CICLOPS 2013

Proceedings of the 13th International Colloquium on Implementation of Constraint and LOgic Programming Systems

Istanbul, Turkey

August 25, 2013

Ricardo Rocha and Christian Theil Have (Eds.)

Preface

This volume contains the proceedings of the 13th International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS 2013), held in Istanbul, Turkey during August 25, 2013. CICLOPS is a well established line of workshops, traditionally co-located with ICLP, that aims at discussing and exchanging experience on the design, implementation, and optimization of constraint and logic programming systems, and other systems based on logic as a means of expressing computations.

This year, CICLOPS received 8 paper submissions. Each submission was reviewed by at least 3 Program Committee members and, at the end, 6 papers were accepted for presentation at the workshop.

We would like to thank the ICLP organizers for their support, the EasyChair conference management system for making the life of the program chairs easier and arxiv.org for providing permanent hosting. Thanks should go also to the authors of all submitted papers for their contribution to make CICLOPS alive and to the participants for making the event a meeting point for a fruitful exchange of ideas and feedback on recent developments. Finally, we want to express our gratitude to the Program Committee members, as the symposium would not have been possible without their dedicated work.

July 2013,

Ricardo Rocha
Christian Theil Have

Workshop Coordinators

Ricardo Rocha	University of Porto, Portugal
Christian Theil Have	Roskilde University, Denmark

Program Committee

Bart Demoen	Department of Computer Science, KU Leuven, Belgium
Christian Theil Have	Roskilde University, Denmark
Daniel Diaz	University of Paris 1, France
Enrico Ponteltr	New Mexico State University, USA
Jan Wielemaker	VU University Amsterdam, Netherlands
Jose F. Morales	IMDEA, Spain
Michael Hanus	Christian-Albrechts-Universität zu Kiel, Germany
Neng-Fa Zhou	The City University of New York, USA
Nicos Angelopoulos	Netherlands Cancer Institute, The Netherlands
Paulo Moura	INESC/CRACS, Portugal
Peter Szeredi	Budapest University of Technology and Economics, Hungary
Ricardo Rocha	University of Porto, Portugal
Salvador Abreu	Universidade de Évora and CENTRIA, Portugal
Terrance Swift	New University of Lisboa, Portugal
Tom Schrijvers	Ghent University, Belgium

Additional Reviewers

Benoit Desouter	Ghent University, Belgium

Web Page

`http://akira.ruc.dk/~cth/ciclops13`

Table Of Contents

A Generic Analysis Server System for Functional Logic Programs 1
Michael Hanus and Fabian Reck

A Prolog Specification of Giant Number Arithmetic 17
Paul Tarau

Experimenting with X10 for Parallel Constraint-Based Local Search 33
Danny Munera, Daniel Diaz and Salvador Abreu

Compilation for QCSP 48
Igor Stéphan

Integrating Datalog and Constraint Solving 63
Benoit Desouter and Tom Schrijvers

Interning Ground Terms in XSB 74
David Warren

A Generic Analysis Server System for Functional Logic Programs

Michael Hanus and Fabian Reck

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
`{mh|fre}@informatik.uni-kiel.de`

Abstract. We present a system, called CASS, for the analysis of functional logic programs. The system is generic so that various kinds of analyses (e.g., groundness, non-determinism, demanded arguments) can be easily integrated. In order to analyze larger applications consisting of dozens or hundreds of modules, CASS supports a modular and incremental analysis of programs. Moreover, it can be used by different programming tools, like documentation generators, analysis environments, program optimizers, as well as Eclipse-based development environments. For this purpose, CASS can also be invoked as a server system to get a language-independent access to its functionality. CASS is completely implemented in the functional logic language Curry as a master/worker architecture to exploit parallel or distributed execution environments.

1 Introduction

Automated program analyses are useful for various purposes. For instance, compilers can benefit from their results to improve the translation of source into target programs. Analysis information can be helpful to programmers to reason about the behavior and operational properties of their programs. Moreover, this information can also be documented by program documentation tools or interactively shown to developers in dedicated programming environments. On the one hand, declarative programming languages provide interesting opportunities for analyzing programs. On the other hand, their complex or abstract execution model demands for good tool support to develop reliable programs. For example, the detection of type errors in languages with higher-order features or the detection of mode problems in the use of Prolog predicates.

This work is related to functional logic languages that combine the most important features of functional and logic programming in a single language (see [4,14] for recent surveys). In particular, these languages provide higher-order functions and demand-driven evaluation from functional programming together with logic programming features like non-deterministic search and computing with partial information (logic variables). This combination has led to new design patterns [2,5] and better abstractions for application programming. Moreover, program implementation and analysis aspects for functional as well as logic languages can be considered in a unified framework. For instance, test cases for functional programs can be generated by executing functions with logic variables as arguments [9].

Automated program analyses have been already used for functional logic programming in various situations. For instance, CurryDoc [11] is an automatic documentation tool for the functional logic language Curry that analyzes Curry programs to document various operational aspects, like the non-determinism behavior or completeness issues. CurryBrowser [12] is an interactive analysis environment that unifies various program analyses in order to reason about Curry applications. KiCS2 [7], a recent implementation of Curry that compiles into Haskell, includes an analyzer to classify higher-order and deterministic operations in order to support their efficient implementation which results in highly efficient target programs. Similar ideas are applied in the implementation of Mercury [24] which uses mode and determinism information to reorder predicate calls. Non-determinism information as well as information about definitely demanded arguments has been used to improve the efficiency of functional logic programs with flexible search strategies [13]. A recent Eclipse-based development environment for Curry [21] also supports the access to analysis information during interactive program development.

These different uses of program analyses is the motivation for the current work. We present CASS (Curry Analysis Server System) which is intended to be a central component of current and future tools for functional logic programs. CASS provides a generic interface to support the integration of various program analyses. Although the current implementation is strongly related to Curry, CASS can also be used for similar declarative programming languages, like TOY [20]. The analyses are performed on an intermediate format into which source programs can be compiled. CASS supports the analysis of larger applications by a modular and incremental analysis. The analysis results for each module are persistently stored and recomputed only if it is necessary. Since CASS is implemented in Curry, it can be directly used in tools implemented in Curry, like the documentation generator CurryDoc, the analysis environment CurryBrowser, or the Curry compiler KiCS2. CASS can also be invoked as a server system providing a text-based communication protocol in order to interact with tools implemented in other languages, like the Eclipse plug-in for Curry. CASS is implemented as a master/worker architecture, i.e., it can distribute the analysis work to different processes in order to exploit parallel or distributed execution environments.

In the next section, we review some features of Curry. Section 3 shows how various kinds of program analyses can be implemented and integrated into CASS. Some uses of CASS are presented in Section 4 before its implementation is sketched in Section 5 and evaluated in Section 6.

2 Curry and FlatCurry

In this section we review some aspects of Curry that are necessary to understand the functionality and implementation of our analysis tool. More details about Curry's computation model and a complete description of all language features can be found in [10,18].

Curry is a declarative multi-paradigm language combining in a seamless way features from functional, logic, and concurrent programming. Curry has a Haskell-like syntax[1] [22] extended by the possible inclusion of free (logic) variables in conditions and right-hand sides of defining rules. Curry also offers standard features of functional languages, like polymorphic types, modules, or monadic I/O which is identical to Haskell's I/O concept [25]. Thus, "`IO` α" denotes the type of an I/O action that returns values of type α.

A *Curry program* consists of the definition of functions and the data types on which the functions operate. Functions are defined by conditional equations with constraints in the conditions. They are evaluated lazily and can be called with partially instantiated arguments. For instance, the following program defines the types of Boolean values and polymorphic lists and functions to concatenate lists (infix operator "`++`") and to compute the last element of a list:

```
data Bool   = True | False
data List a = []   | a : List a

(++) :: [a]  →  [a]  →  [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

last xs | _ ++ [x] =:= xs  = x  where x free
```

The data type declarations define `True` and `False` as the Boolean constants and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over all types and the type "`List a`" is usually written as `[a]` for conformity with Haskell). The (optional) type declaration ("`::`") of the function `(++)` specifies that `(++)` takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type.[2]

The operational semantics of Curry [1,10] is a conservative extension of lazy functional programming (if free variables do not occur in the program or the initial goal) and (concurrent) logic programming. To describe this semantics, compile programs, or implement analyzers and similar tools, an intermediate representation of Curry programs has been shown to be useful. Programs of this intermediate language, called FlatCurry, contain a single rule for each function where the pattern matching strategy is represented by case/or expressions. The basic structure of FlatCurry is defined as follows:[3]

$$\begin{array}{ll}
P ::= D_1 \ldots D_m & e ::= x \\
 & \quad \mid\ c\, e_1 \ldots e_n \\
D ::= f\, x_1 \ldots x_n = e & \quad \mid\ f\, e_1 \ldots e_n \\
 & \quad \mid\ \mathit{case}\ e_0\ \mathit{of}\ \{\overline{p_k \to e_k}\} \\
p\ ::= c\, x_1 \ldots x_n & \quad \mid\ \mathit{fcase}\ e_0\ \mathit{of}\ \{\overline{p_k \to e_k}\} \\
 & \quad \mid\ e_1\ \mathit{or}\ e_2 \\
 & \quad \mid\ \mathit{let}\ \overline{x_k}\ \mathit{free\ in}\ e
\end{array}$$

A program P consists of a sequence of function definitions D with pairwise dif-

[1] Variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

[2] Curry uses curried function types where α->β denotes the type of all functions mapping elements of type α into elements of type β.

[3] $\overline{o_k}$ denotes a sequence of objects $o_1, \ldots, o_k$.

ferent variables in the left-hand sides. The right-hand sides are expressions e composed by variables, constructor and function calls, case expressions, disjunctions, and introduction of free (unbound) variables. A case expression has the form $(f)case\ e\ of\ \{c_1\ \overline{x_{n_1}} \to e_1, \ldots, c_k\ \overline{x_{n_k}} \to e_k\}$, where e is an expression, $c_1, \ldots, c_k$ are different constructors of the type of e, and $e_1, \ldots, e_k$ are expressions. The *pattern variables* $\overline{x_{n_i}}$ are local variables which occur only in the corresponding subexpression e_i. The difference between $case$ and $fcase$ shows up when the argument e is a free variable: $case$ suspends (which corresponds to residuation) whereas $fcase$ nondeterministically binds this variable to the pattern in a branch of the case expression (which corresponds to narrowing).

Note that it is possible to translate other functional logic languages, like TOY [20], or even Haskell into this intermediate format. Since our analysis tool is solely based on FlatCurry, it can also be used for other source languages provided that there is a translator from such languages into FlatCurry.

Mature implementations of Curry, like PAKCS [15] or KiCS2 [7], provide support for meta-programming by a library containing data types for representing FlatCurry programs and an I/O action for reading a module and translating its contents into the corresponding data term. For instance, a module of a Curry program is represented as an expression of type

```
data Prog = Prog String [String] [TypeDecl] [FuncDecl] [OpDecl]
```

where the arguments of the data constructor `Prog` are the module name, the names of all imported modules, the list of all type, function, and infix operator declarations. Furthermore, a function declaration is represented as

```
data FuncDecl = Func QName Int Visibility TypeExpr Rule
```

where the arguments are the qualified name (of type `QName`, i.e., a pair of module and function name), arity, visibility (`Public` or `Private`), type, and rule (of the form "`Rule` *arguments expr*") of the function. Finally, the data type for expressions just reflects its formal definition:[4]

```
data Expr = Var Int
          | Lit Literal
          | Comb CombType QName [Expr]
          | Case CaseType Expr [(Pattern,Expr)]
          | Or Expr Expr
          | Free [Int] Expr
data CombType = FuncCall | ConsCall
```

Thus, variables are numbered, literals (like numbers or characters) are distinguished from combinations (`Comb`) which have a first argument to distinguish constructor applications and applications of defined functions. The remaining data type declarations for representing Curry programs are similar but we omit them for brevity.

[4] We present a slightly simplified version of the actual type definitions.

3 Implementing Program Analyses

Basically, a program analysis can be considered as a mapping that associates a program element with information about some aspect of its semantics. Since most interesting semantical aspects are not computable, they are approximated by some abstract domain where each abstract value describes some set of concrete values [8]. For instance, an "overlapping rules" analysis determines whether a function is defined by a set of overlapping rules, which means that some ground call to this function can be reduced in more than one way. An example for an operation that is defined by overlapping rules is the "choice" operation

```
x ? y = x
x ? y = y
```

For this analysis one can use `Bool` as the abstract domain so that the abstract value `False` is interpreted as "defined by non-overlapping rules" and `True` is interpreted as "defined by overlapping rules". Hence, the "overlapping rules" analysis has the type

```
FuncDecl  →  Bool
```

which means that we associate a `Bool` value to each function definition. Based on the data type definitions sketched in Section 2 and some standard functions, such an analysis can be defined by looking for occurrences of `Or` in the defining expression as follows:

```
isOverlapping :: FuncDecl  →  Bool
isOverlapping (Func _ _ _ _ (Rule _ e))   = orInExpr e

orInExpr :: Expr  →  Bool
orInExpr (Var _)        = False
orInExpr (Lit _)        = False
orInExpr (Comb _ _ es) = any orInExpr es
orInExpr (Case _ e bs) = orInExpr e || any (orInExpr . snd) bs
orInExpr (Or _ _)       = True
orInExpr (Free _ e)     = orInExpr e
```

Many interesting aspects require a more sophisticated analysis where dependencies are taken into account. For instance, consider a "non-determinism" analysis with the abstract domain

```
data DetDom = Det | NonDet
```

Here, `Det` is interpreted as "the operation evaluates in a deterministic manner on ground arguments." However, `NonDet` is interpreted as "the operation *might* evaluate in different ways for given ground arguments." The apparent imprecision is due to the approximation of the analysis. For instance, if the function `f` is defined by overlapping rules and the function `g` might call `f`, then `g` is judged as non-deterministic. In order to take into account such dependencies, the non-determinism analysis requires to examine the current function as well as all directly or indirectly called functions for overlapping rules. Due to recursive function definitions, this analysis cannot be done in one shot but requires a fixpoint computation. In order to make things simple for the analysis developer, CASS supports such fixpoint computations and requires from the developer only the implementation of an operation of type

```
FuncDecl → [(QName,a)] → a
```

where "a" denotes the type of abstract values. The second argument of type `[(QName,a)]` represents the currently known analysis values for the functions *directly* used in this function declaration. Hence, the non-determinism analysis can be implemented as follows:

```
nondetFunc :: FuncDecl → [(QName,DetDom)] → DetDom
nondetFunc (Func f _ _ _ (Rule _ e)) calledFuncs =
  if orInExpr e || freeVarInExpr e ||
     any (==NonDet) (map snd calledFuncs)
  then NonDet
  else Det
```

Thus, it computes the abstract value `NonDet` if the function itself is defined by overlapping rules or contains free variables that might cause non-deterministic guessing (we omit the definition of `freeVarInExpr` since it is quite similar to `orInExpr`), or if it depends on some non-deterministic function. The actual analysis is performed by defining some start value for all functions (the "bottom" value of the abstract domain, here: `Det`) and performing a fixpoint computation for the abstract values of these functions. CASS uses a working list approach as default but also supports other methods to compute fixpoints. The termination of the fixpoint computation can be ensured by standard assumptions in abstract interpretation [8], e.g., by choosing a finite abstract domain and monotonic operations, or by widening operations.

To support the inclusion of different analyses in CASS, there are an abstract type "`Analysis a`" denoting a program analysis with abstract domain "a" and several constructor operations for various kinds of analyses. Each analysis has a name provided as a first argument to these constructors. The name is used to store the analysis information persistently and to pass specific analysis tasks to workers (see below for more details). For instance, a simple function analysis which depends only on a given function definition can be created by

```
funcAnalysis :: String → (FuncDecl → a) → Analysis a
```

where the analysis name and analysis function are provided as arguments. Thus, the overlapping analysis can be specified as

```
overlapAnalysis :: Analysis Bool
overlapAnalysis = funcAnalysis "Overlapping" isOverlapping
```

A function analysis with dependencies can be constructed by

```
dependencyFuncAnalysis ::
  String → a → (FuncDecl → [(QName,a)] → a) → Analysis a
```

Here, the second argument specifies the start value of the fixpoint analysis, i.e., the bottom element of the abstract domain. Thus, the complete non-determinism analysis can be specified as

```
nondetAnalysis :: Analysis DetDom
nondetAnalysis = dependencyFuncAnalysis "Deterministic" Det nondetFunc
```

It should be noted that this definition is sufficient to execute the analysis with CASS since the analysis system takes care of computing fixpoints, calling the analysis functions with appropriate values, analyzing imported modules, etc. Thus, the programmer can concentrate on implementing the logic of the analysis and is freed from many tedious implementation details.

Sometimes one is also interested in analyzing information about data types rather than functions. For instance, the Curry implementation KiCS2 [7] has an optimization for higher-order deterministic operations. This optimization requires some information about the higher-order status of data types, i.e., whether a term of some type might contain functional values. CASS supports such analyses by appropriate analysis constructors. A simple type analysis which depends only on a given type declaration can be specified by

```
typeAnalysis :: String → (TypeDecl → a) → Analysis a
```

A more complex type analysis depending also on information about the types used in the type declaration can be specified by

```
dependencyTypeAnalysis ::
  String → a → (TypeDecl → [(QName,a)] → a) → Analysis a
```

Similarly to a function analysis, the second argument is the start value of the fixpoint analysis and the third argument computes the abstract information about the type names used in the type declaration.

The remaining entities in a Curry program that can be analyzed are data constructors. Since their definition only contains the argument types, it may seem uninteresting to provide a useful analysis for them. However, sometimes it is interesting to analyze their context so that there is a analysis constructor of type

```
constructorAnalysis :: String → (ConsDecl → TypeDecl → a)
                    → Analysis a
```

Thus, the analysis operation of type `(ConsDecl -> TypeDecl -> a)` gets for each constructor declaration the corresponding type declaration. This information could be used to compute the sibling constructors, e.g., the sibling for the constructor `True` is `False`. The information about sibling constructors is useful to check whether a function is completely defined, i.e., contains a case distinction for all possible patterns. For instance, the operation (in FlatCurry notation)

```
not x = case x of True  → False
                  False → True
```

is completely defined whereas

```
cond x y = case x of True → y
```

is incompletely defined since it fails on `False` as the first argument. To check this property, information about sibling constructors is obviously useful. But how can we provide this information in an analysis for functions?

For this and similar purposes, CASS supports the combination of different analyses. Thus, an analysis developer can define an analysis that is based on information computed by another analysis. To make analysis combination possible, there is an abstract type "`ProgInfo a`" to represent the analysis information of type "`a`" for a given module and its imports. In order to look up analysis information about some entity, there is an operation

```
lookupProgInfo:: QName  → ProgInfo a  → Maybe a
```

One can use the analysis constructor

```
combinedFuncAnalysis :: String  → Analysis b
                   →  (ProgInfo  b  → FuncDecl  → a)  → Analysis a
```

to implement a function analysis depending on some other analysis. The second argument is some base analysis computing abstract values of type "b" and the analysis function gets, in contrast to a simple function analysis, the analysis information computed by this base analysis.

For instance, if the sibling constructor analysis is defined as

```
siblingCons :: Analysis [QName]
siblingCons = constructorAnalysis ...
```

then the pattern completeness analysis might be defined by

```
patCompAnalysis :: Analysis Bool
patCompAnalysis =
  combinedFuncAnalysis "PatComplete" siblingCons isPatComplete

isPatComplete :: ProgInfo [QName]  → FuncDecl  → Bool
isPatComplete siblinginfo fundecl = ...
```

Similarly, other kinds of analyses can be also combined with some base analysis by using the following analysis constructors:

```
combinedTypeAnalysis :: String  → Analysis b
                   →  (ProgInfo  b  → TypeDecl  → a)  → Analysis a
combinedDependencyFuncAnalysis :: String  → Analysis b  → a
    →  (ProgInfo b  → FuncDecl  → [(QName,a)]  → a)  → Analysis a
combinedDependencyTypeAnalysis :: String  → Analysis b  → a
    →  (ProgInfo b  → TypeDecl  → [(QName,a)]  → a)  → Analysis a
```

For instance, an analysis for checking whether a function is totally defined, i.e., always reducible for all ground arguments, can be based on the pattern completeness analysis. It is a dependency analysis so that it can be defined as follows (in this case, `True` is the bottom element since the abstract value `False` denotes "might not be totally defined"):

```
totalAnalysis :: Analysis Bool
totalAnalysis =
  combinedDependencyFuncAnalysis "Total" patCompAnalysis True isTotal

isTotal :: ProgInfo Bool  → FuncDecl  → [(QName,Bool)]  → Bool
isTotal pcinfo fdecl calledfuncs =
  (maybe False id (lookupProgInfo (funcName fdecl) pcinfo))
  && all snd calledfuncs
```

Hence, a function is totally defined if it is pattern complete and depends only on totally defined functions.

Further examples of combined analyses are the higher-order function analysis used in KiCS2 (see above) where the higher-order status of a function depends on the higher-order status of its argument types, and the non-determinism analysis of [6] where non-determinism effects are analyzed based on groundness information.

In order to integrate some implemented analysis in CASS, one has to register it. In principle, this could be done dynamically, but currently only a static registration is supported. For the registration, the implementation of CASS contains a constant

```
registeredAnalysis :: [RegisteredAnalysis]
```

keeping the information about all available analyses. To register a new analysis, it has to be added to this list of registered analyses and CASS has to be recompiled. Each registered analysis must provide a "show" function to map abstract values into strings to be shown to the user.[5] This allows for some flexibility in the presentation of the analysis information. For instance, showing the results of the totally definedness analysis can be implemented as follows:

```
showTotal :: Bool → String
showTotal True  = "totally defined"
showTotal False = "possibly partially defined"
```

An analysis can be registered with the auxiliary operation

```
cassAnalysis :: Analysis a → (a → String) → RegisteredAnalysis
```

For instance, we can register our analyses presented in this section by the definition

```
registeredAnalysis = [cassAnalysis overlapAnalysis  showOverlap
                     ,cassAnalysis nondetAnalysis   showDet
                     ,cassAnalysis siblingCons      showSibling
                     ,cassAnalysis patCompAnalysis  showComplete
                     ,cassAnalysis totalAnalysis    showTotal    ]
```

in the CASS implementation. After compiling CASS, they are immediately available as shown in the next section.

4 Using the Analysis System

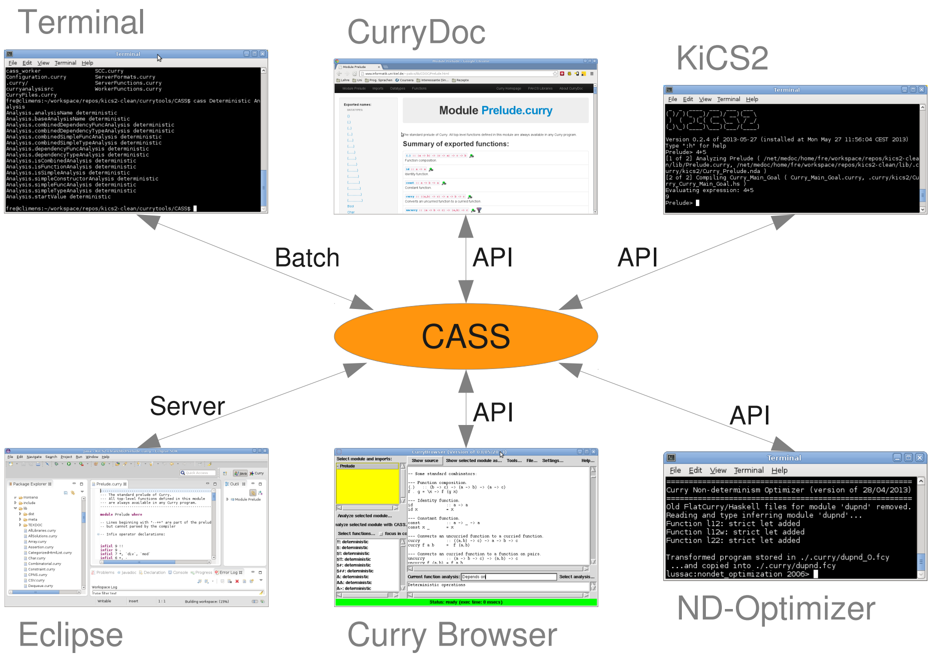
As mentioned above, a program analysis is useful for various purposes, e.g., the implementation and transformation of programs, tool and documentation support for programmers, etc. Therefore, the results computed by some analysis registered in CASS can be accessed in various ways. Currently, there are three methods for this purpose:

Batch mode: CASS is started with a module and analysis name. Then this analysis is applied to the module and the results are printed (using the analysis-specific show function, see above).

API mode: If the analysis information should be used in an application implemented in Curry, the application program could use the CASS interface operations to start an analysis and use the computed results for further processing.

Server mode: If the analysis results should be used in an application implemented in some language that does not have a direct interface to Curry, one can start CASS in a server mode. In this case, one can connect to CASS via some socket using a communication protocol that is specified in the documentation of CASS.

