univariate equation, or an equation with a singly occurring variable that either stands alone or is part of a term. The subset of univariate equations with solution formulae can be directly solved, the remainder is simplified. Should this not be directly applicable, then some substitutions are to be done to transform the equation set into one that can be solved by the above mentioned method. Figure 1 gives the flow diagram of this solving algorithm.

The equations' terms are ordered before the check for equations in the directly solvable cases is made, to be sure that the leading term is always the simplest term, when needing to isolate it or to take it as a reference point for subsitutions. The order of a term depends on the powers of its variables and the second term variable is taken as the reference point to give the priority in isolation to the variable x, thus colexicographic ordering is used. The colexicographic ordering

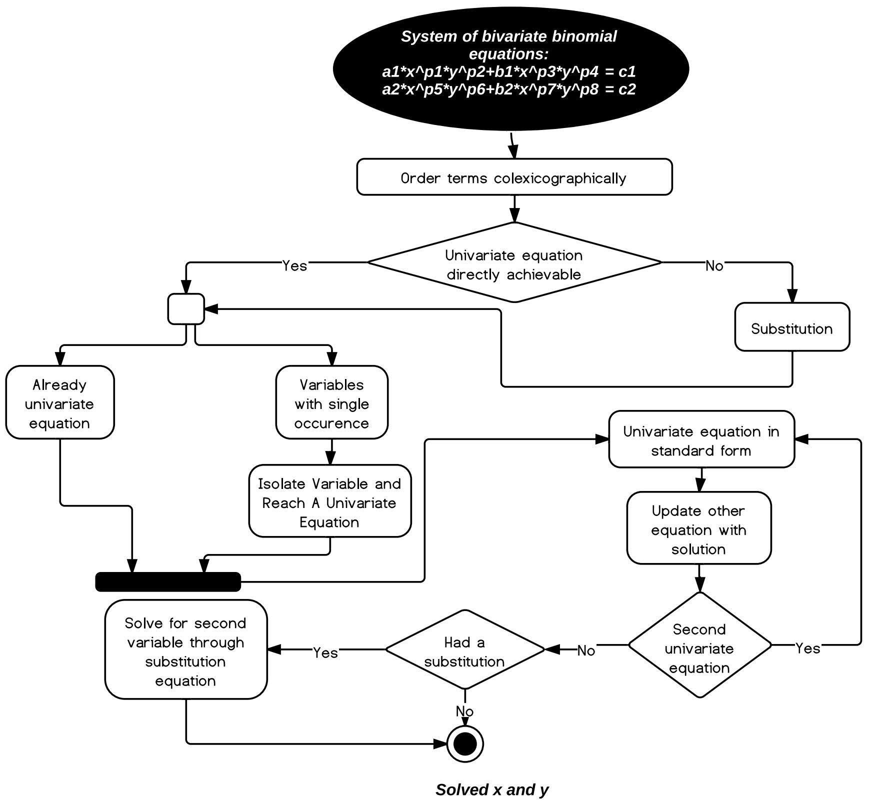

System of bivariate binomial equations:
a1*x^p1*y^p2+b1*x^p3*y^p4 = c1
a2*x^p5*y^p6+b2*x^p7*y^p8 = c2
Order terms colexicographically
Univariate equation directly achievable
Yes
No
Substitution
Already univariate equation
Variables with single occurence
Isolate Variable and Reach A Univariate Equation
Univariate equation in standard form
Update other equation with solution
Second univariate equation
Yes
No
Had a substitution
Yes
No
Solve for second variable through substitution equation
Solved x and y

Fig. 1. Flow diagram of the solver algorithm

of two pairs $x^{n1}y^{m1}$ and $x^{n2}y^{m2}$ is defined as follows:

$$colex : x^{n1}y^{m1} \leq x^{n2}y^{m2} \Leftrightarrow m1 < m2 \vee (m1 = m2 \wedge n1 \leq n2). \qquad (1)$$

```
order_eq_exchange @ order_eq([H1,H2] eq C) <=> lex(H2,H1)
                                              | [H2,H1] eq C.
```

The helper predicate `lex(H1,H2)` is true if `H1` is colexicographically less than `H2`. If univariate equations are found they are transformed into the standard form, else the most optimal system variable is isolated in one equation and used to render the other equation univariate. If the set is not in a directly solvable state then the in 3.2 explained substitution is applied before the set is sent back to the initial solving state. All univariate equations in standard form with degrees less than five are solved and their solution produces a second univariate equation to be solved, possibly with the help of the substitution equation. An example for the realization of part of the quartic formula [15] is:

```
A*X^4+B*X^4-E ueq 0 ==> C=0, D=0, F is C-(3*B**2/8), G is (D+(B**3/8))-(B*C/2),
                        H is (E-(3*B**4/256))+((B**2*C/16)-(B*D/4)),
                        I is F/2, J is (F**2-4*H)/16, K is (-G**2)/64,
                        1*Z^3+I*Z^2+J*Z^1+K ueq 0.

A*X^3+B*X^2+C*X^1+D ueq 0 <=> F is (((3*C)/(A))-(B**2/A**2))/3,
                 G is ((2*B**3/A**3)-(9*B*C/A**2)+((27*D)/(A)))/27,
                 H is (G**2/4)+(F**3/27), H > 0
               | R is (-G/2)+(H**(1/2)), cubic_root(R,S),
                 T is (-G/2)-(H**(1/2)), cubic_root(T,U),
                 X1 is (-((S+U)/2)-(B/3*A)),
                 XI1 is ((S-U)*(3**(1/2))/2),
                 XI2 is (-((S-U)*(3**(1/2))/2)),
                 solved_img(X1,XI1,X1,XI2).

solved_img(Y1,YI1,Y1,YI2),A*X^4+B*X^3-E ueq 0 <=>
                     C=0, D=0, G is (D+(B**3/8))-(B*C/2),
                     img_sqrt(Y1,YI1,PR,PI), S is (B)/(4*A),
                     R is (-G)/(8*(PR**2+PI**2)), X1 is (PR+PR)+(R-S),
                     X2 is (R-S)-(PR+PR), (X=X1;X=X2), solved(X).
```

This shows how the quartic equation is reduced to a cubic one whose solutions are then substituted back into the orginal quartic equation to solve it. In case the helper cubic equation has two imaginary and one real solution, the imaginary solutions are chosen and their real and imaginary parts are seperatly forwarded to the quartic equation using the `solved_img/4` constraint. The auxiliary `img_sqrt/2` predicate, extracts the cubic root of a negative number, as this option is not supported by Prolog.

The algorithm is characterized by its simplicity while covering a wide subset. This is enabled by the different equation representations used and the substitution scheme.

3.1 Different Equation Representations

The simplicity and efficiency of the algorithm is ensured by using different representations and notations for equations, to distinguish between types of equations and phases of the algorithm. Equations are expressed as constraints to benefit from the features of CHR. As the equations are notated differently depending on the state they are in, the rules are fired voluntarily and no explicit iterations need to be done. This ensures that any possible solutions are calculated and possible simplifications are done at any given point, exploiting the online property of CHR. There are two notations for equations in this solver, and the different equality constraints belong to different notations.

The standard equation notation `A*X^P1*Y^P2+B*X^P3*Y^P4 eq C` is where the equation most resembles the normal mathematical form of equations. The '=' sign is replaced by other constraints e.g. the `eq` constraint, depending on the phase of execution of the algorithm. The `ueq/2` constraint for example indicates a univariate equation, while the `req/2` constraint means an equation with an isolated variable and the `deq/2` indicates a not yet handled equation and that the other equation has been already simplified or solved.

```
X req W \ L eq C <=>
            L = [C1*pot(X,P1)*pot(Y,P2), C2*pot(X,P3)*pot(Y,P4)]
           | C1*W^P1*Y^P2+C2*W^P3*Y^P4-C ueq 0.

solved(X) \ C1*X^P1*Y^P2+C2*X^P3*Y^P4 deq C <=>
                                 L1 is C1*X**P1, L2 is C2*X**P3,
                                 L1*Y^P2+L2*Y^P4-C ueq 0.
```

While the standard notation is easier for users to understand, it does not give full access to the components of the equation, which is why the pot/4 (potency) constraints were introduced. Each pot(X,E,T,P) constraint comprises a variable X, its power P and the equation and the term it originates from- E and T respectively. A term consists of a constant and two or three pot constraints, and an equation is represented as the following list of constraints
[A*pot(X,E,1,P1)*pot(Y,E,1,P2),B*pot(X,E,2,P3)*pot(Y,E,2,P4)] eq C, mathematically equivalent to $A{*}X^{P1}{*}Y^{P2}{+}B{*}X^{P3}{*}Y^{P4} = C$. At the beginning of the solver's run, the input equations are transformed into the pot notation and the pot constraints are added to the constraint store. pot constraints give a global insight wether there are univariate equations or singly occuring variables by cross-referencing powers of variables and the term and equation they are in, as each variable is directly accessible through its pot constraint. For example having two pot constraints A*pot(X,1,1,1) and pot(Y,1,1,0) means that Y occurs at most once in equation one, as the first term does not contain it. To decide wether it stands alone or is embedded in a term, the second term needs to be checked. The pot constraints are also used for realizing the substitutions in 3.2. For readability, the term and equation identifiers will be removed for the code samples demonstrated here.

```
[C1*pot(X,0)*pot(Y,P1),C2*pot(X,0)*pot(Y,P2)] eq C,
 pot(X,0),pot(X,0) <=> C1*Y^P1+C2*Y^P2-C ueq 0.

[C1*pot(X,0)*pot(Y,P2), C2*pot(X,P3)*pot(Y,P4)] eq C <=>
                              X req ((C-C1*Y^P2)/(C2*Y^P4))^(1/P3).
```

3.2 Substitution Scheme

The solvable subset is extended by introducing a substitution scheme that simplifies the initial problem set to one of the solvable states. If there is no variable x to solve for, without turning the function $f_i : A_i * T_j + B_i * T_j = C_i$, $i \in \{1, 2\}$, $j \in \{1, 2, 3, 4\}$ where $T_j = x^{P1} * y^{P2}$, into a more complicated one, then some substitutions are applied to the equation terms until one of the solution cases is applicable.

The substitution function $s : F^2 \times F^2 \to G^3 \times G^3$ assigns each bivariate term to an equivalent trivariate one, by introducing a substitution variable a. Substituting all terms within the initial system of equations $(f_1(x, y) = 0, f_2(x, y) = 0)$ results in a trivariate equation set $(g_1(x, y, a) = 0, g_2(x, y, a) = 0)$, as given by the function s. The function s is represented by the sub/2 constraint, where the first attribute is the original bivariate term and the second the trivariate substitution term. The input equation set is transformed into the almost identical output set g_i with $T_j = a^{P1} * x^{P2} * y^{P3}$. The chosen substitution scheme describes system terms in terms of each other while adding only one additional variable to form the base substitution, to be eliminated later. This is efficient, as it excludes the possibility of producing numerous new variables, that cause the system of equations to be strongly under-defined. For the creation of the substitution keys, namely the sub constraints, there are four different substitution cases, which should be checked in the order given below:

1. Direct substitution is applicable if there are two identical terms, except for their respective coefficients: $\exists i : T_j = T_i | i \neq j,\ i, j \in \{1, 2, 3, 4\}$.
```
sub(pot(X,P1)*pot(Y,P2),pot(Var,1)) \ pot(X,P1), pot(Y,P2) <=>
                  P1\=0, P2\=0 | pot(Var,1), pot(X,0), pot(Y,0),
                                 sub(pot(X,P1)*pot(Y,P2), pot(Var,1)).
```
2. Multiples substitution is utilized if there exists a term that is the multiple of another: $\exists n \in \mathbb{Z} : T_i^n = T_j, i \neq j$.
```
pot(X,P1),pot(Y,P2),pot(X,P3),pot(Y,P4) <=>
                     P1\=0,P2\=0,P3\=0,P4\=0,divides(P3,P1,Q1),
                     divides(P4,P2,Q2),Q1 == Q2
                   | pot(Var,1),pot(X,0),pot(Y,0),pot(Var,Q1),pot(X,0),
                     pot(Y,0),sub(pot(Y,P2)*pot(X,P1),pot(Var,1)),
                     sub(pot(Y,P4)*pot(X,P3),pot(Var,Q1)).
```
3. Product substitution can be applied, if one term can be expressed as the product of two other terms: $\exists i : \exists j : T_k = T_j * T_i \wedge i \neq j$.
```
sub(pot(X,P3)*pot(Y,P4),pot(Var1,N1)*pot(Y,N2)*pot(X,N3)),
 sub(pot(X,P5)*pot(Y,P6),pot(Var2,N4)*pot(Y,N5)*pot(X,N6)) \
  pot(X,P1),pot(Y,P2) <=> Q1 is P3+P5,P1==Q1,Q2 is P4+P6,
  P2==Q2,F1 is N1+N4,F2 is N2+N5,F3 is N3+N6
| sub(pot(X,P1)*pot(Y,P2),pot(Var1,F1)*pot(Y,F2)*pot(X,F3)),
  pot(Var1,F1), pot(Y,F2),pot(X,F3).
```
4. If none of the above cases apply, the only unsubstituted terms remaining will be those that could be expressed as the multiplication of a term with one of the system variables: $\exists i : T_j = T_i * v, v \in \{x, y\}$.
```
sub(pot(X,P1)*pot(Y,P4), pot(Var,N)) \
 pot(X,P1),pot(Y,P2) <=> P1\=0, P2\=0,P2 > P4, Diff is P2-P4
              | pot(Var,E1,T1,N),pot(X,E1,T1,0),pot(Y,E1,T1,Diff),
                sub(pot(Y,P2)*pot(X,P1),pot(Var,N)*pot(Y,Diff)).
```

Even though the equation set is three dimensional, the trivariate equation set will then match one of the solvable cases, as each equation on its own is still bivariate, which is ensured by the chosen substitution scheme. This matching has to happen as one whole term must have been taken as a reference point by s and fully substituted by the substitution variable a. After the sub constraints are created, all equations are iterated over and the actual substitution is done. After one of the variables has been solved, the equation $A = X^{P1} * Y^{P2}$ from the base substitution `sub(pot(X,P1)*pot(Y,P2), pot(A,1)*pot(X,0)*pot(Y,0))` is added, to enable the solving of the remaining variables.

3.3 Evaluation

This solver computes solutions to univariate polynomial equations of degrees up to four and binomial bivariate equation sets without approximations and relying only on substitution and direct methods. While various solvers, especially computer algebra systems, for the chosen subset exist, no tools were found that could solve binomial bivariate equations without adding approximative methods, especially none using CHR.

Most of said computer algebra systems, in particular Mathematica [26, 27] and Maple [28, 29], solve the whole set covered by the implemented CHR solver and a vast amount of other mathematical problems, relying primarily on symbolic evalutions of equations and approximative, iterative methods like the Newton method for calculating numeric results. Although the renowned computer algebra tools cover a much wider solution set, the CHR solver is capable of giving the numerical solutions of some systems of equations directly through substitutions, while Mathematica or Maple for example would have yielded to approximative methods instead. In case no solutions can be calculated, both the CHR solver and computer algebra tools, symbolically simplify the equation set the farthest possible.

The majority of the existing solvers for univariate polynomial equations and computer algebra results, display all complex results, whereas the implemented solver only gives the real results of a system of equations, as Prolog is currently only defined in the domain of real numbers.

The method used in this solver is straightforward and thus does not have a high complexity. Depending on the input set, one or at most two full iterations are done and thus the results are achieved almost directly by firing the correct rules. CHR enables the direct translation of the human-based solution method into a program which facilitates the solving of the whole subset, which is not a commonly used method in other non-linear equation solving tools The addition of the substitution system renders otherwise non-solvable systems of equations solvable without adding much complexity, thus extending the solvable subset.

4 Conclusion and Future Work

4.1 Conclusion

Deriving new ways to solve non-linear algebraic systems of equations or improving existing ones is the concern of many fields in mathematical computing. Most of these solution systems are based on approximation methods. This work aimed at finding an exactly solvable, tractable subset of non-linear equation sets, deriving a method to solve said subset and realizing this method using CHR. After investigating different pre-existing solving mechanisms and the non-linear subsets they solve, it was decided to constrict the solution field to bivariate polynomial systems of equations. A substitution-based method for solving bivariate equation sets was derived. The algorithm reduces one of the system equations into a solvable univariate one and then uses the solution to reduce the second one. As proof of concept, the method was modeled for binomial equation sets, as any extensions to multinomial sets, would follow analogously. The implemented solver extends the solvability of the chosen subset through substitutions, without resorting to approximative methods. The solver was implemented using K.U.Leuven's CHR implementation with Prolog as the host language. The roots for any consistent system of equations are obtained, given that the resulting univariate equations are standardizable and quantifiable through finite formulae. Otherwise the highest possible simplification is attained.

4.2 Future Work

There is a large scope of extensions for this solver, depending on the needed functionalities, as this solver was intended to prove a concept based on a subset from which multidirectional expansions are possible. The solver could be extended to solve all consistent univariate systems of polynomial equations. Furthermore, the scope of this work could be broadened to cover multinomial bivariate and overdefined non-linear systems. Finally, the same concept of the solver could be a basis for solving more complex types of non-linear equation sets, such as trivariate polynomial equation sets or bivariate non-linear non-polynomial equation sets, e.g. trigonometric functions for which transformative substitutions exist.

