IMPERIAL COLLEGE LONDON
Department of Computing

An abductive reasoning system in Java

Michael Ford
mtf09@ic.ac.uk

Supervisor: Dr. Alessandra Russo
Co-Supervisor: Dr. Jiefei Ma

June 19, 2012

Abstract

Abductive reasoning is a powerful logic inference mechanism whereby explanations are computed based on a given set of observations and background knowledge. This power comes from the ability to collect assumptions during answer computation for each query, which makes abduction suitable for reasoning over incomplete knowledge.

Abduction has many possible applications, however its usage is currently constrained by the lack of any implementation in a popular, general-purpose language such as Java and C++. In this project I develop an extensible, self-contained abductive logic programming system on the Java platform. The system features both an API for integration with current software projects, and command line facilities that allow parsing of and reasoning over abductive theories specified in a prolog-like syntax. The challenge in this task is in translating the abductive procedures from the logic-based environment from which it is suited to a more traditional imperative programming environment and deciding on the data structures that should be used for this translation.

In the spirit of open-source development and with the general aim being to make abductive logic programming available outside of academia, the system has been made available on GitHub in order to support further development.

Finally, for evaluation purposes the system has been profiled using existing tools available for the Java platform and also qualitatively and quantitatively compared to an existing implementation of Abductive Logic Programming in Prolog.

Acknowledgements

I would like to thank both my supervisor, Dr. Alessandra Russo and my co-supervisor Dr. Jiefei Ma for providing the idea behind the project and for providing continuous guidance, enthusiasm and motivation throughout its duration. I would also like to thank my second marker Dr. Krysia Broda for her kind words and encouragement and also to the lecturers in the Department of Computing for imparting upon me the knowledge and skills I needed to complete this project. Finally, I would like to thank both my friends and family for their continuous financial and emotional support through my three years at Imperial College London.

Contents

Part I
Introduction

In this chapter we introduce the problem tackled in this project and the difficulties that are inherent in doing so. Section 1 explains the problem and the motivations for implementing a solution. Section 1 is an outline of the main objectives to be completed and a brief summary of the steps taken to develop a solution. Section 1 outlines the main contributions made during the project and finally, Section 1 provides a summary of how this report is organised.

1 Motivation

Abduction is a powerful technique that is suitable for reasoning over incomplete knowledge through the collection of assumptions known as *abducibles*. It can be viewed informally as the inverse of deduction and as the process of finding an explanation for a given set of observations. For example if we *observed* a football moving along the ground, one possible *explanation* could be that someone has just kicked that football.

Abductive Logic Programming (ALP) [KKT92] is the combination of abductive reasoning and Logic Programming. Logic Programming is the use of first order logic (or at least substantial subsets of it) used as a programming language. Combination of abduction and Logic Programming results in an expressive system through which to reason about incomplete knowledge bases and has been used in a variety of real-world applications such as Cognitive Robotics [Sha05], Planning [Esh88][Sha00] and others.

ALP has not had many mainstream applications due in part to its implementations being constrained to Prolog. Both interaction and integration with Prolog is difficult for the uninitiated and hence we believe it will be more likely to be adopted given an available API in a popular language. This is the main motivation for the project and it is hoped that development of an open-source, customisable, extensible implementation will encourage its use in existing software projects on the Java platform.

There are several abductive reasoning algorithms and implementations in existence that can be leveraged in development of an ALP, a number of which I explore in the background section. These implementations are built upon variations of prolog and hence have very little choice in the choice of data structures used.

2 Objectives

The aim of this project was to develop a native abductive logic programming system in Java, leveraging existing algorithms, data structures and APIs available as part of the Java SDK.
The main objectives for this project were as follows. The system must implement all the necessary Logic Programming components needed for ALP which includes unification and both an equality and inequality solver. The system should implement an efficient existing ALP algorithm such as the ASystem as its main component. An API for abductive logic reasoning should be implemented for use on the Java platform as well as tools for dealing with abductive theories and queries over the command line.

3 Contributions

To the best of our knowledge, at the time of writing of this report there have been no implementations of ALP outside of the prolog environment. The final outcome of this project is a fully tested abductive reasoning system on the java platform that takes abductive theories specified in Java or in a prolog-like syntax and queries over those theories. The output is a collection of java objects that form possible explanations for those queries or output to a command line interface in a similar but more user-friendly fashion to the prolog interpreter.

In the spirit of open-source development, and with there being so much potential for integration of extra features, this system has also been made available on GitHub in the hope that development may continue into the future.

The following list summarises the main contributions of this project:

- An implementation of the ASystem state-rewriting algorithm for abductive reasoning.
- Self-contained implementations of both an equality and inequality solver.
- Loosely coupled integration of an existing Java constraint solver to provide Constraint Logic Programming features.
- An API for use in integrating ALP in existing Java projects.
- A command line tool for loading theories from files containing prolog-style code and executing queries upon them.
- An interpreter in the same vein as prolog for specification of theories via a console interface and executing queries upon those theories.
- A visualisation tool for exploring the derivation of explanations.

4 Report Structure

The report is further organised as follows:

- Chapter 2 outlines the relevant work in the fields of Logic Programming, ALP and Constraint Logic Programming as well as the various conventions that are used to make reading and understanding of this report easier.
- Chapter 3 provides a brief introduction to the system implemented, the way in which users can interact with the system and also the technologies used in its development. It is meant as a way in which to set the scene before delving into it's design and implementation.
- Chapter 4 focuses on the design and development of the software, discussing the design goals, exploring the various different options available and explaining the implementation of the various components that work together to form the abductive reasoning system.
- Chapter 5 is the application of qualitative and quantitative profiling and comparisons to existing systems at which point we discuss both the advantages of the system and its limitations at this point in time.
- Chapter 6 summarises the achievements of the project and highlights various future developments that would be advantageous to the system.

- Lastly, the appendix contains various examples of abductive theories that can be used with the system, example abductive derivations, as well as a user guide to both integration with the Java API and use of the command-line tools and visualizer.

Part II
Background

This section summarises the necessary background knowledge for understanding this report as well as introducing the concepts of abduction and abductive logic programming.

5 Logic Programming

The idea of Logic Programming began in the early 1970's [LL87] as a direct outgrowth of work in automatic theorem proving and artificial intelligence. Logic programming is the use of first order logic (or at least substantial subsets of it) used as a programming language.

5.1 Terminology

This section summarises the main terminology and conventions on first-order logic and logic programming used in this report.

A first order logic consists of:

- constants e.g. $\{john, mycar\}$
- variables e.g. $\{X, Y, X4, Var\}$
- function symbols e.g. $\{in/2, length/1\}$
- predicate symbols e.g. $\{stomachBug/1, likes/2\}$
- connectives e.g. $\{\neg, \wedge(,), \vee, \leftarrow\}$
- quantifiers e.g. $\{\forall, \exists\}$

By convention, constants, function names and predicate names are strings starting with a lowercase letter whereas variables are those starting with an uppercase letter.

The signature L of a first order logic consists of a set of constants, function symbols and predicate symbols. Given a signature a *term* is defined as a constant, variable or a function $f(t_1, ..., t_n)$ where f is a function of arity n.

An *atomic formula* or *atom* in short, is a predicate $p(t_1, ..., t_n)$ i.e. a formula with no deeper propositional structure.

Given a signature {L}, a formula is defined as follows:

- If A is an atom then A is a formula.
- If A and B are formulas, then so are $\neg A, A \wedge B, A \vee B, A \rightarrow B$ and $A \leftrightarrow B$
- If A is a formula then so are $\exists X.A$ and $\forall X.A$

A variable X is said to be *quantified* or *bound* by a quantifier $\exists$ or $\forall$. A variable in a formula that is not bound is called a *free variable*. A formula without any free variable is called a closed formula. A closed formula is often called a *sentence*.

A literal L is a *positive literal* or a *negative* literal. A positive literal is an atom A, and a negative literal is the negation of that atom $\neg A$.

A *clause* is a formula of the form $\forall X_1, ..., X_n.(L_1 \vee ... \vee L_m)$, where L is a literal.
Clauses are very common in logic programming and so a special notation is adopted. A clause $\forall X_1, ..., X_n.(A_1 \vee ... \vee A_n, \neg B_1 \vee ... \vee \neg B_n)$ is denoted $A_1, ..., A_n \leftarrow B_1, ..., B_n$. In this notation all variables are assumed to be universally quantified. The commas are used to denote disjunction.

A *definite* clause is a clause of the form $A \leftarrow B_1, ..., B_n$ which contains precisely one atom in the consequent. A is called the head of the clause and $B_1, ..., B_n$ is called the body. . The informal semantics of a definite clause is "for each assignment of each variable, if $B_1, ..., B_n$ is true, then A is true."

A *unit* clause takes the form $A \leftarrow$. Unit clauses are unconditional. The informal semantics of a unit clause is "for each assignment of each variable, A is true. This is sometimes known as a fact if A consists of only constants.

A *normal clause* is a clause whose body may contain negative literals.

A *definite logic program* is a finite set of definite clauses. A *normal logic program* is a finite set of normal clauses. A normal logic program is often simply referred to as a logic program.

A clause is *ground* if it does not contain any variable. A ground instance of a clause is obtained by replacing all of its variables with ground terms.

A *horn clause* is a definite clause or a normal clause. These are often referred to as *rules*.

A denial is a rule without a head and takes the form $\leftarrow L_1, ..., L_n$. A denial with a non-empty body is called an integrity constraint. A denial with an empty body is equivalent to falsity (i.e. $\bot$).