[5] Alternative visualizations of analysis information, e.g., as graphs, are planned for the future.



```
cass_worker                 SCC.curry
Configuration.curry         ServerFormats.curry
.curry/                     ServerFunctions.curry
curryanalysisrc             WorkerFunctions.curry
CurryFiles.curry
fre@climens:~/workspace/repos/kics2-clean/currytools/CASS$ cass Deterministic An
alysis
Analysis.analysisName deterministic
Analysis.baseAnalysisName deterministic
Analysis.combinedDependencyFuncAnalysis deterministic
Analysis.combinedDependencyTypeAnalysis deterministic
Analysis.combinedSimpleFuncAnalysis deterministic
Analysis.combinedSimpleTypeAnalysis deterministic
Analysis.dependencyFuncAnalysis deterministic
Analysis.dependencyTypeAnalysis deterministic
Analysis.isCombinedAnalysis deterministic
Analysis.isFunctionAnalysis deterministic
Analysis.isSimpleAnalysis deterministic
Analysis.simpleConstructorAnalysis deterministic
Analysis.simpleFuncAnalysis deterministic
Analysis.simpleTypeAnalysis deterministic
Analysis.startValue deterministic

fre@climens:~/workspace/repos/kics2-clean/currytools/CASS$
```

```
Version 0.2.4 of 2013-05-27 (installed at Mon May 27 11:56:04 CEST 2013)
Type ":h" for help
Prelude> 4+5
[1 of 2] Analyzing Prelude ( /net/medoc/home/fre/workspace/repos/kics2-clea
n/lib/Prelude.curry, /net/medoc/home/fre/workspace/repos/kics2-clean/lib/.c
urry/kics2/Curry_Prelude.nda )
[2 of 2] Compiling Curry_Main_Goal ( Curry_Main_Goal.curry, .curry/kics2/Cu
rry_Curry_Main_Goal.hs )
Evaluating expression: 4+5
9
Prelude>
```

```
================================================================
Curry Non-determinism Optimizer (version of 28/04/2013)
================================================================
Old FlatCurry/Haskell files for module 'dupnd' removed.
Reading and type inferring module 'dupnd'...
Function l12: strict let added
Function l12w: strict let added
Function l22: strict let added

Transformed program stored in ./.curry/dupnd_O.fcy
 ...and copied into ./.curry/dupnd.fcy
lussac:nondet_optimization 2006>
```

Fig. 1. Using CASS in different contexts

Figure 1 shows some uses of CASS which are discussed in the following. The use of CASS in batch mode is obvious. This mode is useful to get a quick access to analysis information so that one can experiment with different abstractions, fixpoint computations, etc.

If one wants to access CASS inside an application implemented in Curry, one can use some interface operation of CASS. For instance, CASS provides an operation

```
analyzeGeneric :: Analysis a  → String
                → IO (Either (ProgInfo a) String)
```

to apply an analysis (first argument) to some module (whose name is given in the second argument). The result is either the analysis information computed for this module or an error message in case of some execution error. This access to CASS is used in the documentation generator CurryDoc [11] to describe some operational aspects of functions (e.g., pattern completeness, non-determinism, solution completeness), the Curry compiler KiCS2 [7] to get information about the determinism and higher-order status of functions, and the non-determinism optimizer described in [13] to obtain information about demanded arguments and non-deterministic functions. Furthermore, there is also a similar operation

```
analyzeModule :: String  → String
               → IO (Either (ProgInfo String) String)
```

which takes an analysis name and a module name as arguments and yields the textual representation of the computed analysis results. This is used in the CurryBrowser [12] which allows the user to browse through the modules of a Curry application and apply and visualize various analyses for each module or function. Beyond some specific analyses like dependency graphs, all function analyses registered in CASS are automatically available in the CurryBrowser.

The server mode of CASS is used in a recently developed Eclipse plug-in for Curry [21] which also supports the visualization of analysis results inside Eclipse. Since this plug-in is implemented in a Java-based framework, the access to CASS is implemented via a textual protocol over a socket connection. This protocol has a command `GetAnalysis` to query the names of all available analyses. This command is used to initialize the analysis selection menus in the Eclipse plug-in. Furthermore, there are commands to analyze a complete module or individual entities inside a module. The analysis results are returned as plain strings or in XML format. Currently, we are working on more options to visualize analysis information in the Eclipse plug-in rather than strings, e.g., term or graph visualizations.

5 Implementation

As mentioned above, CASS is implemented in Curry using the features for meta-programming as sketched in Section 2. Since the analysis programmer only provides operations to analyze a function, type, or data constructor, as shown in Section 3, the main task of CASS is to supply these operations with the appropriate parameters in order to compute the analysis results.

CASS is intended to analyze larger applications consisting of many modules. Thus, a simple implementation by concatenating all modules into one large program to be analyzed would not be efficient enough. Hence, CASS performs a separate analysis of each module by the following steps:

1. The imported modules are analyzed.
2. The analysis information of the interface of the imported modules are loaded.
3. The module is analyzed. If the analysis is a dependency analysis, they are evaluated by a fixpoint computation where the specified start value is used as initial values for the locally defined (i.e., non-imported) entities.

Obviously, this scheme can be simplified in case of a simple analysis without dependencies, since such an analysis does not require the imported entities. For a combined analysis, the base analysis is performed before the main analysis is executed.

It should be noted that the separate analysis of each module allows only a bottom-up but not a top-down analysis starting with the initial goal. A bottom-up analysis is sufficient for interactive systems where the initial goal is not known at analysis time. Nevertheless, it is sometimes possible to express "top-down oriented" analyses, like a groundness analysis, in a bottom-up manner by choosing appropriate abstract domains, as shown in [6] where a type and effect system is used to analyze groundness and non-determinism information.

In order to speed up the complete analysis process, CASS implements a couple of improvements to this general analysis process sketched above. First, the analysis information for each module is persistently stored. Hence, before a module is analyzed, it is checked whether there already exists a storage with the analysis information of this module and whether the time stamp of this information is newer than the source program with all its direct or indirect imports. If the storage is found and is still valid, the stored information is used. Otherwise, the information is computed as described above and then persistently stored. This has the advantage that, if only the main module has changed and needs to be re-analyzed, the analysis time of a large application is still small.

To exploit multi-core or distributed execution environments, the implementation of CASS is designed as a master/worker architecture where a master process coordinates all analysis activities and each worker is responsible to analyze a single module. Thus, when CASS is requested to analyze some module, the master process computes all import dependencies together with a topological order of all dependencies. Therefore, the standard prelude module (without import dependencies) is the first module to be analyzed and the main module is the last one. Then the master process iterates on the following steps until all modules are analyzed:

- If there is a free worker and all imports of the first module are already analyzed, pass the first module to the free worker and delete it from the list of modules.
- If the first module contains imports that are not yet analyzed, wait for the termination of an analysis task of a worker.
- If a worker has finished the analysis of a module, mark all occurrences of this module as "analyzed."

Since contemporary Curry implementations do not support thread creation, the workers are implemented as processes that are started at the beginning and terminated at the end of the entire execution. The number of workers can be defined by some system parameter.

The current distribution of CASS[6] contains fourteen program analyses, including the analyses discussed in Section 3. Further analyses include a "solution completeness" analysis (which checks whether a function might suspend due to residuation), a "right-linearity" analysis (used to improve the implementation of functional patterns [3]), an analysis of demanded arguments (used to optimize non-deterministic computations [13]), or a combined groundness/non-determinism analysis based on a type and effect system [6]. New kinds of analyses can easily be added, since, as shown in Section 3, the infrastructure provided by CASS simplifies their definition and integration.

6 Practical Evaluation

We have already discussed some practical applications of CASS in Section 4. These applications demonstrate that the current implementation with a module-wise analysis, storing analysis information persistently, and incremental re-analysis is good enough

[6] CASS is part of the distributions of the Curry systems KiCS2 [7] and PAKCS [15].

Application:	KiCS2 REPL		CASS		CurryBrowser		ModuleDB	
Modules:	32		46		71		85	
Analysis:	Demand	Ground	Demand	Ground	Demand	Ground	Demand	Ground
1 worker:	8.09	8.25	10.25	10.30	19.53	19.36	27.97	28.15
2 workers:	5.75	5.82	6.87	7.48	12.33	12.49	18.32	18.56
4 workers:	5.41	5.47	6.17	6.47	10.20	10.38	16.98	17.15
Re-analyze:	1.40	1.38	1.26	1.26	2.01	1.99	2.34	2.34

Table 1. Using CASS in different contexts

to use CASS in practice. In order to get some ideas about the efficiency of the current implementation, we made some benchmarks and report their results in this section. Since all analyses contained in CASS have been developed and described elsewhere (see the references above), we do not evaluate their precision but only their execution efficiency.

CASS is intended to analyze larger systems. Thus, we omit the data for analyzing single modules but present the analysis times for four different Curry applications: the interactive environment (read/eval/print loop) of KiCS2, the analysis system presented in this paper, the interactive analysis environment CurryBrowser [12], and the module database, a web application generated from an entity/relationship model with the web framework Spicey [17]. In order to get an impression of the size of each application, the number of modules (including imported system modules) is shown for each application. Typically, most modules contain between 100-300 lines of code, where the largest one has more than 900 lines of code.

Table 1 contains the elapsed time (in seconds) needed to analyze these applications for different numbers of workers. We ran two kinds of fixpoint analysis: an analysis of demanded arguments [13] and a groundness analysis [6]. Each analysis has always been started from scratch, i.e., all persistently stored information were deleted at the beginning, except for the last row which shows the times to re-analyze the application where only the main module has been changed. In this case, the actual analysis time is quite small but most of the total time is spent to check all module dependencies for possible updates. The benchmarks were executed on a Linux machine running Ubuntu 12.04 with an Intel Core i5 (2.53GHz) processor where CASS was compiled with KiCS2 (Version 0.2.4).

The speedup related to the number of workers is not optimal. This might be due to the fact that the dependencies between the modules are complex so that there are not many opportunities for an independent analysis of modules, i.e., workers might have to wait for the termination of the analysis of modules which are imported by many other modules. Nevertheless, the approach shows that there is a potential to exploit the computing power offered by modern computers. Furthermore, the absolute run times are acceptable. It should also be noted that, during system development, the times are lower due to the persistent storing of analysis results.

7 Conclusions

In this paper we presented CASS, a tool to analyze functional logic programs. CASS supports various kinds of program analyses by a general notion of analysis functions that map program entities into analysis information. In order to implement an analysis that also depends on information about other entities used in a definition, CASS supports "dependency analyses" that require a fixpoint computation to yield the final analysis information. Moreover, different analyses can be combined so that one can define an analysis that is based on the results of another analysis. Using these different constructions, the analysis developer can concentrate on defining the logic of the analysis and is freed from the details to invoke the analysis on modules and complete application systems. To analyze larger applications efficiently, CASS performs a modular and incremental analysis where already computed analysis information is persistently stored. Thus, CASS does not support top-down or goal-oriented analyses but only bottom-up analyses which is acceptable for large applications or interactive systems with unknown initial goals. The implementation of CASS supports different modes of use (batch, API, server) so that the registered analyses can be accessed by various systems, like compilers, program optimizers, documentation generators, or programming environments. Currently, CASS produces output in textual form. The support for other kinds of visualizations is a topic for future work.

The analysis of programs is an important topic for all kinds of languages so that there is a vast body of literature. Most of such works is related to the development and application of various analysis methods (where some of them related to functional logic programs have already been discussed in this paper), but there are less works on the development or implementation of program analyzers. An example of such an approach, that is in some aspects similar to our work, is Hoopl [23]. Hoopl is a framework for data flow analysis and transformation. As our framework does, Hoopl eases the definition of analyses by offering high-level abstractions and releases the user from tasks like writing fixpoint computations. In contrast to our work, Hoopl works on a generic representation of data flow graphs, whereas CASS performs incremental, module-wise analyses on an already existing representation of functional logic programs. Another related system is Ciao [19], a logic programming system with advanced program analysis features to optimize and verify logic programs. CASS has similar goals but supports strongly typed analysis constructors to make the analysis construction reliable.

There are only a few approaches or tools directly related to the analysis of combined functional logic programs, as already discussed in this paper. The examples in this paper show that this combination is valuable since analysis aspects of pure functional and pure logic languages can be treated in this combined framework, like demand and higher-order aspects from functional programming and groundness and determinism aspects from logic programming. An early system in this direction is CIDER [16]. CIDER supports the analysis of single Curry modules together with some graphical tracing facilities. A successor of CIDER is CurryBrowser [12], already mentioned above, which supports the analysis and browsing of larger applications. CASS can be considered as a more efficient and more general implementation of the analysis component of CurryBrowser.

For future work, we will add further analyses in CASS with more advanced abstract domains. Since this might lead to analyses with substantial run times, the use of parallel architectures might be more relevant. Thus, it would be also interesting to develop advanced methods to analyze module dependencies in order to obtain a better distribution of analysis tasks between the workers.

Acknowledgements. The authors are grateful to Heiko Hoffmann for his contribution to an initial version of the analysis system and Sandra Dylus for her suggestions to improve the paper.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, Vol. 40, No. 1, pp. 795–829, 2005.
2. S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pp. 67–87. Springer LNCS 2441, 2002.
3. S. Antoy and M. Hanus. Declarative Programming with Function Patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pp. 6–22. Springer LNCS 3901, 2005.
4. S. Antoy and M. Hanus. Functional Logic Programming. *Communications of the ACM*, Vol. 53, No. 4, pp. 74–85, 2010.
5. S. Antoy and M. Hanus. New Functional Logic Design Patterns. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pp. 19–34. Springer LNCS 6816, 2011.
6. B. Braßel and M. Hanus. Nondeterminism Analysis of Functional Logic Programs. In *Proceedings of the International Conference on Logic Programming (ICLP 2005)*, pp. 265–279. Springer LNCS 3668, 2005.
7. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pp. 1–18. Springer LNCS 6816, 2011.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pp. 238–252, 1977.
9. S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pp. 63–74. ACM Press, 2007.
10. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
11. M. Hanus. CurryDoc: A Documentation Tool for Declarative Programs. In *Proc. 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)*, pp. 225–228. Research Report UDMI/18/2002/RR, University of Udine, 2002.
12. M. Hanus. CurryBrowser: A Generic Analysis Environment for Curry Programs. In *Proc. of the 16th Workshop on Logic-based Methods in Programming Environments (WLPE'06)*, pp. 61–74, 2006.

13. M. Hanus. Improving Lazy Non-Deterministic Computations by Demand Analysis. In *Technical Communications of the 28th International Conference on Logic Programming*, volume 17, pp. 130–143. Leibniz International Proceedings in Informatics (LIPIcs), 2012.
14. M. Hanus. Functional Logic Programming: From Theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pp. 123–168. Springer LNCS 7797, 2013.
15. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2013.
16. M. Hanus and J. Koj. CIDER: An Integrated Development Environment for Curry. In *Proc. of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, pp. 369–373. Report No. 2017, University of Kiel, 2001.
17. M. Hanus and S. Koschnicke. An ER-based Framework for Declarative Web Programming. In *Proc. of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*, pp. 201–216. Springer LNCS 5937, 2010.
18. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.
19. M. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J.F. Morales, and G. Puebla. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming*, Vol. 12, No. 1-2, pp. 219–252, 2012.
20. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
21. M. Palkus. An Eclipse-Based Integrated Development Environment for Curry. Master's thesis, Christian-Albrechts-Universität zu Kiel, 2012.
22. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
23. N. Ramsey, J. Dias, and S. Peyton Jones. Hoopl: a modular, reusable library for dataflow analysis and transformation. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell (Haskell 2010)*, pp. 121–134. ACM Press, 2010.
24. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, Vol. 29, No. 1-3, pp. 17–64, 1996.
25. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.

A Prolog Specification of Giant Number Arithmetic

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
e-mail: tarau@cs.unt.edu

Abstract. The tree based representation described in this paper, *hereditarily binary numbers*, applies recursively a run-length compression mechanism that enables computations limited by the structural complexity of their operands rather than by their bitsizes. While within constant factors from their traditional counterparts for their worst case behavior, our arithmetic operations open the doors for interesting numerical computations, impossible with traditional number representations.
We provide a complete specification of our algorithms in the form of a purely declarative Prolog program.
Keywords: *hereditary numbering systems, compressed number representations, arithmetic computations with giant numbers, tree-based numbering systems, Prolog as a specification language.*

1 Introduction

While *notations* like Knuth's "up-arrow" [1] or tetration are useful in describing very large numbers, they do not provide the ability to actually *compute* with them - as, for instance, addition or multiplication with a natural number results in a number that cannot be expressed with the notation anymore.

The novel contribution of this paper is a tree-based numbering system that *allows computations* with numbers comparable in size with Knuth's "arrow-up" notation. Moreover, these computations have a worst case complexity that is comparable with the traditional binary numbers, while their best case complexity outperforms binary numbers by an arbitrary tower of exponents factor. Simple operations like successor, multiplication by 2, exponent of 2 are practically constant time and a number of other operations of practical interest like addition, subtraction and comparison benefit from significant complexity reductions. For the curious reader, it is basically a *hereditary number system* similar to [2], based on recursively applied *run-length* compression of a special (bijective) binary digit notation.

A concept of structural complexity is also introduced, based on the size of our tree representations. It provides estimates on worst and best cases for our algorithms and it serves as an indicator of the expected performance of our arithmetic operations.

We have adopted a *literate programming* style, i.e. the code contained in the paper forms a self-contained Prolog program (tested with SWI-Prolog, Lean Prolog and Styla), also available as a separate file at `http://logic.cse.unt.edu/tarau/research/2013/hbn.pl` . We hope that this will encourage the reader to experiment interactively and validate the technical correctness of our claims.

The paper is organized as follows. Section 2 gives some background on representing bijective base-2 numbers as iterated function application and section 3 introduces hereditarily binary numbers. Section 4 describes practically constant time successor and predecessor operations on tree-represented numbers. Section 5 shows an emulation of bijective base-2 with hereditarily binary numbers and section 6 discusses some of their basic arithmetic operations. Section 7 defines a concept of structural complexity studies best and worst cases and comparisons with bitsizes. Section 8 discusses related work. Section 9 concludes the paper and discusses future work. Finally, an optimized general multiplication algorithm is described in the Appendix.

2 Bijective base-2 numbers as iterated function applications

Natural numbers can be seen as represented by iterated applications of the functions $o(x) = 2x + 1$ and $i(x) = 2x + 2$ corresponding the so called *bijective base-2* representation [3] together with the convention that 0 is represented as the empty sequence. As each $n \in \mathbb{N}$ can be seen as a unique composition of these functions we can make this precise as follows:

Definition 1 *We call bijective base-2 representation of $n \in \mathbb{N}$ the unique sequence of applications of functions o and i to ϵ that evaluates to n.*

With this representation, and denoting the empty sequence ϵ, one obtains $0 = \epsilon, 1 = o(\epsilon), 2 = i(\epsilon), 3 = o(o(\epsilon)), 4 = i(o(\epsilon)), 5 = o(i(\epsilon))$ etc. and the following holds:

$$i(x) = o(x) + 1 \tag{1}$$

2.1 Properties of the iterated functions o^n and i^n

Proposition 1 *Let f^n denote application of function f n times. Let $o(x) = 2x+1$ and $i(x) = 2x+2$, $s(x) = x+1$ and $s'(x) = x-1$. Then $k > 0 \Rightarrow s(o^n(s'(k)) = k2^n$ and $k > 1 \Rightarrow s(s(i^n(s'(s'(k)))) = k2^n$. In particular, $s(o^n(0)) = 2^n$ and $s(s(i^n(0))) = 2^{n+1}$.*

Proof. By induction. Observe that for $n = 0, k > 0, s(o^0(s'(k)) = k2^0$ because $s(s'(k))) = k$. Suppose that $P(n) : k > 0 \Rightarrow s(o^n(s'(k))) = k2^n$ holds. Then, assuming $k > 0$, P(n+1) follows, given that $s(o^{n+1}(s'(k))) = s(o^n(o(s'(k)))) = s(o^n(s'(2k))) = 2k2^n = k2^{n+1}$. Similarly, the second part of the proposition also follows by induction on n.

The underlying arithmetic identities are:

$$o^n(k) = 2^n(k+1) - 1 \tag{2}$$

$$i^n(k) = 2^n(k+2) - 2 \tag{3}$$

and in particular

$$o^n(0) = 2^n - 1 \tag{4}$$

$$i^n(0) = 2^{n+1} - 2 \tag{5}$$

3 Hereditarily binary numbers

3.1 Hereditary Number Systems

Let us observe that conventional number systems, as well as the bijective base-2 numeration system described so far, represent *blocks of contiguous 0 and 1 digits* appearing in the binary representation of a number somewhat naively - one digit for each element of the block. Alternatively, one might think that counting the blocks and representing the resulting counters as *binary numbers* would be also possible. But then, the same principle could be applied recursively. So instead of representing each block of 0 or 1 digits by as many symbols as the size of the block – essentially a *unary* representation – one could also encode the number of elements in such a block using a *binary* representation.

This brings us to the idea of hereditary number systems.

3.2 Hereditarily binary numbers as a data type

First, we define a data type for our tree represented natural numbers, that we call *hereditarily binary numbers* to emphasize that *binary* rather than *unary* encoding is recursively used in their representation.

Definition 2 *The data type $\mathbb{T}$ of the set of hereditarily binary numbers is defined inductively as the set of Prolog terms such that:*

$$X \in \mathbb{T} \text{ if and only if } X = e \textit{ or } X \text{ is of the form } v(T, Ts) \textit{ or } w(T, Ts) \tag{6}$$

where $T \in \mathbb{T}$ and Ts stands for a finite sequence (list) of elements of $\mathbb{T}$.

The intuition behind the set $\mathbb{T}$ is the following:

- The term e (empty leaf) corresponds to zero
- the term $v(T, Ts)$ counts the number $T+1$ (as counting starts at 0) of `o` applications followed by an *alternation* of similar counts of `i` and `o` applications in Ts
- the term $w(T, Ts)$ counts the number $T+1$ of `i` applications followed by an *alternation* of similar counts of `o` and `i` applications in Ts
- the same principle is applied recursively for the counters, until the empty sequence is reached

One can see this process as run-length compressed bijective base-2 numbers, represented as trees with either empty leaves or at least one branch, after applying the encoding recursively.

Definition 3 *The function $n : \mathbb{T} \to \mathbb{N}$ shown in equation* **7** *defines the unique natural number associated to a term of type $\mathbb{T}$.*

$$n(T) = \begin{cases} 0 & \text{if } T = \texttt{e}, \\ 2^{n(X)+1} - 1 & \text{if } T = \texttt{v(X,[])}, \\ (n(U)+1)2^{n(X)+1} - 1 & \text{if } T = \texttt{v(X,[Y|Xs])} \text{ and } U = \texttt{w(Y,Xs)}, \\ 2^{n(X)+2} - 2 & \text{if } T = \texttt{w(X,[])}, \\ (n(U)+2)2^{n(X)+1} - 2 & \text{if } T = \texttt{w(X,[Y|Xs])} \text{ and } U = \texttt{v(Y,Xs)}. \end{cases} \tag{7}$$

For instance, the computation of `N` in `?- n(w(v(e, []), [e, e, e]),N)` expands to $(((2^{0+1}-1+2)2^{0+1}-2+1)2^{0+1}-1+2)2^{2^{0+1}-1+1}-2 = 42$. The Prolog equivalent of equation (7) (using bitshifts for exponents of 2) is:

```
n(e,0).
n(v(X,[]),R) :-n(X,Z),R is 1<<(1+Z)-1.
n(v(X,[Y|Xs]),R):-n(X,Z),n(w(Y,Xs),K),R is (K+1)*(1<<(1+Z))-1.
n(w(X,[]),R):-n(X,Z),R is 1<<(2+Z)-2.
n(w(X,[Y|Xs]),R):-n(X,Z),n(v(Y,Xs),K),R is (K+2)*(1<<(1+Z))-2.
```

The following example illustrates the values associated with the first few natural numbers.

```
0:e, 1:v(e,[]), 2:w(e,[]), 3:v(v(e,[]),[]), 4:w(e,[e]), 5:v(e,[e])
```

Note that a term of the form `v(X,Xs)` represents an odd number $\in \mathbb{N}^+$ and a term of the form `w(X,Xs)` represents an even number $\in \mathbb{N}^+$. The following holds:

Proposition 2 *$n : \mathbb{T} \to \mathbb{N}$ is a bijection, i.e., each term canonically represents the corresponding natural number.*

Proof. It follows from the identities (2), (3) by replacing the power of 2 functions with the corresponding iterated applications of o and i.

4 Successor and predecessor

We will now specify successor and predecessor through a *reversible* Prolog predicate `s(Pred,Succ)` holding if `Succ` is the successor of `Pred`.

```
s(e,v(e,[])).
s(v(e,[]),w(e,[])).
s(v(e,[X|Xs]),w(SX,Xs)):-s(X,SX).
s(v(T,Xs),w(e,[P|Xs])):-s(P,T).
s(w(T,[]),v(ST,[])):-s(T,ST).
s(w(Z,[e]),v(Z,[e])).
s(w(Z,[e,Y|Ys]),v(Z,[SY|Ys])):-s(Y,SY).
s(w(Z,[X|Xs]),v(Z, [e,SX|Xs])):-s(SX,X).
```

It can be proved by structural induction that Peano's axioms hold and as a result $< \mathbb{T}, e, s >$ is a Peano algebra.

Note that recursive calls to `s` in `s` happen on terms that are (roughly) logarithmic in the bitsize of their operands. One can therefore assume that their complexity, computed by an *iterated logarithm*, is practically constant[1]. Note also that by using a single reversible predicate `s` for both successor and predecessor, while the solution is always unique, some backtracking occurs in the latest case. One can eliminate this by using two specialized predicates for successor and predecessor.