References

1. Online function plotter. http://rechneronline.de/function-graphs/,June 2012.
2. Pascal Triangle. http://matheuropa.lfs-koeln.de/pascal/binformel.htm, June 2012.
3. Online diagram drawer. http://www.lucidchart.com, June 2012.
4. J. Xue, Y. Li, Y. Feng, and Z. Liu, An intelligent hybrid algorithm for solving nonlinear polynomial systems., in International Conference on Computational Science (M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds.), vol. 3037 of Lecture Notes in Computer Science, pp. 2633, Springer, 2004.
5. P. Van Hentenryck, D. McAllester, and D. Kapur, Solving polynomial systems using a branch and prune approach, SIAM Journal of Numerical Analisys, vol. 34, pp. 797827, April 1997.
6. L. D. Koninck, T. Schrijvers, and B. Demoen, Inclp(r) - interval-based nonlinear constraint logic programming over the reals, in Workshop on Logic Programming, pp. 91100, 2006.
7. T. Frühwirth, Constraint handling rules. Cambridge University Press, 2009.
8. K. I. Joy, Bernstein polynomials, in On-Line Geometric Modeling Notes, 1996.
9. F. Benhamou, D. A. McAllester, and P. V. Hentenryck, Clp(intervals) revisited, in SLP, pp. 124138, 1994.
10. H. Collavizza, F. Delobel, and M. Rueher, A note on partial consistencies over continuous domains, in In Proc. CP98 (Fourth International Conference on Principles and Practice of Constraint Programming, pp. 147161, Springer Verlag, 1998.
11. L. Granvilliers, A symbolic-numerical branch and prune algorithm for solving nonlinear polynomial systems, Journal of Universal Computer Science, vol. 4, pp. 125 146, February 1998.

12. D. G. Sotiropoulos and T. N. Grapsa, Optimal centers in branch-and-prune algorithms for univariate global optimization, Applied Mathematics and Computation, vol. 169, no. 1, pp. 247277, 2005.
13. V. Stahl, Interval Methods for Bounding the Range of Polynomials and Solving Systems of Nonlinear Equations. PhD thesis, Forschungsinstitut fur Symbolisches Rech- nen, Technisch-Naturwissenschaftliche Fakultat, Johannes Kepler Universtat Linz, 1995.
14. S. Herbort and D. Ratz, Improving the efficiency of a nonlinear-system-solver using a componentwise newton method, in Berichte aus dem Forschungsschwerpunkt Computerarithmetik, Intervallrechnung und Numerische Algorithmen mit Ergebnisverifikation, 1997.
15. Online quartic calculator. http://www.1728.org/quartic2.htm, June 2012.
16. J. Verschelde, Homotopy Continuation Methods For Solving Polynomial Systems. PhD thesis, K.U.Leuven, Belgium, 1996.
17. X.-W. CHANG, Solving a nonlinear equation. Lecture Slides, COMP 350 Numerical Computing, McGill School of Computer Science, 2011.
18. P. D. D. Nesselmann, Grobner-basen und nicht-lineare gleichungssysteme, in Manuskript zur Vorlesung, pp. 8798, Universitat Rostock, 2009.
19. E. C. Sherbrooke and N. M. Patrikalakis, Computation of the solutions of nonlinear polynomial systems Computer Aided Geometry Design, vol. 10, pp. 379–405, October 1993.
20. B. Mourrain and J.-P. Pavone, Subdivision methods for solving polynomial equations, Journal of Symbolic Computation, vol. 44, no. 3, pp. 292–306, 2009.
21. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, Numerical Recipes 3rd Edition: The Art of Scientific Computing. New York, NY, USA: Cambridge University Press, 2007.
22. M. I. Rosen, Niels Hendrik Abel and Equations of Fifth Degree, American Mathematical Monthly, Vol. 102, 1995, pp. 495 505.
23. R. J. Drociuk, On the Complete Solution to the Most General Fifth Degree Polynomial, 3-May 2000, http://arxiv.org/abs/ math.GM/0005026/.
24. Abel, U., H., Beweis der Unmoglichkeit algebraische Gleichugen von hoheren Graden als dem vierten allgemein aufzulosen, J, fur die reine und angew. Math., Bd. 1 (1826) 65-84.
25. C. Grosan and A. Abraham, A new approach for solving nonlinear equations systems, IEEE Transactions on Systems, Man and Cybernetics Part A, 2007.
26. Equation Solving Using Mathematica., http://reference.wolfram.com/mathematica/guide/EquationSolving.html, Wolfram Research, Inc., August 2012.
27. Wolfram Mathematica Tutorial Collection: Mathematics and Algorithms. Wolfram Research,2008.
28. Maple-The Essential Math Software Tool, http://www.maplesoft.com/products/Maple/, Maplesoft, August 2012.
29. Holistic Numerical Methods- Transforming Numerical Methods Education for the STEM Undergraduate, http://numericalmethods.eng.usf.edu/mtl/gen/03nle/index.html, Autar Kaw, University of South Florida, August 2012.

Coinductive Proofs over Streams as CHR Confluence Proofs*

Rémy Haemmerlé

Technical University of Madrid

Abstract. Coinduction is an important theoretical tool for defining and reasoning about unbounded data structures (such as streams, infinite trees, rational numbers ...), and infinite-behavior systems. Confluence is a fundamental property of Constraint Handling Rules (CHR) since, as in other rewriting formalisms, it guarantees that the computations are not dependent on rule application order, and also because it implies the logical consistency of the program's declarative view. In this paper, we illustrate how the confluence of CHR can be used to prove universal coinductive properties. In particular we give several examples of bisimulation proofs over streams.

1 Introduction

Induction and coinduction are contrasting terms for ways of describing and reasoning about a system. Whereas, the classical notion of inductive reasoning begins with some primitive properties (or definitions) and uses constructive operations on these to iteratively infer a whole set of conclusions, coinductive reasoning [4, 5, 20] starts from a set of conceivable properties (or definitions) and iteratively dismisses those that break the self-consistency of the whole set. Despite the fact that coinduction is less known than induction, it has started to receive attention in recent years in computer science. For instance, coinduction has been employed to define process equivalences in Concurrency Theory [17, 20], to study lazy evaluations in functional languages [9], or to deal with infinite data-structures and infinite computations in Logic Programming [21, 16].

Constraint Handling Rules (CHR) is a committed-choice constraint logic programming language, introduced by solvers. It has matured into a general-purpose concurrent programming language. Operationally, a CHR program consists of a set of guarded rules that rewrite multisets of constrained atoms. Declaratively, a CHR program can be viewed as a set of logical implications executed on a deduction principle.

* The research leading to these results has received funding from the Programme for Attracting Talent / young PHD of the MONTEGANCEDO Campus of International Excellence (PICD), the Madrid Regional Government under the CM project P2009/TIC/1465 (PROMETIDOS), and the Spanish Ministry of Science under the MEC project TIN-2008-05624 (DOVES).

Confluence is a basic property of rewriting systems. It refers to the fact that any two finite computations starting from a common state can be prolonged so as to eventually meet in a common state again. Confluence is an important property for any rule-based language, because it is desirable for computations to not be dependent on a particular rule application order. In the particular case of CHR, this property is even more desirable, as it guarantees the correctness of a program [3, 13]: any confluent program has a consistent logical reading.

In this paper, we illustrate how the proof of confluence of non-terminating CHR programs can be used to establish coinductive properties. In practice, we propose a simple encoding of streams[1] and bisimulation[2] as non-terminating CHR programs. Then, we explain how the confluence of the resulting programs provides an effective coinductive proof. Finally, we show how the confluence of these programs can be inferred by using a criterion we recently introduced [11]. The preliminary results presented in this paper are embedded in the more general goal of the research we started in [10]: Understanding relationships between coinduction and CHR.

The remainder of this paper is structured as follows: Sect. 2 gently introduces the notion of coinduction and stream coalgebra. In Sect. 3, we recall some preliminaries on CHR. Sect. 4 presents a criteria we recently introduced for proving confluence of non-terminating CHR programs. In Sect. 5, we present how to encode stream coalgebras in CHR. Then we show how using CHR confluence to infer bisimulation over such coalgebras, before concluding in Sect. 6.

2 Streams and Coinduction

In this section, we introduce the notion of stream, the canonical example of coinductive object. By using few bases from the theory of universal coalgebra, we explain how to define streams by coinduction. We conclude by showing how to prove equality of streams by coinduction.

This introduction is freely inspired from the one by Rutten [19]. It is deliberately kept short. The reader may refer to Rutten's works to get a more general picture of streams, coalgebras, and coinduction.

2.1 Streams

Let $\mathcal{A}$ be an arbitrary set. We define a *stream* over $\mathcal{A}$ as a function from natural numbers (the *positions*) to $\mathcal{A}$ (the *values*). Hence $\mathcal{A}^\omega$ is the set respecting the equation:

$$\mathcal{A}^\omega = (\mathbb{N} \to \mathcal{A})$$

For convenience, we may denote such a stream s by the informal notation:

$$s = [s(0), s(1), s(2), \dots]$$

[1] The canonical example of coinductive data-structure.
[2] The canonical example of coinductive property.

$$\begin{tikzcd}
\mathrm{zero} \arrow[loop right, "0"] & \mathrm{one} \arrow[loop right, "1"] & \mathrm{blink}_0 \arrow[r, bend left, "0"] & \mathrm{blink}_1 \arrow[l, bend left, "1"]
\end{tikzcd}$$

Fig. 1. LTS view of the coalgebra $(\mathcal{X}_3, \mathrm{h}_3, \mathrm{t}_3)$.

Example 1. The stream containing only 0's (i.e. $[0,0,0,\dots]$) is the constant function $(x \mapsto 0)$. Similarly the stream containing only 1's (i.e. $[1,1,1,\dots]$) is the function $(x \mapsto 1)$. The streams alternating 0 and 1 (i.e. $[0,1,0,\dots]$ and $[1,0,1,\dots]$) are the respective functions $(x \mapsto (x \mod 2))$ and $(x \mapsto (x+1 \mod 2))$.

Following analogy between finite lists and streams, we call *head* of a stream s the first value of s, i.e. $\mathrm{h}(s) = s(0)$ and call the *tail* of s the stream obtained by removing the head, i.e. $\mathrm{t}(s) = (x \mapsto s(x+1))$.

Example 2. Consider the streams given in the previous example. The head and the tail of these streams respect the following equations:

$$\begin{array}{ll}
\mathrm{h}([0,0,0,\dots]) = 0 & \mathrm{t}([0,0,0,\dots]) = [0,0,0,\dots] \\
\mathrm{h}([1,1,1,\dots]) = 1 & \mathrm{t}([1,1,1,\dots]) = [1,1,1,\dots] \\
\mathrm{h}([0,1,0,\dots]) = 0 & \mathrm{t}([0,1,0,\dots]) = [1,0,1,\dots] \\
\mathrm{h}([1,0,1,\dots]) = 1 & \mathrm{t}([1,0,1\dots]) = [0,1,0,\dots]
\end{array}$$

2.2 Coalgebras

A *(stream) coalgebra* is a triple $(\mathcal{X}, \mathrm{h}_{\mathcal{X}}, \mathrm{t}_{\mathcal{X}})$ consisting of a set $\mathcal{X}$ of *states* together with an *output function*: $\mathrm{h}_{\mathcal{X}} : \mathcal{X} \to \mathcal{A}$ and a *transition function* $\mathrm{t}_{\mathcal{X}} : \mathcal{X} \to \mathcal{X}$. In the following we may refer to these two functions as the *destructors*.

Example 3. The triple $(\mathcal{X}_3, \mathrm{h}_3, \mathrm{t}_3)$ where $\mathcal{X}_3 = \{\mathrm{one}, \mathrm{zero}, \mathrm{blink}_0, \mathrm{blink}_1\}$ and $\mathrm{h}_3 : \mathcal{X}_3 \to \mathcal{A}$ and $\mathrm{t}_3 : \mathcal{X}_3 \to \mathcal{X}_3$ satisfying the equations below is a coalgebra.

$$\begin{array}{rr}
\mathrm{h}_3(\mathrm{zero}) = 0 & \mathrm{t}_3(\mathrm{zero}) = \mathrm{zero} \\
\mathrm{h}_3(\mathrm{one}) = 1 & \mathrm{t}_3(\mathrm{one}) = \mathrm{one} \\
\mathrm{h}_3(\mathrm{blink}_0) = 0 & \mathrm{t}_3(\mathrm{blink}_0) = \mathrm{blink}_1 \\
\mathrm{h}_3(\mathrm{blink}_1) = 1 & \mathrm{t}_3(\mathrm{blink}_1) = \mathrm{blink}_0
\end{array}$$

Alternatively, a coalgebra can be viewed as a (possibly) infinite automaton or a Labelled Transition System (LTS) $(\mathcal{X}, \mathcal{A}, \to)$ verifying:

$$s \xrightarrow{h} t \qquad \text{if and only if} \qquad h = \mathrm{h}_{\mathcal{X}}(s) \;\&\; t = \mathrm{t}_{\mathcal{X}}(s)$$

Example 4. Figure 1 represents the coalgebra given in Example 3.

$$\begin{tikzcd}
\text{zero} \arrow[loop right, "0"] \arrow[d, dotted, "{[\![\cdot]\!]_{\mathcal{X}_3}}"] & \text{one} \arrow[loop right, "1"] \arrow[d, dotted, "{[\![\cdot]\!]_{\mathcal{X}_3}}"] & \text{blink}_0 \arrow[r, bend left, "0"] \arrow[d, dotted, "{[\![\cdot]\!]_{\mathcal{X}_3}}"] & \text{blink}_1 \arrow[l, bend left, "1"] \arrow[d, dotted, "{[\![\cdot]\!]_{\mathcal{X}_3}}"] \\
{[0,0,0,\dots]} \arrow[loop right, "0"] & {[1,1,1,\dots]} \arrow[loop right, "1"] & {[0,1,0,\dots]} \arrow[r, bend left, "0"] & {[1,0,1,\dots]} \arrow[l, bend left, "1"]
\end{tikzcd}$$

Fig. 2. LTS view of the coalgebra $(\mathcal{X}_3, \mathrm{h}_3, \mathrm{t}_3)$.

Intuitively, within a coalgebra $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ a state $x \in \mathcal{X}$ "represents" a unique stream $s \in \mathcal{A}^\omega$, while the output and transition functions associate to x the head and (a state $x' \in \mathcal{X}$ representing) the tail of s. In order to formalize this intuition, we introduce now the notion of homomorphism and finality.

A *homomorphism* between two coalgebras $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ and $(\mathcal{Y}, \mathrm{h}_\mathcal{Y}, \mathrm{t}_\mathcal{Y})$ is a function $\phi : \mathcal{X} \to \mathcal{Y}$ that respects the destructors, i.e.:

$$\mathrm{h}_\mathcal{Y}(\phi(x)) = \mathrm{h}_\mathcal{X}(x) \qquad \text{and} \qquad \mathrm{t}_\mathcal{Y}(\phi(x)) = \phi(\mathrm{t}_\mathcal{X}(x))$$

The set of streams $\mathcal{A}^\omega$ can be viewed as the coalgebra $(\mathcal{A}^\omega, \mathrm{h}, \mathrm{t})$. This coalgebra has the following property:

Theorem 1 (Finality). *The coalgebra* $(\mathcal{A}^\omega, \mathrm{h}, \mathrm{t})$ *is* final *among the set of all coalgebras. That is, for any coalgebra* $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ *there exists a unique homomorphism* ϕ *from* $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ *to* $(\mathcal{A}^\omega, \mathrm{h}, \mathrm{t})$*.*

The finality of the set of streams gives us the formal basis to define what represents a state. Let $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ be a coalgebra. We say that a state $x \in \mathcal{X}$ represents the stream $[\![x]\!]_\mathcal{X} = \phi(x)$ where ϕ is a homomorphism from $(\mathcal{X}, \mathrm{h}_\mathcal{X}, \mathrm{t}_\mathcal{X})$ to $(\mathcal{A}^\omega, \mathrm{h}, \mathrm{t})$. The finality of $(\mathcal{A}^\omega, \mathrm{h}, \mathrm{t})$ ensures us that this definition is meaningful, i.e. each state represents one and only one stream.

Example 5. Consider the coalgebra $(\mathcal{X}_3, \mathrm{h}_3, \mathrm{t}_3)$ given in Example 3. Let $\phi : \mathcal{X}_3 \to \mathcal{A}^\omega$ be the function satisfying the equations:

$$\begin{aligned} \phi(\text{zero}) &= [0,0,0,\dots] \qquad & \phi(\text{blink}_0) &= [0,1,0,\dots] \\ \phi(\text{one}) &= [1,1,1,\dots] \qquad & \phi(\text{blink}_1) &= [1,0,1,\dots] \end{aligned}$$

One can easily verify that ϕ is a homomorphism. Hence the states zero, one, blink_0, and blink_1 respectively represent the streams $[0,0,0,\dots]$, $[1,1,1,\dots]$, $[0,1,0,\dots]$, and $[1,0,1,\dots]$. Figure 2 represents graphically this correspondence.

2.3 Proof by Coinduction

In order to prove that two streams s and s' are equal, it is necessary and sufficient to prove that

$$\text{for all } n \in \mathbb{N} \quad s(n) = s'(n)$$

An obvious method for establishing equality between streams s and s' consists of an induction on the natural number n (i.e. prove $s(0) = s'(0)$ and show that $s(n) = s'(n)$ implies $s(n+1) = s'(n+1)$). In this section, we present an alternative way, based on coinduction. This method is often more natural, especially for those streams defined using coalgebras.

A *bisimulation (relation)* between two coalgebras $(\mathcal{X}, \mathrm{h}_{\mathcal{X}}, \mathrm{t}_{\mathcal{X}})$ and $(\mathcal{Y}, \mathrm{h}_{\mathcal{Y}}, \mathrm{t}_{\mathcal{Y}})$ is a relation $\mathcal{S} \subseteq \mathcal{X} \times \mathcal{Y}$ such that for all streams $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ if whenever $x \,\mathcal{S}\, y$ holds then both $\mathrm{h}_{\mathcal{X}}(x) = \mathrm{h}_{\mathcal{Y}}(y)$ and $\mathrm{t}_{\mathcal{X}}(x) \,\mathcal{S}\, \mathrm{t}_{\mathcal{Y}}(y)$. If there exists a bisimulation $\mathcal{S}$ with $x \,\mathcal{S}\, y$, we will write $x \sim y$, and say that x and y are *bisimilar*. Hence two states are bisimilar, if they have the same head and bisimilar tails. The Coinduction theory guarantees that two bisimilar states represent the same stream.