The following conventions are adopted from [NU04] in order to make reading of this report easier:

- A tuple of variables $X_1, X_2, X_3, ...$ is represented as $\bar{X}$
- A tuple of terms $t_1, t_2, t_3, ...$ is represented as $\bar{t}$
- Normally the writing of $\exists$ is avoided and any variables not bound by a universal quantifier are to be assumed existentially quantified rather than free.
- Terms signified by t or s are always existentially quantified.
- Terms signified by u or v can be either free or universally quantified.
- An assignment to a variable X to a term t is signified by X/t

5.2 Unification

Unification is an algorithmic process by which we attempt to show that two terms are identical or equal. [Rob65]. To do this we find substitutions for the variables in the two terms until these are the same. If this cannot be achieved, then the two terms (or atoms) cannot be unified.
In the following, U is a unifier for p and q which are sentences in first order logic.

$$Unify(p, q) = U \ where \ subst(U, p) = subst(U, q)$$

The unification of p and q is the result of substituting U in both sentences.
If L is a set of sentences, U is the most general unifier (m.g.u) iff

$$\forall U'.subst(U', L) = subst(s, subst(U, L))$$

5.2.1 Occurs Check

The occurs check is a function used to avoid the unification algorithm from building infinite structures. [Rob65] It simply checks to see if a variable is contained within the structure of a first-order sentence.

For example, unifying data(X,name(X)) with data(Y,Y) would then match name(X) against name(name(X)). This process would continue into infinity.

Some implementations of unification do not include the occurs check or include it as an optional parameter for efficiency reasons. Prolog for example would return name(name(...)) as a result to signify the infinite structure.

Figure 1 is the occurs check as defined by robinson in java pseudo-code. Given a variable x, a term t and a set of substitutions sub the algorithm iterates over all variables in the term checking to see if any are equal to x. Each variable in t is substituted with any relevant terms in sub. These terms are added to the stack. The algorithm continues until either the stack is empty in which case *true* is returned, or until x is found in the variables of t in which case *false* is returned.

```
// Return true or false
robOccursCheck(Var x, Term t, Substitution sub) {
  Stack stack;
  stack.put(term);

  while (!stack.empty()) {
    t = stack.pop();
    for (Var y:t) {
      if (y==t) {
        return false;
    }
    if (sub.contains(y)) { // y is bound in sub
      stack.push(y.sub(sub));
    }
    }
  }
  return true;
}
```

Figure 1: Occurs Check

5.2.2 Algorithm

The original algorithm was presented in [Rob65] and is currently considered the most efficient unification algorithm and has therefore been integrated into the implementation of JALP. [HV2009].

Figure 2 displays the algorithm is java pseudocode. It takes two parameters s and t which are both terms and returns the computed substitutions. These rules are summarised by way of example in figure 3.

The algorithm uses recursion to delve deeper into the compound terms. If one of the two terms is a variable then a substitution is made from that variable to the term. If the two terms are functions of the same name and arity then the algorithm is recursively called on the component terms of that compound term. An occurs check is performed at each stage to avoid infinite recursion should the case arise.

```
// Unifies s and t or fails.
// Returns null on failure.
rob(Term s, Term t) {
  Stack<{Term, Term}> stack;
  stack.put({s,t});
  Sub sub; // Empty substitution;
  while (!stack.empty()) {
    (s,t) = stack.pop();
    while (sub.contains(s)) s = s.sub(sub);
    while (sub.contains(t)) t = t.sub(sub);
    if (s/=t) {
      if (isVar(s)&&isVar(t)) { // Both variables.
        sub.put(s,t);
      }
      if (isVar(s)&&isTerm(t)) { // Variable and term.
        if (robOccursCheck(s,t,sub)) {
          sub.put(s, t)
        }
        else {
          return null;
        }
      }
      if (isTerm(s)&&isVar(t)) { // Variable and term.
        if (robOccursCheck(s,t,sub)) {
          sub.put(s, t)
        }
        else {
          return null;
        }
      }
      if (isFunc(s)&&isFunc(t)) { // Functions.
        if (s==t) {
          sub.putAll(s.getTerms(),t.getTerms());
        }
        else { // Functions aren't equivalent.
          return null;
        }
      }
    }
  }
  return sub;
}
```

Figure 2: Robinson Unification Algorithm

5.2.3 Examples

Examples are as per conventions explained in the first order logic section.

- $unify(a, a) = \{\}$
  - If two constants are equal, then they unify trivially with no substitutions.
- $unify(a, b) = fail$
  - If two constants are unequal then unification fails.
- $unify(X, X) = \{\}$
  - If two variables are the same then they unify trivially with no substitutions.
- $unify(a, X) = X/a$
  - A variable unified with a constant results in a substitution of that constant to the variable.
- $unify(f(a, X), f(a, b)) = \{X/b\}$
  - The unification of two functions of same name and same arity results in the unification of the matching terms.
- $unify(f(a, X), g(a, b)) = fail$
  - The unification of two functions with different names fails unification.
- $unify(f(X), f(a, b)) = fail$
  - The unification of two functions of different arity results in failure of unification.
- $unify(f(g(X), X), f(Y, a)) = \{X/a, Y/g(a)\}$

Figure 3: Unification Examples

5.3 Resolution

Resolution is a deductive rule of inference, that is, a way in which to derive logical conclusions from premises assumed to be true. [Res12]

Resolution is used as an automated theorem-proving mechanism whereby we prove that a propositional formula or first-order logical sentence is satisfiable via a proof by contradiction.

The general resolution rule can be defined as follows (where formulae are assumed to be represented in disjunctive normal form):

$$\frac{a \vee b, \neg a \vee c}{b \vee c}$$

More generally:

$$\frac{a_1 \vee ... \vee a_i \vee ... \vee a_n, b_1 \vee ... \vee b_i \vee ... \vee b_n}{a_1 \vee ... \vee a_{i-1} \vee ... \vee a_{i+1}, b_1 \vee ... \vee b_{j-1} \vee b_{j+1} \vee ... \vee b_m}$$

where b_i is the complement of a_i.

5.4 SLD Resolution

SLD resolution is the basic inference rule used in logic programming. [JG2003] It is based on resolution and designed to be both sound and refutation complete for horn clause in logic programs.

Given a goal clause:

$$\neg L_1 \vee ... \vee \neg L_i \vee ... \vee \neg L_n$$

with selected literal $\neg L_i$ and an input definite clause:

$$L \vee \neg K_1 \vee ... \vee \neg K_n$$

L unifies with with atom L_i of the selected literal $\neg L_i$ and derives another goal clause where the selected literal is replaced by the negative literals of the input clause and a substitution θ is applied as a result of unification to get:

$$(\neg L_1 \vee ... \vee \neg K_1 \vee ... \vee \neg K_n \vee ... \vee \neg L_n)\theta$$

5.5 Backward Chaining

Backward chaining is an inference method used in logic programming languages that is informally defined as 'working backwards from the goals'. It is implemented using SLD resolution and is defined as follows: [FT12]

$$A \rightarrow B \wedge A \therefore B$$

Given a logic program P and a query Q, we solve Q by:

1. If there is a matching fact Q' in P, return unifier θ where $Q\theta = Q'\theta$
2. For each rule $Q' \leftarrow Q_1, ..., Q_n$ in S whose head G' matches Q, solve the set of new goals $Q_1\theta, ..., Q_n\theta$ where $Q\theta = Q'\theta$
3. Repeat until nothing can be proved.

Figure 4 demonstrates a backward chaining proof for the following example:

The US law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, and enemy of America, has some missiles of type M1, and all of its missiles were sold to it by Colonel West, an American. Show that Colonel West is a criminal [FT12].

5.6 Negation

In first order logic a negative literal is the negation of an atom $\neg A$. If A is false then $\neg A$ is true [LL87]. Negation is an interesting case in logic programming and below I summarize two approaches.

5.6.1 Negation as Failure

Negation as failure (NAF) is an inference rule used in logic programming [LL87]. The notation used for negation as failure is usually *not p* as opposed to $\neg p$ of classical logic. With NAF, *not p* holds if we finitely fail to derive p using a chosen method of reasoning. This is a non-monotonic form of inference i.e. p is assumed not to hold if we cannot derive it.

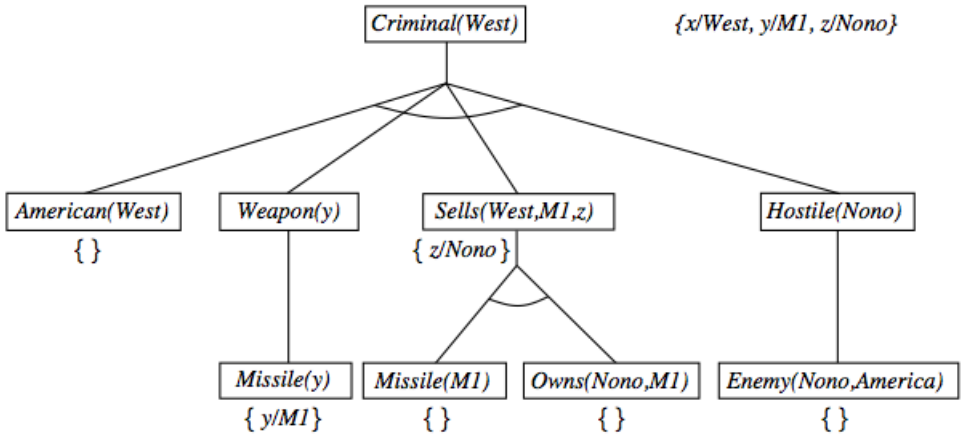


Figure 4: Backward Chaining Example

5.6.2 Constructive Negation

Constructive negation refers to a method of dealing with negation in logic programming whereby rather than proving truth through failure, we prove truth through construction of new logical sentences.