5 Emulating the bijective base-2 operations o, *i*

To be of any practical interest, we will need to ensure that our data type $\mathbb{T}$ emulates also binary arithmetic. We will first show that it does, and next we will show that on a number of operations like exponent of 2 or multiplication by an exponent of 2, it significantly lowers complexity.

Intuitively, the first step should be easy, as we need to express single applications or "un-applications" of `o` and `i` in terms of their iterated applications encapsulated in the terms of type $\mathbb{T}$.

First we emulate single applications of `o` and `i` seen in terms of `s`. Note that `o/2` and `i/2` are also *reversible* predicates.

```
o(e,v(e,[])).
o(w(X,Xs),v(e,[X|Xs])).
o(v(X,Xs),v(SX,Xs)):-s(X,SX).

i(e,w(e,[])).
i(v(X,Xs),w(e,[X|Xs])).
i(w(X,Xs),w(SX,Xs)):-s(X,SX).
```

Finally the "recognizers" `o_` and `i_` simply detect `v` and `w` corresponding to `o` (and respectively `i`) being the last operation applied and `s_` detects that the number is a successor, i.e., not the empty term `e`.

[1] Empirically, when computing the successor on the first $2^{30} = 1073741824$ natural numbers (with a deterministic, functional equivalent of our reversible `s` and its inverse), there are in total 2381889348 calls to `s`, averaging to 2.2183 per successor and predecessor computation. The same average for 100 successor computations on 5000 bit random numbers also oscillates around 2.21.

```
s_(v(_,_)).    s_(w(_,_)).

o_(v(_,_)).    i_(w(_,_)).
```

Note that each of the predicates `o` and `i` calls `s` and on a term that is (on the average) logarithmically smaller. As a worst case, the following holds:

Proposition 3 *The costs of* `o` *and* `i` *are within a constant factor from the cost of* `s`*.*

Definition 4 *The function $t : \mathbb{N} \rightarrow \mathbb{T}$ defines the unique tree of type $\mathbb{T}$ associated to a natural number as follows:*

$$t(x) = \begin{cases} \mathtt{e} & \text{if } x = \mathtt{0}, \\ \mathtt{o}(t(\frac{x-1}{2})) & \text{if } x > 0 \text{ and } x \text{ is odd}, \\ \mathtt{i}(t(\frac{x}{2} - 1)) & \text{if } x > 0 \text{ and } x \text{ is even} \end{cases} \tag{8}$$

We can now define the corresponding Prolog predicate that converts from terms of type $\mathbb{T}$ to natural numbers. Note that we use bitshifts (`>>`) for division by 2.

```
t(0,e).
t(X,R):-X>0, X mod 2=:=1,Y is (X-1)>>1, t(Y,A),o(A,R).
t(X,R):-X>0, X mod 2=:=0,Y is (X>>1)-1, t(Y,A),i(A,R).
```

The following holds:

Proposition 4 *Let* `id` *denote $\lambda x.x$ and "$\circ$" function composition. Then, on their respective domains*

$$t \circ n = id, \quad n \circ t = id \tag{9}$$

Proof. By induction, using the arithmetic formulas defining the two functions.

Note also that the cost of t is proportional to the bitsize of its input and the cost of n is proportional to the bitsize of its output.

6 Arithmetic operations

6.1 A few low complexity operations

Doubling a number `db` and reversing the `db` operation (`hf`) are quite simple, once one remembers that the arithmetic equivalent of function `o` is $\lambda x.2x + 1$.

```
db(X,Db):-o(X,OX),s(Db,OX).
hf(Db,X):-s(Db,OX),o(X,OX).
```

Note that efficient implementations follow directly from our number theoretic observations in section 2. For instance, as a consequence of proposition 1, the operation `exp2` computing an exponent of 2 , has the following simple definition in terms of `s`.

```
exp2(e,v(e,[])).
exp2(X,R):-s(PX,X),s(v(PX,[]),R).
```

Proposition 5 *The costs of* `db, hf` *and* `exp2` *are within a constant factor from the cost of* `s`*.*

Proof. It follows by observing that at most 2 calls to `s, o` are made in each.

6.2 Reduced complexity addition and subtraction

We now derive efficient addition and subtraction operations similar to the successor/predecessor `s`, that *work on one run-length encoded bloc at a time*, rather than by individual `o` and `i` steps.

We first define the predicates `otimes` corresponding to $o^n(k)$ and `itimes` corresponding to $i^n(k)$.

```
otimes(e,Y,Y).
otimes(N,e,v(PN,[])):-s(PN,N).
otimes(N,v(Y,Ys),v(S,Ys)):-add(N,Y,S).
otimes(N,w(Y,Ys),v(PN,[Y|Ys])):-s(PN,N).
```

```
itimes(e,Y,Y).
itimes(N,e,w(PN,[])):- s(PN,N).
itimes(N,w(Y,Ys),w(S,Ys)):-add(N,Y,S).
itimes(N,v(Y,Ys),w(PN,[Y|Ys])):-s(PN,N).
```

They are part of a chain of *mutually recursive predicates* as they are already referring to the `add` predicate, to be implemented later. Note also that instead of naively iterating, they implement a more efficient "one bloc at a time" algorithm. For instance, when detecting that its argument counts a number of applications of `o`, `otimes` just increments that count. On the other hand, when the last predicate applied was `i`, `otimes` simply inserts a new count for `o` operations. A similar process corresponds to `itimes`. As a result, performance is (roughly) logarithmic rather than linear in terms of the bitsize of argument `N`. We will use this property for implementing a low complexity multiplication by exponent of 2 operation.

We also need a number of arithmetic identities on $\mathbb{N}$ involving iterated applications of o and i.

Proposition 6 *The following hold:*

$$o^k(x) + o^k(y) = i^k(x + y) \tag{10}$$

$$o^k(x) + i^k(y) = i^k(x) + o^k(y) = i^k(x + y + 1) - 1 \tag{11}$$

$$i^k(x) + i^k(y) = i^k(x + y + 2) - 2 \tag{12}$$

Proof. By (2) and (3), we substitute the 2^k-based equivalents of o^k and i^k, then observe that the same reduced forms appear on both sides.

The corresponding Prolog code is:

```
oplus(K,X,Y,R):-add(X,Y,S),itimes(K,S,R).

oiplus(K,X,Y,R):-add(X,Y,S),s(S,S1),itimes(K,S1,T),s(R,T).

iplus(K,X,Y,R):-add(X,Y,S),s(S,S1),s(S1,S2),itimes(K,S2,T),s(P,T),s(R,P).
```

Note that the code uses `add` that we will define later and that it is part of a chain of mutually recursive predicate calls, that together will provide an intricate but efficient implementation of the intuitively simple idea: *we want to work on one run-length encoded block at a time.*

The corresponding identities for subtraction are:

Proposition 7

$$x > y \;\Rightarrow\; o^k(x) - o^k(y) = o^k(x-y-1)+1 \tag{13}$$

$$x > y+1 \;\Rightarrow\; o^k(x) - i^k(y) = o^k(x-y-2)+2 \tag{14}$$

$$x \geq y \;\Rightarrow\; i^k(x) - o^k(y) = o^k(x-y) \tag{15}$$

$$x > y \;\Rightarrow\; i^k(x) - i^k(y) = o^k(x-y-1)+1 \tag{16}$$

Proof. By (2) and (3), we substitute the 2^k-based equivalents of o^k and i^k, then observe that the same reduced forms appear on both sides. Note that special cases are handled separately to ensure that subtraction is defined.

The Prolog code, also covering the special cases, is:

```
ominus(_,X,X,e).
ominus(K,X,Y,R):-sub(X,Y,S1),s(S2,S1),otimes(K,S2,S3),s(S3,R).
```

```
iminus(_,X,X,e).
iminus(K,X,Y,R):-sub(X,Y,S1),s(S2,S1),otimes(K,S2,S3),s(S3,R).
```

```
oiminus(_,X,Y,v(e,[])):-s(Y,X).
oiminus(K,X,Y,R):-s(Y,SY),s(SY,X),exp2(K,P),s(P,R).
oiminus(K,X,Y,R):-
  sub(X,Y,S1),s(S2,S1),s(S3,S2),s_(S3), % S3 <> e
  otimes(K,S3,S4),s(S4,S5),s(S5,R).
```

```
iominus(K,X,Y,R):-sub(X,Y,S),otimes(K,S,R).
```

Note the use of `sub`, to be defined later, which is also part of the mutually recursive chain of operations.

The next two predicates extract the iterated applications of o^n and respectively i^n from `v` and `w` terms:

```
osplit(v(X,[]), X,e).
osplit(v(X,[Y|Xs]),X,w(Y,Xs)).
```

```
isplit(w(X,[]), X,e).
isplit(w(X,[Y|Xs]),X,v(Y,Xs)).
```

We are now ready for defining addition. The base cases are:

```
add(e,Y,Y).
add(X,e,X):-s_(X).
```

In the case when both terms represent odd numbers, we apply with `auxAdd1` the identity (10), after extracting the iterated applications of o as `a` and `b` with the predicate `osplit`.

```
add(X,Y,R):-o_(X),o_(Y),osplit(X,A,As),osplit(Y,B,Bs),cmp(A,B,R1),
  auxAdd1(R1,A,As,B,Bs,R).
```

In the case when the first term is odd and the second even, we apply with `auxAdd2` the identity (11), after extracting the iterated application of o and i as `a` and `b`.

```
add(X,Y,R):-o_(X),i_(Y),osplit(X,A,As),isplit(Y,B,Bs),cmp(A,B,R1),
  auxAdd2(R1,A,As,B,Bs,R).
```

In the case when the first term is even and the second odd, we apply with `auxAdd3` the identity (11), after extracting the iterated applications of i and o as, respectively, `a` and `b`.

```
add(X,Y,R):-i_(X),o_(Y),isplit(X,A,As),osplit(Y,B,Bs),cmp(A,B,R1),
  auxAdd3(R1,A,As,B,Bs,R).
```

In the case when both terms represent even numbers, we apply with `auxAdd4` the identity (12), after extracting the iterated application of i as `a` and `b`.

```
add(X,Y,R):-i_(X),i_(Y),isplit(X,A,As),isplit(Y,B,Bs),cmp(A,B,R1),
  auxAdd4(R1,A,As,B,Bs,R).
```

Note the presence of the comparison operation `cmp`, to be defined later, also part of our chain of mutually recursive operations. Note also that in each case we ensure that a block of the same size is extracted, depending on which of the two operands `a` or `b` is larger. Beside that, the auxiliary predicates `auxAdd1, auxAdd2, auxAdd3` and `auxAdd4` implement the equations of Prop. 6.

```
auxAdd1('=',A,As,_B,Bs,R):- s(A,SA),oplus(SA,As,Bs,R).
auxAdd1('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),otimes(S,As,R1),oplus(SB,R1,Bs,R).
auxAdd1('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),otimes(S,Bs,R1),oplus(SA,As,R1,R).
```

```
auxAdd2('=',A,As,_B,Bs,R):- s(A,SA),oiplus(SA,As,Bs,R).
auxAdd2('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),otimes(S,As,R1),oiplus(SB,R1,Bs,R).
auxAdd2('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),itimes(S,Bs,R1),oiplus(SA,As,R1,R).
```

```
auxAdd3('=',A,As,_B,Bs,R):- s(A,SA),oiplus(SA,As,Bs,R).
auxAdd3('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),itimes(S,As,R1),oiplus(SB,R1,Bs,R).
auxAdd3('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),otimes(S,Bs,R1),oiplus(SA,As,R1,R).
```

```
auxAdd4('=',A,As,_B,Bs,R):- s(A,SA),iplus(SA,As,Bs,R).
auxAdd4('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),itimes(S,As,R1),iplus(SB,R1,Bs,R).
auxAdd4('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),itimes(S,Bs,R1),iplus(SA,As,R1,R).
```

The code for the subtraction predicate `sub` is similar:

```
sub(X,e,X).
sub(X,Y,R):-o_(X),o_(Y),osplit(X,A,As),osplit(Y,B,Bs),cmp(A,B,R1),
  auxSub1(R1,A,As,B,Bs,R).
```

In the case when both terms represent odd numbers, we apply the identity (13), after extracting the iterated applications of o as `a` and `b`. For the other cases, we use, respectively, the identities 14, 15 and 16:

```
sub(X,Y,R):-o_(X),i_(Y),osplit(X,A,As),isplit(Y,B,Bs),cmp(A,B,R1),
  auxSub2(R1,A,As,B,Bs,R).
```

```
sub(X,Y,R):-i_(X),o_(Y),isplit(X,A,As),osplit(Y,B,Bs),cmp(A,B,R1),
  auxSub3(R1,A,As,B,Bs,R).
```

```
sub(X,Y,R):-i_(X),i_(Y),isplit(X,A,As),isplit(Y,B,Bs),cmp(A,B,R1),
  auxSub4(R1,A,As,B,Bs,R).
```

Note also the auxiliary predicates `auxSub1, auxSub2, auxSub3` and `auxSub4` that implement the equations of Prop. 7.

```
auxSub1('=',A,As,_B,Bs,R):- s(A,SA),ominus(SA,As,Bs,R).
auxSub1('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),otimes(S,As,R1),ominus(SB,R1,Bs,R).
auxSub1('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),otimes(S,Bs,R1),ominus(SA,As,R1,R).
```

```
auxSub2('=',A,As,_B,Bs,R):- s(A,SA),oiminus(SA,As,Bs,R).
auxSub2('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),otimes(S,As,R1),oiminus(SB,R1,Bs,R).
auxSub2('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),itimes(S,Bs,R1),oiminus(SA,As,R1,R).
```

```
auxSub3('=',A,As,_B,Bs,R):- s(A,SA),iominus(SA,As,Bs,R).
auxSub3('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),itimes(S,As,R1),iominus(SB,R1,Bs,R).
auxSub3('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),otimes(S,Bs,R1),iominus(SA,As,R1,R).
```

```
auxSub4('=',A,As,_B,Bs,R):- s(A,SA),iminus(SA,As,Bs,R).
auxSub4('>',A,As,B,Bs,R):-
  s(B,SB),sub(A,B,S),itimes(S,As,R1),iminus(SB,R1,Bs,R).
auxSub4('<',A,As,B,Bs,R):-
  s(A,SA),sub(B,A,S),itimes(S,Bs,R1),iminus(SA,As,R1,R).
```

6.3 Defining a total order: comparison

The comparison operation `cmp` provides a total order (isomorphic to that on $\mathbb{N}$) on our type $\mathbb{T}$. It relies on `bitsize` computing the number of applications of o and i that build a term in $\mathbb{T}$, which is also part of our mutually recursive predicates, to be defined later.

We first observe that only terms of the same bitsize need detailed comparison, otherwise the relation between their bitsizes is enough, *recursively*. More precisely, the following holds:

Proposition 8 *Let* `bitsize` *count the number of applications of o or i operations on a bijective base-2 number. Then* $\texttt{bitsize}(x) < \texttt{bitsize}(y) \Rightarrow x < y$.

Proof. Observe that, given their lexicographic ordering in "big digit first" form, the bitsize of bijective base-2 numbers is a non-decreasing function.

```
cmp(e,e,'=').
cmp(e,Y,('<')):-s_(Y).
cmp(X,e,('>')):-s_(X).
cmp(X,Y,R):-s_(X),s_(Y),bitsize(X,X1),bitsize(Y,Y1),
  cmp1(X1,Y1,X,Y,R).

cmp1(X1,Y1,_,_,R):- \+(X1=Y1),cmp(X1,Y1,R).
cmp1(X1,X1,X,Y,R):-reversedDual(X,RX),reversedDual(Y,RY),
  compBigFirst(RX,RY,R).
```

The predicate `compBigFirst` compares two terms known to have the same `bitsize`. It works on reversed (big digit first) variants, computed by `reversedDual` and it takes advantage of the block structure using the following proposition:

Proposition 9 *Assuming two terms of the same bitsizes, the one starting with i is larger than one starting with o.*

Proof. Observe that "big digit first" numbers are lexicographically ordered with $o < i$.

As a consequence, `cmp` only recurses when *identical* blocks head the sequence of blocks, otherwise it infers the "<" or ">" relation.

```
compBigFirst(e,e,'=').
compBigFirst(X,Y,R):- o_(X),o_(Y),
  osplit(X,A,C),osplit(Y,B,D),cmp(A,B,R1),
  fcomp1(R1,C,D,R).
```

```
compBigFirst(X,Y,R):-i_(X),i_(Y),
  isplit(X,A,C),isplit(Y,B,D),cmp(A,B,R1),
  fcomp2(R1,C,D,R).
compBigFirst(X,Y,('<')):-o_(X),i_(Y).
compBigFirst(X,Y,('>')):-i_(X),o_(Y).
```

```
fcomp1('=',C,D,R):-compBigFirst(C,D,R).
fcomp1('<',_,_,'>').
fcomp1('>',_,_,'<').
```

```
fcomp2('=',C,D,R):-compBigFirst(C,D,R).
fcomp2('<',_,_,'<').
fcomp2('>',_,_,'>').
```

The predicate `reversedDual` reverses the order of application of the o and i operations to a "biggest digit first" order. For this, it only needs to reverse the order of the alternative blocks of o^k and i^k. It uses the predicate `len` to compute with `auxRev1` and `auxRev2` the number of these blocks. Then, it infers that if the number of blocks is odd, the last block is of the same kind as the first; otherwise it is of its alternate kind (`w` for `v` and vice versa).

```
reversedDual(e,e).
reversedDual(v(X,Xs),R):-reverse([X|Xs],[Y|Ys]),len([X|Xs],L),
  auxRev1(L,Y,Ys,R).
reversedDual(w(X,Xs),R):-reverse([X|Xs],[Y|Ys]),len([X|Xs],L),
  auxRev2(L,Y,Ys,R).
```

```
auxRev1(L,Y,Ys,R):-o_(L),R=v(Y,Ys).
auxRev1(L,Y,Ys,R):-i_(L),R=w(Y,Ys).
```

```
auxRev2(L,Y,Ys,R):-o_(L),R=w(Y,Ys).
auxRev2(L,Y,Ys,R):-i_(L),R=v(Y,Ys).
```

```
len([],e).
len([_|Xs],L):- len(Xs,L1),s(L1,L).
```

6.4 Computing `bitsize`

The predicate `bitsize` computes the number of applications of the `o` and `i` operations. It works by summing up the *counts* of `o` and `i` operations composing a tree-represented natural number of type $\mathbb{T}$.

```
bitsize(e,e).
bitsize(v(X,Xs),R):-tsum([X|Xs],e,R).
bitsize(w(X,Xs),R):-tsum([X|Xs],e,R).
```

```
tsum([],S,S).
tsum([X|Xs],S1,S3):-add(S1,X,S),s(S,S2),tsum(Xs,S2,S3).
```

`Bitsize` concludes our chain of *mutually recursive* predicates. Note that it also provides an efficient implementation of the integer log_2 operation `ilog2`.

```
ilog2(X,R):-s(PX,X),bitsize(PX,R).
```

6.5 Fast multiplication by an exponent of 2

The predicate `leftshiftBy` uses Prop. 1, i.e., the fact that repeated application of the `o` operation (`otimes`) provides an efficient implementation of multiplication with an exponent of 2.

```
leftShiftBy(_,e,e).
leftShiftBy(N,K,R):-s(PK,K),otimes(N,PK,M),s(M,R).
```

The following holds:

Proposition 10 *Assuming* `s` *constant time,* `leftshiftBy` *is (roughly) logarithmic in the bitsize of its arguments.*

Proof. It follows by observing that at most one addition on data logarithmic in the bitsize of the operands is performed.

7 Structural complexity

As a measure of structural complexity we define the predicate `tsize` that counts the nodes of a tree of type $\mathbb{T}$ (except the root).

```
tsize(e,e).
tsize(v(X,Xs),R):- tsizes([X|Xs],e,R).
tsize(w(X,Xs),R):- tsizes([X|Xs],e,R).
```

```
tsizes([],S,S).
tsizes([X|Xs],S1,S4):-tsize(X,N),add(S1,N,S2),s(S2,S3),tsizes(Xs,S3,S4).
```

It corresponds to the function $c : \mathbb{T} \to \mathbb{N}$ defined by equation (17):

$$c(T) = \begin{cases} 0 & \text{if } \texttt{T} = \ \texttt{e}, \\ \sum_{Y \in \texttt{[X|Xs]}} (1 + c(Y)) & \text{if T} = \texttt{v(X,Xs)}, \\ \sum_{Y \in \texttt{[X|Xs]}} (1 + c(Y)) & \text{if T} = \texttt{w(X,Xs)}. \end{cases} \tag{17}$$

The following holds:

Proposition 11 *For all terms $T \in \mathbb{T}$,* `tsize(T)` $\leq$ `bitsize(T)`.

Proof. By induction on the structure of T, by observing that the two predicates have similar definitions and corresponding calls to `tsize` return terms assumed smaller than those of `bitsize`.

The following example illustrates their use:

```
?- t(123456,T),tsize(T,S1),n(S1,TSize),bitsize(T,S2),n(S2,BSize).
T = w(e, [w(e, [e]), e, v(e, []), e, w(e, []), w(e, [])]),
S1 = w(e, [e, e]), TSize = 12,
S2 = w(e, [w(e, [])]), BSize = 16 .
```

After defining the predicate `iterated`, that applies `K` times the predicate `F`

```
iterated(_,e,X,X).
iterated(F,K,X,R):-s(PK,K),iterated(F,PK,X,R1),call(F,R1,R).
```

we can exhibit a best case, of minimal structural complexity for its size

```
bestCase(K,Best):-iterated(wtree,K,e,Best).

wtree(X,w(X,[])).
```

and a worst case, of maximal structural complexity for its size

```
worstCase(K,Worst):-iterated(io,K,e,Worst).

io(X,Z):-o(X,Y),i(Y,Z).
```

The following examples illustrate these predicates:

```
?- t(3,T),bestCase(T,Best),n(Best,N).
T = v(v(e, []), []), Best = w(w(w(e, []), []), []), N = 65534 .

?- t(3,T),worstCase(T,Worst),n(Worst,N).
T = v(v(e, []), []), Worst = w(e, [e, e, e, e, e]), N = 84 .
```

It follows from identity (5) that the predicate `bestCase` computes the iterated exponent of 2 (tetration) and then applies the predecessor to it twice, i.e., it computes $2^{2^{\cdots^{2}}} - 2$. A simple closed formula (easy to proof by induction) can also be found for `worstCase`:

Proposition 12 *The predicate* `worstCase k` *computes the value in* $\mathbb{T}$ *corresponding to the value* $\frac{4(4^k-1)}{3} \in \mathbb{N}$.

The average space-complexity of our number representation is related to the average length of the *integer partitions of the bitsize of a number* [4]. Intuitively, the shorter the partition in alternative blocks of o and i applications, the more significant the compression is, but the exact study, given the recursive application of run-length encoding, is likely to be quite intricate.

The following example shows that computations with towers of exponents 20 and 30 levels tall become possible with our number representation.

```
?- t(20,X),bestCase(X,A),t(30,Y),bestCase(Y,B),add(A,B,C),
|       tsize(C,S),n(S,TSize),write(TSize),nl,fail.
314
```

Note that the structural complexity of the result (that we did not print out) is still quite manageable: `250`. *This opens the door to a new world where tractability of computations is not limited by the size of the operands but only by their structural complexity.*

8 Related work

Several notations for very large numbers have been invented in the past. Examples include Knuth's *arrow-up* notation [1] covering operations like the *tetration* (a notation for towers of exponents). In contrast to our tree-based natural numbers, such notations are not closed under addition and multiplication, and consequently they cannot be used as a replacement for ordinary binary or decimal numbers.

The first instance of a hereditary number system, at our best knowledge, occurs in the proof of Goodstein's theorem [2], where replacement of finite numbers on a tree's branches by the ordinal ω allows him to prove that a "hailstone sequence" visiting arbitrarily large numbers eventually turns around and terminates.

Arithmetic packages similar to our bijective base-2 view of arithmetic operations are part of libraries of proof assistants like Coq [5].

Arithmetic computations based on recursive data types like the free magma of binary trees (isomorphic to the context-free language of balanced parentheses) are described in [3], where they are seen as Gödel's `System T` types, as well as combinator application trees. In [6] a type class mechanism is used to express computations on hereditarily finite sets and hereditarily finite sequences. In [7] integer decision diagrams are introduced providing a compressed representation for sparse integers, sets and various other data types.

9 Conclusion and future work

We have shown that *computations* like addition, subtraction, exponent of 2 and bitsize can be performed with giant numbers in constant time or time proportional to their structural complexity rather than their bitsize. As *structural complexity* is bounded by bitsize, our computations are within constant time from their traditional counterparts, as also illustrated by our best and worst case complexity cases.

The fundamental theoretical challenge raised at this point is the following: *can more number-theoretically interesting operations expressed succinctly in terms of our tree-based data type? Is it possible to reduce the complexity of some other important operations, besides those found so far?*

The general multiplication algorithm in the `Appendix` shows a first step in that direction.