Theorem 2 (Coinduction). *Let $(\mathcal{X}, \mathrm{h}_{\mathcal{X}}, \mathrm{t}_{\mathcal{X}})$ and $(\mathcal{Y}, \mathrm{h}_{\mathcal{Y}}, \mathrm{t}_{\mathcal{Y}})$ be two coalgebras. For all states $x \in \mathcal{X}$ and $y \in \mathcal{Y}$, if $x \sim y$ then $[\![x]\!]_{\mathcal{X}} = [\![y]\!]_{\mathcal{Y}}$.*

This theorem gives us a proof principle: to prove that two streams represented by two states are equal, it is sufficient to exhibit a bisimulation that relates the states.

Example 6. Consider the coalgebra $(\mathcal{X}_3, \mathrm{h}_3, \mathrm{t}_3)$ given in Example 3 together with the coalgebra $(\mathcal{Z}_6, \mathrm{h}_6, \mathrm{t}_6)$ with $\mathcal{Z}_6 = \{z, z'\}$ and satisfying the equations:

$$\mathrm{h}_6(z) = 0 \qquad \mathrm{t}_6(z) = z' \qquad \mathrm{h}_6(z') = 0 \qquad \mathrm{t}_6(z') = z$$

One may want to prove that zero and z represent the same stream. For this purpose assume the relation $\mathcal{S} = \{(\text{zero}, z), (\text{zero}, z')\}$. It is straightforward to demonstrate $\mathcal{S}$ is a bisimulation, i.e. $[\![\text{zero}]\!] = [\![z]\!]$.

We finish the section about coalgebra and coinduction by some observations that may help readers non familiar with the concepts presented in this section.

The coinduction proof principle can be seen as a systematic way of strengthening the statement one is trying to prove. In the previous example instead of proving the identity $[\![\text{zero}]\!]_{\mathcal{X}_3} = [\![z]\!]_{\mathcal{Z}_6}$, we extended the relation $\{(\text{zero}, z)\}$ to a more general relation $\mathcal{S} = \{(\text{zero}, z), (\text{zero}, z')\}$ with zero $\mathcal{S}\, z$.

It is also worth noting that a coalgebra can contain an arbitrary number of representatives for a given stream. For instance $[\![\text{blink}_0]\!]_{\mathcal{X}_3}$ is represented by no state within $(\mathcal{Z}_6, \mathrm{h}_6, \mathrm{t}_6)$. Conversely, $[\![\text{zero}]\!]_{\mathcal{X}_3}$ is represented twice in $(\mathcal{Z}_6, \mathrm{h}_6, \mathrm{t}_6)$. Indeed by Theorem 2, we have $[\![z]\!]_{\mathcal{Z}_6} = [\![\text{zero}]\!]_{\mathcal{X}_3} = [\![z']\!]_{\mathcal{Z}_6}$.

3 Constraint Handling Rules

In this section, we recall briefly the syntax and the semantics of CHR. Frühwirth's book [7] can be referred to for a more general overview of the language.

3.1 Syntax

The formalization of CHR assumes a language of *(built-in) constraints* containing equality over some theory $\mathcal{C}$, and defines *(user-defined) atoms* using a different set of predicate symbols. In the following, $\mathcal{R}$ will denote an arbitrary set of identifiers. By a slight abuse of notation, we allow confusion of conjunctions and multiset unions, omit braces around multisets, and use the comma for multiset union. We use $\mathrm{fv}(\phi)$ to denote the set of free variables of a formula ϕ. The notation $\exists_{\text{-}\psi}\phi$ denotes the existential closure of ϕ with the exception of free variables of ψ.

A *(CHR) program* is a finite set of eponymous rules of the form:

$$(r \mathbin{@} \mathbb{K}\backslash\mathbb{H} \Longleftrightarrow \mathbb{G} \mid \mathbb{B}; \mathbb{C})$$

where $\mathbb{K}$ (the *kept head*), $\mathbb{H}$ (the *removed head*), and $\mathbb{B}$ (the *user body*) are multisets of atoms, $\mathbb{G}$ (the *guard*) and $\mathbb{C}$ (the *built-in body*) are conjunctions of constraints and, $r \in \mathcal{R}$ (the *rule name*) is an identifier assumed unique in the program. Rules in which both heads are empty are prohibited. An empty guard $\top$ (resp. an empty kept head) can be omitted with the symbol $\mid$ (resp. with the symbol $\backslash$). The *local variables* of rule are the variables occurring in the guard and in the body but not in the head that is $\mathrm{lv}(r) = \mathrm{fv}(\mathbb{G}, \mathbb{B}, \mathbb{C}) \setminus \mathrm{fv}(\mathbb{K}, \mathbb{H})$. Rules are divided into two classes: *simplification rules*[3] if the removed head is non-empty and *propagation rules* otherwise. Propagation rules can be written using the alternative syntax:

$$(r \mathbin{@} \mathbb{K} \Longrightarrow \mathbb{G} \mid \mathbb{B}; \mathbb{C})$$

3.2 Operational semantics

In this section, we recall the equivalence-based operational semantics ω_e of Raiser et al. [18]. It is equivalent to the *very abstract* semantics ω_{va} of Frühwirth [6], which is the most general operational semantics of CHR. We prefer the former because it includes a rigorous notion of equivalence, which is an essential component of confluence analysis.

A *(CHR) state* is a tuple $\langle \mathbb{E}; \mathbb{C}; \bar{x} \rangle$, where $\mathbb{E}$ (the *user store*) is a multiset of atoms, $\mathbb{C}$ (the *built-in store*) is a conjunction of constraints, and $\bar{x}$ (the *global variables*) is a finite set of variables. Unsurprisingly, the *local variables* of a state are those variables of the state which are not global. When no confusion can occur, we will syntactically merge user and built-in stores. We may futhermore omit the global variables component when states have no local variables. In the following, we use Σ to denote the set of states. Following Raiser et al., we will always implicitly consider states modulo a structural equivalence. Formally, this *state equivalence* is the least equivalence relation $\equiv$ over states satisfying the following rules:

[3] Unlike standard presentations, our definition does not distinguish simplification rules form the so-called simpagation rules.

- $\langle \mathbb{E}; \mathbb{C}; \bar{x}\rangle \equiv \langle \mathbb{E}; \mathbb{D}; \bar{x}\rangle$ if $\mathcal{C} \vDash \exists_{\text{-}(\mathbb{E},\bar{x})}\mathbb{C} \leftrightarrow \exists_{\text{-}(\mathbb{E},\bar{x})}\mathbb{D}$
- $\langle \mathbb{E}; \bot; \bar{x}\rangle \equiv \langle \mathbb{F}; \bot; \bar{y}\rangle$
- $\langle \mathbb{E}, c; \mathbb{C}, c{=}d; \bar{x}\rangle \equiv \langle \mathbb{E}, d; \mathbb{C}, c{=}d; \bar{x}\rangle$
- $\langle \mathbb{E}; \mathbb{C}; \bar{x}\rangle \equiv \langle \mathbb{E}; \mathbb{C}; \{y\} \cup \bar{x}\rangle$ if $y \notin \mathrm{fv}(\mathbb{E}, \mathbb{C})$.

Once states are considered modulo equivalence, the operational semantics of CHR can be expressed by a single rule. Formally the operational semantics of a program $\mathcal{P}$ is given by the binary relation $\xrightarrow{\mathcal{P}} \mathcal{P}$ on states satisfying the rule:

$$\frac{(r \ @\ \mathbb{K}\backslash\mathbb{H} \iff \mathbb{G}|\mathbb{B}; \mathbb{C}) \in \mathcal{P}\rho \quad \mathrm{lv}(r) \cap \mathrm{fv}(\mathbb{E}, \mathbb{D}, \bar{x}) = \emptyset}{\langle \mathbb{K}, \mathbb{H}, \mathbb{E}; \mathbb{G}, \mathbb{D}; \bar{x}\rangle \xrightarrow{\mathcal{P}} \langle \mathbb{K}, \mathbb{B}, \mathbb{E}; \mathbb{G}, \mathbb{C}, \mathbb{D}; \bar{x}\rangle}$$

where ρ is a renaming. If a program $\mathcal{P}$ contains a sole rule r, we may write $\xrightarrow{r}$ for $\xrightarrow{\{r\}}$. For any transition $\xrightarrow{P}$, the symbol $\xleftarrow{P}$ will denote its converse, $\xrightarrow{\mathcal{P}}{}^{\equiv}$ its reflexive closure, and $\overset{\mathcal{P}}{\twoheadrightarrow}$ its transitive-reflexive closure. We will use $\xrightarrow{\mathcal{P}} \cdot \xrightarrow{\mathcal{Q}}$ to denote the left-composition of all binary relations $\xrightarrow{\mathcal{P}}$ and $\xrightarrow{\mathcal{Q}}$.

We will say a program $\mathcal{P}$ is *terminating* if there is no infinite sequence of the form $e_0 \xrightarrow{\mathcal{P}} e_1 \xrightarrow{\mathcal{P}} e_2 \ldots$ Furthermore, we will say that $\mathcal{P}$ is *confluent* if for all states S, S_1, and S_2 satisfying $S \overset{\mathcal{P}}{\twoheadrightarrow} S_1$ and $S \overset{\mathcal{P}}{\twoheadrightarrow} S_2$, there exists a state S' such that $S_1 \overset{\mathcal{P}}{\twoheadrightarrow} S'$ and $S_2 \overset{\mathcal{P}}{\twoheadrightarrow} S'$.

3.3 Declarative semantics

Owing to its origins in the tradition of CLP, the CHR language features declarative semantics through direct interprestation in first-order logic. Formally, the *logical reading* of a rule of the form:

$$\mathbb{K}\backslash\mathbb{H} \iff \mathbb{G} \mid \mathbb{B}; \mathbb{C}$$

is the guarded equivalence:

$$\forall\big((\mathbb{K} \wedge \mathbb{G}) \rightarrow \big(\mathbb{H} \leftrightarrow \exists_{\text{-}(\mathbb{K},\mathbb{H})}(\mathbb{G} \wedge \mathbb{C} \wedge \mathbb{B})\big)\big)$$

The *logical reading* of a program $\mathcal{P}$ within a theory $\mathcal{C}$ is the conjunction of the logical readings of its rules with the constraint theory $\mathcal{C}$. It is denoted by $\mathcal{CP}$.

Operational semantics is sound and complete with respect to this declarative semantics [6, 3, 10]. Furthermore, we recently established that any confluent program $\mathcal{P}$ is correct and has a logical model expressible by a CLP program, called the CLP projection. The *(CLP) projection* of a CHR program $\mathcal{P}$ is a set $\pi(\mathcal{P})$ of CLP clauses defined as:

$$\pi(\mathcal{P}) = \{(a \leftarrow \mathbb{G}, \mathbb{C}, \mathbb{K}, \mathbb{B}) \mid (\mathbb{K}\backslash\mathbb{H} \iff \mathbb{G} \mid \mathbb{B}, \mathbb{C}) \in \mathcal{P} \text{ and } a \in \mathbb{H}\}$$

Theorem 3 ([13]). *Let $\mathcal{S}$ be an arbitrary model of the constraint theory $\mathcal{C}$. A confluent CHR program and its projection have the same least $\mathcal{S}$-model.*

It is worth noting that in the current state of knowledge Theorem 3 only holds when programs are considered with respect to the most general operational semantics for CHR, namely the very abstract semantics. In particular, the proofs of the theorem do not appear to be adaptable to more concrete semantics such as for instance Abdennadher's token-based semantics [1].

4 Diagrammatic Confluence for CHR

This section sums up some recent results about conlfuence of non-terminating CHR programs. More details can be found in [11].

4.1 Critical Peaks

In term rewriting systems, the basic techniques used to prove confluence consist of showing various confluence criteria on a finite set of special cases, called *critical pairs*. Critical pairs are generated by a superposition algorithm, in which one attempts to capture the most general way the left-hand sides of the two rules of the system may overlap. The notion of critical pairs has been successfully adapted to CHR by Abdennadher et al. [2]. Here, we introduce a slight extension of the notion.

Let us assume that r_1 and r_2 are CHR rules (form possible disctinct programs) renamed apart:

$$(r_1 \ @ \ \mathbb{K}_1 \backslash \mathbb{H}_1 \iff \mathbb{G}_1 \mid \mathbb{B}_1; \mathbb{C}_1) \in \mathcal{P}_1 \qquad (r_2 \ @ \ \mathbb{K}_2 \backslash \mathbb{H}_2 \iff \mathbb{G}_2 \mid \mathbb{B}_2; \mathbb{C}_2) \in \mathcal{P}_2$$

A *critical ancestor (state)* S_c for the rules r_1 and r_2 is a state of the form:

$$S_c = \langle \mathbb{H}_1^{\Delta}, \mathbb{H}_1^{\cap}, \mathbb{H}_2^{\Delta}; \mathbb{D}; \bar{x} \rangle$$

satisfying the following properties:

- $(\mathbb{K}_1, \mathbb{H}_1) = (\mathbb{H}_1^{\Delta}, \mathbb{H}_1^{\cap})$, $(\mathbb{K}_2, \mathbb{H}_2) = (\mathbb{H}_2^{\Delta}, \mathbb{H}_2^{\cap})$, $\mathbb{H}_1^{\cap} \neq \emptyset$, and $\mathbb{H}_2^{\cap} \neq \emptyset$;
- $\bar{x}_1 = \mathrm{fv}(\mathbb{K}_1, \mathbb{H}_1)$, $\bar{x}_2 = \mathrm{fv}(\mathbb{K}_2, \mathbb{H}_2)$ and $\bar{x} = \bar{x}_1 \cup \bar{x}_2$;
- $\mathbb{D} = (\mathbb{H}_1^{\cap} = \mathbb{H}_2^{\cap}, \mathbb{G}_1, \mathbb{G}_2)$ and $\exists \mathbb{D}$ is $\mathcal{C}$-satisfiable;
- $\mathbb{H}_1^{\cap} \not\subseteq \mathbb{K}_1$ or $\mathbb{H}_2^{\cap} \not\subseteq \mathbb{K}_2$.

Then the following tuple is called a *critical peak* between r_1 and r_2 at S_c:

$$\langle \mathbb{K}_1, \mathbb{B}_1, \mathbb{H}_2^{\Delta}; \mathbb{D}, \mathbb{C}_1; \bar{x} \rangle \xleftarrow{r_1} \cdot \xrightarrow{r_2} \langle \mathbb{K}_2, \mathbb{B}_2, \mathbb{H}_1^{\Delta}; \mathbb{D}, \mathbb{C}_2; \bar{x} \rangle$$

4.2 Rule-decreasingness

In this section, we present the so-called *rule-decreasingness criterion*. This criterion derived from the decreasing diagrams technique [22] is a novel criterion on CHR critical pairs that generalizes both local confluence [3] and strong confluence [14] criteria.

Rule-decreasingness criterion assumes the set $\mathcal{R}$ of rule identifiers is defined as a disjoint union $\mathcal{R}_i \uplus \mathcal{R}_c$. For a given program $\mathcal{P}$, we denote by $\mathcal{P}^i$ (resp. $\mathcal{P}^c$) the set of rules form $\mathcal{P}$ built with $\mathcal{R}_i$ (resp. $\mathcal{R}_c$). We call $\mathcal{P}^i$ the *inductive* part of $\mathcal{P}$, because we will subsequently assume that $\mathcal{P}^i$ is terminating, while $\mathcal{P}^c$ will be called *coinductive*, as it will be typically non-terminating. A critical peak is *inductive* if it involves only inductive rules (i.e. a critical peak of $\mathcal{P}^i$), or

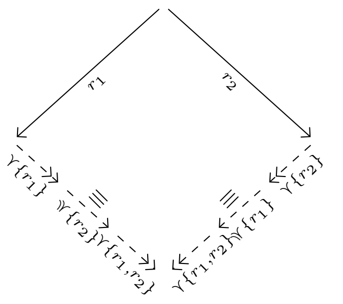


Fig. 3. Local decreasingness

coinductive if it involves at least one coinductive rule (i.e. a critical peak between $\mathcal{P}^c$ and $\mathcal{P}$).

In the rest of this paper, we will say that a preorder $\succcurlyeq$ is *wellfounded*, if the strict preorder $\succ$ associated with $\succcurlyeq$ (i.e. $\alpha \succ \beta$ iff $\alpha \succcurlyeq \beta$ but not $\beta \succcurlyeq \alpha$) is a terminating relation. A preorder $\succcurlyeq$ on rule identifiers is *admissible*, if any inductive rule identifier is strictly smaller than any coinductive one (i.e. for any $r_i \in \mathcal{R}_i$ and any $r_c \in \mathcal{R}_c$, $r_c \succ r_i$ holds).