For example, given logic program $P = \{p(X) \leftarrow \ not\ q(X), q(1)\}$, NAF takes $not\ q(X)$ to be true if it cannot find an assignment to X and hence returns false. One method of constructive negation could return $X \neq 1$ which means $not\ q(X)$ is true given that X never takes the assignment 1.

5.7 Prolog

Prolog is a general-purpose logic programming language and also the first of such. [Wam99]. It is now the standard programming environment for logic programming of which there are several implementations such as Sicstus and Yap.

Prolog uses resolution refutation (reductio ad absurdum) with backward chaining and negation as failure to determine whether or not a query succeeds. If an instantiation for all free variables in a query is found when combined with the set of horn clauses in the logic program then the query is a consequence of the logic program and succeeds.

Prolog has a powerful unification mechanism which explains why it has been used to implement abductive logic programming systems in the past. ALP implementations rely heavily on unification in each stage of the proof procedure.

6 Abductive Logic Programming

Charles Peirce identified three types of logical reasoning: [Pei31]

- Deduction: All men are mortal; Socrates is a man; Therefore Socrates is mortal.
- Abduction: A football is rolling across the ground; One possible explanation is that someone kicked it; One possible explanation is that someone rolled it.
- Induction: All the swans we've seen so far are white; All swans are therefore white.

Informally, abduction can be viewed as the reverse process of deduction. Whilst deduction can be used to predict the effects of a given set of causes, abduction can be used to explain effects of the causes. Abduction is hence particularly suited for reasoning about incomplete knowledge.

In other words, abductive reasoning is a form of logical inference whereby given a set of observations O we attempt to explain these observations and arrive at a hypothesis H that consists of various assumptions that explain O.

This project involves developing an abductive logic programming system, that is, when given a query and a logic program, derive logical explanations that would entail the query.

Abductive Logic Programming (ALP) describes a declarative programming approach whereby we write O as a theory consisting of rules (horn clauses), denials and abducible predicates that form the basis of possible hypotheses H. An abductive theory T consists of a tuple (P, A, IC) where P is a set of horn clauses, A is a set of abducible predicates that form the basis of possible explanations and IC is a set of denials that constrain the combinations of abducibles that can be collected as well as assignments to variables in those abducibles.

A goal G refers to the observations that we are seeking an abductive explanation for e.g. $onTable(box_1, T_1) \wedge onTable(box_2, T_2)$.

In ALP, predicates are divided into two disjoint sets: abducible and non-abducible. An abducible predicate is a predicate that can be collected as part of an explanation.

An abductive explanation is a tuple (Δ, θ) such that given a query Q , (Δ, θ) explains Q if: [JM11]

1. Δ is a set of abducible atoms and θ is a set of variable substitutions, i.e. $\Delta\theta \subseteq A$
2. $P \cup \Delta \models Q\theta$
3. $P \cup \Delta \models IC$

The second condition means that the abductive explanation and the background knowledge must be able to prove the query. The third condition means that the abductive explanation and background knowledge must be consistent with the integrity constraints. [JM11]

6.1 Kakas and Mancarella

The Kakas and Mancarella algorithm is one of the earlier attempts at an abductive reasoning algorithm presented in [KM90] as applied to deductive databases.

Given a query Q and an abductive framework A the algorithm iteratively computes an abductive solution Δ and performs consistency checks on whatever is added to this solution. All branches failing indicate success in a consistency check. If Δ is successfully computed then it represents an explanation of the query with regards to P. If not, then there are no possible explanations for Q.

One feature of this proof procedure is that negative literals are treated as abducibles.

Example

$$A = (P, AB, IC)$$

$$P : \{p(x) \leftarrow \neg q(x), q(x) \leftarrow B(x).B(a)\}$$
$$AB : \{B, \neg B, \neg p, \neg q\}$$
$$IC : \{\leftarrow q(x), \neg q(x), \leftarrow p(x), \neg p(x), \leftarrow B(x), \neg B(x)\} \cup \{q(x) \vee \neg q(x), p(x) \vee \neg p(x), B(x) \vee \neg B(x)\}$$
$$Q = p(a)$$

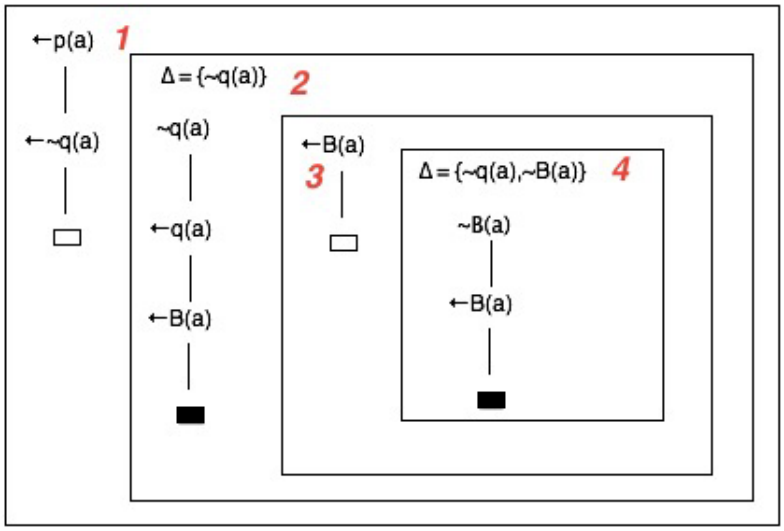


Figure 5: Kakas and Mancerella example

1. Resolve $p(a)$ with P to get $\neg q(a)$
2. Consistency check on $\neg q(a)$. We want $q(a)$ to fail.
   (a) Add $\neg q(a)$ to Δ
   (b) Convert into a denial.
   (c) Expand rule that defines $q(a)$ to get $B(A)$
3. We want the denial of $B(a)$ to succeed to prove failure of $q(a)$ in 2.
4. In which case
   (a) We add $\neg B(a)$ to our solution.
   (b) Perform a consistency check. We've already failed this in 2.

Hence our computed abductive solution is $\Delta = \{\neg q(a), \neg B(a)\}$

6.2 ASystem

ASystem, like K&M is an abductive proof procedure but with numerous advantages. Unlike K&M it allows non-ground abductive solutions and performs constructive negation instead of negation as failure. It also has support for equalities and finite constraints and performs dynamic collection of integrity constraints as an addition to the abductive explanation.
For example given a logic program $P = \{p(X) \leftarrow \neg q(X), q(1)\}$ negation as failure takes $\neg q(X)$ to be tru if it fails to find an assignment X. Because 1 can be assigned X it simply fails. With constructive negation however, $X \neq 1$ will be returned. The inequality will be generated as a

result of transitions between states.

These benefits are achieved through the use of a state-rewriting algorithm, that is, the computation takes the form of a tree. The branches of the tree represent possible state transitions.

6.2.1 Components

The following summarises the main components in the ASystem proof procedure.

ASystem Theory

An ASystem theory is an abductive framework (P, A, IC):

- P is a logic program: a set of horn clauses.
- A is a set of abducibles: a set of predicates that can be used in computing explanations for queries.
- IC is a set of integrity constraints in the form of denials. These constrain the combinations of abducibles that can be used to provide explanations. Additionally all constraints have the special property that the head is an abducible predicate. This is as a result of the sequential application of the inference rules.

ASystem State

An ASystem state is a pair (G, ST) where G is a set of goal formulas derived from the query and the logic program P.

A goal formula is either a conjunction of literals $l_1 \wedge ... \wedge l_n$ or a denial $\forall \bar{X} \leftarrow l_1 \wedge ... \wedge l_n$

ASystem Store

ST is a set of basic formulas known as the store.

A basic formula is a formula that cannot be reduced by any ASystem inference rule.

There are numerous types of basic formula that can be taken as output from an inference rule at each iteration of the ASystem procedure. The store is split into four sub-stores.

- Δ is a set of collected abducibles.
- Δ^* is a set of collected denials.
- ε is a set of collected (in-)equalities.
- FD is a set of collected finite domain constraints.

ASystem Inference Rules

ASystem has a set of inference rules that are iteratively applied to the goal formulas in G to produce new states. These take a goal formula as input and both basic and goal formulas as output.

These are explained in the detail in the next section.

ASystem Query

An query Q is a set of basic and goal formulas for which we are attempting to compute an abductive explanation.

6.2.2 ASystem Derivation

Given an abductive framework (P, A, IC) and a query Q, an ASystem derivation starts from an initial state $S_0 = (\{Q\} \cup IC, ST^{\emptyset}), ST^{\emptyset} = (\emptyset, \emptyset, \emptyset, \emptyset)$.

An ASystem derivation tree is a tree in which every node is a state, the root is S_0 and the children of a node are all states that can be constructed from that node. A leaf node is labeled either as failed (due to an inconsistency) or successful (a solution state).

A derivation fails for a query Q when all leaf nodes are labeled failed. A derivation succeeds when it reaches a solution state.

An answer to query Q is the pair (θ, S_{sol}) where S_{sol} is a leaf node labeled as successful and θ is a set of variable substitutions for all free variables in S_{sol}. Normally a user is most interested in the set of abducible atoms Δ_{sol} and so the response is generally restricted to (θ, Δ_{sol}).