References

1. Knuth, D.E.: Mathematics and Computer Science: Coping with Finiteness. Science **194**(4271) (1976) 1235 –1242
2. Goodstein, R.: On the restricted ordinal theorem. Journal of Symbolic Logic (9) (1944) 33–41
3. Tarau, P., Haraburda, D.: On Computing with Types. In: Proceedings of SAC'12, ACM Symposium on Applied Computing, PL track, Riva del Garda (Trento), Italy (March 2012) 1889–1896
4. Corteel, S., Pittel, B., Savage, C.D., Wilf, H.S.: On the multiplicity of parts in a random partition. Random Struct. Algorithms **14**(2) (1999) 185–197

5. The Coq development team: The Coq proof assistant reference manual. LogiCal Project. (2012) Version 8.4.
6. Tarau, P.: Declarative modeling of finite mathematics. In: PPDP '10: Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming, New York, NY, USA, ACM (2010) 131–142
7. Vuillemin, J.: Efficient Data Structure and Algorithms for Sparse Integers, Sets and Predicates. In: Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on. (june 2009) 7 –14

Appendix

Reduced complexity general multiplication

We can devise a similar optimization as for `add` and `sub` for multiplication

Proposition 13 *The following holds:*

$$o^n(a)o^m(b) = o^{n+m}(ab + a + b) - o^n(a) - o^m(b) \tag{18}$$

Proof. By 2, we can expand and then reduce as follows: $o^n(a)o^m(b) = (2^n(a+1) - 1)(2^m(b+1) - 1) = 2^{n+m}(a+1)(b+1) - (2^n(a+1) + 2^m(b+1) + 1 = 2^{n+m}(a+1)(b+1) - 1 - (2^n(a+1) - 1 + 2^m(b+1) - 1 + 2) + 2 = o^{n+m}(ab + a + b + 1) - (o^n(a) + o^m(b)) - 2 + 2 = o^{n+m}(ab + a + b) - o^n(a) - o^m(b)$

The corresponding Prolog code starts with the obvious base cases:

```
mul(_,e,e).
mul(e,Y,e):-s_(Y).
```

When both terms represent odd numbers we apply the identity (18):

```
mul(X,Y,R):-o_(X),o_(Y),osplit(X,N,A),osplit(Y,M,B),
  add(A,B,S),mul(A,B,P),add(S,P,P1),s(N,SN),s(M,SM),
  add(SN,SM,K),otimes(K,P1,P2),sub(P2,X,R1),sub(R1,Y,R).
```

The other cases are reduced to the previous one by the identity $i = s \circ o$.

```
mul(X,Y,R):-o_(X),i_(Y),s(PY,Y),mul(X,PY,Z),add(X,Z,R).
mul(X,Y,R):-i_(X),o_(Y),s(PX,X),mul(PX,Y,Z),add(Y,Z,R).
mul(X,Y,R):-i_(X),i_(Y),
  s(PX,X),s(PY,Y),add(PX,PY,S),mul(PX,PY,P),add(S,P,R1),s(R1,R).
```

Note that when the operands are composed of large blocks of alternating o^n and i^m applications, the algorithm works (roughly) in time proportional to the number of blocks rather than the number of digits. The following example illustrates a multiplication with two "tower of exponent" terms:

```
?- t(30,X),bestCase(X,A),s(A,N),t(40,Y),bestCase(Y,B),s(B,M),
   mul(M,N,P),tsize(P,S),n(S,TSize),write(TSize),nl,fail.
668
```

The structural complexity of the result, `668` is in indicator that such computations are still tractable. Note however, that the predicate `mul` can be still optimized, by using in its last 3 clauses identities similar to (18), so that it works in all cases one block at a time and it reduces to addition / subtraction operations proportional to the number of blocks.

Experimenting with X10 for Parallel Constraint-Based Local Search

Danny Munera[1], Daniel Diaz[1], and Salvador Abreu[2]

[1] University of Paris 1-Sorbonne, France
{Danny.Munera,Daniel.Diaz}@univ-paris1.fr
[2] Universidade de Évora and CENTRIA, Portugal
spa@di.uevora.pt

Abstract. In this study, we have investigated the adequacy of the PGAS parallel language X10 to implement a Constraint-Based Local Search solver. We decided to code in this language to benefit from the ease of use and architectural independence from parallel resources which it offers. We present the implementation strategy, in search of different sources of parallelism in the context of an implementation of the Adaptive Search algorithm. We extensively discuss the algorithm and its implementation. The performance evaluation on a representative set of benchmarks shows close to linear speed-ups, in all the problems treated.

1 Introduction

Constraint Programming has been successfully used to model and solve many real-life problems in diverse areas such as planning, resource allocation, scheduling and product line modeling [18, 19]. Classically constraint satisfaction problems (CSPs) may be solved exhaustively by complete methods which are able to find all solutions, and therefore determine whether any solutions exist. However efficient these solvers may be, a significant class of problems remains out of reach because of exponential growth of search space, which must be exhaustively explored. Another approach to solving CSPs entails giving up completeness and resorting to (meta-) heuristics which will guide the process of searching for solutions to the problem. Solvers in this class make choices which limit the part of the search space which actually gets visited, enough so to make problems tractable. For instance a complete solver for the *magic squares* benchmark will fail for problems larger than 15×15 whereas a local search method will easily solve a 100×100 problem instance within the lower resource bounds. On the other hand, a local search procedure may not be able to find a solution, even when one exists.

However, it is unquestionable that the more computational resources are available, the more complex the problems that may be solved. We would therefore like to be able to tap into the forms of augmented computational power which are actually available, as conveniently as feasible. This requires taming various forms of explicitly parallel architectures.

Present-day parallel computational resources include increasingly multi-core processors, General Purpose Graphic Processing Units (GPGPUs), computer clusters and grid computing platforms. Each of these forms requires a different programming model and the use of specific software tools, the combination of which makes software development even more difficult.

The foremost software platforms used for parallel programming include POSIX Threads [1] and OpenMP [17] for shared-memory multiprocessors and multicore CPUs, MPI [22] for distributed-memory clusters or CUDA [16] and OpenCL [13] for massively parallel architectures such as GPGPUs. This diversity is a challenge from the programming language design standpoint, and a few proposals have emerged that try to simultaneously address the multiplicity of parallel computational architectures.

Several modern language designs are built around the Partitioned Global Address Space (PGAS) memory model, as is the case with X10 [21], Unified Parallel C [9] or Chapel [6]. Many of these languages propose abstractions which capture the several forms in which multiprocessors can be organized. Other, less radical, approaches consist in supplying a library of inter-process communication which relies on and uses a PGAS model.

In our quest to find a scalable and architecture-independent implementation platform for our exploration of high-performance parallel constraint-based local search methods, we decided to experiment with one of the most promising new-generation languages, X10 [21].

The remainder of this article is organized as follows: Section 2 present some necessary background in Constraint-Based Local Search. Section 3 discusses the PGAS Model and presents X10 language. The sources of parallelism in the Adaptive Search algorithm and its X10 implementation are the object of Section 4. Section 5 describes the benchmarks used in our experiments. The results and a discussion of the performance evaluation may be found in Section 6. A short conclusion ends the paper.

2 Constraint-Based Local Search

It is possible to classify the method for constraint solving in two groups: *complete* or *incomplete* methods. Complete methods always find a solution of the problem, if any exists. There are two main groups of algorithms: search and inference. Firstly, a representative complete search method is depth-first backtracking, which incrementally builds a solution exploring as far as possible along each branch of the search space, such that no constraint is violated. If from some point no value allows to extend the current partial assignment, it backtracks and a previous assignment is reconsidered. Secondly, the inference methods use constraints to reduce the number of legal values for a variable, and propagate this new domain to reduce other domain variables, finding a solution.

Incomplete methods, like Local Search, try to found a solution using limited resources, however this type of methods does not guarantee neither to find one solution nor to detect an inconsistency problem (no possible solution). Local search is suitable for optimization problems with a cost function which makes possible to evaluate the quality of a given configuration (assignment of variables to current values). It also needs a transition function which defines, for each configuration, a set of neighbors. The simplest Local Search algorithm start from a random configuration, explores the neighborhood and then selects a promising neighbor and moves there. This is a iteratively process that continues until a solution is found.

In this study, a Local Search method, the Adaptive Search algorithm is selected. The Adaptive Search [3, 4] is a generic, domain-independent constraint-based local search method. This meta-heuristic takes advantage of the CSP formulation and makes it possible to structure the problem in terms of variables and constraints and to analyze the current assignment of variables more precisely than an optimization of a global cost function e.g. the number of constraints that are not satisfied. Adaptive Search also includes an adaptive memory inspired in Tabu Search [11] in which each variable leading to a local minimum is marked and cannot be chosen for the next few iterations. A local minimum is a configuration for which none of the neighbors improve the current configuration. The input of the Adaptive Search algorithm is a CSP, for each constraint an error function is defined. This function is a heuristic value to represent the degree of satisfaction of a constraint and gives an indication on how much the constraint is violated.

Adaptive Search is based on iterative repair from the variables and constraint error information, trying to reduce the error in the worse variable. The basic idea is to calculate the error function for each constraint, and then combine for each variable the errors of all constraints in which it appears, thus projecting constraint errors on involved variables. Then, the algorithm chooses the variable with the maximum error as a “culprit” and selects it to modify later its value.

The purpose is to select the best neighbor move for the culprit variable, this is done by considering all possible changes in the value of this variable (neighbors) and selecting the lower value of the overall cost function. Finally, the algorithm also includes partial resets in order to escape stagnation around local minima; and it is possible to restart from scratch when the number of iterations becomes too large. Algorithm 1 presents a particular implementation of the Adaptive Search algorithm dedicated to permutation problems. In this case all N variables have the same initial domain of size N and are subject to an implicit *all-different* constraint.

3 PGAS model and X10

The current arrangement of tools to exploit parallelism in machines are strongly linked to the platform used. As it was said above, two broad programming models stand out in this matter: *distributed* and *shared memory* models. For large distributed memory systems, like clusters and grid computing, Message Passing Interface (MPI) [22] is a de-facto programming standard. The key idea in MPI is to decompose the computation over a collection of processes with private memory space. This processes can communicate with each other through message passing, generally over a communication network.

With the recent growth of many-core architectures, the shared memory approach have increased its popularity. This model decomposes the computation in multiple threads of execution sharing a common address space, communicating with each other by reading and writing shared variables. Actually, this is the model used by traditional programming tools like Fortran or C through libraries like *pthreads* [1] or OpenMP [17].

The PGAS model tries to combine the advantages of the two approaches mentioned so far. This model extends shared memory to a distributed memory setting. The execution model allows having multiple processes (like MPI), multiple threads in a process (like OpenMP), or a combination (see Figure 1). Ideally, the user would be allowed to decide how tasks get mapped to physical resources. X10 [21], Unified Parallel C [24] and Chapel [6] are examples of PGAS-enabled languages, but there exist also PGAS-based IPC libraries such as GPI [15], for use in traditional programming languages. For the experiments described herein, we used the X10 language.

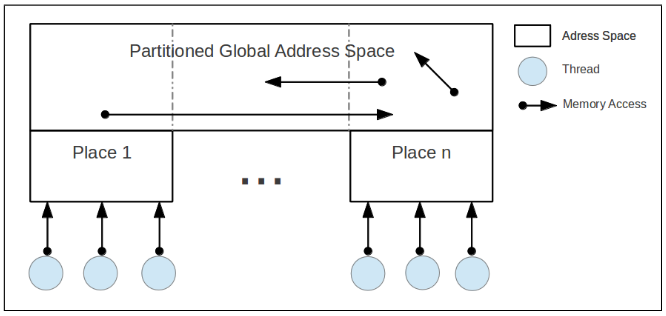


Fig. 1. PGAS Model

3.1 X10 in a Nutshell

X10 [21] is a general-purpose language developed by IBM, which provides a PGAS variation: Asynchronous PGAS (APGAS). APGAS extends the PGAS model making it flexible, even in non-HPC platforms [20]. Through this model X10 can support different levels of concurrency with simple language constructs.

There are two main abstractions in X10 model: *places* and *activities*. A *place* is the abstraction of a virtual shared-memory process, it has a coherent portion of the address space together with threads (activities) that operate on that memory. The X10 construct for creating a place in X10 is *at*, and is commonly used to create a place for each processing unit in the platform. An *activity* is the mechanism to abstract the single threads that perform computation within a place. Multiple activities may be active simultaneously in a place.

With these two components of X10 implement the main concepts of the PGAS model. Howevere, the language includes other interesting tools with the goal of improving the abstraction level of the language. Synchronization is supported thanks to various operations such as *finish*, *atomic* and *clock*. The operation *finish* is used to wait the termination of a set of activities, it behaves like a traditional barrier. The constructs *atomic* ensures an exclusive access to a critical portion of code. Finally, the construct *clock* is the standard way to ensure the synchronization between activities or places. X10 supports the distributed array construct, which makes it possible to divide an array into sub-arrays which are mapped to available places. Doing this ensures a local access from each place to the related assigned sub-array. A detailed examination of X10, including tutorial, language specification and examples can be consulted at http://x10-lang.org.

4 X10 Adaptive Search Parallel Implementation

In order to get advantage of the parallelism it is necessary to identify the sources of parallelism of the algorithm. In [5], the authors survey the state-of-the-art of the main parallel meta-heuristic strategies and discuss general design and implementation principles. This study raises a number of important issues in the taxonomy of parallel sources that lies in meta-heuristics. They classify the decomposition of activities for parallel work in two main groups: *functional parallelism* and *data parallelism.*

On the one hand, in *functional parallelism* generally different tasks work in parallel on the same data, allocated in different compute instances. On the other hand, *data parallelism* refers to the methods in which the problem domain or the associated search space is decomposed. A particular solution methodology is used to address the problem on each of the resulting components of the search space. Based on this study, we explored the parallelism in the Adaptive Search method in both *functional parallelism* and *data parallelism* and this article mostly reports on the latter, because this is where we expect the most significant gains to show.

4.1 Adaptive Search X10 sequential implementation

The first stage of the X10 implementation was to develop a sequential algorithm, which is described as algorithm 1.

Figure 2 shows the class diagram of the basic X10 project. The class *ASPermutSolver* contains the Adaptive Search permutation specialized method implementation. This class inherits the basic functionality from a general implementation of the Adaptive Search solver (in class *AdaptiveSearchSolver*), which in turn inherits a very simple Local Search method implementation from the class *LocalSearchSolver*. This class is then specialized for different parallel approaches, which we experimented with.[3]

[3] We experimented with two versions of Functional Parallism (FP1 and FP2) and a Random Walk version (RW.)

Algorithm 1 Adaptive Search Base Algorithm

Input: problem given in CSP format:

- set of variables $V = \{X_1, X_2 \cdots\}$ with their domains
- set of constraints C_j with error functions
- function to project constraint errors on vars (positive) cost function to minimize
- T: Tabu tenure (number of iterations a variable is frozen on local minima)
- RL: number of frozen variables triggering a reset
- MI: maximal number of iterations before restart
- MR: maximal number of restarts

Output: a solution if the CSP is satisfied or a quasi-solution of minimal cost otherwise.

```
1:  $Restart \leftarrow 0$
2:  repeat
3:      $Restart \leftarrow Restart + 1$
4:      $Iteration \leftarrow 0$
5:      Compute a random assignment $A$ of variables in $V$
6:      $Opt\_Sol \leftarrow A$
7:      $Opt\_Cost \leftarrow cost(A)$
8:      repeat
9:          $Iteration \leftarrow Iteration + 1$
10:         Compute errors constraints in $C$ and project on relevant variables
11:         Select variable $X$ with highest error: $MaxV$
12:                                                  ▷ not marked Tabu
13:         Select the move with best cost from $X$: $MinConflictV$
14:         if no improvement move exists then
15:             mark $X$ as Tabu for $T$ iterations
16:             if number of variables marked Tabu $\geq RL$ then
17:                 randomly reset some variables in $V$
18:                                                  ▷ and unmark those Tabu
19:             end if
20:         else
21:             swap($MaxV$,$MinConflictV$),
22:                                                  ▷ modifying the configuration $A$
23:             if $cost(A) < Opt\_Cost$ then
24:                 $Opt\_Sol \leftarrow A$
25:                 $Opt\_Cost \leftarrow costs(A)$
26:             end if
27:         end if
28:     until $Opt\_Cost = 0$ (solution found) or $Iteration \geq MI$
29: until $Opt\_Cost = 0$ (solution found) or $Restart \geq MR$
30: $output(Opt\_Sol, Opt\_Cost)$
```

Moreover, a simple CSP model is described in the class *CSPModel*, and specialized implementations of each CSP benchmark problem are contained in the classes *PartitModel, MagicSquareModel, AllIntervallModel* and *CostasModel*, which have all data structures and methods to implement the error function of each problem.

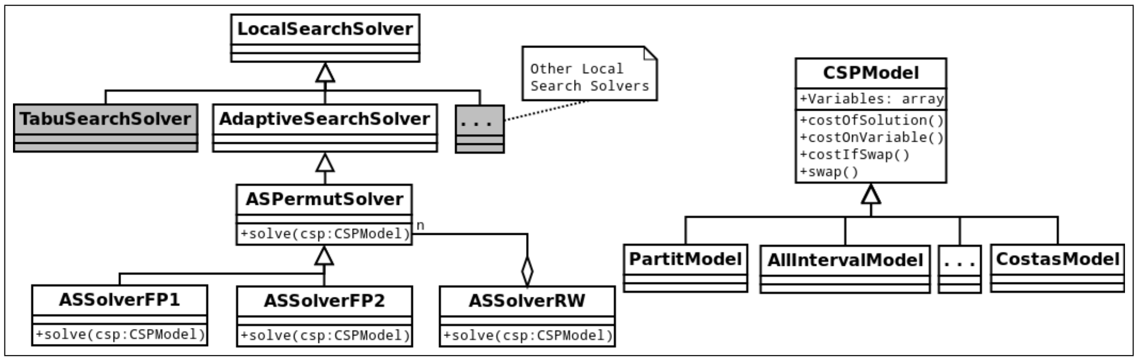


Fig. 2. X10 Class Diagram basic project

Listing 1.1 shows a simplified skeleton code of our X10 sequential implementation, based on Algorithm 1. The core of the Adaptive Search algorithm is implemented in the method solve. The *solver* method receives a *CSPModel* instance as a parameter. In line 8, the CSP variables of the model are initialized with a random permutation, in the next line the total cost of the current configuration is computed. The sentence *while* on line 10 begins the main loop of the algorithm. The *selectVarHighCost* function (Line 12) selects the variable with the maximum error and saves the result in *maxI* variable. The *selectVarMinConflict* function (Line 13) selects the best neighbor move from the high cost variable *maxI*, and saves the result index in *minJ* variable. Finally, if no local minimum is detected, the algorithm swaps the variables *maxI* and *minJ* (permutation problem) and recompute the total cost of the current configuration (Line 16). The solver function ends if the *totalCost* variable reaches the 0 value or if the algorithm reaches the maximum number of iterations.

Listing 1.1. Simplified AS X10 Sequential Implementation

```
1   class ASPermutSolver {
2     var totalCost: Int;
3     var maxI: Int;
4     var minJ: Int;
5
6     public def solve (csp: CSPModel): Int {
7       // (...local variables...)
8       csp.initialize();
9       totalCost = csp.costOfSolution();
10      while (totalCost != 0) {
11        // (...restart code...)
12        maxI = selectVarHighCost (csp);
13        minJ = selectVarMinConflict (csp);
```

```
      // (...local min tabu list, reset code...)
      csp.swapVariables (maxI, minJ);
      totalCost = csp.costOfSolution ();
    }
    return totalCost;
  }
}
```

4.2 Adaptive Search X10 Parallel Implementation

We first experimented with functional parallelism which consisted in executing the inner loop in parallel, but the results were not expressive, mostly because of thread handling overhead. We then implemented a data parallel variant, which turns out natural because the sequential Adaptive Search algorithm can be used as an isolated search instance. Furthermore, the search space is divided using different random start points (i.e configurations). This strategy is called Random Walks (RW) or Multi Search (MPSS, Multiple initial Points, Same search Strategies) [5] and has proven to be very efficient [7, 14]. The main point of our study is to explore the adequacy of a programming language based on a PGAS model: we will discuss the strengths and weaknesses of this language when applied to Independent Random Walks, without any further tuning of the parallel execution.

The implementation strategy is very different from the functional parallelism: the key of this algorithm is to have several independent and isolated instances of the Adaptive Search Solver applied to the problem model. Then it is necessary to initialize the problem variables with a random assignment of values for the variables and to distribute it to the available processing resources in the computer platform. Finally, when one instance reaches a solution, a termination detection communication strategy is used to finalize the remaining running instances. This simple parallel version has no inter-process communication, making it *Embarrassingly* or *Pleasantly Parallel*. The skeleton code of the algorithm is shown in the Listing 1.2.

Listing 1.2. Adaptive Search *data parallel* X10 implementation

```
public class ASSolverRW{
  val solDist : DistArray[ASPermutSolver];
  val cspDist : DistArray[CSPModel];
  def this( ){
    solDist=DistArray.make[ASPermutSolver](Dist.makeUnique());
    cspDist=DistArray.make[CSPModel](Dist.makeUnique());
  }
  public def solve(){
    val random = new Random();
    finish for(p in Place.places()){
      val seed = random.nextLong();
      at(p) async {
        cspDist(here.id) = new CSPModel(seed);
14            solDist(here.id) = new ASPermutSolver(seed);
15            cost = solDist(here.id).solve(cspDist(here.id));
16            if (cost==0){
17               for (k in Place.places())
18                  if (here.id != k.id)
19                     at(k) async{
20                        solDist(here.id).kill = true;
21                     }
22            }
23         }
24      }
25      return cost;
26   }
```

For this implementation the *ASSolverRW* class was created. This class has two global distributed arrays *solDist* and *cspDist* (lines 2 and 3). As explained in Section 3.1, the *DistArray* class creates an array which is spread across multiple X10 places. In this case, an instance of *ASPermutSolver* and *CSPModel* are spread over all the available places in the program. On line 10 a finish operation is executed over a for loop that goes through all the places in the program (*Place.places()*). Then, an activity is created in each place with the sentence *at(p) async* on line 12. Into the *async* block, a new instance of the solver (*new ASPermutSolver(seed)*) and the problem (*new CSPModel(seed)*) are created (lines 13 and 14) and a random seed is passed. In line 15, the solving process is executed and the returned cost is assigned to the *cost* variable. If this cost is equal to 0, the solver in a place has reached a valid solution, it is then necessary to send a termination signal to the remaining places (lines 16- 22). For this, every place (i.e. every solver), checks the value of a *kill* variable at each iteration. When it becomes equal to **true** the main loop of the solver is broken and the activity is finished. To set a *kill* remote variable from any X10 place it was necessary to create a new activity into each remaining place (sentence *at(k) async* in line 19) and into the *async* block to change the value of the *kill* variable. Line 18 with the sentence *if (here.id != k.id)* filters all the places that not are the winner place (*here*). Finally, the function returns the solution of the fastest place in line 25.

5 Benchmark description

In this study we used a set of benchmarks composed by four classical problems in constraint programming: the magic square problem, the number partitioning problem and the all-interval problem, all three taken from the CSPLib [10]; also we include the Costas Array Problem (CAP) introduced in [12], which is a very challenging real problem.

Magic Square Problem (MSP) The Magic Square Problem (`prob019` in CSPLib) consists of placing on a $N \times N$ square all the numbers in $\{1, 2, \ldots, N^2\}$

such as the sum of the numbers in all rows, columns and the two diagonal are the same. It can therefore be modeled in CSP by considering N^2 variables with initial domains $\{1, 2, \ldots, N^2\}$ together with linear equation constraints and a global *all-different* constraint stating that all variables should have a different value. The constant value that should be the sum of all rows, columns and the two diagonals can be easily computed to be $N(N^2 + 1)/2$.

All-Interval Problem (AIP) The All-Interval Problem (`prob007` in CSPLib) consists of composing a sequence of N notes such that all are different and tonal intervals between consecutive notes are also distinct. This problem is equivalent to finding a permutation of the N first integers such that the absolute difference between two consecutive pairs of numbers are all different. This amounts to finding a permutation $(X_1, \ldots, X_N)$ of $(0, \ldots, N-1)$ such that the list $(abs(X_1 - X_2), abs(X_2 - X_3), \ldots, abs(X_{N-1} - X_N))$ is a permutation of $(1, \ldots, N-1)$. A possible solution for $N = 8$ is $(3, 6, 0, 7, 2, 4, 5, 1)$ because all consecutive distances are different.

Number Partitioning Problem (NPP) The Number Partitioning Problem (`prob049` in CSPLib) consists of finding a partition of numbers $\{1, \ldots, N\}$ into two groups A and B of the same cardinality such that the sum of numbers in A is equal to the sum of numbers in B and the sum of squares of numbers in A is equal to the sum of squares of numbers in B. A solution for $N = 8$ is $A = \{1, 4, 6, 7\}$ and $B = \{2, 3, 5, 8\}$.

Costas Array Problem (CAP) The Costas Array Problem consists of filling an $N \times N$ grid with N marks such that there is exactly one mark per row and per column and the $N(N-1)/2$ vectors joining the marks are all different. It is convenient to see the Costas Array Problem as a permutation problem by considering an array of N variables $(X_1, \ldots, X_n)$ which forms a permutation of $\{1, 2, \ldots, N\}$ subject to some *all-different* constraints (see [8] for a detailed modeling). This problem has many practical applications and currently it has a whole community active working around it (http://www.costasarrays.org/).