A critical peak $S_1 \xleftarrow{r_1} \cdot \xrightarrow{r_2} S_2$ is decreasing with respect to a preorder $\succcurlyeq$ if the following property holds:

$$S_1 \xrightarrow{\curlyvee\{r_1\}}\!\!\!\!\twoheadrightarrow \cdot \xrightarrow{\underline{\curlyvee}\{r_2\}}{}^{\equiv} \cdot \xrightarrow{\curlyvee\{r_1,r_2\}}\!\!\!\!\twoheadrightarrow \cdot \twoheadleftarrow\!\!\!\!\xleftarrow{\curlyvee\{r_1,r_2\}} \cdot \xleftarrow{\underline{\curlyvee}\{r_1\}}{}^{\equiv} \cdot \twoheadleftarrow\!\!\!\!\xleftarrow{\curlyvee\{r_2\}} S_2 \qquad (\star)$$

where for any set K of rule identifiers, $\underline{\curlyvee} K$ stands for $\{r \in \mathcal{R} \mid \exists r' \in K.r' \succcurlyeq r\}$ and $\curlyvee K$ for $\{r \in \mathcal{R} \mid \exists r' \in K.r' \succ r\}$. Property $(\star)$ is graphically represented in Figure 3. A program $\mathcal{P}$ is *rule-decreasing* with respect to an admissible preorder $\succcurlyeq$ if:

– the inductive part of $\mathcal{P}$ is terminating,
– any inductive critical peak of is joinable by $\xrightarrow{\mathcal{P}^i}\!\!\!\!\twoheadrightarrow \cdot \twoheadleftarrow\!\!\!\!\xleftarrow{\mathcal{P}^i}$, and
– any coinductive critical peaks is decreasing with respect to $\succcurlyeq$.

A program is *rule-decreasing* if it is rule-decreasing with respect to some admissible preorder.

Theorem 4 ([11]). *Rule-decreasing programs are confluent.*

To illustrate the use of the theorem, we recall now one example form [11].

Example 7 (Partial order constraint). Let $\mathcal{P}_7$ be the classic CHR introductory example, namely the constraint solver for partial order. This consists of the following four rules, which define the meaning of the *user-defined* symbol $\leq$

$$\begin{tikzcd}
 & \langle x \leq x, x \leq y \rangle \arrow[dl, "reflex."'] \arrow[dr, "trans."] & \\
\langle x \leq y \rangle & & \langle x \leq x, x \leq y, x \leq y \rangle \arrow[dl, dotted, "reflex."] \\
 & \langle x \leq y, x \leq y \rangle \arrow[ul, dotted, "dupl."] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \langle x \leq y, y \leq z, z \leq y \rangle \arrow[dl, "anti."'] \arrow[dr, "trans."] & \\
\langle x \leq y, y = z \rangle & & \langle x \leq y, y \leq z, z \leq y, x \leq z \rangle \arrow[dl, dotted, "anti."] \\
 & \langle x \leq y, x \leq z, y = z \rangle \arrow[ul, dotted, "dupl."] &
\end{tikzcd}$$

Fig. 4. Some rule-decreasing critical peaks for $\mathcal{P}_7$

using the built-in equality constraint $=$:

$$\begin{array}{lll}
\textit{duplicate} & @ & x \leq y \setminus x \leq y \iff \top \\
\textit{reflexivity} & @ & x \leq x \iff \top \\
\textit{antisymmetry} & @ & x \leq y, y \leq x \iff x = y \\
\textit{transitivity} & @ & x \leq y, y \leq z \implies x \leq z
\end{array}$$

Since $\mathcal{P}_7$ is trivially non-terminating (indeed, it uses propagation rules) one cannot apply local confluence criterion [3]. Nonetheless, confluence of $\mathcal{P}_7$ can be deduced using the full generality of Theorem 4. For this purpose, assume that all rules except *transitivity* are inductive and take any admissible preorder. Clearly the inductive part of $\mathcal{P}_7$ is terminating. Indeed the application of any one of the three first rules strictly reduces the number of atoms in a state. Then by a tedious but simple analysis, we prove that critical peaks of $\mathcal{P}_7$ can be joined while respecting the hypothesis of rule-decreasingness. In fact all critical peaks can be joined without using the *transitivity* rule. Some rule-decreasing diagrams involving the *transitivity* rule are given as examples in Figure 4.

5 Proving Stream Bisimulation Using CHR Confluence

In this section, we illustrate the power of CHR confluence and the rule-decreasingness criterion to prove coinductive properties.

5.1 Coalgebra in CHR

Following preliminary ideas for encoding coalgebras into CHR with the standard Herbrand constraint system [10], we use first-order terms as states, and define the destructors by means of a single user-defined atom $\mathrm{d}(s, h, t)$. Here the $\mathrm{d}(s, h, t)$ predicate must be understood as the function that returns for a given state s its head $h = \mathrm{h}(s)$ and its tail $t = \mathrm{t}(s)$. To enforce functionality of the destructor, we start our program $\mathcal{P}$ with the following simplification rule:

$$\mathrm{fun_d} \ @ \ \mathrm{d}(s, h_2, t_2) \backslash \mathrm{d}(s, h_1, t_1) \iff h_1 = h_2, t_1 = t_2$$

Now we use the terms zero, one, blink_0, and blink_1 as states for the respective streams containing only 0's, only 1's, alternations of 0 and 1, and alternations of 1 and 0. Then we add to our program the following rules that specify the behaviour of the destructor on these states:

$$\begin{array}{llll}
\mathrm{d_{zero}} & @ & \mathrm{d}(\mathrm{zero}, h, t) & \iff h = 0, t = \mathrm{zero}. \\
\mathrm{d_{one}} & @ & \mathrm{d}(\mathrm{one}, h, t) & \iff h = 1, t = \mathrm{one}. \\
\mathrm{d_{blink_0}} & @ & \mathrm{d}(\mathrm{blink}_0, h, t) & \iff h = 0, t = \mathrm{blink}_1. \\
\mathrm{d_{blink_1}} & @ & \mathrm{d}(\mathrm{blink}_1, h, t) & \iff h = 1, t = \mathrm{blink}_0.
\end{array}$$

We can go even further and define operators on streams. For this purpose, we use fresh function symbols as new operators and use recursive simplification rules to encode the behaviour of the destructors on these operators. For instance, we will encode the functions odd() and even(), which return the stream formed by the elements in the odd and even positions, respectively, and the function zip() which interlaces the elements from the two given streams. Hence, we add to $\mathcal{P}$ the rules:

$$\begin{array}{llll}
\mathrm{d}_{even} & @ & \mathrm{d}(\mathrm{even}(x), h, t) & \iff \mathrm{d}(x, \_, t_1), \mathrm{d}(t_1, h, t_2), t = \mathrm{even}(t_2). \\
\mathrm{d}_{odd} & @ & \mathrm{d}(\mathrm{odd}(x), h, t) & \iff \mathrm{d}(x, h, t_1), t = \mathrm{even}(t_1). \\
\mathrm{d_{zip}} & @ & \mathrm{d}(\mathrm{zip}(x, y), h, t) & \iff \mathrm{d}(x, h, t_1), t = \mathrm{zip}(y, t_1).
\end{array}$$

It is not difficult to convince oneself that $\mathcal{P}$ is terminating. For this reason, we fix all the rules we have defined so far as inductive. Note that what is inductive here is solely the definition of the destructors. The encoded coalgebra still has a coinductive nature, in the sense that the destructor can be indefinitely called, in order to get all the elements composing a stream.

5.2 Proving simple Coinductive properties in CHR

The definition of bisimilation can translated into CHR by a single coinductive rule added to our program:

$$\sim\ @\ s_1 \sim s_2 \iff \mathrm{d}(s_1, h, t_1), \mathrm{d}(s_2, h, t_2), t_1 \sim t_2$$

From a logical point of view, the declarative reading of this rule ensures that two states s_2 and s_1 are bisimilar if and only if there exists a model for the program $\mathcal{P}$ we have built so far, that contains the atom $s_1 \sim s_2$. Thanks to Theorem 3, we know that this is the case precisely if the program augmented with a "query" rule of the form $s_1 \sim s_2 \iff \top$ is confluent. For instance, in order to prove that the stream zip(zero, one) is equal to the stream blink_0, we can show that $\mathcal{P}$ together with the following rule is confluent:

$$\mathrm{q}_1\ @\ \mathrm{zip}(\mathrm{zero}, \mathrm{one}) \sim \mathrm{blink}_0 \iff \top$$

Unfortunately this is not the case. Indeed, rules $\sim$ and q_1 yield a non-joinable peak:

$$\langle \mathrm{zip}(\mathrm{one}, \mathrm{zero}) \sim \mathrm{blink}_1 \rangle \overset{\mathcal{P}}{\twoheadleftarrow} \cdot \overset{\mathcal{P}}{\leftarrow}_{\sim} \langle \mathrm{zip}(\mathrm{zero}, \mathrm{one}) \sim \mathrm{blink}_0 \rangle \overset{\mathcal{P}}{\rightarrow}_{q_1} \langle \top \rangle$$

$$\begin{tikzcd}
\langle \mathrm{z}(\mathrm{z},\mathrm{o}) \sim \mathrm{b}_0 \rangle \arrow[r, "\sim"'] \arrow[d, "\mathrm{q}_1"] & {} \arrow[r, dotted, two heads, "\mathcal{P}_i"] & \langle \mathrm{z}(\mathrm{o},\mathrm{z}) \sim \mathrm{b}_1 \rangle \arrow[dll, dotted, "\kappa_1"] \\
\langle \top \rangle & &
\end{tikzcd}
\qquad
\begin{tikzcd}
\langle \mathrm{z}(\mathrm{o},\mathrm{z}) \sim \mathrm{b}_1 \rangle \arrow[r, "\sim"'] \arrow[d, "\kappa_1"] & {} \arrow[r, dotted, two heads, "\mathcal{P}_i"] & \langle \mathrm{z}(\mathrm{z},\mathrm{o}) \sim \mathrm{b}_0 \rangle \arrow[dll, dotted, "\mathrm{q}_1"] \\
\langle \top \rangle & &
\end{tikzcd}$$

Fig. 5. Diagrammatic proofs for $\mathrm{z}(\mathrm{z},\mathrm{o}) \sim \mathrm{b}_0$

One idea for circumventing this problem is to "complete" the program, i.e. to add rules to it, in order to make it confluent. Indeed, if an interpretation is a model for the completed program, it is obviously a model for the original program. In the case of our example, we can just add a rule closing the previous peak:

$$\kappa_1 \text{ @ } \mathrm{zip}(\mathrm{one}, \mathrm{zero}) \sim \mathrm{blink}_1 \Longleftrightarrow \top$$

This time, the resulting program is confluent. We can use rule-decreasing criteria to prove this. For the proof passing through, we have to assume the rules q_1 and κ_1 are coinductive and strictly greater than the rule $\sim$. The proofs of the decreasingness of all the critical peaks involving at least one coinductive rule are graphically represented in Figure 5. For the sake of conciseness, symbols have been shortened to their initials in the figure.

5.3 Proving universal coinductive properties in CHR

In the previous section, we have proved a coinductive property for a particular stream. Here we are concerned with proving similar properties for arbitrary streams. The idea is to take benefit of the implicit universal quantification of rule-head variables to prove coinductive properties true with respect to arbitrary streams. However, we first have to formally define in our framework what a stream is.

As with bisimulation, being a stream is a coinductive property. It can be translated into CHR by the coinductive rule:

$$\mathbf{str} \text{ @ } \mathbf{str}(x) \Longleftrightarrow \mathrm{d}(x, \_, t), \mathbf{str}(t).$$

Basically, x is a stream if it can be deconstructed by d into a head and into a tail which is itself a stream. We now add to our program the following coinductive rules to specify which terms are actual streams:

$$\begin{array}{lll}
\mathbf{str}_{\mathrm{zero}} \text{ @ } \mathbf{str}(\mathrm{zero}) & \Longleftrightarrow & \top \\
\mathbf{str}_{\mathrm{even}} \text{ @ } \mathbf{str}(\mathrm{even}(x)) & \Longleftrightarrow & \mathbf{str}(x). \\
\mathbf{str}_{\mathrm{zip}} \text{ @ } \mathbf{str}(\mathrm{zip}(x,y)) & \Longleftrightarrow & \mathbf{str}(x), \mathbf{str}(y)
\end{array}$$

These rules state respectively that zero is a stream, even(x) is a stream if and only if x is a stream, and zip(y,z) is a stream if and only if both y and z are streams. From a typing point of view, rule **str** can be viewed as the definition of a (coinductive) type, and the rules $\mathbf{str}_x$ as type declarations. We do not

$$\begin{tikzcd}
\langle \mathrm{z}(x_0, y_0) \sim z_0 \rangle \arrow[r, "\sim"'] \arrow[d, "\kappa_4"] & \left\langle \begin{array}{c} \mathbf{d}(x_0, h_0, x_1), \mathbf{d}(z_0, h_0, z_1), \\ \mathrm{z}(y_0, x_1) \sim z_1 \end{array} \right\rangle \arrow[r, dotted, two heads, "\kappa_4"'] & \left\langle \begin{array}{c} \mathbf{d}(x_0, h_0, x_1), \mathbf{d}(z, h_0, z_1), \\ \mathrm{o}(z_1) \sim y_0, \mathrm{e}(z_1) \sim x_1 \end{array} \right\rangle \arrow[d, dotted, "\sim"] \\
\langle \mathrm{o}(z_0) \sim x_0, \mathrm{e}(z_0) \sim y_0 \rangle \arrow[rr, dotted, two heads, "\sim"'] & & \left\langle \begin{array}{c} \mathbf{d}(x_0, h_0, x_1), \mathbf{d}(z, h_0, z_1), \mathrm{e}(z_1) \sim x_1, \\ \mathbf{d}(y_0, h_1, y_1), \mathbf{d}(z_1, h_1, z_2), \mathrm{e}(z_2) \sim y_1 \end{array} \right\rangle
\end{tikzcd}$$

$$\begin{tikzcd}
\langle \mathrm{z}(\mathrm{o}(x_0), \mathrm{e}(x_0))) \sim x_0 \rangle \arrow[r, "\mathrm{q}_2"'] \arrow[d, "\sim"] & \langle \mathbf{s}(x_0)^4 \rangle \arrow[r, dotted, two heads, "\mathbf{s}"'] & \langle \mathbf{d}(x_0, \_, x_1), \mathbf{s}(x_1)^2, \mathbf{d}(x_1, \_, x_2), \mathbf{s}(x_2)^2 \rangle \\
\langle \mathbf{d}(x_0, \_, x_1), \mathrm{z}(\mathrm{e}(x_0), \mathrm{e}(x_1)) \sim x_1 \rangle \arrow[d, dotted, "\kappa_4"] & & \\
\langle \mathbf{d}(x_0, \_, x_1), \mathrm{o}(x_1) \sim \mathrm{e}(x_0), \mathrm{e}(x_1) \sim \mathrm{e}(x_1) \rangle \arrow[rr, dotted, "\sim"'] & & \left\langle \begin{array}{c} \mathbf{d}(x_0, \_, x_1), \mathbf{d}(x_1, \_, x_2), \\ \mathrm{e}(x_2) \sim \mathrm{e}(x_2), \mathrm{e}(x_1) \sim \mathrm{e}(x_1) \end{array} \right\rangle \arrow[uu, dotted, two heads, "\kappa_2"]
\end{tikzcd}$$

Fig. 6. Diagrammatic proofs for $\mathrm{zip}(\mathrm{odd}(x), \mathrm{even}(x)) \sim x$

explicitly present similar declarations for the streams one, blink_0, and blink_1, or for the operator odd. As we have done previously with bisimulation, we verify the coherency of these declarations by checking the confluence of the whole program. (We assume rules $\mathbf{str}_x$ are strictly greater than rule $\mathbf{str}$.)

Once streams have been properly defined and declared in our framework, we can try to prove that any stream x is bisimilar to $(\mathrm{zip}(\mathrm{odd}(x), \mathrm{even}(x)))$. For this purpose, we add the following coinductive rules to the current program:

$$\begin{array}{lcll}
q_2 & @ & \mathrm{zip}(\mathrm{odd}(x), \mathrm{even}(x)) \sim x & \Longleftrightarrow \mathbf{str}(x)^4 \\
\kappa_2 & @ & x \sim x & \Longleftrightarrow \mathbf{str}(x)^2 \\
\kappa_3 & @ & \mathrm{even}(\mathrm{zip}(x, y)) \sim y & \Longleftrightarrow \mathbf{str}(x), \mathbf{str}(y) \\
\kappa_4 & @ & \mathrm{zip}(x, y) \sim z & \Longleftrightarrow \mathrm{odd}(z) \sim x, \mathrm{even}(z) \sim y
\end{array}$$

The first rule corresponds to the property we want to prove, while the other rules are there to complete the program, which would otherwise be non-confluent.

Once again, we are able to establish the confluence of the resulting program by using the rule-decreasingness criterion (assuming $q_2 \succ \kappa_4 \succ \kappa_3 \succ \kappa_2 \succ \mathbf{str}_x$). Note that, in order to simplify the proof, some atoms are duplicated. The exponents in the rules q_2 and κ_2 indicate how many times the atom $\mathbf{str}(x)$ is repeated. In practice, these repetitions are helpful because of the multiset nature of the user store. They are effective in closing a critical peak between κ_2 and κ_4 on the one hand, and a critical peak between q_2 and $\sim$ on the other hand. From a theoretical point of view, the repetition of atoms is not problematic, since an atom has a declarative meaning equivalent to the declarative meaning of several copies of it. Proofs of decreasingness for some relevant critical peaks of the program are graphically represented in Figure 6. Within the figure, states are implicitly normalized using the inductive part of the program.

When the program is proved confluent, the CLP projection provides us a logical model in the form of a CLP program. (In the practical case of our example, the projection is obtained by replacing the symbol $\iff$ by $\leftarrow$ in all rules expcet for $\mathrm{fun}_{\mathrm{d}}$ which can be safely ignored.) This finite representation is convenient since the model is in fact infinite: the domain of discourse of the model contains any stream obtained by composition of zero, one, blink_0, blink_1, even(), odd(), and zip().