A state progression consists of selecting goal formula from G and then applying an inference rule to obtain the next state (G and/or ST will be modified in someway). If the resultant state of a progression is found to be inconsistent then we mark the node as failed and backtrack and try another state.

6.2.3 ASystem Rules

I now present the inference rules used in the state rewriting of the ASystem derivation tree. The following the conventions are used as per [NU04].

- $G_i^- = G_i - \{F\}$ where F is the selected goal formula from the goal stack G_i of state S_i.
- OR and $SELECT$ are non-deterministic choices in an inference rule i.e. separate branches in the derivation tree.
- Q is a possibly empty conjunction of literals.
- Universal quantification of variable is explicit whereas existential quantification is not.
- Variables denoted by letters s and t will never be universally quantified.
- Variables denoted by letters u and v can be universally quantified.
- $vars(u)$ denotes the possibly empty set of variables in u
- $\bar{X}, \bar{Y}$ denotes a possibly empty set of variables.

Basic Rules The basic rules describe the state transition base cases.

1. Conjunction:
   - $true \wedge Q : G_{i+1} = G_i^- \cup \{Q\}$
   - $false \wedge Q : fail$
   - $\forall \bar{X} \leftarrow true \wedge Q : G_{i+1} = G_i^- \cup \{\forall(\bar{X}).Q\}$
   - $\forall \bar{X} \leftarrow false \wedge Q : G_{i+1} = G_i^-$
2. Superfluous Variables
   - $\forall \bar{X}, Y.\ Q \text{ and } Y \notin vars(Q) : G_{i+1} = G_i^- \cup \{\forall \bar{X}.Q\}$

Defined Predicates These inference rules unfold the bodies of the horn clauses in P and corresponds with standard resolution.

D1

For a positive conjunction rule D1 performs resolution with an arbitrary defined predicate to produce one or more new states i.e. only one of the defined predicates needs to be true.

$p(\bar{t}) \wedge Q$: Let $p(\bar{s}_i) \leftarrow B_i \in P$ $(1, ..., n)$ be n clauses with p in the head.

Then:

$G_{i+1} = G_i^- \cup \{\bar{t} = \bar{s_1} \wedge B_1 \wedge Q\}$ OR ... OR $G_{i+1} = G_i^- \cup \{\bar{t} = \bar{s_n} \wedge B_n \wedge Q\}$

D2

In the denial case D2 creates a new denial for every matched rule head as each and every expansion must lead to an inconsistency for the branch of the derivation tree to succeed.

$\forall \bar{X}. \leftarrow p(\bar{u}) \wedge Q$:

$G_{i+1} = G_i^- \cup \{\forall \bar{X}, \bar{Y}. \leftarrow \bar{u} = \bar{v} \wedge B \wedge Q \mid p(\bar{v}) \leftarrow B \in P \text{ and } \bar{Y} = vars(B) \cup vars(\bar{v})\}$

Examples

$G = \{p(c)\}$ and $P = \{p(X) \leftarrow q(X), p(X) \leftarrow r(X)\}$.

Then:

$G_{i+1} = \{X = c, q(X)\}$ OR $G_{i+1} = \{X = c, r(X)\}$

$G = \{\leftarrow p(c)\}$ and $P = \{p(X) \leftarrow q(X), p(X) \leftarrow r(X)\}$.

Then:

$G_{i+1} = \{\leftarrow X = c, q(X), \leftarrow X = c, r(X)\}$

Abducibles The abducible inference rules are used in the generation of hypotheses and constraints and also in testing consistency between Δ and Δ^*.

A1

Rule A1 generates multiple states. It unifies the abducible with an already collected abducible from Δ or adds the new abducible to the hypothesis whilst ensuring that consistency is maintained through generation of new denials based on those already collect with a matching abducible head.

$a(\bar{t}) \wedge Q$:

SELECT an arbitrary $a(\bar{s}) \in \Delta_i$ such that

$G_{i+1} = G_i^- \cup \{Q\} \cup \{\bar{s} = \bar{t}\}$ OR

$G_{i+1} = G_i^- \cup \{Q\}\{\forall \bar{X}. \leftarrow \bar{u} = \bar{t} \wedge R \mid \forall \bar{X}. \leftarrow a(\bar{u}) \wedge R \in \Delta_i^*\} \cup$

$\{\bar{s} \neq \bar{t} \mid a(\bar{s}) \in \Delta_i\}$ and

$\Delta_{i+1} = \Delta_i \cup \{a(\bar{t})\}$

A2

Rule A2 deals with the denial case by creating a constraint and checking that no abducibles that have already been collected break this new constraint.

$\forall \bar{X}. \leftarrow a(\bar{u}) \wedge Q$:

$G_{i+1} = G_i^- \cup \{\forall \bar{X}. \leftarrow \bar{s} = \bar{u} \wedge Q \mid a(\bar{s}) \in \Delta_i\}$ *and*

$\Delta^*_{i+1} = \Delta^*_i \cup \{\forall \bar{X}. \leftarrow a(\bar{u}) \wedge Q\}$

Examples

$G = \{p(X, Y, Z)\}$ and $\Delta = \{p(a, b, c)\}$

Then:

$G_{i+1} = \{X = a, X = b, Z = c\}$, $\Delta = \{p(a, b, c)\}$

OR

$G_{i+1} = \{\leftarrow X = a, X = b, Z = c\}$, $\Delta = \{p(a, b, c), p(X, Y, Z)\}$

$G = \{\leftarrow p(X, Y, Z)\}$ and $\Delta = \{p(a, b, c)\}$ and $\Delta^* = \{\}$

Then:

$G_{i+1} = \{\leftarrow X = a, X = b, Z = c\}$, $\Delta = \{p(a, b, c)\}$ and $\Delta^* = \{\leftarrow p(X, Y, Z)\}$

Negations The negation inference rules simply convert a negative literal into a denial and hence allows the use of constructive negation.

N1

$\neg p(\bar{t}) \wedge Q$:

$G_{i+1} = G_i^- \cup \{Q, \leftarrow p(\bar{t}) \notin \bar{X}\}$

N2

$\forall \bar{X}. \leftarrow \neg p(\bar{t}) \wedge Q$ where $vars(\bar{t}) \notin \bar{X}$:

$G_{i+1} = G_i^- \cup p(\bar{t})$ *OR* $G_{i+1} = G_i^- \cup \{\neg p(\bar{t}), \forall \bar{X}. \leftarrow Q\}$

Examples

$G = \{not\ p(X, Y, Z)\}$

Then:

$G_{i+1} = \{\leftarrow p(X, Y, Z)\}$

$G = \{\leftarrow not\ p(X, Y, Z), q(X)\}$

Then:

$G_{i+1} = \{p(X, Y, Z)\}$ OR $G_{i+1} = \{not\ p(X, Y, Z), \leftarrow q(X)\}$

Equalities The equality inference rules isolate the (in)equalities so that they can be evaluated by the equality solver which must deal with both universally and existentially quantified variable assignments.

E1

Rule E1 collects the equality and performs any unifications. Variables in positive literals are always existentially quantified.

$s = t \wedge Q$:

$G_{i+1} = G_i^- \cup \{Q\}$ *and* $\varepsilon_{i+1} = \varepsilon_i \cup \{s = t\}$

E2

Rule E2 deals with the denial case. The equality is reduced to the equational solved form by the equality solver which is then dealt with by a base case.

$\forall \bar{X}. \leftarrow v = u \wedge Q :$

$G_{i+1} = G_i^- \cup \{\forall \bar{X}. \leftarrow E_s \wedge Q\}$ where E_s is the equational solved from of $v = u$

This rule reduces the denial to the basic case:

E2b

Rule E.2.b deals with an equality between a variable and some existentially or universally quantified term. It produces two possible states. Either the equality is false and we collect the inequality and the containing denial succeeds or we perform a variable assignment and must find another element of the denial to fail.

$\forall \bar{X}, \bar{Y}. \leftarrow Z = u \wedge Q$ where $Z \notin \bar{X} \cup \bar{Y}$ and $vars(u) \subseteq \bar{Y} :$

$\varepsilon_{i+1} = \varepsilon_i \cup \{\forall \bar{Y}. Z \neq u\}$ OR

$G_{i+1} = G_i^- \cup \{\forall \bar{X}, \bar{Y}. \leftarrow Q(Z/u)\}$

A special case of this rule is:

E2c

Rule E.2.c is a special case of E2b whereby u is a universally quantified variable.

$\forall \bar{X}, Y. \leftarrow Z = Y \wedge Q$ where $Z \notin \bar{X} :$

$G_{i+1} = G_i^- \cup \{\forall \bar{X}. \leftarrow Q(Y/Z)\}$

Examples

$G = \{X = u\}$ and $\varepsilon = \{\}$

Then

$G_{i+1} = \{\}$ and $\varepsilon = \{X = u\}$

$G = \{\forall X. \leftarrow p(X) = p(Y)\}$

Then

$G_{i+1} = \{\forall X. \leftarrow X = Y\}\}$

$G = \{\leftarrow X = p(Y), q(Y)\}$

Then

$G_{i+1} = \{X \neq p(Y)\}\}$ OR $G_{i+1} = \{\leftarrow q(Y)\}, \theta = \{X/p(Y)\}$

$G = \{\forall Y. \leftarrow X = Y, q(Y)\}$

Then

$G_{i+1} = \{\leftarrow q(X)\}$

Finite-Domain Constraints The finite domain rules are similar to the equality rules in that they simply isolate any collected constraints so that they can be evaluated by the constraint solver.