6 Performance Analysis

This section presents and discusses our experimental results of the X10 implementation of the Adaptive Search algorithm.

We present the experimental results of the Adaptive Search algorithm in its X10 data parallel implementation. We do not present results for functional parallelism because the tests we carried out show that the granularity of individual threads is too fine to yield any performance improvement.

The testing environment used in each running was a non-uniform memory access (NUMA) computer, with 2 Intel Xeon W5580 CPUs each one with 4 hyper-threaded cores running at 3.2GHz. This system has 12 GB of main memory.

At the software level, the X10 runtime system can be deployed in two different backends: Java backend and `C++` backend; they differ in the native language used to implement the X10 program (Java or `C++`), also they present different trade-offs on different machines. Currently, the `C++` backend seems relatively mature and faster, therefore, we have chosen it for this experimentation.

Regarding the stochastic nature of the Adaptive Search behavior, several executions of the same problem were done and the times averaged. We ran 100 samples for each experimental case in the benchmark.

In this presentation, all tables use the same format: the first column identifies the problem instance, the second column is the execution time of the problem in the sequential implementation, the next group of columns contains the corresponding speed-up obtained with a varying number of cores (activities or places), and the last column presents the execution time of the problem with the highest number of cores.

Magic Square Problem. Table 1 presents the data obtained solving several large instances of MSP. Both raw times (average of 100 runs) and relative speed-ups are reported. The results show quasi-linear speed-ups (which seem independent of the size of the problem).

Problem	time (s)	speed-up with k places				time (s)
instance	seq.	2	4	6	8	8 places
40	0.47	2.12	3.20	4.14	4.42	0.11
60	1.59	1.58	2.68	3.25	3.62	0.44
80	5.10	1.84	2.92	3.72	4.19	1.22
100	11.88	1.69	2.92	3.64	4.36	2.72

Table 1. Magic Square: data parallel (timings and speed-ups)

All-Interval Problem. Table 2 shows the average time of several instances of AIP together with the speed-ups acquired with different number of places. Here again the speed-ups are practically linear up to 5.38 with 8 places. Moreover, in this case the speed-up tends to increase with the size of the problem.

Number Partitioning Problem. Table 3 shows the results of NPP. From this data, it can be seen that the corresponding speed-up obtained is almost the ideal speed-up for each number of places used. Also, the speed-up increases linearly with the number of cores to reach a maximum of 7.64 with 8 places. Moreover, this speed-up seems to increase with the size of the problem.

Problem instance	time (s) seq.	speed-up with k places 2	4	6	8	time (s) 8 places
50	0.027	2.25	3.24	4.46	4.94	0.005
100	0.47	2.12	3.21	4.32	4.96	0.09
150	2.36	1.74	3.47	4.65	5.15	0.46
200	8.29	1.84	3.66	5.28	5.38	1.54

Table 2. All-interval: data parallel (timings and speed-ups)

Problem instance	time (s) seq.	speed-up with k places 2	4	6	8	time (s) 8 places
1400	1.07	1.50	2.60	4.49	5.00	0.21
1600	1.94	1.78	3.59	5.10	6.97	0.28
1800	2.30	1.62	3.48	3.98	5.49	0.42
2000	4.31	2.34	4.87	7.06	7.64	0.56

Table 3. Partition: data parallel (timings and speed-ups)

Costas Array Problem. Table 4 presents the average time for several instances of the Costas Array Problem together with the speed-up obtained when using different numbers of places. The data confirm the trends above observed. Note that the best speed-up (9.56) is super-linear and is obtained for the most difficult instance of CAP (size of 19): the execution time is drastically reduced from 103.98 seconds to only 10.87 seconds on 8 places.

Problem instance	time (s) seq.	speed-up with k places 2	4	6	8	time (s) 8 places
16	0.26	1.79	3.52	4.00	7.09	0.04
17	1.94	1.96	4.46	6.47	9.84	0.19
18	10.90	1.68	4.05	5.09	6.88	1.59
19	103.98	2.13	4.63	5.81	9.56	10.87

Table 4. Costas Array: data parallel (timings and speed-ups)

Figure 3 shows the speed-ups reached on the most difficult instance of each problem. It can be seen that the speed-up increases almost linearly with the number of places used in the X10 program.

The performance evaluation developed in this work shows that a parallel Local Search solver implemented in X10 has good performance[4] using a data parallel strategy. The study has identified constant behavior of the speed-up with relation to the size of the problem (for some problems the speed-up improves

[4] Within a bound of 4-5 w.r.t. our sequential C implementation.

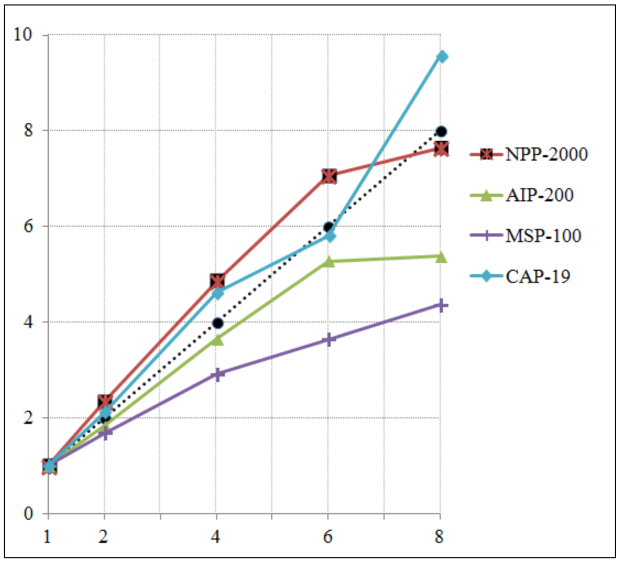


Fig. 3. Speed-ups for the most difficult instance of each problem

with the size of the problem). The resulting average runtime and the speed-ups obtained in the entire experimental test performed are as good as reported in the literature when using other IPC frameworks such as MPI [7, 8, 2] and seems to lie within the predictable bounds proposed by [23].

7 Conclusion and Future Work

We presented a parallel X10 implementation of an effective Local Search algorithm, Adaptive Search. We first experimented with functional parallelism, i.e. trying to divide the inner loop of the algorithm into various concurrent tasks. As expected, this yielded no speed-up, mainly because of the bookkeeping overhead (creation, scheduling and synchronization) that are too fine-grained.

We then proceeded with a data parallel implementation, in which the search space is decomposed in possible different random initial configurations of the problem and getting isolated solver instances to work on each point concurrently. We got a good level of performance for the X10 data-parallel implementation, reaching a maximum speed-up of 9.84 with 8 *places* for the Costas Array Problem. Linear (or close) speed-ups have been recorded in all problems we studied and they remain constant (or increasing) wrt the size of the problem.

X10 has proved a suitable platform to exploit parallelism in different ways for constraint-based local search solvers, ranging from single shared memory interprocess parallelism to more external distributed memory programming model. Additionally, the use of the X10 implicit communication mechanisms shows that X10 enables one to abstract the complexity of the parallelism with a very simple model, e.g. the distributed arrays and the termination detection system in our data parallel implementation.

Future work will focus on the implementation of a cooperative Local Search parallel solver using data parallelism. The key idea is to take advantage of all communications tools available in this APGAS model, to exchange information between different solver instances in order to obtain a more efficient and scalable solver implementation. We also plan to test the behavior of a cooperative implementation, under different HPC architectures, such as the many-core Xeon PHI, GPGPU accelerators and grid computing platforms like Grid5000.

References

1. David Butenhof. *Programming With Posix Threads.* Addison-Wesley Professional, 1997.
2. Yves Caniou, Philippe Codognet, Daniel Diaz, and Salvador Abreu. Experiments in parallel constraint-based local search. In Peter Merz and Jin-Kao Hao, editors, *Evolutionary Computation in Combinatorial Optimization - 11th European Conference, EvoCOP 2011, Torino, Italy, April 27-29, 2011. Proceedings*, volume 6622 of *Lecture Notes in Computer Science*, pages 96–107. Springer, 2011.
3. Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In Kathleen Steinhöfel, editor, *Stochastic Algorithms: Foundations and Applications*, pages 342–344. Springer Berlin Heidelberg, London, 2001.
4. Philippe Codognet and Daniel Diaz. An Efficient Library for Solving CSP with Local Search. In *5th international Conference on Metaheuristics*, pages 1–6, Kyoto, Japan, 2003.
5. Teodor Gabriel Crainic and Michel Toulouse. Parallel Meta-Heuristics. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, number May, pages 497–541. Springer US, 2010.
6. Cray Inc. *Chapel Language Specification Version 0.91.* 2012.
7. Daniel Diaz, Salvador Abreu, and Philippe Codognet. Targeting the Cell Broadband Engine for constraint-based local search. *Concurrency and Computation: Practice and Experience (CCP&E)*, 24(6):647–660, 2011.
8. Daniel Diaz, Florian Richoux, Yves Caniou, Philippe Codognet, and Salvador Abreu. Parallel local search for the costas array problem. In *PCO'12,Parallel Computing and Optimization*, Shanghai, China, 2012. IEEE.
9. Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *Wiley: UPC: Distributed Shared Memory Programming - Tarek El-.* Wiley, 2005.
10. I.P. Gent and T. Walsh. "CSPLib: a benchmark library for constraints. Technical report, 1999.
11. Fred Glover and Manuel Laguna. *Tabu Search.* Kluwer Academic Publishers, July 1997.
12. Serdar Kadioglu and Meinolf Sellmann. Dialectic Search. In *Principles and Practice of Constraint Programming (CP)*, volume 5732, pages 486–500, 2009.
13. Khronos OpenCL Working Group. *OpenCL Specification.* 2008.
14. Rui Machado, Salvador Abreu, and Daniel Diaz. Parallel Local Search : Experiments with a PGAS-based programming model. In *12th International Colloquium on Implementation of Constraint and Logic Programming Systems*, pages 1–17, Budapest, Hungary, 2012.

15. Rui Machado and Carsten Lojewski. The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. *Computer Science - R&D*, 23(3-4):125–132, 2009.
16. NVIDIA. CUDA C Programming Guide, 2013.
17. OpenMP. The OpenMP API specification for parallel programming.
18. Francesca Rossi, Peter Van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier Science, 2006.
19. Camille Salinesi, Raul Mazo, Olfa Djebbi, Daniel Diaz, and Alberto Lora-michiels. Constraints : the Core of Product Line Engineering. In *Conference on Research Challenges in Information Science (RCIS)*, number ii, pages 1–10, Guadeloupe, French West Indies, France, 2011.
20. Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The Asynchronous Partitioned Global Address Space Model. In *The First Workshop on Advances in Message Passing*, pages 1–8, Toronto, Canada, 2010.
21. Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification - Version 2.3. Technical report, 2012.
22. Mark Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI : The Complete Reference*. The MIT Press, 1996.
23. Charlotte Truchet, Florian Richoux, and Philippe Codognet. Prediction of parallel speed-ups for las vegas algorithms. *CoRR*, abs/1212.4287, 2012.
24. UPC Consortium, editor. *UPC Language Specifications*. 2005.

Compilation for QCSP

Igor Stéphan

LERIA, University of Angers, France
email: igor.stephan@info.univ-angers.fr

Abstract. We propose in this article a framework for compilation of quantified constraint satisfaction problems (QCSP). We establish the semantics of this formalism by an interpretation to a QCSP. We specify an algorithm to compile a QCSP embedded into a search algorithm and based on the inductive semantics of QCSP. We introduce an optimality property and demonstrate the optimality of the interpretation of the compiled QCSP.

1 Introduction

A constraint satisfaction problem (CSP) requires a value, selected from a given finite domain, to be assigned to each variable in the problem, so that all constraints relating the variables are satisfied [14, 8]. A quantified constraint satisfaction problem (QCSP) [9, 6] is an extension of a constraint satisfaction problem in which some of the variables are universally quantified (since the remaining variables are still existentially quantified). In this latter framework, variables take value in discrete domains. Universally quantified variables may be considered to represent certain kind of uncertainty: a choice of nature or an opponent. A QCSP can formalize many AI problems including planning under uncertainty and playing a game against an opponent. In this second application, the goal of the QCSP is to make a robust plan against the opponent. Whereas finding a solution of a CSP is generally NP-complete, finding a solution for a QCSP is generally PSPACE-complete [9].

Most of the recent decision procedure for QCSP [3, 5, 12, 7] are based on a search algorithm (except [18] which is based on a bottom-up approach and [13] which is based on a translation to quantified boolean formulas) and off-line procedures (except [2] which is an on-line real-time algorithm based on Monte Carlo game tree search and [17] which is based on standard game tree search techniques). Such an algorithm chooses a variable, branches on the different values of the domain, verifies if the subproblems have some solutions and combines, according to the semantics of the quantifier associated to the variable, those solutions into a solution to the problem.

Knowledge compilation is considered in many AI applications where quick on-line responses are expected. In general, a knowledge base is compiled off-line into a target language which is then used on-line to answer some queries. The goal is to have a lesser complexity for the query computation of the compiled knowledge base than for the initial knowledge base. This principle is for example applied in product configuration where the set of possible configurations is compiled [1].

As far as we know, the problem of compiling a knowledge base represented as a QCSP has not been treated but only for the related domain of quantified Boolean formulas [16, 11]. Our first contribution is a new formalism as compilation target language: the QCSP base. Our second contribution is a definition of an optimality property for QCSP bases in order to give a polytime answer to the next move choice problem [16] which raises the issue of whether one can change for another solution during the game. Our third contribution is a compilation algorithm embedded in a search algorithm which is proved to compile a QCSP in an optimal QCSP base.

This article is organized as follows: Section 2 establishes the necessary preliminaries, section 3 presents our framework and target language for the compilation of QCSP, section 4 specifies an algorithm to compile a QCSP in our target language, section 5 concludes with a discussion and some further works.

2 Preliminaries

Symbol $\exists$ stands for existential quantifier and symbol $\forall$ stands for universal quantifier. Symbol $\wedge$ stands for logical conjunction, symbol $\top$ stands for what is always true and symbol $\bot$ stands for what is always false. A QCSP is a tuple $(\mathbf{V}, \text{rank}, \text{quant}, \mathbf{D}, \mathbf{C})$: $\mathbf{V}$ is a set of n variables, rank is a bijection from $\mathbf{V}$ to $[1..n]$, quant is a mapping from $\mathbf{V}$ to $\{\exists, \forall\}$ (quant(v) is the quantifier associated to the variable v), $\mathbf{D}$ is a mapping from $\mathbf{V}$ to a set of domains $\{D(v_1), \ldots, D(v_n)\}$ where, for every variable $v_i \in \mathbf{V}$, $D(v_i)$ is the finite domain of all the possible values ($D(v)$ is the domain associated to the variable v), $\mathbf{C}$ is a set of contraints. If $v_{j_1}, \ldots, v_{j_m}$ are the variables of a constraint $c_j \in \mathcal{C}$ then the relation associated to c_j is a subset of the Cartesian product $D(v_{j_1}) \times \ldots \times D(v_{j_m})$. In what follows, we denote for every $i \in [1..n]$, $q_i = \text{quant}(v_i)$ and $D_i = D(v_i)$. A QCSP $(\mathbf{V}, \text{rank}, \text{quant}, \mathbf{D}, \mathbf{C})$ on n variables will be denoted as follows to simplify notation:

$$q_1 v_1 \ldots q_n v_n \bigwedge_{c_j \in \mathcal{C}} c_j$$

with $v_1 \in D_1, \ldots, v_n \in D_n$, $\text{rank}(v_i) = i$, for every $i \in [1..n]$; $q_1 v_1 \ldots q_n v_n$ is the binder.

The QCSP $(\{x, y, z, t\}, \text{rank}, \text{quant}, \{\{0, 1, 2\}\}, \mathcal{C})$ with

$$\begin{cases} \text{rank} = \{(1, x), (2, y), (3, z), (4, t)\}, \\ \text{quant} = \{(x, \exists), (y, \exists), (z, \forall), (t, \exists)\}, \\ D(x) = D(y) = D(z) = D(t) = \{0, 1, 2\} \text{ and} \\ \mathcal{C} = \{(x = (y * z) + t)\} \end{cases}$$

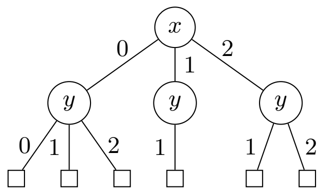
is, for example, denoted : $\exists x \exists y \forall z \exists t (x = (y * z) + t)$ with $x, y, z, t \in \{0, 1, 2\}$.

In a binder, a maximal homogeneous sequence of quantifiers forms a bloc ; the first one (and also the outermost) is the leftmost.

The set $\mathcal{T}_i(Q)$ with $v_1 \in D_1, \ldots, v_n \in D_n$, $Q = q_1 v_1 \ldots q_n v_n$ for $1 \leq i \leq n$ is the set of trees such that

- every leaf node is labeled by the symbol $\square$ and is at depth i,
- every internal node at depth k, $0 \leq k < i-1$ is labeled with the variable v_{k+1},
- every edge linking a node at depth k to one of its children's nodes is labeled with an element of D_k,
- all the labels of the edges linking a node to its children nodes are different.

The following tree is, for example, an element of the set $\mathcal{T}_2(\exists x \exists y \forall z \exists t)$ with $x, y, z, t \in \{0, 1, 2\}$:

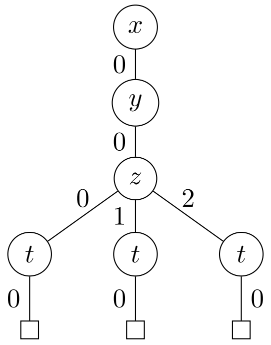


Let $(\mathbf{V}, \text{rank}, \text{quant}, \mathbf{D}, \mathbf{C})$ be a QCSP such that $\mathbf{V} = \{v_1, \ldots, v_n\}$, with $v_1 \in D_1, \ldots, v_n \in D_n$, then a scenario is the sequence of the labels $val_1, \ldots, val_n$ on the path $(v_1, val_1), \ldots, (v_n, val_n)$, $val_i \in D_i$ for every i, $1 \leq i \leq n$, of a tree of $\mathcal{T}_n(q_1 v_1 \ldots q_n v_n)$ and a strategy is a tree of $\mathcal{T}_n(q_1 v_1 \ldots q_n v_n)$ such that

- every node labeled with an existentially quantified variable has a unique child node and
- every node labeled with a universally quantified variable whose associated domain is of size k admits k children nodes.

A scenario $val_1, \ldots, val_n$ for a QCSP $(\mathbf{V}, \text{rank}, \text{quant}, \mathbf{D}, \mathbf{C})$ such that $\mathbf{V} = \{v_1, \ldots, v_n\}$ is a winning scenario if $(\bigwedge_{1 \leq i \leq n} v_i = val_i) \wedge (\bigwedge_{c_j \in \mathcal{C}} c_j)$ is true ; such a scenario corresponds to the complete instantiation $[v_1 \leftarrow val_1], \ldots, [v_n \leftarrow val_n]$; it is a winning scenario if the instantiation satisfies all the constraints. A strategy is a winning strategy if all the scenarios are winning scenarios. If there is no quantifier, the $\square$ strategy is always a winning strategy.

The scenario $0, 0, 2, 0$, which corresponds to the complete instantiation $[x \leftarrow 0]$, $[y \leftarrow 0]$, $[z \leftarrow 2]$ and $[t \leftarrow 0]$, is a winning scenario, since $0 = (0 * 2) + 0$. The following strategy is a winning strategy



for the QCSP $\exists x \exists y \forall z \exists t (x = (y * z) + t), x, y, z, t \in \{0, 1, 2\}$ since $(0 = (0 * 0) + 0)$, $(0 = (0 * 1) + 0)$ and $(0 = (0 * 2) + 0)$.

We can give a more intuitive and recursive decision semantics for QCSP as follows: A QCSP $\forall x QC$ with $x \in D$ admits a winning strategy if and only if, for every $val \in D$, $Q(C \wedge (x = val))$ admits a winning strategy and a QCSP $\exists x QC$ with $x \in D$ admits a winning strategy if and only if, for at least one $val \in D$, $Q(C \wedge (x = val))$ admits a winning strategy.

3 Base for QCSP

From a complexity point of view and under some classical assumptions, winning strategies are exponential in space in worst case w.r.t. the number of variables of the QCSP [10]. But the number of winning strategies may also be exponential in worst case. A naive way to compile a QCSP would be to store all the winning strategies in a set but this approach is intractable in practice. For example, the QCSP $\forall x \forall y \exists z \exists t (x = (y * z) + t), x, y, z, t \in \{0, 1, 2\}$ admits 324 winning strategies. Another way is to store a tree which contains only the scenarios present in the winning strategies. This approach is not very useful too from the knowledge representation point of view since there is no direct access to the possibilities of an existentially quantified variable except for those of the first bloc.

We define in this section our formalism as a target language for QCSP compilation: the *QCSP base*. We also define the semantics of QCSP bases in terms of QCSP. We introduce a property of optimality for QCSP bases and prove a very interesting result about optimal QCSP.

3.1 Definitions for QCSP bases

Intuitively, a QCSP base is a set of strategies organized according to a mechanism of guards for every existentially quantified variable and every value of the domain. Such a guard is a pair of a value and a tree which is the expression of what have already been played by both opponents.

Definition 1 (QCSP base). *A QCSP base is either*

- *the symbol bl_top*
- *the symbol bl_bottom*
- *a pair $\langle Q \mid G \rangle$ with $n > 0$, $Q = q_1 v_1 \dots q_n v_n$ and $G = [G_{e_1}, \dots, G_{e_m}]$ a list such that*
  - *$e_1, \dots, e_m$ is the set of indexes of the existentially quantified variables*[1] *;*
  - *every G_{e_k}, $1 \leq k \leq m$ is a function with non-empty graph $\{(val_1 \mapsto T_1), \dots, (val_{j_k} \mapsto T_{j_k})\}$, $val_1, \dots, val_{j_k} \in D(v_{e_k})$ and $T_1, \dots, T_{j_k} \in \mathcal{T}_{e_k}(Q)$.*

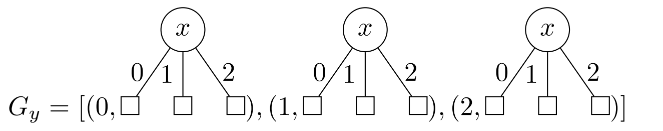
[1] $u_1, \dots, u_p$ is the set of indexes of the universally quantified variables, $\{e_1, \dots, e_m\} \cup \{u_1, \dots, u_p\} = [1..n]$, $\{e_1, \dots, e_m\} \cap \{u_1, \dots, u_p\} = \emptyset$, quant$(v_{e_i}) = \exists$, for every i, $1 \leq i \leq n$, quant$(v_{u_i}) = \forall$, for every i, $1 \leq i \leq p$.

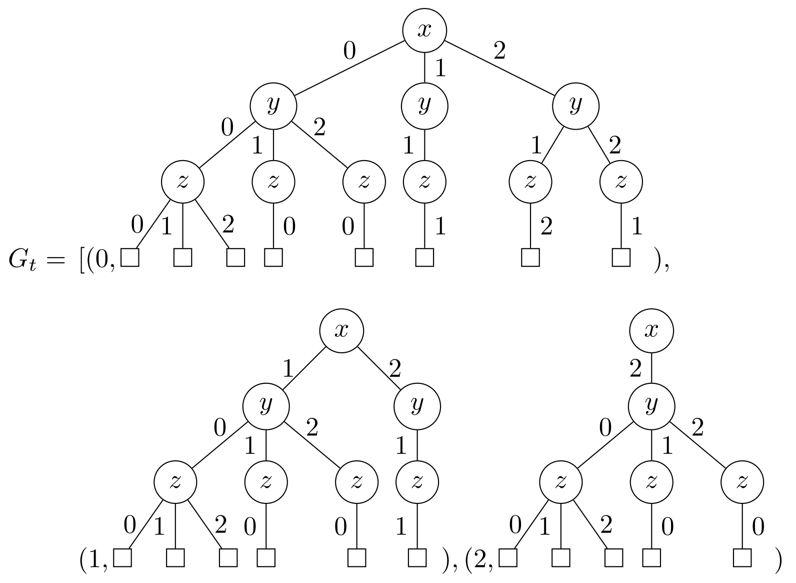
A pair $(val, T) \in G_{e_k}$ is a guard for the existentially quantified variable v_{e_k}.

In what follows, the QCSP bases *bl_top* and *bl_bottom* are semantically interpreted as respectively what is always true and what is always false and algorithmically as respectively what admits every strategy as a winning strategy and what admits no winning strategy at all.

Example 1. The following guard sets G_x, G_y and G_z are guard sets of the QCSP base $B = \langle \exists x \exists y \forall z \exists t \mid [G_x, G_y, G_t] \rangle$ with $x, y, z, t \in \{0, 1, 2\}$.

$G_x = [(0, \square), (1, \square), (2, \square)]$,





3.2 Interpretation

The semantics of a QCSP base is expressed by an interpretation to the QCSP. First of all, we interpret the trees of the guards of the QCSP bases as tuples of values.