From a model-theory point of view, we are not aware of any technique that is able to directly prove satisfiable formulas such as the ones presented in the last part of this section. In particular, the classical techniques, such as SMT solvers, inference-based theorem provers, or the analytic tableaux all seem inadequate for dealing with non-valid formulas mixing universal and existential quantifications, and which also have only infinite models. From a purely coinductive point of view, there do exist frameworks, such as circular coinductive rewriting [8], which are able to fully automatically prove bisimulations similar to the ones presented here. We argue nonetheless that our framework is more general, as coinductive properties are not hard-coded, but user-defined. Furthermore our work also has the advantage of shedding new light on the relationship between coinduction, confluence, and first-order satisfiability.

6 Conclusion

In this paper, we continued the study of relationships between CHR and coinduction started in [10]. Relying on universal coalgebra theory, we present a simple encoding of coalgebras and coinduction properties in CHR. Then, using the rule-decreasingness criterion we recently introduced, we realized effective coinductive proofs of bisimulations over streams. All the diagrammatic proofs sketched in the paper have been systematically verified by a prototype of a diagrammatic confluence checker [12] written in Ciao Prolog [15].

References

[1] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proceedings of the Int'l Conference on Principles and Practice of Constraint Programming (CP)*, volume 1330 of *LNCS*, pages 252–266, Berlin, Germany, 1997. Springer.

[2] Slim Abdennadher, Thom Frühwirth, and Holger Meuss. On confluence of Constraint Handling Rules. In *Proceedings of the Int'l Conference on Principles and Practice of Constraint Programming (CP)*, volume 1118 of *LNCS1*. Springer, 1996.

[3] Slim Abdennadher, Thom Frühwirth, and Holger Meuss. Confluence and semantics of Constraint Simplification Rules. *Constraints*, 4(2):133–165, 1999.

[4] P. Aczel. *Non-well-founded sets*. CSLI Publications, 1988.

[5] Jon Barwise and Larry Moss. *Vicious circles.* CSLI Publications, 1996.

[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] Thom Frühwirth. *Constraint Handling Rules.* Cambridge University Press, 2009.

[8] Joseph A. Goguen, Kai Lin, and Grigore Rosu. Circular coinductive rewriting. In *Automated Software Engineering*, pages 123–132, 2000.

[9] Andrew D. Gordon. A tutorial on co-induction and functional programming. In *In Proceedings of Glasgow Function Programming Workshop*, pages 78–95. Springer, 1994.

[10] R. Haemmerlé. (Co)-Inductive semantics for Constraint Handling Rules. *Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP'11) Special Issue*, 11(4–5):593–609, July 2011.

[11] R. Haemmerlé. Diagrammatic confluence for Cosntraint Handling Rules. *Theory and Practice of Logic Programming, 28th Int'l. Conference on Logic Programming (ICLP'12) Special Issue*, 2012. To appear.

[12] R. Haemmerlé. `metaCHR` package, 2012. available online at `http://clip.dia.fi.upm.es/~remy/metaCHR/`.

[13] R. Haemmerlé, P. López, and M. Hermenegildo. CLP projection for Constraint Handling Rules. In *Proceedings of the 13th Int'l ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 137–148. ACM Press, July 2011.

[14] Rémy Haemmerlé and François Fages. Abstract critical pairs and confluence of arbitrary binary relations. In *Proceedings of the Int'l Conference on Rewriting Techniques and Applications (RTA)*, number 4533 in LNCS, pages 214–228, Berlin, Germany, 2007. Springer.

[15] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, 2012.

[16] Ekaterina Komendantskaya and John Power. Coalgebraic semantics for derivations in logic programming. *In Proceedings of the Int'l Conference on Algebra and Coalgebra in Computer Science (CALCO)*, volume 6859 of *LNCS*, pages 268–282. Springer, 2011.

[17] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[18] Frank Raiser, Hariolf Betz, and Thom Frühwirth. Equivalence of CHR states revisited. In *Proceedings of Int'l Workshop on Constraint Handling Rules*, Report CW 555, pages 34–48. Kath. Univ. Leuven, 2009.

[19] Jan J. M. M. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005.

[20] D. Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2011.

[21] Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. In *ICLP*, volume 4079 of *LNCS*, pages 330–345, 2006.

[22] Vincent van Oostrom. Confluence by decreasing diagrams. *Theor. Comput. Sci.*, 126(2):259–280, 1994.

Qualitative Velocity Model: Representation, Reasoning and Application

Ester Martinez-Martin and M. Teresa Escrig and Angel P. del Pobil

Abstract On the way to autonomous systems, one of the key issues concerns spatial reasoning what involves solving problems with uncertainty. From the starting point of human nature, information must be represented in a way that any system can reason with imprecise knowledge about different physical aspects and make correct decisions from them. Keeping this idea in mind, this paper presents the qualitative model of velocity including representation, reasoning process and a real robotic application.

1 Introduction

Recent research is aimed at building autonomous systems that help human beings in their daily tasks, specially when they are tedious and/or repetitive. These *common* tasks can concern poorly defined situations. On this matter, humans have a remarkable capability to solve them without any measurements and/or any computations. Familiar examples are parking a car, cooking a meal, or summarizing a story. That is, humans make decisions based on information that is mostly perception, rather than accurate measurements as pointed out in [31]. Thus, qualitative reasoning properly fits this problem since it works on representation formalisms close to human conceptual schemata for reasoning about the surrounding physical environment (e.g. [30, 26]).

Ester Martinez-Martin
Universitat Jaume-I, Castellón, Spain e-mail: `emartine@icc.uji.es`

M. Teresa Escrig
Cognitive Robot, S.L. Parque Científico, Tecnológico y Empresarial (ESPAITEC), Universitat Jaume-I, Castellón, Spain e-mail: `mtescrig@c-robots.com`

Angel P. del Pobil
Universitat Jaume-I, Castellón, Spain e-mail: `pobil@icc.uji.es`

In qualitative spatial reasoning, it is common to consider a particular aspect of the physical world, that is, a magnitude, and to develop a system of qualitative relationships between entities which cover that aspect of the world to some degree. Examples of that can be found in many disciplines (e.g. geography [29], psychology [19], ecology [5, 28], biology [18, 15], robotics [22, 21, 17, 25] and Artificial Intelligence [7]). Actually, a qualitative representation of a magnitude results from an abstraction process and it has been defined in [26, 16] as that representation that *makes only as many distinctions as necessary to identify objects, events, situations, etc. in a given context for that magnitude*. The way to define those distinctions depends on two different aspects:

1. **the level of *granularity***. In this context, *granularity* refers to a matter of precision in the sense of the amount of information which is included in the representation. Therefore, a fine level of granularity will provide a more detailed information than a coarse level.
2. **the distinction between *comparing* and *naming* magnitudes** (as stated in [6]). This distinction refers to the usual comparison between *absolute* and *relative*. From a spatial point of view, this controversy corresponds to the way of representing the relationships among objects (see Fig. 1). As Levinson pointed out in [20], *absolute* defines an object's location in terms of arbitrary bearings such as cardinal directions (e.g. North, South, East, West), by resulting in binary relationships. Instead, *relative* leads to ternary relationships. Consequently, for *comparing* magnitudes, an object b is *any compared relationship* to another object a from the same Point of View (PV). It is worth noting that the comparison depends on the orientation of both objects *with respect to* (wrt) the PV, since objects a and b can be at any orientation wrt the PV. On the other hand, *naming* magnitudes divides the magnitude of any concept into intervals (sharply or overlapped separated, depending on the context) such that qualitative labels are assigned to each interval. Note that the result of reasoning with regions of this kind can provide imprecision. This imprecision will be solved by providing disjunction in the result. That is, if an object can be found in several qualitative regions, q_i or q_{i+1} or $\ldots$ or q_n, then all possibilities are listed as follows $\{q_i, q_{i+1}, \ldots, q_n\}$ by indicating this situation.

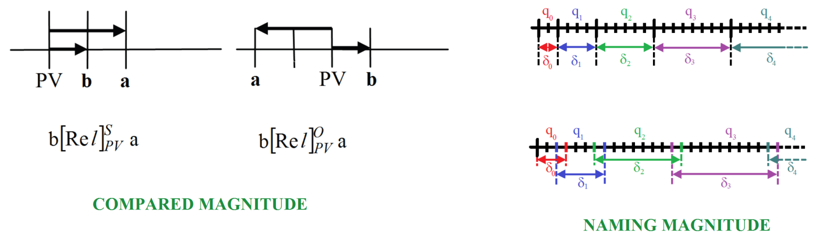

PV
b
a
$b[Rel]^S_{PV}$ a
a
PV
b
$b[Rel]^O_{PV}$ a
COMPARED MAGNITUDE
q_0 q_1 q_2 q_3 q_4
δ_0 δ_1 δ_2 δ_3 δ_4
NAMING MAGNITUDE

Fig. 1 An example of *compared* distances as represented in [9] (left) and an example of structure relations in *naming* magnitudes with sharply and overlapped separated qualitative areas (right)

From that starting point, and on the way to develop intelligent abilities to solve some service robotics problems, in this paper, we present a qualitative naming velocity model including its qualitative representation, the reasoning process and a real robotic application. With that aim, the structure of this paper is as follows: the proposed *qualitative velocity* model is analyzed in Section 2, while a real application is presented in Section 3. Finally, some conclusions are presented in Section 4.

2 Qualitative Velocity Model

The velocity is the physical concept that measures the distance travelled by an object per unit of time. From a physical point of view, this concept can be mathematically defined as:

$$Velocity = \frac{Distance\ Travelled}{Time\ of\ Travel} \tag{1}$$

2.1 Representation

The first issue to be solved refers to the way to represent the magnitude to be modelled, that is, the *velocity*. So, from the previous definition of velocity and focusing on developing its qualitative naming model, the velocity representation will be composed of three elements:

1. the **number of objects** implied in each relation (i.e. *arity*). From the physical definition of velocity, the relationships to be defined imply two objects such that an object acts as reference and the other one is referred.
2. the **set of velocity relations between objects**. It depends on the considered level of *granularity*. In a formal way, this set of relations is expressed by means of the definition of a *Reference System (**RS**)* consisting of:

   - a *set of qualitative symbols* in increasing order represented by $Q = \{q_0, q_1, ..., q_n\}$, where q_0 is the qualitative symbol closest to the *Reference Object (RO)* and q_n is the one furthest away, going to infinity. Here, by cognitive considerations, the acceptance areas have been chosen in increasing size. Note that this set defines the different areas in which the workspace is divided and the number of them will depend on the *granularity* of the task, as abovementioned
   - the *structure relations*, $\Delta r = \{\delta_0, \delta_1, ..., \delta_n\}$, describe the acceptance areas for each qualitative symbol q_i. So, δ_0 corresponds to the acceptance area of qualitative symbol q_0; δ_1 to the acceptance area of symbol q_1 and so on. These acceptance areas are quantitatively defined by means of a set of close or open intervals delimited by two extreme points: the initial point of the interval j,

δ_j^i, and the ending point of the interval j, δ_j^e. Thus, the structure relations are rewritten by:

$$\begin{cases} \Delta r = \left\{ \left[\delta_0^i, \delta_0^e\right[, \left[\delta_1^i, \delta_1^e\right[, \ldots, \left[\delta_n^i, \delta_n^e\right[\right\} & \text{if open intervals are considered} \\ \Delta r = \left\{ \left[\delta_0^i, \delta_0^e\right], \left[\delta_1^i, \delta_1^e\right], \ldots, \left[\delta_n^i, \delta_n^e\right] \right\} & \text{otherwise} \end{cases}$$

As a consequence, the acceptance area of a particular velocity entity, $AcAr(entity)$, is δ_j if its value is between the initial and ending points of δ_j, that is, $\delta_j^i \leq value(entity) \leq \delta_j^e$

3. the **operations**. The number of operations associated to a representation corresponds to the possible change in the PV. In this case, as only two objects are implied in the velocity relationships, only one operation can be defined: *inverse*.

By way of illustration, the velocity representation for a given context could be:

- the set of qualitative symbols: $Q_1 = \{$zero, slow, normal, quick$\}$
- the structure relations: $\Delta r_1 = \{[0,0[, [0, ud/2ut[, [ud/2ut, ud/ut[, [ud/ut, \infty[\}$, such that *ud* indicates the unit of distance or space travelled by an object while *ut* is the unit of time. Note that both values are context-dependent by being able to be set to metres and seconds, respectively; miles and hours or kilometres and hours, by depending on which velocity unit has been used
- The operations: as it is a binary relationship, the only allowed operation is *inverse*

2.2 The Basic Step of the Inference Process

The *Basic Step of the Inference Process (**BSIP**)* can be defined as: "given two relationships, (1) the object *b wrt a reference system, RS1*, and (2) the object *c wrt another reference system, RS2*, such that the object *b* is included into the second reference system, the BSIP obtains the relationship *c wrt RS1*" (see Fig. 2 for a graphical example of the BSIP)

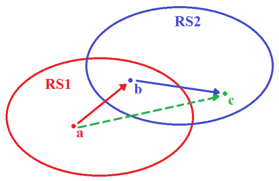
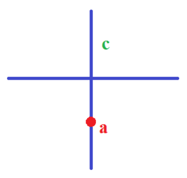


Fig. 2 A graphical example of the BSIP for qualitative models not based on projections, such that the dashed line represents the relationship to be inferred

For the concept of velocity, the BSIP can be defined as: given two velocity relationships between three spatio-temporal entities *a*, *b* and *c*, we want to find the velocity relationship between the two entities which is not initially given. However, it is important to take into account that the relative movement of the implied objects can be at any direction. For that reason, the BSIP is studied integrating the velocity concept with a qualitative orientation model. Note that, for this case, the qualitative orientational model of Freksa and Zimmerman [12, 11] has been redefined as depicted in Fig. 3 since the reference object is always on the *b* object. In that way, it is possible to reason with the extreme angles of the defined structure relations for the *Orientation Reference System (**ORS**)*.



ORS = {Q$_o$, Δr$_o$}

Q$_o$ = {front-left (fl), straight-front (sf), front-right (fr), left (l), none (n), right (r), back-left (bl), straight-back (sb), back-right (br)}

Δr$_o$ = {]90, 180[, [90, 90],]0, 90[, [180, 180], _, [0, 0],]180, 270[, [270, 270],]270, 360[}

Fig. 3 Redefinition of the Orientation Reference System (ORS) of Freksa and Zimmerman [12, 11] by means of its set of qualitative symbols (Q_o) and its structure relations (Δ_o)

Given that we are working with qualitative areas expressed by intervals, we use the two operations to add and subtract qualitative intervals presented in our previous work [24]. In particular, the functions to be performed are *qualitative_sum* (obtains the sum of two qualitative intervals), *qualitative_difference* (provides the subtraction of two qualitative intervals), *Find_UB_qualitative_sum* (obtains the qualitative interval corresponding to the upper bound of the qualitative sum of two qualitative intervals) and *Find_LB_qualitative_difference* (provides the qualitative interval corresponding to the lower bound of the qualitative subtraction of two qualitative intervals). In addition, five new functions are defined: *pythagorean_theorem_LB* and *pythagorean_theorem_UB* that obtain the qualitative interval corresponding to, respectively, the lower and upper bounds when the Pythagorean theorem is applied; *intermediate_orientation* provides the orientations existing between the two ones given as input (e.g. *intermediate_orientation(right, straight-front)* will be *front-right*); *open_interval*, from an orientation defined with a closed interval and another with an open interval, returns that corresponding to an open interval; and, *all_orientation_relationships* returns all the defined qualitative orientations (i.e. *right, front-right, straight-front, front-left, left, back-left, straight-back* and *back-right*).

Therefore, the BSIP for velocity has been solved as follows (see Fig. 4 for the graphical resolution): when any velocity relationship is zero, both velocity and orientation will be equal to the other involved relationship. When the two velocity relationships have the same orientation, the resulting relationship has the same orientation and its value corresponds to the qualitative sum of both relationships. On the contrary, if the relationships have an opposite orientation, the resulting relationship will be obtained as their qualitative difference and its orientation will be equal to that of higher velocity value. On the other hand, in the case both relationships have the same orientation but it corresponds to an open interval, the resulting relationship has the same orientation, although its value will be a disjunction of velocity relationships from the result of applying the Pythagorean theorem to the upper bound (UB) of the qualitative sum. When the orientation relationships corresponds to an open and a close interval such that one extreme of an interval matches up with an extreme of the other interval, then the resulting relationship will have the orientation of the open interval, while its value will be obtained from the Pythagorean theorem and the qualitative sum. The last special case refers to the case two orientation relationships are perpendicular. In that situation, the resulting relationship results of the Pythagorean theorem, whereas its orientation is the orientation between the orientations of the initial relationships. Finally, the remaining situations are solved by means of qualitative difference and the Pythagorean theorem. With regard to its orientation, it corresponds to all the possible orientation relationships.

The performance of the proposed method has been tested by comparing the results with those obtained by hand. The results obtained for the same orientation have been compared to the handwritten ones [10] by being the same.

2.3 The Complete Inference Process

From the BSIP definition, the Complete Inference Process (CIP) can be defined. It is necessary when more than two objects are involved in the reasoning mechanism. Mainly, it consists of repeating the BSIP as many times as possible with the initial information and the information provided by some BSIP until no more information can be inferred.

As knowledge about relationships between entities is often given in the form of *constraints*, the CIP can be formalized as a *Constraint Satisfaction Problem (CSP)* (see [30, 27, 8] for a survey). Note that a CSP is *consistent* if it has a *solution*. Moreover, a CSP can be represented by a *constraint network* where each node is labelled by a variable X_i or by the variable index i, and each directed edge is labelled by the relationship between the variables it links. Consequently, a path consistency algorithm can be used as a heuristic test for whether the defined constraint network is *consistent* [2], and, therefore, if the CSP has a *solution*. Thus, a number of algorithms for path consistency has been developed from its definition: a constraint graph is *path consistent* if for pairs of nodes (i, j) and all paths $i - i_1 - i_2 - \ldots - i_n - j$ between them, the direct constraint $c_{i,j}$ is tighter than the indirect constraint along the path, i.e. the composition of constraint $c_{i,i_1} \otimes \ldots \otimes c_{i_n,j}$ [13, 14].