F1

$c(\bar{t}) \wedge Q :$

$G_{i+1} = G_i^- \cup \{Q\}$ *and* $FD_i + 1 = FD_i \cup \{c(\bar{t})\}$

F2

$\forall \bar{X}. \leftarrow c(\bar{t}) \wedge Q$ where $vars(\bar{t}) \not\subseteq \bar{X} :$

$FD_{i+1} = FD_i \cup \{\neg c(\bar{t})\}$ *OR*

$FD_{i+1} = FD_i \cup \{c(\bar{t})\}$ *and* $G_{i+1} = G_i^- \cup \{\forall \bar{X}. \leftarrow Q)\}$

Examples

$G = \{X < Y, p(X)\}$

Then:

$G_{i+1} = \{p(X)\}$ and $FD = \{X < Y\}$

$G = \{\leftarrow X < Y, p(X)\}$

Then:

$G_{i+1} = \{\leftarrow p(X)\}$ and $FD = \{X < Y\}$ OR $G_{i+1} = \{\}$ and $FD = \{X >= Y\}$

Additional Inequality Rules These rules allow collection of inequalities to the equality store which are then evaluated by the inequality solver.

E1

$s \neq t \wedge Q :$

$G_{i+1} = G_i^- \cup \{Q\}$ *and* $\varepsilon_{i+1} = \varepsilon_i \cup \{s \neq t\}$

E2

$\forall \bar{X}. \leftarrow s \neq t \wedge Q : vars(s) \cup vars(t) \not\subseteq \bar{X}$

$G_{i+1} = G_i^- \cup \{s = t\}$ OR

$G_{i+1} = G_i^- \cup \{\forall \bar{X}. \leftarrow Q\} \cup s \neq t$

Examples

$G = \{X \neq Y, p(X)\}$

Then:

$G_{i+1} = \{p(X)\}$ and $\varepsilon = \{X \neq Y\}$

$G = \{\leftarrow X \neq Y, p(X)\}$

Then:

$G_{i+1} = \{X = Y\}$ OR $G_{i+1} = \{X \neq Y, \leftarrow p(X)\}$

Additional Finite Domain Constraint Rules Rules F1 and F2 are general rules that apply to any constraint expression. This additional inference rule describes the case whereby a constraint constrains a universally quantified variable Y which has a domain determined by the constraint $c[Y]$. $Domain(Y)\downarrow_c$ denotes that the domain of Y is determined by c.

F2b

$\forall \bar{X}, Y. \leftarrow c[Y] \wedge Q : vars(c) - \{Y\} \not\subset \bar{X}$ and $Domain(Y)\downarrow_c$ is bounded.

$G_{i+1} = G_i^- \cup \{\forall \bar{X}.(\leftarrow c \wedge Q)(Y/d) | d \in Domain(Y)\downarrow_c\}$

Examples

$G = \{\forall Y. \leftarrow Y \ in \ [1,2,3], p(Y)\}$

Then:

$G = \{\leftarrow p(1)\}$ OR $G = \{\leftarrow p(2)\}$ OR $G = \{\leftarrow p(3)\}$

6.2.4 ASystem Example

Below is a simple example logic program P describing the types of birds that can fly.

```
flies(X) :- bird(X), not abnormal(X).

abnormal(X) :- penguin(X).

bird(X) :- penguin(X).
bird(X) :- eagle(X).

penguin(tweety).
eagle(sam).
```

Given a query $Q = \{flies(X)\}$ we are attempting to find an explanation, that is, an assignment to X such that X is a bird that can fly.

The initial state is:

$$G = \{flies(X)\}, ST = \{\emptyset, \emptyset, \emptyset\}$$

The first rule we apply is D1 to obtain:

$$G = \{bird(X), \neg abnormal(X)\}, ST = \{\emptyset, \emptyset, \emptyset\}$$

We then apply D1 again. D1 generates two possible states here due to there being two definitions for bird. We will take the eagle branch.

$$G = \{eagle(X), \neg abnormal(X)\}, ST = \{\emptyset, \emptyset, \emptyset\}$$

We then apply D1 a third time.

$$G = \{X = sam, \neg abnormal(X)\}, ST = \{\emptyset, \emptyset, \emptyset\}$$

And now E1 which results in an assignment to X.

$$G = \{\neg abnormal(sam)\}, ST = \{\emptyset, \emptyset, \{X = sam\}\}$$

We must now check that sam is not abnormal. This an application of N1.

$$G = \{\leftarrow abnormal(sam)\}, ST = \{\emptyset, \emptyset, \{X = sam\}\}$$

We then apply D2 to obtain:

$$G = \{\leftarrow X' = sam, penguin(X)\}, ST = \{\emptyset, \emptyset, \{X = sam\}\}$$

And E2 to obtain:

$$G = \{\leftarrow penguin(sam)\}, ST = \{\emptyset, \emptyset, \{X = sam\}\}$$

And D2 to obtain:

$$G = \{\leftarrow tweety = sam\}, ST = \{\emptyset, \emptyset, \{X = sam\}\}$$

And E2 to obtain:

$$G = \{\leftarrow \bot\}, ST = \{\emptyset, \emptyset, \{X = sam\}\}$$

And a basic rule to obtain

$$G = \{\top\}, ST = \{\emptyset, \emptyset, \{X = sam\}\}$$

And hence we succeed with an explanation $\{X = sam\}$. More examples can be found in the appendix.

7 Constraint Logic Programming

Constraint programming [Bar99] is a form of declarative programming that involves expressing the relations between variables through the use of constraints. A constraint can be an arithmetic constraint such as $X > Y$ and $T = T1 + 5$ or constraints connected by boolean connectives such as $(X > 4) \wedge (X < 6)$ and $\neg(X - Y \geq 2) \vee (Y < X)$. Given a set of such constraints, the problem of finding the numerical assignments to the variables so that all constraints are true is called the constraint satisfaction problem.

These constraints are often embedded within other programming languages and are common in logic programming implementations. The integration of constraint programming into logic programming is known as constraint logic programming. [JM94] In CLP, a rule takes the form $H \leftarrow L_1, ..., L_n, C_1, ..., C_n$, where $L_1, ..., L_n$ are literals and $C_1, ..., C_n$ are constraints.

For example prolog is able to support constraints on variables that exist within the bodies of rules e.g. $p(X) \leftarrow X \geq 3, X \leq 10$. which constrains the value of X to the domain $\{3, 4, ..., 10\}$. List constraints are also often supported such as $p(X) \leftarrow X \ in \ [3, 4, 5]$ which is equivalent to defining the facts $p(1), p(2), p(3)$.

Prolog has a native implementation of a constraint solver that handles these cases. As an extension to this project I will investigate integration of existing constraint solver APIs written in Java.

As an example see the block world example in the appendix. When this is successfully executed using ASystem inference rules it results in the following constraints in the finite domain store:

$$\{T, E_1, E_2, E_3, E_4 \ in \ [1..6], E_1 < T, E_3 < T, E_4 < T, E_2 < E_1, E_3 < E_2, E_2 < E_4, E_1 < E_4\}$$

Execution of the constraint solver results in the following substitutions θ:

- $\{E_1/3, E_2/2, E_3/1, E_4/4, T/5\}$
- $\{E_1/3, E_2/2, E_3/1, E_4/4, T/6\}$
- $\{E_1/3, E_2/2, E_3/1, E_4/5, T/6\}$
- $\{E_1/4, E_2/2, E_3/1, E_4/5, T/5\}$
- $\{E_1/4, E_2/3, E_3/1, E_4/5, T/6\}$
- $\{E_1/4, E_2/3, E_3/2, E_4/5, T/6\}$

In the ASystem, these constraints are collected during the state rewriting by inference rules F1, F2 or used in creation of a choice based on substitutions by F2b.

Part III
JALP, Java Abductive Logic Programming

This section provides a brief overview of the implementation which I have named Java Abductive Logic Programming and sets the scene before I delve into the design details in the next section. Whilst this section covers the basics, more detailed usage of the system and its components can be found in the user guide.

JALP is an abductive logic programming system that consists of several components. It has a state rewriting system that constructs a derivation tree that acts as a *proof* of the explanation. It also features an integrated equality solver, inequality solver and a finite domain constraint solver which can used separately from the system. A Java API has also been made available for integration with other software projects as well as command line tools for parsing abductive theories specified in a prolog-like syntax which means previous prolog users will find the system intuitive to use.

8 Shell

Theories and queries can be executed via the shell for example:

```
$ java -jar jalp.jar examples/fact/one-fact.alp -q likes(X,Y)
$ Loading basic/fact/one-fact.alp
$ Computed 1 explanation in 23429 microseconds.
$
$ Query
$  likes(X,Y)
$ Substitutions
$  Y/jane
$  X/john
$
Exiting...
```

9 Interpreter

Users can load theories from files and run queries from a command line interface. Alternatively components of the theory can be asserted step by step e.g.

```
$ java -jar jalp.jar
Welcome to JALP. Type :h for help.
JALP->likes(john,jane).
JALP->likes(john,maria).
JALP->:q likes(X,Y).
Computed 2 explanations in 29858 microseconds.

Query
  likes(X,Y)
Substitutions
  Y/maria
```

```
  X/john

There are 1 results remaining. See next? (y/n): n
JALP->
```

10 Visualizer

The visualizer uses the JSON representation of the derivation tree to construct an interactive graphical tree. This was developed mainly as a debug tool but can be useful in understanding the system. Figure 6 shows a screenshot of the visualizer.