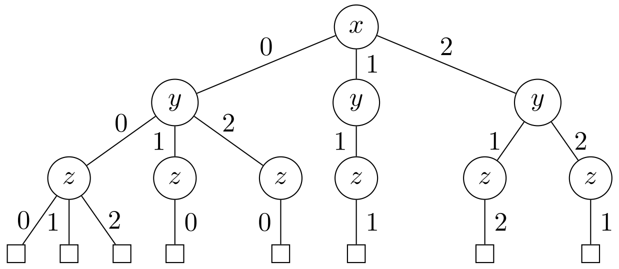
Definition 2 (interpretation of a tree). *The interpretation of a tree T of a guard (val, T) according to a value val is the set*

$$I^{val}(T) = \{(val, e_1, \ldots, e_n) | e_1 \ldots e_n \text{ a branch of a tree } T\}.$$

In particular, $I^{val}(\square) = \{val\}$. The interpretation of a set G of guards (value, tree) is by extension :

$$I(G) = \bigcup_{(val,T)\in G} I^{val}(T).$$

Example 2. (Example 1 continued.) The interpretation of the tree T extracted from the set of guards G_t :



according to the value 0 is the set

$$I^0(T) = \{(0,0,0,0),(0,0,0,1),(0,0,0,2),\\ (0,0,1,0),(0,0,2,0),(0,1,1,1),\\ (0,2,1,2),(0,2,2,1)\}.$$

and

$$I(G_t) = \{(0,0,0,0),(0,0,0,1),(0,0,0,2),\\ (0,0,1,0),(0,0,2,0),(0,1,1,1),(0,2,1,2),\\ (0,2,2,1),(1,1,0,0),(1,1,0,1),(1,1,0,2),\\ (1,1,1,0),(1,1,2,0),(1,2,1,1),(2,2,0,0),\\ (2,2,0,1),(2,2,0,2),(2,2,1,0),(2,2,2,0)\}$$

One can remark that for all $(val_t, val_x, val_y, val_z) \in I(G_t)$, the instantiation $[t \leftarrow val_t][x \leftarrow val_x][y \leftarrow val_y][z \leftarrow val_z]$ satisfies the constraint $(x = (y*z)+t)$.

We now define the interpretation of a QCSP base.

Definition 3 (interpretation of a QCSP base). *The interpretation function $(\cdot)^*$ of a QCSP base to a QCSP is defined as follows ($Q = q_1v_1 \ldots q_nv_n$) :*
$(bl\_top)^* = \top$
$(bl\_bottom)^* = \bot$
$(\langle Q \mid [G_{e_1}, \ldots, G_{e_m}]\rangle)^* =$
$Q \bigwedge_{e_i\in[e_1,\ldots,e_m]}((v_{e_k}, v_1, \ldots, v_{e_k-1}) \in I(G_{e_k}))$

The interpretation of a QCSP base is a QCSP but only on table constraints.

Example 3. (Examples 1 and 2 continued.)

$$(B)^* = \exists x\exists y\forall z\exists t((x \in I(G_x)) \wedge ((y,x) \in I(G_y)) \wedge ((t,x,y,z) \in I(G_t)))$$

with $x,y,z,t \in \{0,1,2\}$, $I(G_x) = \{0,1,2\}$ and $I(G_y) = \{0,1,2\}^2$.

3.3 Properties of QCSP bases

To a given QCSP, many different QCSP bases may be such that their interpretations have exactly the same set of winning strategies as that QCSP. We define this property as the *compatibility property.*

Definition 4 (compatibility of a QCSP base). *A QCSP base is compatible with a QCSP if its interpretation has exactly the same winning strategy.*

In what follows, we will see that the set of compatible QCSP bases may be seen as a good candidate as a target for a compilation language.

Example 4. (Examples 1, 2 and 3 continued.) The QCSP base B is compatible with the QCSP $\exists x \exists y \forall z \exists t (x = (y*z)+t)$ with $x, y, z, t \in \{0, 1, 2\}$. If, for example, the pair $(2, \square)$ of G_x is discarded then the resulting QCSP base is no more compatible with the QCSP since two of the four winning strategies are lost.

The following theorem establishes immediately the completeness of the QCSP base formalism w.r.t. QCSP.

Theorem 1 (completeness). *For every QCSP there exists a compatible base.*

When a QCSP represents a finite two-player game, one of the most important issues for the existential player, at each turn during the game, is the following: "What do I have to play to be certain to win the game?" If a winning strategy has been already computed before the game begins, the player has only to follow it. But if the uncertainty was not completely known and if the current winning strategy can not be applied anymore, the existential player has to compute again a new strategy and has to pay also the complete algorithmic price.

Definition 5 (next move choice problem). *The* next move choice problem *is defined as follows.*

- *Instance : A QCSP* $q_1 v_1 \ldots q_n v_n \bigwedge_{c \in \mathcal{C}} c$ *with* $v_1 \in D_1, \ldots, v_n \in D_n$ *and a sequence of instantiations* $[v_1 \leftarrow val_1], \ldots, [v_i \leftarrow val_i]$ *obtained from a winning strategy for a QCSP with* $quant(v_i) = \exists$ *and* $val_1 \in D_1, \ldots, val_i \in D_i$.
- *Query : Is there any winning strategy for a QCSP*

$$q_{i+1} v_{i+1} \ldots q_n v_n \bigwedge_{c \in \mathcal{C}} c \wedge (v_1 = val_1) \wedge (v_{i-1} = val_{i-1}) \wedge (v_i = val'_i)$$

with $v_{i+1} \in D_{i+1}, \ldots, v_n \in D_n$, $val'_i \in D_i$, $val'_i \neq val_i$ *?*

Clearly enough the next move choice problem is still a PSPACE-complete problem since $q_{i+1} v_{i+1} \ldots q_n v_n \bigwedge_{c \in \mathcal{C}} c \wedge (v_1 = val_1) \wedge (v_{i-1} = val_{i-1}) \wedge (v_i = val'_i)$ with $v_{i+1} \in D_{i+1}, \ldots, v_n \in D_n$, $val'_i \in D_i$, $val'_i \neq val_i$ is a QCSP.

We introduce a new property for a QCSP base which guarantees that the next move choice problem is no more PSPACE-complete but polytime w.r.t. the size of the QCSP base. A QCSP base is *optimal* if all the guards associated to the moves played by the existential player are verified then this player is sure to follow a winning strategy.

Definition 6 (optimality). *Let $B = \langle q_1 v_1 \dots q_n v_n \mid [G_{e_1}, \dots, G_{e_m}]\rangle$ be a QCSP base and $(B)^* = q_1 v_1 \dots q_n v_n C$ with $v_1 \in D_1, \dots, v_n \in D_n$. This base is optimal if the following property is verified. For every i, $i \in [1 \dots m]$, let C_i be the set of constraints $\{(v_{e_k} = val_{e_k}) | 1 \leq k < i\}$ such that $(val_{e_k}, val_{e_1}, \dots, val_{e_{k-1}}) \in I^{val_{e_k}}(a_{e_k})$, $(val_{e_k}, a_{e_k}) \in G_{e_k}$.*

Then for every guard $(val, a) \in G_{e_i}$, $(val, val_{e_1}, \dots, val_{e_{i-1}}) \in I^{val}(a)$ if and only if $q_{e_i+1} v_{e_i+1} \dots q_n v_n (C \wedge (v_{e_i} = val) \wedge \bigwedge_{c \in C_i} c)$ admits a winning strategy.

The underlying order of this notion of optimality is the number of winning scenarios which are not a branch of any winning strategy. In case of the interpretation of an optimal base this number is zero.

Example 5. The following guard sets G_x^{opt}, G_y^{opt} and G_t^{opt} are guard sets for the QCSP base $B^{opt} = \langle \exists x \exists y \forall z \exists t \mid [G_x^{opt}, G_y^{opt}, G_t^{opt}]\rangle$ which is optimal and compatible with the QCSP : $\exists x \exists y \forall z \exists t (x = (y * z) + t)$ with $x, y, z, t \in \{0, 1, 2\}$.

$G_x^{opt} = [(0, \square), (1, \square), (2, \square)]$

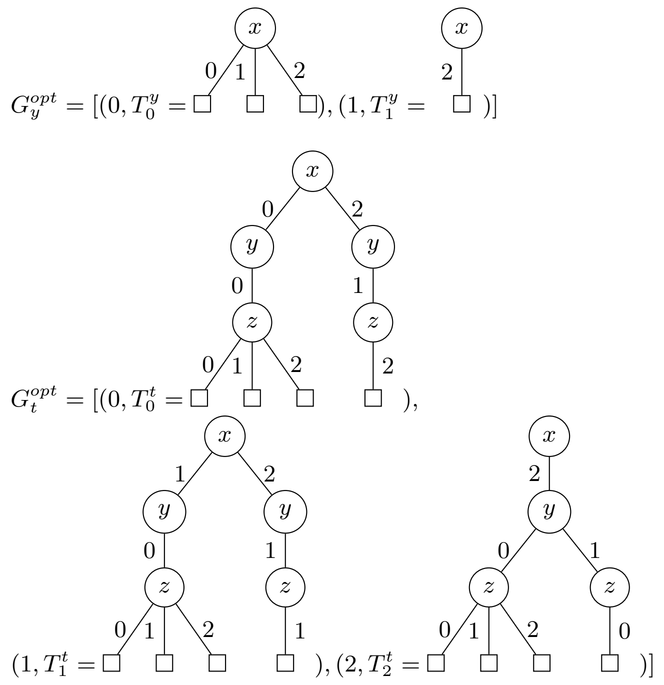


Fig. 1. Guard sets for an optimal QCSP base.

We explicit hereafter the optimality of the QCSP base but we use the QCSP constraint $(x = (y * z) + t)$ instead of the table constraints in order to simplify.

- $i = 1$ (i.e. $v_{e_i} = x$) then $C_i = \emptyset$ and for every $K \in \{0,1,2\}$, $K \in I^K(\Box) = \{K\}$ and $\exists y \forall z \exists t (x = (y * z) + t) \wedge (x = K)$, with $y, z, t \in \{0,1,2\}$, admits a winning strategy.
- $i = 2$ (i.e. $v_{e_i} = y$) then
  - for every $K \in \{0,1,2\}$ $(K, \Box) \in G_x^{opt}$, $I^0(T_0^y) = \{(0,0),(0,1),(0,2)\}$ and for every $K \in \{0,1,2\}$, $(0,K) \in I^0(T_0^y)$ and $\forall z \exists t (x = (y * z) + t) \wedge (y = 0) \wedge (x = K)$, with $z, t \in \{0,1,2\}$, admits a winning strategy ;
  - $(1, \Box) \in G_x^{opt}$, $I^1(T_1^y) = \{(1,1)\}$ and $(1,1) \in I^1(T_1^y)$ and $\forall z \exists t (x = (y * z) + t) \wedge (y = 1) \wedge (x = 1)$, with $z, t \in \{0,1,2\}$, admits winning strategy ; $(1,0) \notin I^1(T_1^y)$ and $\forall z \exists t (x = (y * z) + t) \wedge (y = 1) \wedge (x = 0)$, with $z, t \in \{0,1,2\}$, does not admit a winning strategy ; $(1,2) \notin I^1(T_1^y)$ and $\forall z \exists t (x = (y * z) + t) \wedge (y = 1) \wedge (x = 2)$, with $z, t \in \{0,1,2\}$, does not admit a winning strategy ;
  - for $y = 2$, the is no pair $(2, T_2^y) \in G_y$ and $\exists x \forall z \exists t (x = (y*z)+t) \wedge (y = 2)$, with $x, z, t \in \{0,1,2\}$, does not admit a winning strategy.
- $i = 3$ (i.e. $v_{e_i} = t$) then (we only treat the case $t = 0$, the others are similar)
  - $(2, \Box) \in G_x^{opt}$, $(1, T_1^y) \in G_y^{opt}$ with $(1,2) \in I^1(T_1^y)$ and $(0,2,1,2) \in I^0(T_0^t)$ and $(x = (y * z) + t) \wedge (x = 2) \wedge (y = 1) \wedge (z = 2) \wedge (t = 0)$ admits a winning strategy ; it is similar for $(0,0,0,0)$, $(0,0,0,1)$ and $(0,0,0,2)$;
  - for all the other cases $(0, val_x, val_y, val_z) \notin I^0(T_0^t)$ and $(x = (y*z)+t) \wedge (x = val_x) \wedge (y = val_y) \wedge (z = val_z) \wedge (t = 0)$ does not admit a winning strategy.

The most important property of optimal QCSP base is that the next move choice problem for the interpretation of compatible optimal base with a QCSP is no more PSPACE-complete but polytime.

Theorem 2 (next move choice problem). *The next move decision problem for the interpretation of an optimal base is polytime in the size of the base.*

4 Compilation of a QCSP to an optimal QCSP base

We present in this section an algorithm based on a search algorithm and establish that the result of the application of this algorithm is an optimal QCSP base compatible with the initial QCSP. Algorithm 1 *rec_comp* computes a compatible QCSP base from a QCSP following the inductive definition of the semantics of the QCSP. This algorithm first computes a fix-point for the set of constraints and returns *bl_bottom* if a contradiction is detected. If it is not the case and the binder is not empty then *bl_top* is returned. Otherwise for every value *val* of the domain of the outermost variable x of the binder, the constraint $(x = val)$ is added to the constraint store and the algorithm is recursively called. If the variable is universally quantified and at least one subproblem returns *bl_bottom* then *bl_bottom* is returned. If the variable is existentially quantified and all the subproblems return *bl_bottom* then *bl_bottom* is returned. In any other cases, operators $\oplus_\exists$ or $\oplus_\forall$ are called to combine the resulting QCSP bases together.

Algorithm 1 *rec_comp*

```
In: $Q$ : a binder of a QCSP
In: $C$ : a set of constraints of a QCSP
Out: a QCSP base or $bl\_top$ or $bl\_bottom$
  if $reach\_fixpoint(C) = failure$ then
    return $bl\_bottom$
  end if
  if $empty(Q)$ then return $bl\_top$ end if
  $qx_D \leftarrow head(Q); listValBase \leftarrow []; d \leftarrow D$
  while $!empty(d)$ do
    $val \leftarrow head(d); d \leftarrow tail(d)$
    $base \leftarrow rec\_comp(tail(Q), C \cup \{x = val\})$
    if $base = bl\_bottom \& q = \forall$ then
      return $bl\_bottom$
    end if
    $listValBase \leftarrow [(val, base)|listValBase]$
  end while
  if $empty(listValBase)$ then
    return $bl\_bottom$
  end if
  if $q = \exists$ then
    return $\oplus_\exists(x_D, Q, listValBase)$
  else
    return $\oplus_\forall(x_D, Q, listValBase)$
  end if
```

Operators $\oplus_\forall$ and $\oplus_\exists$ specified respectively by the algorithms 2 and 3 work as follows. First we describe the $\oplus_\forall$ operator. Function *constants*(l) checks if the list of pairs as argument l does not contain only *bl_top* or *bl_bottom* for second element. If the check *constants*(l) is verified, it is necessarily only a list of *bl_top* associated with all the values of the domain of the variable in case of an innermost bloc of universal quantifiers and then *bl_top* is returned. Otherwise, the operator $\oplus$ defined hereafter is applied and the result is returned since the universally quantified variables are not associated to guards. Now we describe the $\oplus_\exists$ operator. If the check *constants*(l) is verified, it is necessarily an innermost existential quantifier and a QCSP base containing only the values associated to the *bl_top* is built thanks to the function *base_case* defined by $base\_case(l) = \{(val, \square)|(val, bl\_top) \in l\}$. Otherwise, the returned QCSP base is built by adding to the result of the $\oplus$ operator the list of the values associated to each of the QCSP bases thanks to the function $first\_values$ defined by $first\_values(l) = \{(val, \square)|(val, a) \in l\}$.

The $\oplus$ operator works as follows. The *decompose* function extracts from the list *lvb* of pairs (value, QCSP bases), for the outermost existentially quantified variable y of the binder, a pair constituted of a list of pairs (val, list of guards)) and a list of pairs (val, remaining of the guards). The *compose* function builds for y its set of guards by distributing the trees for the different values. Functions

Algorithm 2 $\oplus_\exists$

```
In: $x_D$ : a variable and its domain
In: $Q$ : a binder
In: $l$ : a list of pairs (value, QCSP base)
Out: a QCSP base
  if $constants(l)$ then
    return $\langle \exists x_D Q \mid case\_base(l) \rangle$
  else
    return $\langle \exists x_D Q \mid [first\_values(l) | \oplus(x, l)] \rangle$
  end if
```

Algorithm 3 $\oplus_\forall$

```
In: $x_D$ : a variable and its domain
In: $Q$ : a binder
In: $l$ : a list of pairs (value, QCSP base)
Out: a QCSP base or $bl\_top$
  if $constants(l)$ then
    return $bl\_top$
  else
    return $\langle \forall x_D Q \mid \oplus(x, l) \rangle$
  end if
```

$first$ and $second$ give access to respectively the first and the second position of a pair.

Algorithm 4 $\oplus$

```
In: $x$ : a variable
In: $lvb$ : a list of pairs (value, QCSP base)
Out: a list of guards
  $lg \leftarrow []$
  $lvg \leftarrow extract\_guards(lvb)$
  while $empty(lvg)$ do
    $dec\_y \leftarrow decompose(lvg)$
    $lg \leftarrow [compose(x, first(dec\_y)) | lg]$
    $lvg \leftarrow second(dec\_y)$
  end while
  return $lg$
```

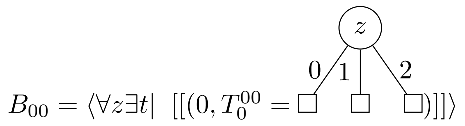
The following example shows how the *rec_comp* algorithm works.

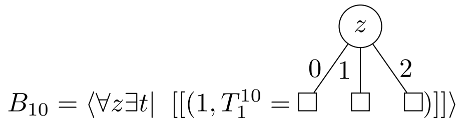
Example 6. We compute for all $val_x, val_y \in \{0, 1, 2\}$ the QCSP base $B_{val_x val_y}$ as the result of the following call :

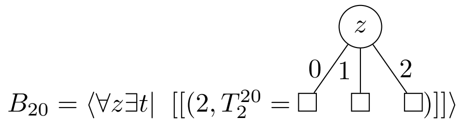
$$rec\_comp(\forall z \exists t, \{(x = (y * z) + t), (x = val_x), (y = val_y)\})$$

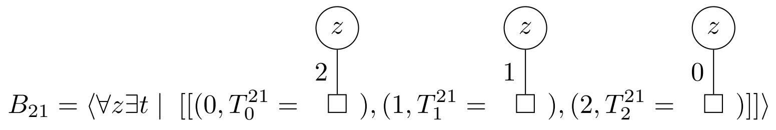
with $z, t \in \{0, 1, 2\}$.

We obtain the QCSP bases (according to $T_{val_t}^{val_x val_y}$) :








$B_{21} = \langle \forall z \exists t \mid \ [[(0, T_0^{21} =$
z
2
$), (1, T_1^{21} =$
z
1
$), (2, T_2^{21} =$
z
0
$)]]\rangle$

of for any other combination, $B_{val_x val_y} = bl\_bottom$.

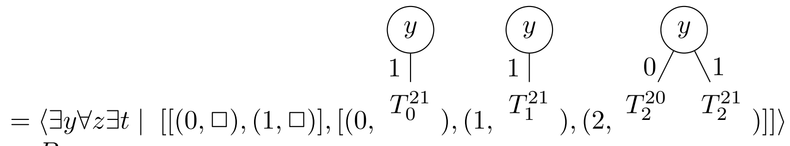
The following example shows how the trees are shared by the $\oplus$ operator and also the distribution of the trees.

Example 7. The operator $\oplus_\exists$ is applied during the execution of the call

$$rec\_comp(\exists y \forall z \exists t, \{(x = (y * z) + t), (x = 2)\})$$

with $y, z, t \in \{0, 1, 2\}$, to the QCSP bases B_{20}, B_{21} and B_{22} which represent compatible QCSP bases with QCSP, respectively, $\forall z \exists t((x = (y * z) + t) \wedge (x = 2) \wedge (y = 0))$, $\forall z \exists t((x = (y * z) + t) \wedge (x = 2) \wedge (y = 1))$, $\forall z \exists t((x = (y * z) + t) \wedge (x = 2) \wedge (y = 2))$.

$\oplus_\exists(y, \forall z \exists t, [(0, B_{20}), (1, B_{21}), (2, B_{22})])$


$= \langle \exists y \forall z \exists t \mid \ [[(0, \square), (1, \square)], [(0,$
y
1
T_0^{21}
$), (1,$
y
1
T_1^{21}
$), (2,$
y
0
1
T_2^{20}
T_2^{21}
$)]]\rangle$

$= B_2$

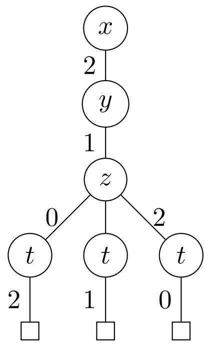
which is a compatible QCSP base with the QCSP $\exists y \forall z \exists t((x = (y * z) + t) \wedge (x = 2))$ with $y, z, t \in \{0, 1, 2\}$.

The following example shows how the existential player can play with an optimal QCSP base instead of only one winning strategy how the optimal QCSP base gives a direct access to the possibilities for a given existentially quantified variable.

Example 8. (Previous examples continued.) Let the following QCSP be the expression of a very simple two-player game:

$$\exists x \exists y \forall z \exists t(x = (y * z) + t), x, y, z, t \in \{0, 1, 2\}$$

Let us suppose that an existential player only knows the following winning strategy:



For his first move, he decides to play $(x = 2)$. Now, following its winning strategy, he is supposed to play $(y = 1)$. Let us suppose that it is no more possible because of an unexpected reason. He can not follow its winning strategy anymore. If he follows the compatible but not optimal base of Example 1 he will follows one of the two other choices thinking that he still has a chance. If he wants to be sure, he will have to pay the full computational price for the QCSP $\forall z \exists t((x = (y * z) + t) \wedge (x = 2) \wedge (y = 0)), z, t \in \{0, 1, 2\}$ and $\forall z \exists t((x = (y * z) + t) \wedge (x = 2) \wedge (y = 2)), z, t \in \{0, 1, 2\}$. If he follows its compatible and optimal QCSP base of Example 5, he knows that he has already lost.

The following theorem establishes that the *rec_comp* algorithm not only computes a compatible QCSP base from a QCSP but also that this QCSP base is optimal.

Theorem 3. *Let QC be a QCSP. rec_comp(Q, C) return an optimal base and compatible with the QCSP.*

5 Conclusion

We have proposed in this article a framework for the compilation of QCSP: the QCSP bases. We have defined an algorithm embedded in the state-of-the-art search algorithm of QCSP solver to compute a QCSP base from a QCSP.

We have shown that the obtained QCSP base is compatible with the initial QCSP and optimal in the sense that the construction of any winning strategy is polytime for the interpreration of a QCSP base.

We have implemented in Prolog the algorithms described in this article and the programs and examples are downloadable. at the following address `http://www.info.univ-angers.fr/pub/stephan/Research/Download.html`. We plan to integrate it in our QCSP solveur developed in the generic constraint development environment Gecode [15].

When a QCSP solver returns there is or there is no winning strategy, there is no way to check if the answer is correct while for CSP the associated result to the decision (a complete instantiation) is easy to check. A certificate for a QCSP which has a winning strategy is any piece of information that provides self-supporting evidence of the existence of a winning strategy for that QCSP. Due to the lack of space, we have not treated certificates for QCSP: our formalism includes QCSP certificates as a particular case. During the execution of the solver, a QCSP base is generated but only with strategies as guards. The interpretation of these trees (cf Definition 2) in tuples permits to verify such a QCSP certificate w.r.t. a QCSP by the resolution of a co-NP-complete problem. (The complexity is similar to the check of a winning strategy for QBF [4].)

We have proposed a recursive construction of the QCSP base but in practice it is often more efficient to consider a cooperation between a solver which emits a trace and a trace analyzer which builds the QCSP base. We have also develop this approach but due to the lack of space we only give here the main two important reasons for this cooperation between a solver and a trace analyzer: If the construction of the QCSP base is embedded into the solver, the memory management of the solver by means of a backtrack stack will have also to keep the state of the current QCSP base. We want to take into account modern architectures with multi-core multithreaded processors.

References

1. J. Amilhastre, H. Fargier, and P. Marquis. Consistency Restoration and Explanations in Dynamic CSPs - Application to Configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.
2. S. Baba, Y. Joe, A. Iwasaki, and M. Yokoo. Real-Time Solving of Quantified CSPs Based on Monte-Carlo Game Tree Search. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 655–661, 2011.
3. F. Bacchus and K. Stergiou. Solution directed backjumping for QCSP. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, pages 148–163, 2007.
4. M. Benedetti. Extracting Certificates from Quantified Boolean Formulas. In *Proceedings of 9th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 47–53, 2005.
5. M. Benedetti, A. Lallouet, and J. Vautard. Reusing CSP propagators for QCSPs. In *Recent Advances in Constraints, 11th Annual ERCIM International, Workshop on Constraint Solving and Contraint Logic Programming (CSCLP'06)*, pages 63–77, 2006.