A straight-forward way to enforce path-consistency on a CSP is to strengthen relationships by successively applying the following operation until a fixed point is reached:

$$c_{ij} := c_{ij} \oplus c_{ik} \otimes c_{kj} \tag{2}$$

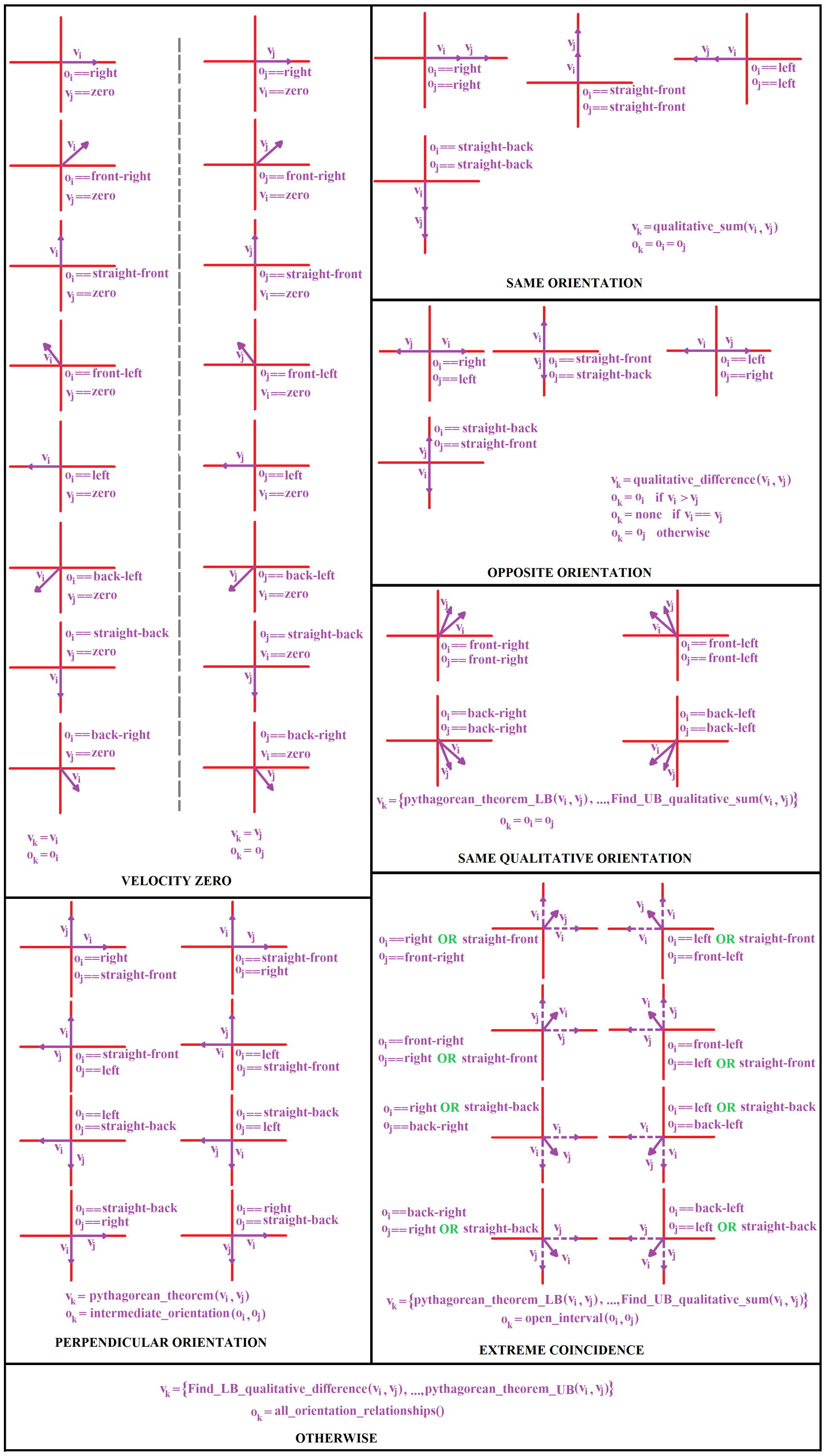

VELOCITY ZERO
$v_k = v_i$, $o_k = o_i$
$v_k = v_j$, $o_k = o_j$
SAME ORIENTATION
v_k = qualitative_sum(v_i, v_j)
$o_k = o_i = o_j$
OPPOSITE ORIENTATION
v_k = qualitative_difference(v_i, v_j)
$o_k = o_i$ if $v_i > v_j$
o_k = none if $v_i == v_j$
$o_k = o_j$ otherwise
SAME QUALITATIVE ORIENTATION
v_k = {pythagorean_theorem_LB(v_i, v_j), ..., Find_UB_qualitative_sum(v_i, v_j)}
$o_k = o_i = o_j$
PERPENDICULAR ORIENTATION
v_k = pythagorean_theorem(v_i, v_j)
o_k = intermediate_orientation(o_i, o_j)
EXTREME COINCIDENCE
v_k = {pythagorean_theorem_LB(v_i, v_j), ..., Find_UB_qualitative_sum(v_i, v_j)}
o_k = open_interval(o_i, o_j)
OTHERWISE
v_k = {Find_LB_qualitative_difference(v_i, v_j), ..., pythagorean_theorem_UB(v_i, v_j)}
o_k = all_orientation_relationships()

Fig. 4 Graphical resolution of the BSIP for velocity, where v_i and v_j represents the velocity relationships given as input with their corresponding orientation relationships (o_i and o_j); and v_k and o_k are the resulting velocity and orientation relationships

where the part $(c_{ik} \otimes c_{kj})$ of the formula computes composition and it obtains the constraint c_{ij}. This result is intersected ($\oplus$) with the preceding computed or user-defined constraints (if they exist). The complexity of such an algorithm is $O\left(n^3\right)$ where n is the number of nodes in the constraint graph [3, 23].

In a similar way, the computation of the full inference process for qualitative velocity can be viewed as an instance of the CSP. So, in order to determine whether a graph is complete we repeatedly compute the following operation until a fixed point is reached:

$$v_{x,y} := v_{x,y} \oplus v_{x,z} \otimes v_{z,y} \tag{3}$$

where $v_{x,y}$ corresponds to the velocity relationship between x and y. Again, The complexity of such an algorithm is $O(n^3)$, where n is the number of nodes in the network, that is, the number of velocity landmarks which are used in the inference process.

However, although path consistency eliminate some values of variable domains that will never appear in a solution, a search algorithm is still needed to solve the CSP. One way of solving this kind of problems is by means of *Constraint Logic Programming (CLP)* extended with *Constraint Handling Rules* (CHRs). So, on the one hand, CLP is a paradigm based on *First Order Predicate Logic* that combines the declarative of logic programming. Moreover, it provides a means to separate *competence* of a program (also called *logic* or *what*) from *performance* (*control* or *how*) with the efficiency of constraint solving. The main idea is to replace unification of terms -the heart of a logic programming system- by constraint handling in a constraint domain such that a constraint (or a set of constraints) is satisfied. The scheme is called CLP(X) where the argument X represents a computational domain.

Thus, a CLP program is defined as a finite set of clauses, while CHRs are logical formulas which basically define *simplification* and *propagation* over user-defined constraints [13]. In such way, *simplification* replaces constraints by a simpler constraints while preserving logical equivalence; and *propagation* adds new constraints logically redundant but being able to cause further simplification. So, repeatedly applying CHRs, the constraints are incrementally solved as in a built-in constraint solver. Consequently, CHRs allow the system to faster achieve an answer without backtracking.

In this context, a *Constraint Solver* (CS) is a CLP+CHRs program composed of a finite set of clauses from the CLP language and from the language of CHRs. So, a CS is defined for solving the CIP for the velocity model by means of a CLP+CHRs implementation (see Algorithm 1). The constraint $v_{x,y}$ is represented in the algorithms by the predicate $ctr\_vel(X,Y,V,\ O)$, where V is the list of primitive velocity relationships and O is the list of primitive orientation relationships forming the disjunctive constraint. That is, the velocity relationships between X and Y are V and the orientations of them are O. For instance, if the oriented velocity relationship between objects a and b is *slow, front-left*, then the corresponding predicate will be $ctr\_vel(a,b,slow,front-left$. Note that V and O represents sets since inferred relationships can consist of more than one relationship. The basic operation of a path consistency, the Equation 3, is implemented by means of two kinds of CHRs. The part of the basic operation in Equation 3 corresponding to the intersection ($c_{x,y} \oplus \ldots$), is implemented by simplification CHRs:

$$ctr\_vel(X,Y,V1,O1), ctr\_vel(X,Y,V2,O2) \Leftrightarrow intersection(V1,V2,V3), \\ intersection(O1,O2,O3) \mid ctr\_vel(X,Y,V3,O3) \tag{4}$$

so that, when we have two different contraints relative to the oriented velocity relationships between two objects X and Y (i.e. $ctr\_vel(X,Y,V1,O1)$ and $ctr\_vel(X,Y,V2,O2)$) what means the oriented velocity relationship between X and Y is $V1$ and $V2$, with their respective orientations $O1$ and $O2$, the system replaces both constraints by only one resulting from the intersection of the oriented velocity relationships $V1-O1$ and $V2-O2$.

On the contrary, the part of the basic operation in Equation 3 corresponding to the composition ($v_{x,z} \otimes v_{z,y}$) is implemented by propagation CHRs:

$$ctr\_vel(X,Y,V1,O1), ctr\_vel(Y,Z,V2,O2) \Rightarrow composition(V1,V2,V3), \\ composition(O1,O2,O3) \mid newc(X,Z,V3,O3) \tag{5}$$

where, given the oriented velocity relationships between objects X and Y and the existing relationships between objects Y and Z, the system introduces a new constraint that provides the oriented velocity relationship between objects X and Z. For that, the BSIP is applied (for simplification, avoiding all the possible cases in solving the velocity BSIP, here it is called *composition*)

Note that termination is guaranteed because the simplification rule 5 replaces $V1$ and $V2$ by the result $V3$ of intersecting $V1$ and $V2$ ($V3$ is the same as $V1$ or $V2$ or smaller than them) as well as the resulting relationship $O3$ of intersecting $O1$ and $O2$ replaces $O1$ and $O2$ ($O3$ is the same as $O1$ or $O2$ or smaller than them) and because propagation CHRs are never repeated for the same constraint goals.

Algorithm 1 CIP for Qualitative Velocity integrated with Qualitative Orientation

```
% Constraint declaration and definition
    (1a) constraints ctr_vel/4, ctr_vel/7.
    (1b) label_with ctr_vel(Nv, No, X, Y, V, O, I) if Nv ≥ 1 and No ≥ 1.
    (1c) ctr_vel(Nv, No, X, Y, V, O, I) :- member(V1, V), member(O1, O),
            ctr_vel(1, 1, X, Y, [V1], [O1], I).
% Initialize
    (2) ctr_vel(X, Y, V, O) ⇔ length(N, Nv), length(O, No) | ctr_vel(Nv, No, X,
            Y, V, O, 1).
% Special cases
    (3a) ctr_vel(Nv, No, X, Y, V, O, I) ⇔ empty(V) | false.
    (3b) ctr_vel(Nv, No, X, Y, V, O, I) ⇔ empty(O) | false.
    (3c) ctr_vel(Nv, No, X, Y, V, O, I) ⇔ Nv = N | true.
```

(3d) **ctr_vel**(Nv, No, X, Y, V, O, I) ⇔ No = 9 | **true**.
(3e) **ctr_vel**(Nv, No, X, X, V, O, I) ⇔ **true**.

% **Intersection**

(4a) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, X, Y, V2, O2, J) ⇔ **intersection**(V1, V2, V3), **length**(V3, Nv3), **intersection**(O1, O2, O3), **length**(O3, No3), K **is** **min**(I, J) | **ctr_vel**(Nv3, No3, X, Y, V3, O3, K).
(4b) **ctr_vel**(Nv1, No1, Y, X, V1, O1, I), **ctr_vel**(Nv2, No2, X, Y, V2, O2, J) ⇔ **inv_op**(V1, V11), **intersection**(V11, V2, V3), **length**(V3, Nv3), **inv_op**(O1, O11), **intersection**(O11, O2, O3), **length**(O3, No3), K **is** **min**(I, J) | **ctr_vel**(Nv3, No3, X, Y, V3, O3, K).
(4c) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, Y, X, V2, O2, J) ⇔ **inv_op**(V2, V12), **intersection**(V1, V12, V3), **length**(V3, Nv3), **inv_op**(O2, O12), **intersection**(O1, O12, O3), **length**(O3, No3), K **is** **min**(I, J) | **ctr_vel**(Nv3, No3, X, Y, V3, O3, K).

% **Composition**

(5a) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, Y, Z, V2, O2, J) ⇔ **velocity_zero**(V1), K **is** I+J | **ctr_vel**(Nv2, No2, X, Z, V2, O2, K).
(5b) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, Y, Z, V2, O2, J) ⇔ **velocity_zero**(V2), K **is** I+J | **ctr_vel**(Nv1, No1, X, Z, V1, O1, K).
(5c) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, Y, Z, V2, O2, J) ⇔ **lb_qualitative_difference**(O1, O2, O11), **ub_qualitative_difference**(O1,O2,O12), **build_result**(O11, O12, O3), **orientation_zero**(O3), **lb_qualitative_sum**(V1, V2, V11), **ub_qualitative_sum**(V1, V2, V12), **build_result**(V11, V12, V3), **length**(V3, Nv3), **higher_orientation**(O1, O2, O3), **length**(O3, No3), K **is** I+J | **ctr_vel**(Nv3, No3, X, Z, V3, O3, K).
(5d) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, Y, Z, V2, O2, J) ⇔ **lb_qualitative_difference**(O1, O2, O11), **ub_qualitative_difference**(O1, O2, O12), **build_result**(O11, O12, O3), **orientation_90degrees**(O3), **pythagoream_theorem**(V1,V2,O1, O2,V3), **length**(V3, Nv3), **maxmin_op**(O1,O2,O3), **length**(O3, No3), K **is** I+J | **ctr_vel**(Nv3, No3, X, Z, V3, O3, K).
(5e) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, Y, Z, V2, O2, J) ⇔ **lb_qualitative_difference**(O1, O2, O11), **ub_qualitative_difference**(O1,O2,O12), **build_result**(O11,O12,O3), **orientation_180degrees**(O3), **lb_qualitative_difference**(V1,V2,V11), **ub_qualitative_difference**(V1,V2,V12), **build_result**(V11,V12,V3), **non_zero**(V3), **length**(V3, Nv3), **higher_orientation**(O1, O2, O3), **length**(O3, No3), K **is** I+J | **ctr_vel**(Nv3, No3, X, Z, V3, O3, K).
(5f) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, Y, Z, V2, O2, J) ⇔ **lb_qualitative_difference**(O1, O2, O11), **ub_qualitative_difference**(O1,O2,O12), **build_result**(O11,O12, O3), **orientation_180degrees**(O3), **lb_qualitative_difference**(V1,V2,V11), **ub_qualitative_difference**(V1,V2,V12), **build_result**(V11,V12,V3),

zero(V3), **length**(V3, Nv3), K **is** I+J | **ctr_vel**(Nv3, _, X, Z, V3, _, K).

(5g) **ctr_vel**(Nv1, No1, Y, X, V1, O1, I), **ctr_vel**(Nv2, No2, Y, Z, V2, O2, J) ⇔ **inv_op**(V1, V11), **velocity_zero**(V11), K **is** I+J | **ctr_vel**(Nv2, No2, X, Z, V2, O2, K).

(5h) **ctr_vel**(Nv1, No1, Y, X, V1, O1, I), **ctr_vel**(Nv2, No2, Y, Z, V2, O2, J) ⇔ **velocity_zero**(V2), **inv_op**(V1,V11), **length**(V11,Nv11), K **is** I+J | **ctr_vel**(Nv11, No1, X, Z, V11, O1, K).

(5i) **ctr_vel**(Nv1, No1, Y, X, V1, O1, I), **ctr_vel**(Nv2, No2, Y, Z, V2, O2, J) ⇔ **lb_qualitative_difference**(O1, O2, O11), **ub_qualitative_difference**(O1,O2,O12), **build_result**(O11,O12,O3), **orientation_zero**(O3), **inv_op**(V1, V11), **lb_qualitative_sum**(V11, V2, V21), **ub_qualitative_sum**(V11,V2,V22), **build_result**(V21, V22, V3), **length**(V3, Nv3), **higher_orientation**(O1, O2, O3), **length**(O3, No3), K **is** I+J | **ctr_vel**(Nv3, No3, X, Z, V3, O3, K).

(5j) **ctr_vel**(Nv1, No1, Y, X, V1, O1, I), **ctr_vel**(Nv2, No2, Y, Z, V2, O2, J) ⇔ **lb_qualitative_difference**(O1, O2, O11), **ub_qualitative_difference**(O1,O2,O12), **build_result**(O11,O12,O3), **orientation_90degrees**(O3), **inv_op**(V1, V11), **pythagoream_theorem**(V11, V2, O1, O2, V3), **length**(V3, Nv3), **maxmin_op**(O1, O2, O3), **length**(O3, No3), K **is** I+J | **ctr_vel**(Nv3, No3, X, Z, V3, O3, K).

(5k) **ctr_vel**(Nv1, No1, Y, X, V1, O1, I), **ctr_vel**(Nv2, No2, Y, Z, V2, O2, J) ⇔ **lb_qualitative_difference**(O1, O2, O11), **ub_qualitative_difference**(O1,O2,O12), **build_result**(O11,O12,O3), **orientation_180degrees**(O3), **inv_op**(V1, V11), **lb_qualitative_difference**(V11, V2, V21), **ub_qualitative_difference**(V11,V2,V22), **build_result**(V21,V22,V3), **non_zero**(V3), **length**(V3, Nv3), **higher_orientation**(O1, O2, O3), **length**(O3, No3), K **is** I+J | **ctr_vel**(Nv3, No3, X, Z, V3, O3, K).