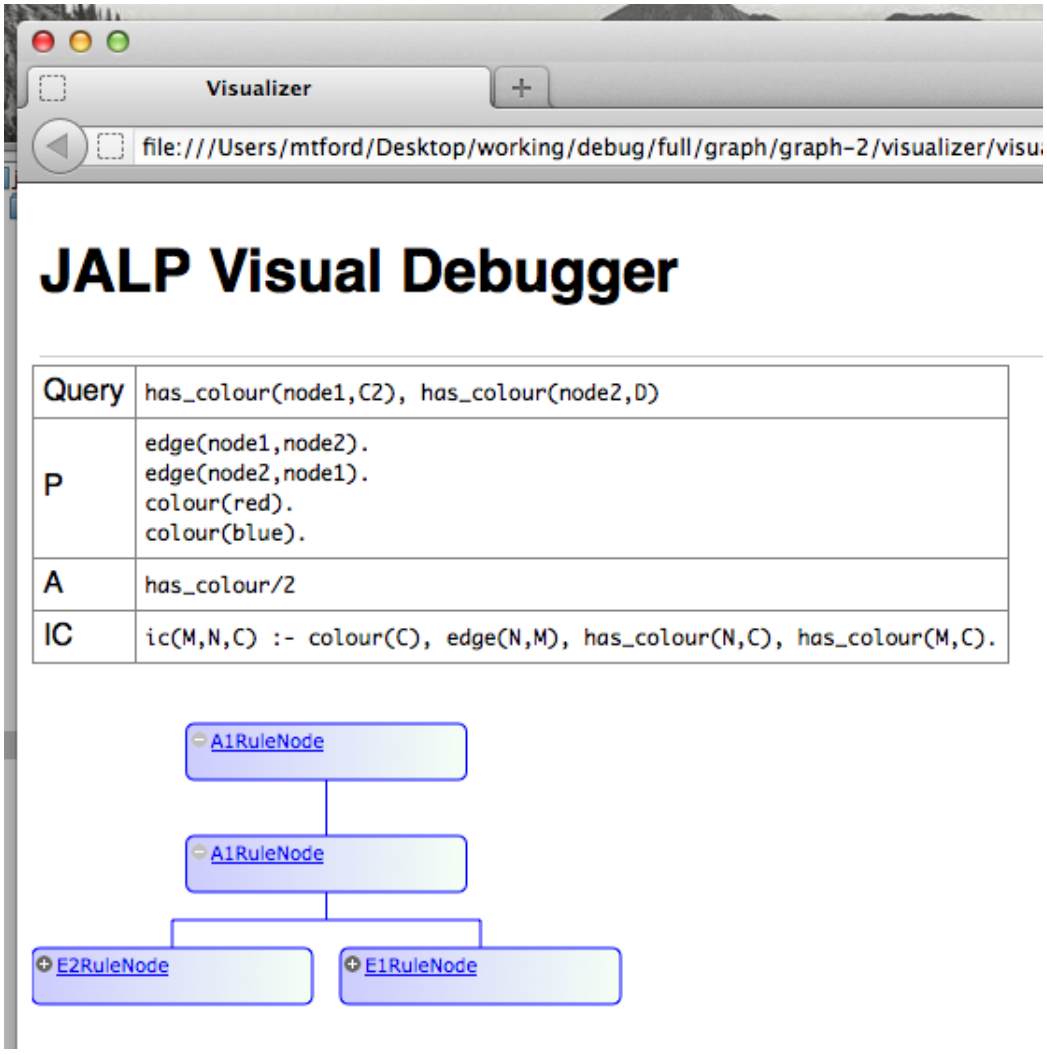


Figure 6: Visualizer Screenshot

11 Syntax

The syntax is based on prolog and consists of:

- Rules e.g. $likes(X, Y) :- boy(X), girl(Y)$.
- Facts e.g. $colour(red)$.
- Integrity constraints e.g. $ic :- migraine(X), not\ headache(X)$.
- Abducibles e.g. $abducible(has\_colour(N, C))$.

For example a simple graph colouring theory with 2 nodes is as follows:

```
edge(node1,node2).
edge(node2,node1).

colour(red).
colour(blue).

ic :- colour(C), has_colour(N,C), edge(N,M), has_colour(M,C).

abducible(has_colour/2).
```

12 Technologies Used

- Java: Implementation language.
- JUnit: Unit testing.
- log4j: Logging.
- HTML/CSS/Javascript: Visualizer.
- JSON: Visualizer representation of derivation tree.
- ECOTree.js: HTML5 tree widget.
- Choco: Finite Domain Constraint Solver
- Javacc: Lexical/Parser generator.
- JProfiler: Java profiling tool.

Part IV
JALP Design

13 Design Aims

The use of java as opposed to declarative languages such as prolog means that it is possible to leverage extensive libraries and also object-oriented design features. In this section I state the design criteria I followed during development of JALP.

The system should be *modular*. Components and implementations should be easily modified, removed and changed. Components that have been integrated from external API's should be loosely coupled with the system so as to avoid dependencies and provide flexibility for future developers.

The system should be *extensible*. This ties in with modularity and loose coupling. There should be room for future growth of the system. Since the system is going to be open-source this is particularly important to future developers.

The system should be as *efficient* as possible, aiming for comparable efficiency to existing implementations in Prolog if possible. An advantage in using Java is access to mature libraries of data structures which have been steadily improved over the years.

The system should remain faithful to the logic programming terminology in again to make future development of JALP easier for those familiar with the topic.

The system should be *user friendly*. It should provide an easy to use API that abstracts and encapsulates implementation details. It should allow command line interaction for general use and provide as familiar a programming environment as possible e.g. through using similar syntax to prolog.

The system should be *robust*. Thorough usage of error reporting, logging and unit testing is necessary for ensuring that the system remains robust in the future.

14 Use Cases

Figure 7 shows a use case diagram for the JALP system. Users must be able to load theories from files or enter them manually. Users must be able to view the result of each query as a graphical tree, in some form of external data representation (XDR) or simply as output to the console. Users must be able to integrate the system in existing Java projects.

15 Grammar

As shown in figure 7, one requirement was for users to specify through a prolog-like syntax over a command-line interface/interpreter. Of course, these theories need to parsed, checked for syntax and converted into an internal Java representation. To save time I decided on the use of a tokenizer/parser generator. By far the most popular choice for java developers is JavaCC or 'Java Compiler Compiler'.

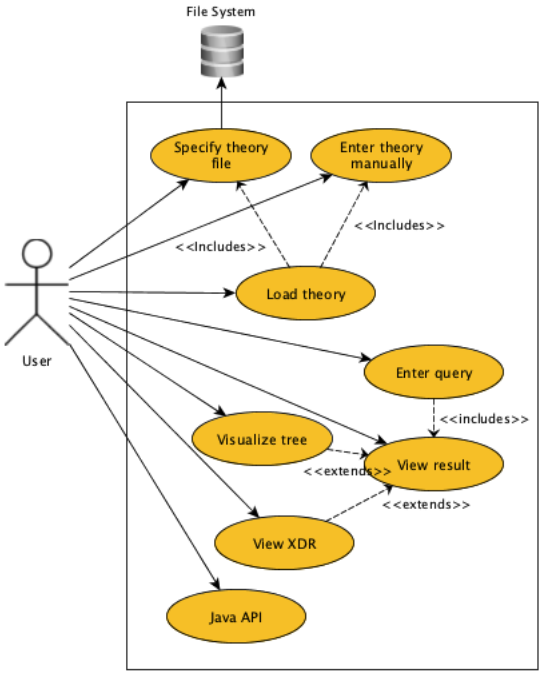


Figure 7: Use Case Diagram

JavaCC generates top-down parsers limited to the $LL(k)$ class of grammars where k is the token lookahead. Lookahead is used where there are conflicts in the grammar for example both constants and predicates start with a lower case name. Input is parsed from left to right and constructs a left most derivation of the sentence. The main limiting factor with regards to LL(k) grammars is that left recursion must be avoided. This means that the grammar can often become complex. For example the following grammar describes an inferable i.e. a logical structure to which we can apply an ASystem inference rule:

```
Equality -> Parameter EQUALS Parameter

Inferable -> PosInferable | NegInferable
PosInferable -> Predicate | Equality | InEquality | Constraint
NegInferable -> NOT PosInferable
```

This had to be changed to:

```
Equality -> Parameter EQUALS Parameter

Inferable -> PosInferable | NegInferable
PosInferable -> PredOrEqualOrInequal | EqualOrInequal | Constraint
NegInferable -> NOT PosInferable

PredicateOrEqualOrInequal -> Predicate (EQUALS Parameter | NOTEQUALS Parameter)?
EqualOrInequal -> (Variable | Constant) (EQUALS Parameter | NOTEQUALS Parameter)
```

in order for the parser to accept equalities involving predicates due to the conflict between lower case names of constants and the lower case names of predicates.