6. L. Bordeaux, M. Cadoli, and T. Mancini. CSP Properties for Quantified Constraints: Definitions and Complexity. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI'05)*, pages 360–365, 2005.
7. L. Bordeaux and E. Monfroy. Beyond NP: Arc-Consistency for Quantified Constraints. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, pages 371–386, 2002.
8. S.C. Brailsford, C.N. Potts, and B.M. Smith. Constraint staifaction problems: Agorithms and applications. *European Journal of Operational Research*, 119:557–581, 1999.
9. H. Chen. The Computational Complexity of Quantified Constraint Satisfaction, PhD thesis, Cornell University, 2004.
10. S. Coste-Marquis, D. Le Berre, F. Letombe, and P. Marquis. Complexity Results for Quantified Boolean Formulae Based on Complete Propositional Languages. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:61–88, 2006.
11. H. Fargier and P. Marquis. On the Use of Partially Ordered Decision Graphs in Knowledge Compilation and Quantified Boolean Formulae. In *Proceedings of the 21th National Conference on Artificial Intelligence (AAAI'06)*, 2006.
12. I. Gent, P. Nightingale, and K. Stergiou. QCSP-solve: A solver for quantified constraint satisfaction problems. In *Proceedings of 9th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 138–143, 2005.
13. I.P. Gent, P. Nightingale, and A. Rowley. Encoding Quantified CSPs as Quantified Boolean Formulae. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, pages 176–180, 2004.
14. A.K. Mackworth. Consistency in networks of relations. In *Artificial Intelligence*, volume 8, pages 99–118, 1977.
15. C. Schulte and G. Tack. Views and Iterators for Generic Constraint Implementations. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, pages 817–821, 2005.
16. I. Stéphan and B. Da Mota. A unified framework for Certificate and Compilation for QBF. In *Proceedings of the 3rd Indian Conference on Logic and its Applications*, pages 210–223, 2009.
17. D. Stynes and K.N. Brown. Realtime Online Solving of Quantified CSPs. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP'09)*, pages 771–786, 2009.
18. G. Verger and C. Bessière. BlockSolve : une approche bottom-up des QCSP. In *Actes des Deuxièmes Journées Francophones de Programmation par Contraintes (JFPC'06)*, pages 337–345, 2006.

Integrating Datalog and Constraint Solving

Benoit Desouter and Tom Schrijvers

Ghent University, Belgium
{Benoit.Desouter,Tom.Schrijvers}@UGent.be

Abstract. LP is a common formalism for the field of databases and CSP, both at the theoretical level and the implementation level in the form of Datalog and CLP. In the past, close correspondences have been made between both fields at the theoretical level. Yet correspondence at the implementation level has been much less explored. In this article we work towards relating them at the implementation level. Concretely, we show how to derive the efficient Leapfrog Triejoin execution algorithm of Datalog from a generic CP execution scheme.

1 Introduction

Constraint programming (CP) is a well-known paradigm in which relations between variables describe the properties of a solution to the problem we wish to solve [1]. The strategy how to actually compute these solutions is left to the system. Databases have been an interesting research topic since the 1960's. Constraint programming and databases span two separate domains, each with their own insights and techniques. They are not immediately similar. However database theory and CP, in particular CLP, actually have much in common [2], at least at the theoretical level. A common language for them is first-order logic, which does not involve any computational aspects.

Cross-fertilization between the two could give us more expressive systems and better results. Hence we look at logic programming as a common computational language: Datalog is a query language for deductive databases used in a variety of applications, such as retail planning, modelling, ... [3].

This paper aims to show that Datalog and CP are also compatible at the implementation level. We do so by showing how a standard CP implementation scheme, as formulated by Schulte and Stuckey [4], can be specialized to obtain a recently documented Datalog execution algorithm called *Leapfrog Triejoin* [5]. Leapfrog Triejoin has a good theoretical complexity and is simple to implement.

This integration opens up the possibility for further cross-fertilization between actual CP and Datalog systems. In particular, we aim to integrate CP propagation techniques in the Datalog join algorithm for query optimization.

2 The Abstract Constraint Solving Scheme

Our starting point is the abstract algorithm for efficient CP propagator engines as formulated by Schulte and Stuckey [4], listed in Figure 1.

The inputs to the algorithm are a set of old (constraint) propagators F_o, a set of new propagators F_n and the domain D of the constraint variables. Initially the set of old propagators is empty and a toplevel calls takes the form **search**$(\emptyset, F_n, D)$. These inputs are derived from a declarative problem specification that relates a finite set of constraint variables $\mathcal{V}$ by means of a number of constraints C:

– All of the variables have an initial set of admissible values, which are captured in D. This domain D is a mapping from $\mathcal{V}$ to the admissible values; we denote the set associated with a variable x as $D(x)$. For example, if the admissible values for x are the integers from one to four, we write $D(x) = \{1, 2, 3, 4\}$.
– The constraints are captured in a set of constraint propagators F (typically one or several per constraint). Such a propagator f is a monotonically decreasing function on domains that removes values that do not feature in any possible solution to the constraint.
  Define, in addition to x and its domain defined above, a variable y with $D(y) = \{3, 4, 5\}$. A constraint propagator for $x = y$ can then eliminate the values $\{1, 2\}$ from $D(x)$ and 5 from $D(y)$.

Given these inputs it is the algorithm's job to figure out whether the constraint problem has a solution. To do so, it alternates between two phases: *constraint propagation* and *nondeterministic choice*.

Constraint propagation computes a fixpoint of the constraint propagators; it is captured in the function **isolv**(F_o, F_n, D) which we will explain in more detail below. This may yield one of three possible outcomes:

1. One of the variables has no more admissible values. Then there is no solution.
2. All of the variables have exactly one admissible value. A solution has been found.
3. At least one of the variables has two or more admissible values.

The first two cases terminate the algorithm. The last case leads to nondeterministic choice. The current search space $C \wedge D$ is partitioned using a set of constraints $\{c_1, \ldots, c_n\}$. Typical approaches include the use of two constraints that each split the domain of a certain variable in half, or constraints that either remove or assign a value. A large number of strategies for choosing a split variable or a value exists. One may pick the variable with the largest domain, the smallest domain, ..., the smallest value or a random one, etc. In all of those cases each of the subspaces is explored recursively in a depth-first order. The ith recursive subcall gets $F_o \cup F_n$ as old propagators[1] and the new propagators of c_i as the new ones.

Incremental Constraint Propagation Figure 2 shows an incremental algorithm for constraint propagation. It takes a set of old propagators F_o, new propagators F_n and a constraint domain as inputs, and returns a reduced domain as output. The invariant is that for every old propagator $f_o \in F_o$, D is a fixpoint (i.e.,

[1] We know that D is a fixpoint of them.

```
search($F_o$, $F_n$, $D$)
    $D :=$ isolv($F_0$,$F_n$,$D$)                                    % propagation
    if ($D$ is a false domain)
        return
    if ($\exists x \in \mathcal{V}. |D(x)| > 1$)
        choose $\{c_1, \ldots, c_m\}$ where $C \wedge D \models c_1 \vee \cdots \vee c_m$     % search strategy
        for $i \in [1..m]$
            search($F_o \cup F_n$, prop($c_i$), $D$)
    else
        yield $D$
```

Fig. 1. General constraint solver

```
isolv($F_o$, $F_n$, $D$)
    $F := F_o \cup F_n$; $Q := F_n$;
    while ($Q \neq \emptyset$)
        $f :=$ first($Q$)
        $Q := Q - \{f\}$; $D' := f(D)$
        if ($D' \neq D$)
            $Q :=$ new($f, F, D, D'$)
        $D := D'$
    return $D$
```

Fig. 2. Incremental constraint propagation

$D = f_o(D)$). The propagators that may still reduce D are in F_n; they are used to initialize a worklist Q.

Then the algorithm repeatedly takes a propagator f from Q and uses it to obtain a possibly reduced domain D'. Then an auxiliary function (not given) new(f, F, D, D') determines what new propagators from F to add to the worklist; these should be propagators for which D' may not be a fixpoint. A valid but highly inefficient implementation of new just returns F, but typical implementations try to be more clever and return a much smaller set of propagators.

When the worklist is empty, the algorithm returns D which is now a fixpoint of all propagators F.

2.1 Instantiation

In practice the generic scheme is instantiated to fill in unspecified details (like how the partition is obtained) and refined to obtain better efficiency. For instance, when D is reduced by a propagator, typically not all variables are affected. The new function would only return those propagators that depend on the affected variables. Moreover, efficient pointer-based datastructures would be used to quickly identify the relevant propagators.

In the rest of the paper we will apply various such instantiations and refinements. Yet our goal is not to obtain a concrete CP system. Instead, we have as target the Datalog Leapfrog-Triejoin execution algorithm.

3 The Datalog Instance

Datalog execution uses rules to derive new facts from known facts. A rule has the form

$$\mathrm{h} \leftarrow \mathrm{b}_1, \ldots, \mathrm{b}_n$$

where $\mathrm{h}, \mathrm{b}_1, \ldots, \mathrm{b}_n$ are atomic formulas. An atomic formula has the form

$$\mathrm{p}(X_1, \ldots, X_n)$$

where p is a predicate with arity n and the X_i are variables. Every predicate refers to a table of facts of the form $\mathrm{p}(\mathrm{c}_1, \ldots, \mathrm{c}_n)$ with the c_i constants. If the body is instantiated by known facts, then the head yields a (possibly) new fact.

The most performance-critical part of the instantiation is the *join* which finds facts that share a common argument. Suppose we have the following facts:

$$\mathrm{p(a,b), p(c,d), p(e,f)}$$

$$\mathrm{q(a,1), q(c,2), q(g,3)}$$

Then the join $\mathrm{p}(X, Y), \mathrm{q}(X, Z)$ gives us the following results: $\{X \mapsto a, Y \mapsto b, Z \mapsto 1\}$ and $\{X \mapsto c, Y \mapsto d, Z \mapsto 2\}$.

As is clear from the example, a join between three unary predicates looks like

$$\mathrm{p}(X), \mathrm{q}(X), \mathrm{r}(X)$$

We can rewrite the rule body to the following form

$$\mathrm{p}(X), \mathrm{q}(Y), \mathrm{r}(Z), X = Y = Z$$

that makes the equalities explicit. Now the following analogy with constraint satisfaction problems becomes more obvious:

- The rule variables correspond to constraint variables.
- The predicates denote the domains of the variables.
- The equalities are constraints on the variables.

Note that Datalog only uses one kind of constraints: the *global equality constraint*. A generic propagator for this constraint is shown in Figure 3, that only performs propagation on the lower bound.

It introduces a variable mapping M from $\{0, \ldots, n-1\}$ to $\mathcal{V}$. The mapping is essentially an array of pointers to the elements of D. We sort M by increasing lower bound of the variables in D. Then, the variable x pointed to by the last position in M has the largest lower bound $l_{\max}$. For each of the other variables, we eliminate values smaller than $l_{\max}$. If this operation leads to a larger lower

```
allequal($D$)
    make a variable mapping $M = \{0 \mapsto x_1, \ldots, n-1 \mapsto x_n\}$ to $D$
    sort $M$ by increasing lower bound in $D$
    $l_{\max} :=$ lowerBound($D[M[n-1]]$)
    $i := 0$
    while (lowerBound($D[M[i]]$) $\neq l_{\max}$)
        $D[M[i]]$.raiseLowerBound($l_{\max}$)
        $l_{\max} :=$ lowerBound($D[M[i]]$)
        $i := (i+1) \bmod n$
    return $D$
```

Fig. 3. Generic equality propagator

bound, we start the entire process again. If, in contrast, the lower bound of all variables is equal to $l_{\max}$, we have found a fixpoint from which we can derive a solution. Thus the algorithm maintains the invariant that the lower bounds of the variables pointed to by the array elements at indices $i \ldots (i+n) \bmod n$ are a sorted series.

As an example consider three variables X, Y and Z with respective domains $D(X) = \{1, 2, 3, 4, 9, 10, 11\}, D(Y) = \{3, 4, 7, 10\}$ and $D(Z) = \{1, 4, 7, 10, 11\}$. Sorting M by increasing lower bound then gives us $\{0 \mapsto X, 1 \mapsto Z, 2 \mapsto Y\}$. In Table 1, we illustrate the process, underlining the domain with the largest lower bound. The absence of a value means that there are no changes with respect to the previous line in the table.

Initially $l_{\max} = 3$. In the first iteration, we increase the lower bound of X to 3 and $l_{\max}$ does not change. In the next iteration, the lower bound of Z is increased to 4. The maximum lower bound $l_{\max}$ is updated accordingly. In the next two iterations the lower bounds of Y and X are again increased to end up at 4. We now have found a solution.

$\boldsymbol{X}$	$\boldsymbol{Z}$	$\boldsymbol{Y}$
$\{1,2,3,4,9,10,11\}$	$\{1,4,7,10,11\}$	$\underline{\{3,4,7,10\}}$
$\underline{\{3,4,9,10,11\}}$		
	$\underline{\{4,7,10,11\}}$	
		$\underline{\{4,7,10\}}$
$\underline{\{4,9,10,11\}}$		

Table 1. Example operation of the propagator for global equality

3.1 Unary Datalog

When we restrict ourselves to unary Datalog, we only solve CP problems with a single equality propagator at a time. For rules with multiple variables like

$$\mathrm{p}(X), \mathrm{q}(X), \mathrm{r}(Y), \mathrm{s}(Y)$$

we calculate one join at a time. The final result is then the Cartesian product of the solutions for X and Y.

In this situation, **isolv** is trivial to implement as a single invocation of the propagator. This is valid because the propagator is idempotent:

$$\textbf{allequal}(\textbf{allequal}(D)) = \textbf{allequal}(D)$$

4 Making a choice

The abstract constraint solving scheme from Section 2 does not specify how to add extra constraints c_i to D when propagation alone does not yield a solution. Recall that the set of constraints c_i added in turn must partition the search space. A well-known technique, the indomain-min strategy, is to select a variable x and either assign or remove its lower bound lb: $c_1 \equiv (x = lb), c_2 \equiv (x > lb)$.

This technique is particularly attractive here because the propagator has already made sure that all variables have the same lower bound. Thus assigning the lower bound of one variable with c_1 requires no work. In particular it requires no further propagation by the **allequal** propagator, so we immediately have a solution that we can yield.

In the other branch, we increment the domain. That means we eliminate the lower bound from a random variable. We then continue as before. We do not need an additional propagator for c_2: by eliminating the lower bound lb, the constraint c_2 is satisfied right away.

In Figure 4 we show the impact of this refinement on the specialized constraint solver. Note that the algorithm is now tail recursive and thus can easily be turned into a **while** loop that runs in constant stack space. Contrast this with conventional CP systems that need a stack to perform depth first search. We illustrate the difference in Figure 5. On the left is a general search tree; on the right the tree searched in our code. The dashed nodes represent the solution found after the call to **allequal**. From this node, we can move on to the rest of the tree by following the dashed arrow. It corresponds to the incDomain operation.

As an example of the approach, again consider the three variables X, Y and Z with the same domains as above. The first solution is $X = Y = Z = 4$. After yielding this solution, the domains are $X = \{4, 9, 10, 11\}, Y = \{4, 7, 10\}$ and $Z = \{4, 7, 10, 11\}$. We now increase the lower bound of variable X. During the next iteration of the while loop, **allequal** is applied again to find the solution $X = Y = Z = 10$. Once again incrementing the lower bound of X leaves us with an empty domain and the while loop terminates.

```
search($D$)
    $D$ := allequal($D$)                              % propagation
    if ($D$ is a false domain)
        return
    yield $D$                                          % lower bounds equal
    $D$ := incDomain($D$,$x$)
    search($D$)

                              ⇓

search($D$)
    while ($true$)
        $D$ := allequal($D$)
        if ($D$ is a false domain)
            return
        yield $D$
        $D$ := incDomain($D$,$x$)
```

Fig. 4. Effect of the indomain-minimum value selection on the constraint solver

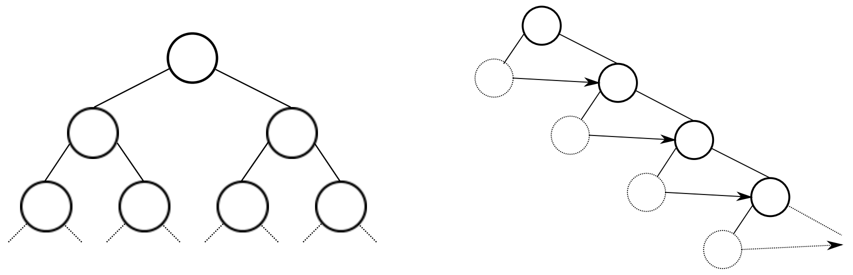


Fig. 5. General search tree (left) vs. tree searched by our tail recursive algorithm (right).

5 Leapfrog Triejoin

When we inline the code for the **allequal** constraint from Figure 3 within the constraint solver with indomain-min value selection, it is clear we can introduce one more optimization. Indeed, we do not need to resort the variable domains on every invocation of the propagator. This is because we know the variable modified by the incDomain operation must be the one that has the new largest lower bound. All other variables have not been modified since we found a solution. To avoid having to change the position $p_{\min}$ where the solution was detected, it is most convenient to increase the variable at position $p_{\min} - 1 \bmod n$ in M. In that way, there is absolutely no work involved in maintaining the ordering. The result can be seen in Figure 6. The algorithm is now exactly the same as Leapfrog Triejoin.

```
search($D$)
    make a variable mapping $M$ to $D$
    sort $M$ by increasing lower bound in $D$
    $p_{\min} := 0$
    $l_{\max} :=$ lowerBound($D[M[n-1]]$)
    while ($D$ is not a false domain)
        $foundSolution := false$
        while($\neg$ ($foundSolution \lor D$ is a false domain))
            $x_{\min} := D[M[p_{\min}]]$
            $l_{\min} :=$ lowerBound($x_{\min}$)
            if ($l_{\min} = l_{\max}$)
                $foundSolution := true$
            else
                $l_{\max} := x_{\min}$.raiseLowerBound($l_{\max}$)
                $p_{\min} := (p_{\min} + 1) \bmod n$
        if ($foundSolution$)
            yield $D$
            $D :=$ incDomain($D$,$D[M[p_{\min} - 1 \bmod n]]$)
```

Fig. 6. Leapfrog Triejoin algorithm

We begin by sorting the array of pointers M to variables x by increasing lower bound. As before, we keep the maximum lower bound in $l_{\max}$. The variable $x_{\min}$ having the smallest lower bound can be found at position $p_{\min}$ in M. As before, we know we have a solution if $l_{\min}$ is equal to $l_{\max}$. The inner while loop either stops because this is the case, or because there is a variable x with an empty domain. In the latter case, all solutions have been found. In the former case, we yield the solution and increment the domain. This is done in such a way that we can immediately start the inner while loop again.

6 Full Datalog Implementation

Iterator implementation We have started from an abstract domain representation D. In CP it is typically represented as a union of intervals $\bigcup [lb_i, ub_i]$ where $lb_{i+1} > ub_i + 1$. We only use a restricted set of operations in the algorithm of Figure 6:

- Access to the lower bound from the domain of a variable x.
- Removing that lower bound from the domain.
- Removing all values smaller than a certain value from the domain.

In a Datalog context, tables are normally stored as trees. But as described in Veldhuizen's work [5], it is perfectly possible to implement the necessary operations on top of trees. The resulting concept is called an iterator.

N-Ary predicates In addition to the operations needed in the unary case, an iterator offers three additional operations for working with general predicates: **open** and **up** are used to move in the tree-based representation of a relation. From a higher level, we can describe this as moving between the variables in a predicate. The function **depth** then indicates which variable we are currently manipulating.

The basic approach for non-unary predicates is to use one Leapfrog Triejoin per set of variables that must be equal. For example, if $\mathrm{p}(X, Y)$ and $\mathrm{q}(Z, Q)$ are two binary predicates and we join on $X = Z, Y = Q$, we first calculate a solution for $X = Z$. Given this configuration, all solutions for $Y = Q$ are looked for. Then we look for the next solution where $X = Y$ and repeat the entire process.

Datalog System A fully functional Datalog system has the ability to store the new facts derived by the program rules. This can be achieved by collecting the answers and storing them in trees. Recursive rules can be handled with a semi-naive algorithm. Both capabilities do not influence the core algorithm described in this paper.

7 Related Work

Much work has been done in coupling logic programming languages to relational databases. The oldest method, relational access, lets Prolog access only one table at a time and combines data from multiple tables using depth-first search. It is clear that this method is very inefficient, since it does not exploit any of the optimizations from the DBMS. Maier *et al.* [6, 7] have stressed the importance of achieving this coupling efficiently. A more recent approach thus translates Prolog database access predicates into appropriate SQL queries [8]. Although arguably more efficient, the integration may have varying degrees of transparency. Queries are generally isolated from the rest of the Prolog program. Therefore, they may not use all information available in the Prolog program to restrict the number of records accessed even more. Furthermore, not all queries expressible in Prolog can be translated to SQL.

Compared to our work, these integration techniques are rather loosely coupled. Back in 1986, Brodie and Jarke stated that tightly integrating logic programming techniques with those of DBMSs will yield a more capable system. They estimated this requires no more work than extending either with some facilities of the other [9]. Unfortunately, to date, no full integration of Prolog and relational databases has gained a significant degree of acceptance [6]. Datalog, on the other hand, has been successfully used as a more integrated approach.

8 Conclusions and Future Work

The integration of Datalog and constraint programming offers many interesting perspectives in join optimization. In this article, we have only described the core ideas behind this integration.

In future, we will first and foremost generalize the approach for non-unary predicates. Intuitively leapfrog triejoin for non-unary predicates corresponds to nested searches. This only allows for propagation between the arguments in one direction. Techniques that allow for more propagation between the arguments definitely deserve our attention.

Furthermore, we will also investigate both impact and advantages of adding propagators for additional constraints. A well-known example is the $X < Y$ constraint. Consider this constraint and the domains $X = \{1, 8\}$ and $Y = \{2, 3, 5, 6, 9\}$. It is clear that after finding all solutions where $X = 1$, one can at once discard the values $\{2, 3, 5, 6\}$ from the domain of Y.

When the less-than constraint is used together with a join, as in

$$\mathrm{p}(X), \mathrm{q}(Y), \mathrm{r}(Y), X < Y$$

we would now first do the join on $\mathrm{q}(Y)$ and $\mathrm{r}(Y)$ and then filter out the values where $X < Y$. It is clear we can improve here with more propagation. Finally, many standard constraint programming optimizations can still be added to the system.

Acknowledgments

We would like to thank LogicBlox, Inc. for their support and for giving us the ability to investigate the integration of Datalog and constraint programming in their system.

References

1. Marriott, K., Stuckey, P.J.: Programming with constraints: an introduction. MIT press (1998)
2. Vardi, M.Y.: Constraint satisfaction and database theory: a tutorial. In: Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. PODS '00, New York, NY, USA, ACM (2000) 76–85
3. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). IEEE Trans. on Knowl. and Data Eng. **1**(1) (1989) 146–166
4. Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. ACM Trans. Program. Lang. Syst. **31**(1) (December 2008) 2:1–2:43
5. Veldhuizen, T.L.: Leapfrog triejoin: a worst-case optimal join algorithm. (2012)
6. Maier, F., Nute, D., Potter, W., Wang, J., Twery, M., Rauscher, H., Knopp, P., Thomasma, S., Dass, M., Uchiyama, H.: Prolog/RDBMS integration in the NED intelligent information system. In: On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE. Volume 2519 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2002) 528–528
7. Maier, F., Nute, D., Potter, W.D., Wang, J., Twery, M., Rauscher, H.M., Dass, M., Uchiyama, H.: Efficient integration of Prolog and relational databases in the NED intelligent information system. In: Proceedings of the 2003 International Conference on Information and Knowledge Engineering (IKE03). (2003) 364–369

8. Draxler, C.: A powerful Prolog to SQL compiler. Technical report, CIS Centre for Information and Language Processing, Ludwig-Maximilians-Universität München (1993)
9. Brodie, M.L., Jarke, M.: On integrating logic programming and databases. In: Proceedings from the first international workshop on expert database systems, Benjamin-Cummings Publishing Co., Inc. (1986) 191–207

Interning Ground Terms in XSB

DAVID S. WARREN [⋆]

Stony Brook University, Stony Brook, NY 11794-4400, USA,
`warren@stonybrook.edu`

Abstract. This paper presents an implementation of interning of ground terms in the XSB Tabled Prolog system. This is related to the idea of "hash-consing". I describe the concept of interning atoms and discuss the issues around interning ground structured terms, motivating why tabling Prolog systems may change the cost-benefit tradeoffs from those of traditional Prolog systems. I describe the details of the implementation of interning ground terms in the XSB Tabled Prolog System and show some of its performance properties. This implementation achieves the effects of that of Zhou and Have [7] but is tuned for XSB's representations and is arguably simpler.

1 Introduction

Prolog implementations (and all implementations of functional languages that I know of) intern atomic constants. An atomic constant (called an "atom" in Prolog) is determined by the character string that constitutes its name. Rather than representing each occurrence of an atom by its character string name, the character strings are kept uniquely in a global table and the atom is represented by a pointer to its string in that table. This (usually) saves space in that multiple occurrences of the same atom are represented by multiple occurrences of a pointer rather than multiple occurrences of its string. But more importantly, comparison of atoms is simplified; two atoms are the same if and only if their pointers are the same. The important direction here is that, since each string appears only once in the global table, two atoms differ if their pointers differ. This makes atom comparison simpler and more efficient.

The atom table is indexed, usually by a hash index, so finding whether a new atom already exists in the table (and adding it if it does not) is a relatively efficient operation. This operation is known as "interning", indicating that an atom representation is converted from a string representation to an internal representation, i.e., the pointer representation.