(5l) **ctr_vel**(Nv1, No1, Y, X, V1, O1, I), **ctr_vel**(Nv2, No2, Y, Z, V2, O2, J) ⇔ **lb_qualitative_difference**(O1, O2, O11), **ub_qualitative_difference**(O1,O2,O12), **build_result**(O11,O12,O3), **orientation_180degrees**(O3), **inv_op**(V1, V11), **lb_qualitative_difference**(V11, V2, V21), **ub_qualitative_difference**(V11,V2,V22), **build_result**(V21,V22,V3), **zero**(V3), **length**(V3, Nv3), K **is** I+J | **ctr_vel**(Nv3, _, X, Z, V3, _, K).

(5m) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, Z, Y, V2, O2, J) ⇔ **velocity_zero**(V1), **inv_op**(V2, V12), **length**(V12, Nv12), K **is** I+J | **ctr_vel**(Nv12, No2, X, Z, V12, O2, K).

(5n) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, Z, Y, V2, O2, J) ⇔ **inv_op**(V2, V12), **velocity_zero**(V12), K **is** I+J | **ctr_vel**(Nv1, No1, X, Z, V1, O1, K).

(5o) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, Z, Y, V2, O2, J) ⇔ **lb_qualitative_difference**(O1, O2, O11), **ub_qualitative_difference**(O1,O2,O12), **build_result**(O11,O12,O3),

orientation_zero(O3), **inv_op**(V2,V12), **lb_qualitative_sum**(V1,V12, V21), **ub_qualitative_sum**(V1,V12,V22), **build_result**(V21,V22,V3), **length**(V3, Nv3), **higher_orientation**(O1, O2, O3), **length**(O3, No3), K **is** I+J | **ctr_vel**(Nv3, No3, X, Z, V3, O3, K).

(5p) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, Z, Y, V2, O2, J) ⇔ **lb_qualitative_difference**(O1, O2, O11), **ub_qualitative_difference**(O1,O2,O12), **build_result**(O11,O12,O3), **orientation_90degrees**(O3), **inv_op**(V2, V12), **pythagoream_theorem**(V1, V12, O1, O2, V3), **length**(V3, Nv3), **maxmin_op**(O1, O2, O3), **length**(O3, No3), K **is** I+J | **ctr_vel**(Nv3, No3, X, Z, V3, O3, K).

(5q) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, Z, Y, V2, O2, J) ⇔ **lb_qualitative_difference**(O1, O2, O11), **ub_qualitative_difference**(O1,O2,O12), **build_result**(O11,O12,O3), **orientation_180degrees**(O3), **inv_op**(V2, V12), **lb_qualitative_difference**(V1, V12, V21), **ub_qualitative_difference**(V1,V12,V22), **build_result**(V21,V22,V3), **non_zero**(V3), **length**(V3, Nv3), **higher_orientation**(O1, O2, O3), **length**(O3, No3), K **is** I+J | **ctr_vel**(Nv3, No3, X, Z, V3, O3, K).

(5r) **ctr_vel**(Nv1, No1, X, Y, V1, O1, I), **ctr_vel**(Nv2, No2, Z, Y, V2, O2, J) ⇔ **lb_qualitative_difference**(O1, O2, O11), **ub_qualitative_difference**(O1,O2,O12), **build_result**(O11,O12,O3), **orientation_180degrees**(O3), **inv_op**(V2, V12), **lb_qualitative_difference**(V1, V12, V21), **ub_qualitative_difference**(V1,V12,V22), **build_result**(V21,V22,V3), **zero**(V3), **length**(V3, Nv3), K **is** I+J | **ctr_vel**(Nv3, _, X, Z, V3, _, K).

Two predicates, *ctr_vel* of arity 4 and 7 are declared in rule $(1a)$. The initial qualitative velocity information is introduced through predicates *ctr_vel/4*. So, the predicates of type *ctr_vel/4* are translated into the predicates *ctr_vel/7* by means of rule $(1b)$ where the length of the velocity relation (N_v), the length of the orientation relation (No) and the length of the shortest path from which the constraint (I) is derived are added. A path length equal to 1 means that the constraint is direct, that is, it is user-defined, not obtained from derivation. All those arguments are included to increase efficiency. The two first ones will avoid compositions between constraints which do not give more information (rules $3c$ and $3d$) because all the qualitative primitive (velocity or orientation) relationships are included in the disjunction. They will also restrict constraints involved in a propagation to be disjunction-free, as it is explained below. The last argument is used to restrict the propagation CHRs to involve at least one constraint. The constraints will be treated by the CLP clause $(1c)$ if the relations, V and O, represent a disjunction of primitive relationships (rule $1b$). In predicate $(1c)$, $member(V1,V)$ and $member(O1,O)$ non-deterministically choose one primitive constraint for velocity and for orientation respectively, V and O, from the disjunctive constraints $V1$ and $O1$, by implementing the backtrack search part of the algorithm.

Special cases are simplification CHRs. (3*a*) and (3*b*) detect inconsistent constraints. When the constraint relates three objects with an empty (velocity or orientation) relationship, the constraint is substituted by the built-in predicate *false* and the full predicate fails. If it is not be expected behaviour when substituting the inconsistent constraint by *true*, that is, deleting this constraint. (3*c*) and (3*d*) delete constraints which contain the full primitive qualitative velocity relationship set or the full primitive qualitative orientation relationship respectively, while (3*e*) deletes constraints which contain only one point instead of two.

Simplification CHRs (4*a*) to (4*c*) perform intersections which allow the simplification of redundant information. Rule (4*a*) implements intersection in the way that it was originally defined in Equation 4, i.e., given two constraints which relate the same three spatial objects, the more restricted relationships (velocity as well as orientation) between both constraints is obtained by the predicates $intersection(V1,V2,V3)$ and $intersection(O1,O2,O3)$ and those constraints are substituted by a new one which relates the same three objects with the new relationships $V3$ and $O3$ among them. On the other hand, rules (4*b*) and (4*c*) solve intersection when inverse operation is respectively applied to the first or the second constraint in the head of the original intersection rule (eq. 4). Therefore, it is possible to obtain intersection if the inverse operation is applied to the relationship or disjunction of relationships in the guard part of the rules. Note that the application of the defined operation to a disjunction of relationships is equivalent to the application of that operation to each relationship included in the disjunction of relation.

Propagation CHRs (5*a*) to (5*r*) perform compositions by using the proposed algorithm. Several special cases have to be considered in order to properly solve the composition equation (Equation 5). These cases depend on the orientation of objects. For that reason, it has to be distinguished between them (predicates *orientation_zero*, *orientation_90degrees* and *orientation_180degrees*). Furthermore, in a similar way to what it happens to the simplification rule eq. 4, the application of the inverse operation to the first constraint of the two which define the head of the original composition rule define the CHRs (5*g*) to (5*l*); whereas if it is applied to the second constraint, the CHRs (5*m*) to (5*r*) are obtained. Hence, a total of 18 propagation CHRs are needed to cover all possible combinations of constraints. Note that another optimization is introduced in order to make composition more efficient. It is based on the fact that the resulting relationship of combining a zero velocity with another velocity relation is that velocity relation different from zero.

3 A Practical Application

A real application of the proposed method is presented. In this case, the qualitative velocity model has been implemented on a mobile robot. The aim of this system is to assist human beings in performing a variety of tasks such as carrying person's tools or delivering parts. One of the major requirements of such robotic assistants is the ability to track and follow a moving person through a non-predetermined, unstructured environment. To achieve this goal, two different tasks have to be carried out: person recognition and segmentation from the surrounding environment, and motion control to follow the person using the recognition results. In particular, in this section, we proposed a qualitative reasoning method to achieve the second task to be performed.

For that, an indoor pan-tilt-zoom (PTZ) camera was mounted on a Pioneer 3-DX mobile platform [1] without restricting its autonomy and flexibility as depicted in Fig. 5. The core of the PTZ system is a Canon VC-C4 analog colour camera [4] with a resolution of $320x240$ pixels, which is integrated with the mobile platform hardware.

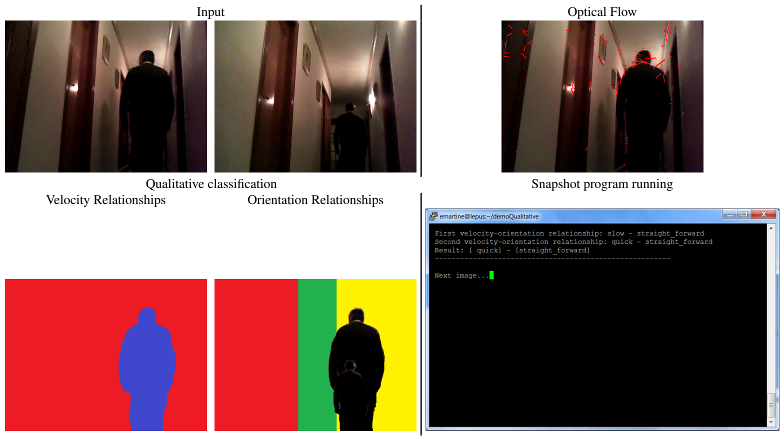
Fig. 5 Experimental set-up: external view of the used mobile platform (left) and a more detailed view of the camera (right)

So, on the one hand, the system knows both its velocity and its orientation through the information obtained from its motors. On the other hand, an image processing based on optical flow provides an estimation of the velocity and orientation relationships corresponding to the person to be followed. Therefore, from these two relationships (the one obtained by the robotic system itself and the other corresponding to the person from image processing), the system is able to determine the required velocity-orientation relationship that allows it to know the required trajectory change to properly follow and assist that person. An example of the obtained results can be seen in Fig. 6.

4 Conclusions

In this paper, we have proposed a qualitative velocity model including representation, reasoning process and a real robotic application of that. From the starting point that the development of any qualitative model consists of a representation of the magnitude at hand and the reasoning process, we have developed a qualitative model for physical velocity such that velocity and orientation are combined, the basic step of the inference process is expressed in terms of qualitative sums and differences, and, given that knowledge about relationships between entities is often provided in the form of *constraints*, the complete inference process is formalized as a *Constraint Satisfaction Problem (CSP)*.

As future work we will investigate the development of new qualitative models based on intervals of aspects such as: time, weight, body sensations (such as hunger,


Input
Optical Flow
Qualitative classification
Velocity Relationships
Orientation Relationships
Snapshot program running
emartine@lepus:~/demoQualitative

```
First velocity-orientation relationship: slow - straight_forward
Second velocity-orientation relationship: quick - straight_forward
Result: [ quick] - [straight_forward]
----------------------------------------------------------

Next image...
```

Fig. 6 Results obtained with the real robot when the qualitative velocity model proposed in the previous section has been used. In this case, velocity relationships are labelled as $Q = \{$zero, slow, normal, quick$\}$ coded in the image by purple, red, green and blue respectively. On the other hand, orientation relationships correspond to the modified Freksa and Zimmermann's approach such that fl is coded by red, sf by green, fr by yellow, l by blue, r by purple, bl by orange, sb by rose and br by olive

sleepiness, tiredness, love, etc.), etc. with the purpose of providing robots with intelligent abilities to solve service robotics problems.

Acknowledgements This research was partly supported by WCU (World Class University) program through the National Research Foundation of Korea funded by the Ministry of Education, Science and Technology (Grant No. R31-2008-000-10062-0), by Ministerio de Ciencia e Innovación (DPI2011-27846), by Generalitat Valenciana (PROMETEO/2009/052) and by Fundació Caixa Castelló-Bancaixa (P1-1B2011-54).

References

1. Adept-Technology: `http://www.mobilerobots.com/researchrobots/researchrobots/pioneerp3dx.aspx` (2004)
2. Allen, J.: Maintaining knowledge about temporal intervals. Communications of the ACM **26**, 832–843 (1983)
3. Bessière, C.: A simple way to improve path consistency processing in interval algebra networks. In: 13th National Conf. on AI, vol. 1, pp. 375–380. AAAI Press, Portland, Oregon (1996)
4. Canon: `http://www.canon-europe.com/For_Home/Product_Finder/Web_Cameras/Web_Cameras/VC-C4/` (2001)
5. Cioaca, E., Linnebank, F., Bredeweg, B., Salles, P.: A qualitative reasoning model of algal bloom in the danube delta biosphere reserve (ddbr). Ecol Informatics **4**(5-6), 282–298 (2009)
6. Clementini, E., Felice, P.D., Hernández, D.: Qualitative representation of positional information. AI **95**(2), 317–356 (1997)

7. Cohn, A., Hazarika, S.: Qualitative spatial representation and reasoning: An overview. Fundamenta Informaticae **46**(1-2), 1–29 (2001)
8. Dimopoulos, Y., Stergiou, K.: Propagation in csp and sat. In: CP, pp. 137–151. Nantes, France (2006)
9. Escrig, M., Toledo, F.: Reasoning with compared distances at different levels of granularity. In: CAEPIA. Gijón, Spain (2001)
10. Escrig, M., Toledo, F.: Qualitative Velocity, *LNAI*, vol. 2504. Springer (2002)
11. Freksa, C.: Using orientation information for qualitative spatial reasoning. In: Theories and Methods of Spatio-Temporal Reasoning in Geographic Space, *LNCS*, vol. 639. Springer-Verlag, Berlin, Germany (1992)
12. Freksa, C., Zimmermann, K.: On the utilization of spatial structures for cognitively plausible and efficient reasoning. In: SMC, pp. 261–266. Chicago, USA (1992)
13. Frühwirth, T.: Constraint Handling Rules, *LNCS*, vol. 910, chap. Constraint Programming: Basic and Trends, pp. 90–107. Springer (1994)
14. Frühwirth, T.: Reasoning with constraint handling rules. Tech. rep., European Computer-Industry Research Center (1994)
15. Guerrin, F., Dumas, J.: Knowledge representation and qualitative simulation of salmon redd functioning. part i: qualitative modeling and simulation. Biosystems **59**(2), 75–84 (2001)
16. Hernández, D.: Qualitative Representation of Spatial Knowledge, *LNAI*, vol. 804. Springer-Verlag (1994)
17. Holzmann, C.: Rule-based reasoning about qualitative spatiotemporal relations. In: 5th Int. workshop on Middleware for pervasive and ad-hoc computing, pp. 49–54. Newport Beach, CA, USA (2007)
18. King, R., Garrett, S., Coghill, G.: On the use of qualitative reasoning to simulate and identify metabolic pathways. Bioinformatics **21**(9), 2017–2026 (2005)
19. Knauff, M., Strube, G., Jola, C., Rauh, R., Schlieder, C.: The psychological validity of qualitative spatial reasoning in one dimension. Spatial Cognition & Computation **4**(2), 167–188 (2004)
20. Levinson, S.: Space in Language and Cognition. Explorations in Cognitive Diversity. Cambridge University Press, UK (2003)
21. Liu, H.: A fuzzy qualitative framework for connecting robot qualitative and quantitative representations. IEEE Trans. on Fuzzy Systems **16**(6), 1522–1530 (2008)
22. Liu, H., Brown, D., Coghill, G.: Fuzzy qualitative robot kinematics. IEEE Trans. on Fuzzy Systems **16**(3), 808–822 (2008)
23. Mackworth, A., Freuder, E.: The complexity of some polynomial networks consistency algorithms for constraint satisfaction problems. AI **25**, 65–74 (1985)
24. Martínez-Martín, E., Escrig, M., del Pobil, A.: A General Framework for Naming Qualitative Models Based on Intervals, *AISC*, vol. 151, pp. 681–688. Springer-Verlag (2012)
25. Moratz, R., Wallgrün, J.: Spatial reasoning about relative orientation and distance for robot exploration. In: Spatial Information Theory. Foundations of Geographic Information Science, LNCS: 2825, pp. 61–74 (2003)
26. Renz, J., Nebel, B.: Qualitative Spatial Reasoning Using Constraint Calculi, pp. 161–215. Springer Verlag, Berlin, Germany (2007)
27. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming, *Foundations of AI*, vol. 35. Elsevier, Amsterdam, The Netherlands (2006)
28. Salles, P., Bredeweg, B.: Modelling population and community dynamics with qualitative reasoning. Ecol Modell **195**(1-2), 114–128 (2006)
29. van de Weghe, N., Cohn, A., de Tré, G., de Maeyer, P.: A qualitative trajectory calculus as a basis for representing moving objects in geographical information systems. Control and Cybernetics **35**(1), 97–119 (2006)
30. Westphal, M., Wölfl, S.: Qualitative csp, finite csp, and sat: Comparing methods for qualitative constraint-based reasoning. In: IJCAI, pp. 628–633 (2009)
31. Zadeh, L.: A new direction in ai. toward a computational theory of perceptions. AI Magazine **22**(1), 73–84 (2001)

Parallel Execution of Constraint Handling Rules on a Graphical Processing Unit

Amira Zaki, Thom Frühwirth, and Ilvar Geller

Faculty of Engineering and Computer Sciences, Ulm University, Germany
`{amira.zaki,thom.fruehwirth,ilvar.geller}@uni-ulm.de`

Abstract. Graphical Processing Units (GPUs) consist of hundreds of small cores, collectively operating to provide massive computation capabilities. The aim of this work is to utilize this technology to execute Constraint Handling Rules (CHR) which are inherently parallel. A translation scheme is defined to transform a subset of CHR rules to C/C++, then to use a GPU to fire the rules on all combinations of constraints. As proof of concept, the scheme was performed on several CHR examples.