I chose JavaCC for two reasons, the first is plenty of previous experience with its usage and the second the breadth of documentation and examples available.
Figure 8 shows the LL(2) grammar used for the theory syntax where 2 is the token lookahead.
Figure 9 shows the grammar for specifying queries either via the command line or interpreter.
A JavaCC specification consists of tokens and productions. Strings of characters are combined into tokens by the JavaCC tokenizer. These tokens are specified as regular expressions, for example:

```
  <LBRACKET: "("> |
  <LCASENAME: ["a"-"z"] ( ["a"-"z","A"-"Z","_","0"-"9"] )*> |
  <DEFINES: ":-"> |
```

Productions are then expanded using these tokens. Java objects representing the logic program are then constructed as a side-effect of these expansions. For example:

```
IntegerConstantInstance IntegerConstant():
{
    Token t;
    Integer integer;
}
{
    t = <INTEGER>
    {
        integer = Integer.parseInt(t.image);
        return new IntegerConstantInstance(integer);
    }
}
```

16 Unification

To emulate the pattern matching provided by functional and declarative programming languages, Unification is implemented in a visitor-like fashion. An IUnifiableInstance is an interface implemented by terms and predicates (atoms) to imply that they can be unified. The visitor-style call backs (via visit and acceptVisitor) are a necessary design feature due to the ambiguity of the class structure e.g. IUnifiableInstance unified with an IUnifiableInstance.

Figure 11 is a UML communications diagram showing the process by which a unification occurs and shows a unification between two variable instances, u_1 and u_2 as requested by a client object. u_1 sends a request for a callback to u_2 so that the type of u_2 can be determined by u_1. u_2 calls the unify(VariableInstance, subst) method of u_1 with itself, and the unification takes place. The use of this callback mechanism meant avoidance of usage of the much disliked instanceof operators.

The implementation of unification in an object-oriented manner is an example of modularity and reusability. The Java API allows construction of these objects external to any abductive framework and so could be used by users for other purposes than in the JALP System.

Another design decision with regards to unification was the avoidance of an occurs check. The occurs check is considered to be inefficient and since unification is used heavily by the ASystem inference rules implemented, I made the decision to perform runtime checks with a timeout when unifying rather than checking the formulas at every opportunity.

```
ALP -> ((Rule | Denial | Abducible))* EOF
Rule -> Predicate (DEFINES Body)? DOT
Denial -> IC DEFINES Body DOT
Body -> Inferable (COMMA Inferable)*
Abducible -> ABDUCIBLE LBRACKET UCASENAME SLASH INTEGER RBRACKET DOT

Inferable -> PosInferable | NegInferable
PosInferable -> PredOrEqualOrInequal | EqualOrInequal | Constraint
NegInferable -> NOT PosInferable

PredicateOrEqualOrInequal -> Predicate (EQUALS Parameter | NOTEQUALS Parameter)?
EqualOrInequal -> (Variable | Constant) (EQUALS Parameter | NOTEQUALS Parameter)

Predicate -> LCASENAME LBRACKET ParameterList RBRACKET
Equality -> Parameter EQUALS Parameter
InEquality -> Parameter NOTEQUALS Parameter
Constraint -> LessThanConstraint | LessThanEqConstraint | GreaterThanConstraint | GreaterThanEqCons
LessThanConstraint -> Term LESSTHAN Term
LessThanEqConstraint -> Term LESSTHANEQ Term
GreaterThanConstraint -> Term GREATERTHAN Term
GreaterThanEqConstraint -> Term GREATERTHANEQ Term
InListConstraint -> Term IN List

ParameterList -> Parameter (COMMA Parameter)*

Term -> (Variable | Constant | List)
Parameter -> (Variable | Constant | Predicate)

List -> IntegerConstantList | CharConstantList
IntegerConstantList -> LSQBRACKET IntegerConstant ... IntegerConstant RSQBRACKET
CharConstantList -> LSQBRACKET CharConstant (COMMA CharConstant)* RSQBRACKET

Variable -> UCASENAME
IntegerConstant -> INTEGER
CharConstant -> LCASENAME
```

Additionally, comments are also supported via the use of Javacc's skip feature which means all white space and newlines are also skipped. Comments start with the % character.

Figure 8: JALP Theory Grammar

```
Query -> Inferable (COMMA Inferable)* EOF

Inferable -> PosInferable | NegInferable
PosInferable -> PredOrEqualOrInequal | EqualOrInequal | Constraint
NegInferable -> NOT PosInferable

PredicateOrEqualOrInequal -> Predicate (EQUALS Parameter | NOTEQUALS Parameter)?
EqualOrInequal -> (Variable | Constant) (EQUALS Parameter | NOTEQUALS Parameter)

Predicate -> LCASENAME (LBRACKET ParameterList RBRACKET)?
Equality -> Parameter EQUALS Parameter
InEquality -> Parameter NOTEQUALS Parameter
Constraint -> LessThanConstraint | LessThanEqConstraint | GreaterThanConstraint | GreaterThanEqCons
LessThanConstraint -> Term LESSTHAN Term
LessThanEqConstraint -> Term LESSTHANEQ Term
GreaterThanConstraint -> Term GREATERTHAN Term
GreaterThanEqConstraint -> Term GREATERTHANEQ Term
InListConstraint -> Term IN List

Term -> (Variable | Constant | List)
Parameter -> (Variable | Constant | Predicate)

List -> IntegerConstantList | CharConstantList
IntegerConstantList -> LSQBRACKET IntegerConstant ... IntegerConstant RSQBRACKET
CharConstantList -> LSQBRACKET CharConstant (COMMA CharConstant)* RSQBRACKET

Variable -> UCASENAME
IntegerConstant -> INTEGER
CharConstant -> LCASENAME
```

Figure 9: JALP Query Grammar

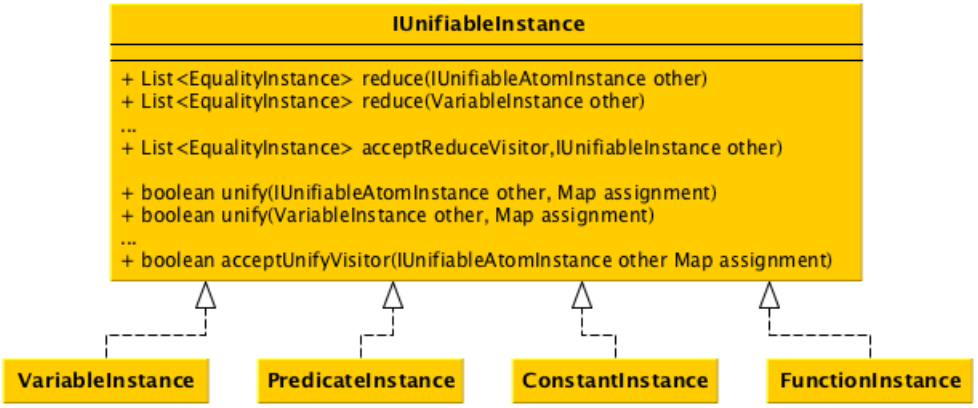


Figure 10: Unification UML Diagram

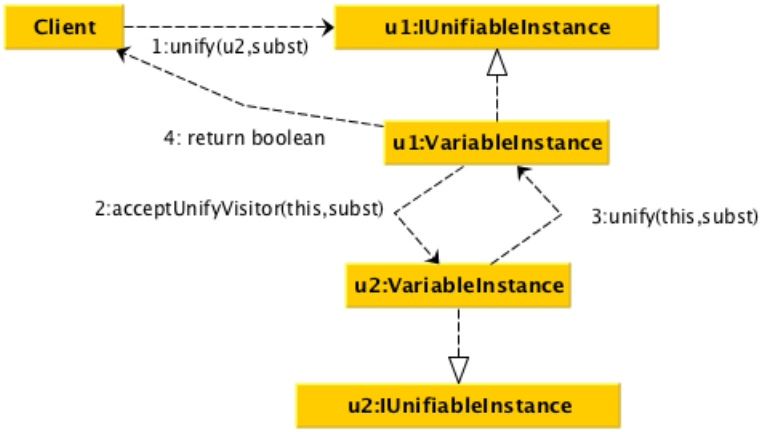


Figure 11: Unification Communication Diagram

17 Equality Solver

The equality solver is an important aspect in JALP due to the ASystem algorithms reliance on unfication. All rules except F1,F2 and F2b either produce or resolve equalities and hence the equality solver must be robust and efficient. The equality solver used in JALP is based on the robinson unification algorithm and leverages the object-oriented unification system described in the previous section.

17.1 Rules

The following rules are used in evaluating equalities collected by the equality inference rules E1 and E2 and stored in the equality store ε. The rules use the following conventions:

- f and g are different predicates/functors.
- X and Y are variables.
- s and t are terms.
- $\bar{s}$ and $\bar{t}$ are tuples of terms.

- $\perp$ means failure and will involve 'backtracking' to a previous state i.e. a previous choice point in the tree.
- $\top$ means success and no further evaluation is necessary.

The standard rules described in figure 12 deal with equalities in the positive context i.e. those equalities collected by inference rule E1. The denial rules described in figure 13 deal with the negative context i.e. those equalities collected by inference rules E2, E2b and E2c. Each rule is applied iteratively and reduce equalities to Equational Solved Form E_s which is an equality of the form $\forall X.Z = u$.

Equations in E_s are not dealt with by the equality solver but by an extension to these rules in the form of the base case inference rules E2b and E2c. These base case rules correctly deal with existentially quantified variables. The equality solver contains no concept of quantification and in fact assumes universal quantification. Figure 14 shows the rules implemented by E2b and E2c. The current implementation has tried to stay as faithful to the ASystem structure as possible but this has resulted in the various elements that deal with equalities being fragmented between the EqualityInstances, the RuleNodeVisitor and the EqualitySolver which is possibly undesirable from a design point of view. Future versions of JALP could integrate these rules into the equality solver and hence provide it with the concept of quantification.