The question arises as to whether this kind of representation might be lifted to more complex terms, i.e., applied not only to atoms but to structured terms. This idea has been explored in the Lisp language community [2] and goes by the name of "hash-consing" (originally proposed by [1]). The name comes from the use of a "hash" table to store the structures, and in Lisp the way that complex

[⋆] Research supported in part by National Science Foundation, Award 0964196

structures are constructed is by means of an operation called "cons". Zhou and Have [7] present an implementation of this concept in B-Prolog. I compare my approach to theirs in more detail later in the paper.

There are several reasons why interning of structured terms is more complex than interning atoms, and its potential advantages over a traditional direct implementation of structured terms less clear. The interning operation itself, while optimized by sophisticated indexing strategies, still takes time. Interning of atoms is required when a new atom is created. Atoms are created (mostly) at read-in time, and sometimes in particular builtins such as atom_codes/2. Execution of pure code (e.g., without builtins) does not cause the creation of new atoms so the overhead of interning atoms is relatively small and localized in most Prolog programs. However, structured terms are created continuously during execution of pure code. I assume a "copy" based implementation of terms, which is the implementation in WAM-based Prolog systems. For example, running the traditional definition of append/3 to concatenate two lists requires the creation of as many structured subterms as there are elements in the first list. So the interning cost for complex terms can be quite high. Significant memory may be saved by interning structured terms, but the space-time tradeoffs are not clear. Many terms may be created and used once but then not used again. Locality of reference is also changed, so caching behavior may be affected, perhaps for the better, perhaps for the worse.

There is another complication in the case of Prolog that does not arise in functional programming systems: Prolog terms may have variables embedded in them. Interning a term containing a variable is problematic. For example terms, such as f(a,X) and f(a,Y), cannot be interned to the same hash table entry, since X may become bound to b and Y to c, in which case the terms are distinct. However, if they are interned to distinct table entries, then if X and Y both become bound to a, the terms are then the same but in this situation there are two distinct copies of the same term in the hash table, which undermines a major reason for interning. These (and other) difficulties strongly mitigate against trying to intern terms in Prolog that contain variables, i.e., terms that are not ground. (See the recent work of Nguyen and Demoen [3] for an interesting, and deeper, discussion of this issue.)

One might explore interning every term when it is created. In the WAM, terms can be created in a variety of ways. In pure code (i.e., not in builtins) terms are built in a top-down way by a sequence of instructions starting with a get-structure instruction and followed by a sequence of unify-something type instructions, one for each field of the term. These instructions could be changed to support checking whether all subfields contain constants or interned subterms, and to intern the constructed term if so. But this would require major surgery to these instructions. A better solution would probably be to modify the WAM instructions and compilation strategy to build terms bottom-up. But again, it is not clear, even with such optimizations, that the overhead of hashing every time a ground term is constructed would be out-weighed by other improvements.

For all these reasons, I believe, general interning of ground complex terms is not a general implementation strategy considered in Prolog systems.

The advent of Prolog systems that support tabling, however, may have changed the cost-benefit analysis of sometimes interning ground terms. When a tabled predicate is called with new arguments, the arguments of the call are copied into the table; and when new answers are returned to a tabled predicate, they are also copied into the table. Answers are also copied out of tables when they are used to satisfy subsequent calls. This copying of calls and answers to and from tables may lead to significant time and table space usage. For example, when using a DCG (Definite Clause Grammar) in the standard way for parsing, the input string (represented as a list) is passed into each nonterminal predicate and the list remaining after the nonterminal has recognized a prefix is passed out. So, for example, when tabling a nonterminal predicate that removes just the first atom in a long list, the entire list is copied into the table once for the call, and the entire list minus the first element is (again) copied into the table as the answer. (The fact that tries are used to represent calls and answers in tables may in special cases reduce the copying, but in general, it is needed.) So when using DCG's to parse lists of terminal symbols, there is much copying of lists into and out of tables. Tabling a DCG can in principle give the performance of Earley recognition, but this extensive copying of the input list adds an extra "unnecessary" linear factor to the complexity, in both space and in time.

Note also that when a term is copied into the table, it must, of course, be traversed. So it adds no extra complexity to check for its goundedness and intern it if it is ground. Another situation in which this happens is in assert. Since an asserted term is fully traversed to convert it into internal "code form" (in the XSB implementation of assert), one can intern ground subterms during that process without increasing complexity. Another opportunity would be in findall/3.

2 Implementation of Interned Ground Terms

2.1 Representation of Interned Ground Terms

I describe the representation of interned ground terms in XSB. In the WAM structured records are represented as a sequence of words, the first is a pointer to a global record for the function (aka structure) symbol. For a structure symbol of arity n, that initial word is followed immediately by n tagged words representing the subfields of that structure. List (or cons) records are just pairs, with the structure symbol optimized away in favor of a tag. A picture of a portion of the state for interned terms, containing the ground term f(1,g(a)), is shown in Figure 1.

Interned structure records of arity n are stored in blocks of records, each of $n + 2$ words, "linked records" in the figure. The records are accessed (for interning) by using the record arity to index into an array to access a hash table for records of that arity. The hash value is computed using the $n+1$ fields of the

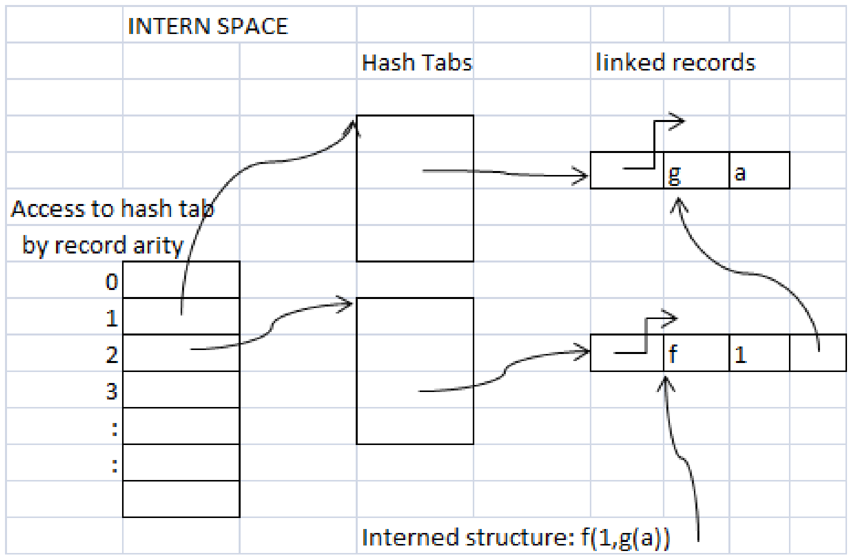


Fig. 1. Storage of Interned Ground Terms

record. The subfields of an interned record can contain only atoms, numbers, or tagged pointers to other interned records, and so the hash value computed from these values will be canonical. The hash value is used to index into the hash table to access a hash bucket chain that can be run to find the desired record. The bucket link field immediately precedes its record. List (or cons) records have their own special hash table.

The representation of interned terms is exactly the same as in the heap; the only difference is that the records are stored in globally allocated blocks, not in the heap. For example, in the Figure, the pointer from the bottom to the f/2 record could well be from the heap, and for any code traversing this representation, the data structure looks exactly the same as it would were in on the heap. This means that all existing code in XSB for accessing and processing structured terms continues to work with interned ground terms.

Whether a structured term pointer is pointing to an interned term or not is determined by examining the pointer itself; if it points into the heap, it is not interned; otherwise it is interned. One can think of this as adding another "tag" to a pointer to a structured term, but the "tag" is implemented using a pointer range, rather than an explicit bit in the pointer. The general unification algorithm is modified to check, when unifying two structured terms, if the terms are both interned, in which case it fails if the addresses are not equal. (Note that the algorithm already succeeds immediately if two pointers to structure records are equal.)

Other builtin functions can be modified to take advantage of knowing a subterm is interned. For example, the builtin ground/2, which checks for groundedness, need not descend into an interned subterm; the builtin copy_term/2 does not need to descend into an interned ground term but can simply copy the reference.

2.2 Interning Ground Terms

A new function (accessible through a builtin) takes a Prolog data object (usually a structured term) and creates a copy of that term in which all ground subterms are interned. The term is traversed bottom-up, using an explicit stack, and the new copy is created on the heap. (Of course, if the term is ground, the new heap copy will be a single word pointing the interned term.) Clearly subterms that are already interned need not be traversed; the reference to the existing interned representation is simply copied. Note this operation is different from the standard copy_term/2, since the new term contains the same variables as the old term, whereas in copy_term/2 the new term contains new variables.

The user can call the builtin intern_term/2 at any time to make a logically identical copy of any term. Since interned ground terms are represented exactly as regular heap terms, except that they reside in a different place in memory, nothing in the XSB system needs to be changed to support the terms created by the new builtin intern_term/2[1]. However to take full advantage of the new term storage mechanism, other system changes can be made. I describe changes made to asserting of dynamic code and to the handling of complex terms in tables.

2.3 Interning before Asserting

Terms are fully traversed in XSB when a clause is asserted to the generate WAM code that will be executed when the clause is called. Also, when that code is called, it may construct a copy of a term on the heap. Thus interned terms can be used to good effect when asserting clauses. New WAM instructions are added for get-intern-structure (and unify-intern-structure) whose (non-register) argument is a pointer to an interned term. If a dynamic predicate is declared as intern, then clauses are automatically interned before they are asserted. This can save space if the same ground term occurs multiple time in asserted clauses. Again, this doesn't increase the complexity of assert, since the clause has to be fully traversed in any case.

Get-intern-structure unifies a term with an interned term. Unlike get-structure, it will never construct any subterm on the heap, since all subterms of the interned term are interned and unifying an interned term with a variable simply sets the variable to point to the interned term. So this can save space on the heap and the time it would otherwise take to construct the term on the heap.

[1] In fact, in XSB the builtin findall/3 copies terms out of the heap and uses the fact that term pointers do or do not point into the heap to determine sharing. Therefore the distinction between findall/3 terms not in the heap and interned terms not in the heap had to be handled carefully within this operation.

Note that indexing is not an issue with asserted clauses in XSB, if they are not trie-indexed. Standard hash indexing still hashes on the same portion of an argument term, whether it is interned or not. Based on the resulting hash value, it chooses the set of clauses that might unify, and then executes the chosen clauses that actually do the unification.

2.4 Interning before Tabling

The computational advantage of interned terms is that the system never needs to make a copy of one; it can simply use its reference. As described above, terms are copied into and out of tables to represent calls and answers. With interned subterms much of this copying can be avoided.

In XSB a variant table can be declared as intern, in which case all calls will be interned before being looked up, and possibly entered, in the table. Similarly, all returned answers will be interned before being (checked and perhaps) added to the table. In XSB terms in a call (and return) table are represented in tries, using a linearization based on a pre-order traversal of the terms. Figure 2 shows schematically a trie that contains interned ground terms.

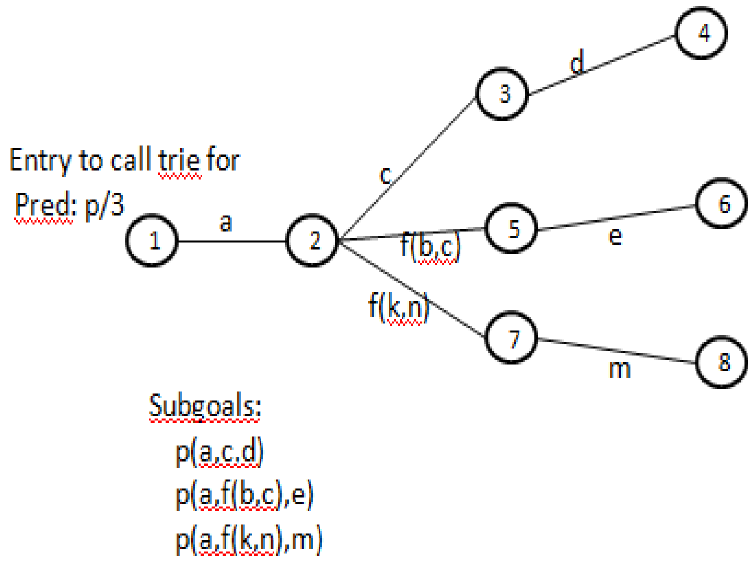


Fig. 2. Trie Containing Interned Ground Terms

The Figure shows a trie containing three calls to p/3: p(a,c,d), p(a,f(b,c),e), and p(a,f(k,n),m), assuming that p/3 is declared as intern. The new feature here is that, for example, f(b,c) is a ground complex term and so is interned before being entered into the trie. So the entire interned subterm is treated as a unit and represented as being on one link, such as between node 2 and node 5. When an interned subterm is encountered when adding (or looking) up a component in such a trie, it is treated as an atomic constant, with the reference treated as the unique identifier. This figure shows a (possibly complex) symbol on each link, but of course, in the implementation that is a pointer to some canonical representation for that symbol. For a constant it is a pointer to the interned string of its name; for an interned structured term, it is a pointer to the canonical representation for that term in the interned term data structure.

Notice that interning the arguments when making a tabled call does not increase the complexity of processing the table, since the terms, were they not interned, would have to be completely traversed in any case. At worst, the constant factor may increase due to the multiple traversals.

It is worth noting how interned terms interact with the indexed lookup of calls (and answers) in the table. Each node in the trie can be indexed, so, for example, a hash index is built (as necessary) on the outgoing links from a node to quickly move to the target node on the right outgoing link. For example, node 2 in the Figure would have a hash table to quickly access nodes 3, 5, or 7. Thus tries ordinarily provide full indexing on every constant and function symbol in a term being looked up. However, as we have seen, interned terms are treated as (unstructured) constants in the trie, and are indexed as constants. This means that there is no indexing on the main function symbol (or indeed any component) within an interned term. So, for example, if a call is made to p/3 of the form p(a,f(k,X),m), when trie traversal reaches node 2, it cannot use the hash table to index at this point to find quickly the one term (on the link to node 7) that matches. Since the input term is f(k,X) is not ground, it is not interned, and the index, which is based on pointers to ground terms cannot be used. Note that were p/3 not tabled as intern, the trie would have more links, and the symbols f/2 and k could be used to index the traversal. But given that p/3 is tabled as intern, the only choice would be to look at every outgoing link from 2, and see whether the possibly complex symbol unifies with the lookup term.

So this loss of indexing may potentially have serious performance consequences. However, if only variants of the source term are to be retrieved, and all ground subterms in both the source lookup term and the trie are known to be interned, which is the case for variant table processing, then this problem is avoided. Note also that tables that are not declared as intern will process interned terms just as they do regular terms, traversing them and processing each atomic component.

The implementation in XSB currently avoids this potential problem by disallowing the entry of interned terms into tries for which retrieval by unification would be required. This may be revisited, since there do seem to be situations in which the benefits of interning could be gained and the pitfalls of the loss of indexing avoided.

3 Performance

All tests were done on a laptop, running Windows 7 Professional, 64-bit OS, on a Intel(R) Core(TM i5) CPU, 2.67 GHz with 8 GB of memory. XSB was compiled using MSVC in 64-bit mode.

Figure 3 shows how long it takes to intern a list of integers. Each run starts with an empty intern table, so every new subterm must be added.

List Length	CPU Time (secs)
100000	0.0160
200000	0.0470
300000	0.0630
400000	0.0940
500000	0.1090
600000	0.1400
700000	0.1720
800000	0.2030
900000	0.2180
1000000	0.2340

Fig. 3. Time to intern a ground list

A simple (but not very realistic) example in which interning can provide great performance improvements (see [7]) is to table a predicate that tests that a term is a proper list:

```
:- table islist/1 as intern.
islist([]).
islist([_|L]) :- islist(L).
```

and call it with a list of distinct integers of various lengths. Figure 4 shows the results. Without interning, each recursive call causes the sublist to be copied

List Len	nonintern Cpu Time (secs)	nonintern Table Space (bytes)	intern Cpu Time (secs)	intern Table Space (bytes)
100	0.0000	427,304	0.0000	27,264
800	0.0160	25,813,640	0.0000	213,600
2700	0.2490	292,321,384	0.0160	721,344
6400	1.3570	1,640,106,376	0.0000	1,722,720
12500	5.8960	6,253,333,160	0.0150	3,333,120

Fig. 4. Interning of islist/2: Space and Time Comparisons

into the table. So every suffix of the initial input list is copied to the table, and the space required is quadratic in the length of the input list. With interning, only a pointer to an interned list is copied into the table, so the space required is linear in the length of the input list.

DCGs normally process lists and can benefit significantly from interned structures. Consider the following DCG for a grammar that recognizes even-length palindromes:

```
:- table epal/2.
epal --> [].
epal --> [X],epal,[X].
```

Figure 5 shows the results of recognizing a list of randomly chosen numbers between 1 and 10,000,000, appended to its reverse, to make an even-length palindrome. The xx's indicate instances that do not run due to memory limitations.

List Len	nonintern Cpu Time (secs)	nonintern Table Space (bytes)	intern Cpu Time (secs)	intern Table Space (bytes)
200	0.0000	3,642,296	0.0000	65,664
1600	0.2340	230,735,064	0.0000	526,944
5400	2.6060	2,625,534,840	0.0160	1,766,336
12800	xx	xx	0.0310	4,213,088
25000	xx	xx	0.0780	8,231,424
43200	xx	xx	0.1090	14,259,296
68600	xx	xx	0.1720	22,751,040
102400	xx	xx	0.2960	34,226,528
145800	xx	xx	0.4680	48,288,128
200000	xx	xx	0.6560	65,848,928

Fig. 5. Palindrome (epal) DCG: Space and Time Comparisons

Note that with interning, palindrome recognition is linear in time and space.

An example of a grammar for which tabling is required is the following left-recursive grammar, which recognizes all strings consisting of just the integers 1, 2, and 3.

```
:- table lr/2.
lr --> [].
lr --> lr,[1].
lr --> lr,[2].
lr --> lr,[3].
```

Figure 6 shows the results of using this grammar to recognize strings, when using and not using interning. The strings are lists of integers 1, 2, and 3 chosen randomly. Again note that interning makes it linear in time and space.

List Len	nonintern Cpu Time (secs)	nonintern Table Space (bytes)	intern Cpu Time (secs)	intern Table Space (bytes)
100	0.0000	388,248	0.0000	5,168
800	0.0620	25,366,688	0.0000	36,752
2700	0.5770	290,570,648	0.0000	116,848
6400	xx	xx	0.0160	322,192
12500	xx	xx	0.0160	631,728
21600	xx	xx	0.0310	995,728
34300	xx	xx	0.0620	1,503,728
51200	xx	xx	0.0940	2,310,800

Fig. 6. Left Recursive (lr) DCG: Space and Time Comparisons

Figure 7 compares the space and time cost of loading (and initializing) a large ontology when interning all ground structures and when interning none. The application loads and initializes a large ontology (and data) using XSB's

nonintern asserted space (bytes)	nonintern cpu Time (secs)	intern asserted space (bytes)	intern interned space (bytes)	intern total space (bytes)	intern cpu Time (secs)
4,456,675,216	484.945	2,765,179,136	506,172,376	3,271,351,512	513.290

Fig. 7. Loading a large database of ontology facts

CDF representation (a package within the XSB System [6]). The CDF represents classes and objects and relationships between them. Classes are represented by small terms, such as cid(local_class_name,name_space), and objects and properties similarly. Also measures (quantity and units) are represented by small (and some not so small containing perhaps 15 symbols) ground terms as well. This ontology has over 1.7 million objects and 7.5 million attributes, represented by facts such as hasAttr_ext(Oid,Rid,Cid), where each id is an object id term, relation id term, or class id term. So there are many ground terms asserted in this database, so interning ground subterms in these asserted facts may save signification space.

When interning ground terms for all dynamic predicates, there is 26.6% decrease in space used traded for a 5.84% increase in load and initialization time. Note that initialization includes more than just the asserting of the facts; so the time overhead for just the interning of asserted facts would be a higher percentage. But this does give an idea of the trade-offs of using interning in a large and complex application.

Another perhaps nonintuitive example is the following simple program that uses tabling and interning to provide asymptotic log access to entries in a sorted list.

```
% find Ent in SortedList, which is Len long
find(Ent,Len,SortedList) :-
    Len > 0,
    split_sorted(Len,SortedList,LoList,HiList),
    HiList = [Mid|_],
    (Mid == Ent
     -> true
     ;  LoLen is Len // 2,
        (Ent @< Mid
         -> find(Ent,LoLen,LoList)
         ;  HiLen is Len - LoLen, HiLen > 1,
            find(Ent,HiLen,HiList)
        ) ).

:- table split_sorted/4 as intern.
% Split a sorted list in half (knowing its length)
split_sorted(Len,List,LoList,HiList) :-
    Len1 is Len // 2, split_off(Len1,List,LoList,HiList).

split_off(Len,List,LoList,HiList) :-
    (Len =< 0
     -> LoList = [], HiList = List
     ;  List = [X|List1], LoList = [X|LoList1], Len1 is Len - 1,
        split_off(Len1,List1,LoList1,HiList)  ).

% Query to build a long list of 500,000 elements,
% and look up 100 elements that are not there.
:- import intern_term/2 from machine.
?- mkevenlist(1000000,L0), cputime(T0),
   (intern_term(L0,L), between(1,I,100), I2 is 100*I+1,
    find(I2,500000,L), fail
    ;
    true  ),
   cputime(T1), Time is T1-T0, writeln(cputime(Time)), fail.
```

This query builds a list of even numbers 500,000 elements long starting from 0 and in increasing order. It interns that ground list, and then uses find/3 to use split_sorted/4 to allow it to do a binary search on the list to look up each of 100 odd numbers (of course finding none of them.) The basic work is done by split_sorted/4, which takes a sorted list and its length and produces two lists: the first half of the list, and the second half, so the middle element of the list is the first element in the second list, which split_sorted/4 makes immediately accessible. Since split_sorted/4 is tabled as intern, the lists that split_sorted/4 generates are interned. So no explicit lists are stored in the trie, only pointers to interned ground lists. This query takes 0.2650 seconds, uses 111,464,680 bytes of space for the interned terms and 466,608 bytes of table space to store the calls to and returns from find/4. Without the explicit call to intern_term/2, this query takes approximately the same space, but over 7 seconds of cputime, since it has to intern the list of 500,000 elements for each of the 100 calls to find/3. I didn't try running this query without interning, for what are, I think, obvious reasons.

4 Related Work

Nguyen and Demoen [3] describe the general issue of sharing term representions in the implementation of Prolog. They motivate the advantages of representation sharing and provide effective implementations. They do not consider its potential impact with respect to tabling. They constrain their approach to avoid any change in the standard representation of terms in their implementations, while our approach does change the representation of terms to the extent of interpreting tagged term references outside of the heap to be interned terms. Our unification algorithm does change, minimally, to take advantage of the new representation.

Zhou and Have [7] present an implementation of hash-consing in B-Prolog with goals similar to those of this work: to eliminate an unnecessary extra linear factor in both the time and space complexity of tabling when naive copying of subgoals and answers into and out of tables is done. Their work is clearly prior to this, but their algorithm is somewhat different, using hash tables instead of tries to store tables and requiring and extra optimization of *hash code memoization* to obtain the improved time complexity. It is intimately connected with their implementation of tabling. Our algorithm can be effectively applied when asserting terms. I was motivated to write this paper, since I believe that this implementation is simpler, clearer, and more general and orthogonal to tabling, and deserves consideration as an alternative implementation.

5 Discussion

The approach to representation sharing described in this paper is simple and designed primarily to improve the complexity of tabling on certain kinds of programs. The implementation in XSB is usable, but further effort is necessary to make if fully robust. Foremost, it must be extended to support the expansion of the size of hash tables used to access the interned records. This is straightforward to do. Secondly, I would like to support garbage collection of the interned records. XSB currently supports garbage collection of the atom space. It is not difficult to add to this function the ability to garbage collect the interned space.

As described in Section 2.4, this implementaiton currently don't allow the use of interned tables when interning might compromise indexing. However, there are cases, in particular when using answer subsumption [5], in which loss of indexing would be acceptable. I would like to revisit and further explore this issue. One option that suggests itself, and might be good for other reasons, is to allow interning of specific designated arguments, rather than interning all or none.

References

1. Andrey Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–6, 1958.
2. Eiichi Goto. Monocopy and associative algorithms in an extended lisp. Technical report, University of Tokyo, Tokyo, Japan, May 1974.
3. Phuong-Lan Nguyen and Bart Demoen. Representation sharing for Prolog. *Theory and Practice of Logic Programming*, 13(1):71–106, January 2103.
4. I. V. Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, and David S. Warren. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming*, 38(1):31–54, Jan 1999.
5. Terrance Swift and David S. Warren. Tabling with answer subsumption: Implementation, applications and performance. In *Logics in Artificial Intelligence, 12th European Conference on Logics, JELIA 2010*. Springer, September 2010.
6. David S. Warren, Terrance Swift, and Konstantinos F. Sagonas. The XSB programmer's manual, Version 3.3.x. Technical report, Department of Computer Science, Stony Brook University, Stony Brook, New York, 11794-4400, Mar 2013. The XSB System is available from xsb.sourceforge.net.
7. Neng-Fa Zhou and Christian Theil Have. Efficient tabling of structured data with enhanced hash-consing. *Theory and Practice of Logic Programming*, 12(4–5):547–563, 2012.