Keywords: Constraint Handling Rules, CUDA, GPU, Parallel

1 Introduction

In recent years, graphics hardware has incurred a rapid increase in terms of performance. Its use has evolved from merely rendering graphics to offering a powerful platform for parallel computations. It has facilitated high performance computing to be readily available on a typical desktop, shipped as the common graphics processing units (GPUs). The powerful technology has become abundant at a relatively low price, hence it is tempting for researchers to harness this power for general-purpose computing to tackle intensive computations.

Furthermore, the introduction of CUDA (Compute Unified Device Architecture) by NVIDIA, a leading GPU manufacturer, gave rise to a new era of computing. CUDA allows users to seamlessly run C, C++ and Fortran code on a GPU, without requiring to resort to assembly language. CUDA has helped unleash the power of GPUs to be easily available to wide range of users [6]. Several works have emerged making use of this computing potential, like several number crunching algorithms [1], graph algorithms [4] and various others.

Constraint Handling Rules (CHR) is a committed-choice rule-based programming language having a well-established formal basis. The abstract semantics of CHR is inherently parallel, it involves multi-set rewriting over a multi-set of constraints [2]. CHR rules can be applied in parallel even to overlapping multi-sets of constraints, if they are removed by at most one rule. Thus it supports a very fine-grained form of parallelism.

A first abstract operational semantics for parallel CHR has been proposed by Thom Frühwirth [3]. Early prototypes for parallel execution of CHR have been developed based on shared-transaction memory (STM) by Edward Lam and Martin Sulzmann [5, 7]. Experimental evaluation of these systems revealed a significant boost and often linear speedup over sequential executions. However, these prototypes showed that conflicts occur with the STM-based approach; this results in a slow down of the execution. More recently, Andrea Triossi [8, 9] has developed a framework for compiling CHR to specialized hardware circuits. A code fragment of CHR is compiled into a low level hardware description language, to generate a specialized digital circuit on a Field Programmable Gate Array (FPGA) for each specific CHR code fragment. The hardware blocks then enable a parallel execution model for the compiled CHR fragment.

In this work, we aim to develop a prototype whilst exploiting the power of graphics processing units to simulate the execution of a subset of CHR by experimenting with different potential execution schemes. A translation scheme from CHR to CUDA is defined in such a manner that the output CUDA code is run in parallel, hence investigating the potential speed up of a parallel execution of the CHR rules.

2 CHR Overview

Constraint Handling Rules (CHR) is a high-level, concurrent, committed-choice, constraint logic programming language [2]. It consists of guarded rules that perform conditional transformation of multi-sets of constraints, known as a constraint store, until a fixed point is reached. CHR utilizes built-in constraints which are predefined by the host language, and other user-defined CHR constraints. A CHR constraint is a predicate having a name and a certain number of arguments. A CHR program typically consists of a finite set of rules, which can be generally represented with a simpagation rule as follows:

rule_name `@` *heads_kept* \ *heads_removed* `<=>` *guard* `|` *built_ins, body_constraints.*

The *rule_name* is an optional unique identifier given to a rule. *heads_kept, heads_removed, body_constraints* are a conjunction of one or more CHR constraints, where the constraints are kept, removed or added respectively. The rule operates by matching the *heads_kept* and *heads_removed* with constraints in the constraint store, then checks for the guard validity. If it holds then the *heads_removed* are removed from the store, and replaced with the *built_ins* and the *body_constraints.* Additionally there are propagation and simplification rules, which do not remove and do not keep any constraints respectively.

3 CUDA

CUDA offers a data parallel programming model that is supported on NVIDIA GPUs [10]. In this model, the host program launches a sequence of kernels, where a kernel is a hierarchy of threads. Threads are grouped into blocks, and blocks are grouped into a grid. The sizes of grids, blocks and threads is hardware dependent but a block typically contains 512 threads.

Each thread has a unique local index in its block (`threadIdx`), and each block of dimension (`blockDim`) has a unique index in the grid (`blockIdx`). The three indexes given are built-in 3-component vectors to access their values. Threads in a single block will be executed on a single multiprocessor, sharing the software data cache, and can synchronize and share data with threads in the same block. Threads in different blocks may be assigned to different multiprocessors concurrently, to the same multiprocessor concurrently, or may be assigned to the same or different multiprocessors at different times, depending on how the blocks are scheduled dynamically.

Thus a kernel is executed N times in parallel by N different CUDA threads. A kernel is defined using C/C++ functions and characterized with the `__global__` declaration specifier indicating that it is callable from the host only. The number of threads per block and the number of blocks per grid is specified using the `<<<...>>>>` statement. Other functions which are callable from the device only are indicated with `__device__` specifier.

4 Translation Scheme

The approach presented in this paper involves translating CHR rules into an imperative form, which can then be easily transformed into CUDA code to run on a graphics card. The CUDA code is run in parallel to simulate the parallel firing of the CHR rules. A subset of the CHR language is used, which includes only simplification rules and simpagation rules that do not introduce more constraints than those removed. This subset is a necessity due to the limited memory of the graphics card.

The CUDA code defines a structure for every CHR constraint, to store the information associated with it. The constraint store is then modeled and stored as an array of fixed length consisting of the structures. The dynamic nature of CHR constraints could be captured more clearly with a dynamic data structure such as a list structure, however this would not be practical on the graphics card. The GPU can not allocate memory in kernel calls because it does not contain a memory management unit. Moreover despite developments to support this feature in the future, there would still be an overhead introduced due to synchronization issues. Moreover, the total number of constraints possible in the lifetime of a program has to be known in advance, due to memory limitations of the GPU. For the scope of this work, a compromise was reached by choosing the subset of CHR that ensures an easy prediction of the number of constraints incurred by a program.

4.1 CHR Constraint Representation

Constraints represent data in a program and can be introduced and removed from the constraint store by CHR rules. Every constraint is a distinguished predicate of first order logic, having a name and a number of arguments. With the CHR Prolog implementation, every CHR constraint used has to be declared with a `chr_constraint/1` declaration by the constraint specifier. In its extended form, a constraint specifier is *constraint_name(type*$_1, \ldots,$ *type*$_n$*)*, where *constraint_name* is the constraint's functor, n its arity and the *type*$_i$ are argument specifiers. An argument specifier is a mode, followed by a type. Similar to the work done in [11], for every constraint a C/C++ structure is defined having the same name as the functor and with a listing of the arguments of the constraint using the provided types. Additional meta-data about the constraint can also be stored within the structure. Thus for a CHR constraint *constraint_name(type*$_1, \ldots,$ *type*$_n$*)*, the corresponding C/C++ structure can be defined accordingly:

```
typedef struct {
      type1 var1;
      ...
      typen varn;
      boolean isRemoved;
} constraint_name;
```

The variables var_i are used to store the arguments of the constraint. Additionally every constraint structure generated should contain a `boolean isRemoved` variable, which indicates the presence of the constraint in the store. It should be changed during the computation if the constraint was removed from the store.

As an example, a CHR constraint to describe a candidate number for the computation of a minimum can be expressed as: `min(+int)`. Using the previously mentioned translation scheme, it can be transformed into the following `MIN` structure:

```
typedef struct {
   int value;
   bool isRemoved;
} MIN;
```

The constraint store which contains N candidate minimum constraints is modeled as an array named `min_store` as follows: `MIN min_store [N]`.

4.2 CHR Rule Representation

The CHR subset chosen, should ensure that the body includes at most as many added constraints as the removed ones. Generic simpagation rules are taken as expressed in section 2. The subset chosen ensures that the number of constraints removed is at most equal to the added body constraints, thus:

$$|heads\_removed| \geq |body\_constraints|$$

A CHR rule can be translated into a function in C/C++, by mapping it to the following form:

```
void rule_name (calling_heads_kept, calling_heads_removed) {
    if(head constraints are not marked as removed
         && matching of variables in heads holds
          && guard holds) {
            equivalent built-ins, setting body constraints
    }
}
```

The name for a rule is optional in CHR but it is needed in C/C++ as a unique identifier for each function. The parameter list contains a listing referencing the structures of the equivalent head constraints. Constraints are fired only if they are actually present in the constraint store, thus first a check must be performed to check that they have not been marked as removed. Then when firing the rule, variables may exist in common between the head constraints and matching is performed. Thus in the translated C/C++ code matching of the variables must be explicitly ensured. Lastly before the rule fires, the guard must be checked if it holds and this must also be performed in the translated code. The guard is a typical condition and contains only built-in constraints which are expressed as straight forward C/C++ built-ins. The body consists of built-in constraints and overwrites existing constraints or deletes them by changing their `isRemoved` status variable.

Added constraints are actually overwritten in the place of removed head constraints. This is done by modifying their respective structure variables, to match the newly produced constraint. Head constraints that are removed and not overwritten, must have their `isRemoved` variable changed.

For example to calculate the minimum of a multi-set of numbers n_i expressed as `min(`n_1`),...min(`n_k`)`, a simpagation rule that takes two `min` candidates and removes the one with the larger value is given as:

```
minimum @  min(A) \ min(B) <=> A=<B | true.
```

The equivalent C/C++ function using the previously mentioned translation scheme is shown below. No variable matching is done in the rule, however both constraints are first checked for being present in the store. The guard is also checked, if all holds then the constraint with the larger value is removed from the store.

```
void minimum(MIN &a, MIN &b) {
    if(!a.isRemoved && !b.isRemoved && a.value <= b.value)
         b.isRemoved = true;
}
```

4.3 CHR Rule Firing

The translation scheme involves transforming the query constraints into the specified structure format and then placing them in an indexed array. The rule is fired on every possible combination of constraints. This exhaustive method can be optimized and changed according to the problem to be solved.

For the running minimum example, it is sufficient to apply the exhaustive firing. This means that the rule is fired for each pair of constraints; for an array `min_store` of N constraints, N^2 pairs are constructed.

The `MIN` constraints are stored in an indexed array and for each constraint pair the rule-function is called. Using the exact same constraint in the rule does not make any sense since a constraint is only present once in the store and should not be fired against itself unless it is present twice in the store, therefore a further if statement is needed. At the end of the loops the result will be a single non-removed constraint in the array which contains the smallest value. Encapsulating this functionality into a fire function is shown below:

```
void fire_minimum(MIN *min_store, int N) {
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; j += 1)
      if (i != j)
        minimum(min_store[i], min_store[j]);
}
```

4.4 Mapping to CUDA

After translating a CHR program into a C/C++ program it can be mapped with little effort into a CUDA program. Every CHR rule was mapped into a C/C++ function, which is now defined to be called by a thread from a device, and thus is redefined by adding the `__device__` declaration specifier. The function is redefined to the following:

```
__device__ void minimum(MIN &a, MIN &b) {
  if(!a.isRemoved && !b.isRemoved && a.value <= b.value)
   b.isRemoved = true;
}
```

The calls to the rule-firing functions, which were shown in the previous section as nested loops, will now be run in parallel. This straightforward translation with nested for-loops is perfectly suitable for the massive parallelism of CUDA. The outer loop is now considered as a block and each block can be designed to have 512 threads working on its content. With this thread layout a large amount of data can be processed.

An alternative approach to parallelize both loops is possible, but the amount of data has to be significantly smaller and a greater overhead is incurred leading to a slow down. The topic of work distribution between the threads remains a subject of future investigations.

The loop-firing function is now declared with `__global__`. The loops are reduced by one dimension, which now is replaced by the index of the handling thread (calculated from one-dimension of the 3-component indexes). For the minimum example, this now becomes:

```
__global__ void fire_minimum(int *min_store, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  for (int j = 0; j < N; j += 1)
   if(i != j)
    minimum(min_store[i], min_store[j]);
}
```

In the CUDA code's main body, the number of threads is initialized. Assuming we have defined an array, `min_store`, of `N` minimum constraints, and a `block_size` equal to 512, then the initialization of the worker threads and assigning them to the firing function is done as given below:

```
int num_blocks = N / block_size + (N % block_size == 0 ? 0 : 1);
fire_minimum <<< num_blocks, block_size >>> (min_store, N);
```

A CUDA kernel launch is asynchronous and returns immediately. Thus to ensure synchronization between the worker threads, `cudaThreadSynchronize()` should be called to block execution until the device has completed all preceding tasks. This would ensure that all worker threads fire a single rule synchronously, and update the needed constraints before launching another kernel round.

5 Dynamic Detection Enhancement: Floyd-Warshall

As a proof of concept, several different algorithms were investigated and translated using the proposed scheme. These algorithms were the Sieve of Eratosthenes, GCD calculation and Floyd-Warshall. Due to the limited space of this short paper, the latter one will only be presented here. It sheds light on the need for an enhancement to the initial translation scheme to allow for dynamic detection of re-firing of rules due to new constraints that have been added.

The Floyd-Warshall algorithm finds the length of the shortest paths between all pairs of vertexes in a weighted graph. An edge can be represented by a CHR constraint `edge(?int,?int,?int)`, with the first two parameters expressing a connection between two connected integer indexed nodes in a graph and the third parameter describing the weight of the edge. The Floyd-Warshall algorithm can be expressed in a single CHR rule:

```
floydw @ edge(I, K, D1), edge(K, J, D2) \ edge(I , J, D3)
      <=> D3 > D1 + D2
       | D4 is D1 + D2, edge(I, J, D4).
```

The `edge` constraints are stored in an array named `edges_store`; each one is modeled using the following structure:

```
typedef struct {
   int from, to, weight;
   bool isRemoved;
} EDGE;
```

The number of constraints in the program life cycle is equal to the number of input constraints, as the rule only overwrites an existing constraint. The `floydw` rule is transformed into the following CUDA function:

```
__device__ void floydw (EDGE &a, EDGE &b, EDGE &c) {
  if(!a.isRemoved && !b.isRemoved && !c.isRemoved
      && a.from == c.from && a.to == b.from && b.to == c.to
       && c.weight > a.weight + b.weight)
         c.distance = a.distance + b.distance;
}
```

Since the rule tries the matching of three heads, it follows that the rule firings require three nested for-loops. Similar to the previous example this gets reduced to two for-loops, and the resulting in the CUDA code is shown below:

```
__global__ void fire_floydw(EDGE *edges_store, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  for (int j = 0; j < N; j += 1)
   for (int k = 0; k < N; k += 1)
    if (k != j && k != i && j != i)
     floydw(edges_store[i], edges_store[j], edges_store[k]);
}
```

However in this example an existing constraint is overwritten and a new constraint has been introduced into the constraint store. This new constraint must be tried in a potential rule application. Thus it is not sufficient to fire the rule on every triplet combination, rather the firings must be performed exhaustively until no changes have been done.

Thus a boolean flag (`update`) is introduced which detects if a new constraint has been added. The flag is changed inside the body of the if-statement of the `floydw` function. Inside a loop, the CUDA threads are initialized and call the kernel fire function. This takes place several times until no new constraint is added to the store. A simplified CUDA code snippet for this would look like:

```
int update = 1;
while (update) {
   update = 0; ...
   fire_floydw <<< n_blocks, block_size >>> (edges_store, N);
   cudaThreadSynchronize();  ...
}
```

6 Conclusion

Constraint Handling Rules is a declarative multi-headed guarded rule-based programming language, which is parallel by nature. Graphics processing units have gained popularity nowadays, and have emerged as a cheap and powerful computation power for parallel executions; their uses have exceeded the rendering of graphics and have become desirable for various computationally expensive tasks. In this work, we described a means to model the parallel execution of CHR onto a graphics processor. Due to the limited memory of graphical units, a subset of CHR was chosen which ensures that the maximal number of CHR constraints present in the constraint store throughout the course of the program is known beforehand. The scheme translates CHR constraints to C/C++ structures, defines an array of these structures to denote the constraint store and each rule into a function that performs the firing action. The firing of rules is simulated by nested for-loops that fire rules on all combinations of constraints available in the store.

The work presented is still in progress, requiring several extensions, benchmarks and generalizations. Benchmarks to access the value of the gained speed up which the translation incurred is missing. Furthermore, an automatic CHR-to-CUDA translator that produces the output CUDA code would be greatly advantageous. Another criterion which further needs optimization is the rule firings methodology and threads work load distribution. The process used in this work was a naive one which exhaustively tries all combinations of constraint pairs, this could be altered and optimized. Benchmarks for the various options for rule firings would then be an interesting open topic to access.

References

1. Chen, H., Cheng, C., Hung, S., Lin, Z.: Integer number crunching on the cell processor. Proceedings of the 39th International Conference on Parallel Processing, 508–515 (2010).
2. Frühwirth, T.: Constraint handling rules. Cambridge University Press (2009).
3. Frühwirth, T.: Parallelizing union-find in constraint handling rules using confluence analysis. Logic Programming: 21st International Conference, 113–127. Springer (2005).
4. Katz, G., Kider, J.: All-pairs shortest-paths for large graphs on the GPU. In Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, 47–55 (2008).
5. Lam, E., Sulzmann, M.: Concurrent goal-based execution of constraint handling rules. Theory and Practice of Logic Programming, 841–879 (2010).
6. Owens, J., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., Purcell, T.: A survey of general-purpose computation on graphics hardware. Eurographics 2005, State of the Art Reports, 21–51, (2005).
7. Sulzmann, M., Lam, E.: Parallel execution of multi-set constraint rewrite rules. Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming, 20–31, ACM Press (2008).
8. Triossi, A.: Hardware execution of constraint handling rules. PhD Thesis (2011).
9. Triossi, A., Orlando, S., Raffaetá, A., Frühwirth, T.: Compiling CHR to parallel hardware. Proceedings of the 14th international ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming, ACM Press (2012, forthcoming).
10. NVIDIA Corporation: NVIDIA CUDA C Programming Guide. Version 4.3 (2012).
11. Wuille, P., Schrijvers, T., Demoen, B.: CCHR: the fastest CHR implementation. Proceedings of the 4th Workshop on Constraint Handling Rules, 123-137 (2007).

Author Index