- $c = d \mapsto \perp$
- $c = c \mapsto \top$
- $f(\bar{s}) = f(\bar{t}) \mapsto \bar{s} = \bar{t}$
- $f(\bar{s}) = g(\bar{t}) \mapsto \perp$
- $X = t \mapsto \perp$ when $X \in vars(t)$
- $X = t \mapsto \sigma(X/t)$ when $X \notin vars(t)$

Figure 12: Standard Equality Rules

17.2 Implementation

This section discusses specific implementation details of the equality solver in JALP.

- $\forall \bar{X}. \leftarrow c = c \wedge Q \mapsto \forall \bar{X}. \leftarrow Q$
- $\forall \bar{X}. \leftarrow c = d \wedge Q \mapsto \top$
- $\forall \bar{X}. \leftarrow f(\bar{s}) = f(\bar{t}) \wedge Q \mapsto \forall \bar{X}. \leftarrow \bar{s} = \bar{t} \wedge Q$
- $\forall \bar{X}. \leftarrow f(\bar{s}) = g(\bar{t}) \wedge Q \mapsto \top$
- $\forall \bar{X}, X. \leftarrow X = t \wedge Q \mapsto \top$ when $X \in vars(t)$
- $\forall \bar{X}, X. \leftarrow X = t \wedge Q \mapsto \forall \bar{X}. \leftarrow Q(X/t)$ when $X \notin vars(t)$

Figure 13: Denial Equality Rules

- $\forall \bar{X}, \bar{Y} \exists Z. \leftarrow Z = u \wedge Q \mapsto Z \neq u \vee \forall \bar{X}, \bar{Y}. \leftarrow Q(Z/u)$
- $\forall \bar{X}, Y \exists Z. \leftarrow Z = Y \wedge Q \mapsto \forall \bar{X}. \leftarrow Q(Y/Z)$

Figure 14: Existentially Quantified Denial Rules

With respect to my design goals in this project, the equality solver is implemented in a modular, re-usable fashion. Like the unification implementation the equality solver can be used external to the abductive system; It simply requires the construction of an EqualitySolver instance through which we then pass EqualityInstances. Alternatively an EqualityInstance can execute equalitySolve directly to produce a list of possible substitutions that are created as a result of application of the equality solver rules. More details with regards to this are presented in the user guide.

The equality solver is integrated as part of the inference rules, that is, it is executed by the RuleNodeVisitor that is responsible for expanding the derivation tree. Nodes in the derivation tree are marked as successful, failed or expanded based on whether or not (amongst other things) the equality solver returns true. This system is discussed in detail in the state rewriting section.

Figure 15 is a UML communications diagram that shows a typical call to the equality solver. The client object (a RuleNodeVisitor in this case) calls the execute() method in the equality solver. This node then consults its store to obtain the equalities. These equalities each have an equality solve method which is sequentially called producing substitutions before returning true or false based on successful application of the rules.

The main design issue with the equality solver was how often to apply it and what to do with processed equalities. Applying at the end of a derivation process would result in the exploration of a much bigger state space but would reduce the costs involved in repeated unification. Applying at each derivation stage would reduce the state space explored, but cost more in unification and applications of substitutions. Without heuristics the latter choice is unsustainable and so I opted to execution of the equality solver as and when equalities are collected to the equality store.

18 Inequality Solver

The introduction of inequalities by the rules and the abductive theory means that the system needs additional logic in order to deal with them. This leads to the idea of an inequality solver and also led to the introduction of further equality inference rules for dealing with inequalities as mentioned in the background section on ASystem. The inequality rules are stated again below for clarity's sake.

E1

$s \neq t \wedge Q :$
$G_{i+1} = G_i^- \cup \{Q\}$ *and* $\varepsilon_{i+1} = \varepsilon_i \cup \{s \neq t\}$

E2

$\forall \bar{X}. \leftarrow s \neq t \wedge Q : vars(s) \cup vars(t) \not\subseteq \bar{X}$
$G_{i+1} = G_i^- \cup \{s = t\}$ OR
$G_{i+1} = G_i^- \cup \{\forall \bar{X}. \leftarrow Q\} \cup s \neq t$

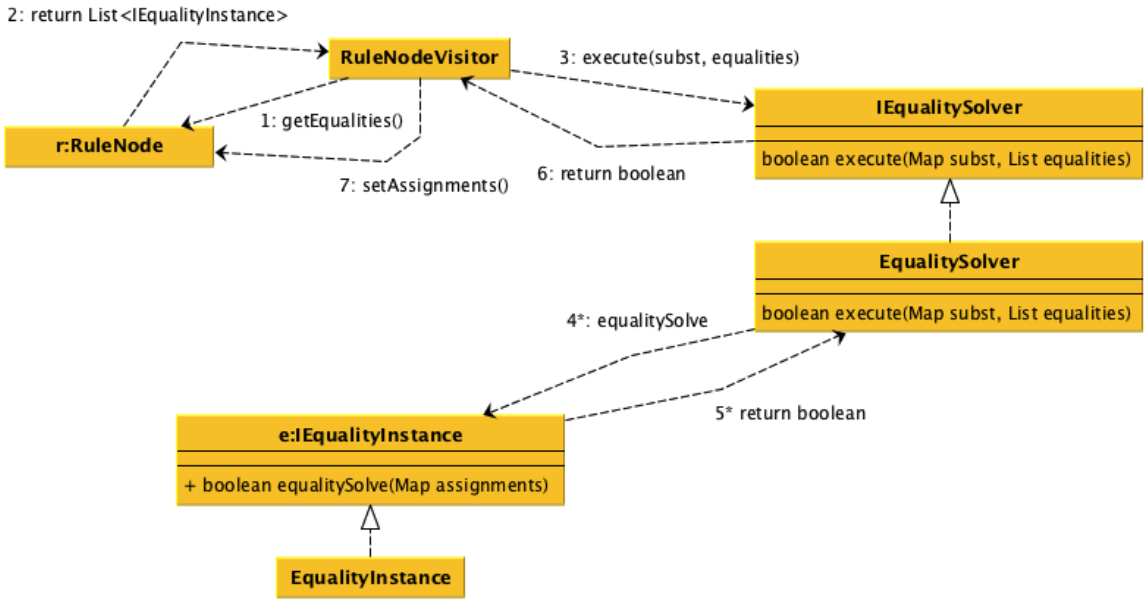


Figure 15: Equality Solver Communication Diagram

Inequalities are collected to the store in the same way that equalities are. They are produced by rules A1 and E2b as well as existing in the abductive theory.

18.1 Rules

The inequality solver takes an inequality of the form $s \neq t$ and applies the following rules in an iterative manner. These rules are specified in figure 16.

- $c \neq c \mapsto \bot$
- $c \neq d \mapsto \top$
- $X \neq X \mapsto \bot$
- $X \neq c \mapsto X \neq c$
- $X \neq Y \mapsto X \neq Y$
- $p(s_1, ..., s_n) \neq q(t_1, ..., t_n) \mapsto \top$
- $p(s_1, ..., s_n) \neq p(t_1, ..., t_m) \mapsto \top$ when $m \neq n$
- $p(s_1, ..., s_n) \neq p(t_1, ..., t_n) \mapsto s_1 \neq t_1 \vee ... \vee s_n \neq t_n$

Figure 16: InEquality Rules

18.2 Implementation

The inequality solver is used by the RuleNodeVisitor as part of the state rewriting process. After an inference rule has been applied the inequality solver is called if any inequalities are in the

store. Each inequality is evaluated separately by the solver which obtains the left and right components of the inequality and then uses the unifiable interfaces 'reduce' method to break each inequality into a list of equalities. For example:

1. $p(q(X), Y, Z) \neq p(q(a), b, c)$ is passed to the inequality solver.
2. $p(X, Y, Z)$ is unified with $p(a, b, c)$ to obtain $\{q(X) = q(a), Y = b, Z = c\}$
3. This is unified again to become $\{X = a, Y = b, Z = c\}$

This set of equalities essentially represents the body of the denial $\leftarrow X = a, Y = b, Z = c$ i.e. for the equality to return true then the following must be the case $X \neq a \vee Y \neq b \vee Z \neq c$.

The inequality solver returns a list of pairs of lists of equality and inequality instance such that repeat answers are avoided. In the above example the following would be returned by the equality solver:

- $\{\}, \{X \neq a\}$
- $\{X = a\}, \{Y \neq b\}$
- $\{X = a, Y = b\}, \{Z \neq c\}$

Due to there being multiple results the application of the inequality solver produces 'choice points' in the derivation tree. The RuleNodeVisitor use the results to generate three child nodes in the derivation tree. The equalities will be applied can used to create substitutions in the next stage, whilst the inequalities will be continuously reevaluated until the variables involved have been substituted and they can return true or false. If this never occurs then the inequality remains in the store and is returned as part of the explanation to the user.

In the case where no possible equality/inequality pairs can be created by the equality solver for example if we had an inequality that contained $c! = c$ then an empty list would be returned by the equality solver, and the RuleNodeVisitor would mark that node as failed.

19 Finite Domain Constraint Solver

The use of a finite domain constraint solver provides additional reasoning capabilities to JALP, in the same way that it does so with existing ASystem implementations based on prolog. Much tedium is avoided in terms of defining large numbers of facts and predicates.

For example instead of:

```
colour(red).
colour(blue).
colour(green).
```

We can write:

```
colour(X) :- X in [red,green,blue]
```

And instead of:

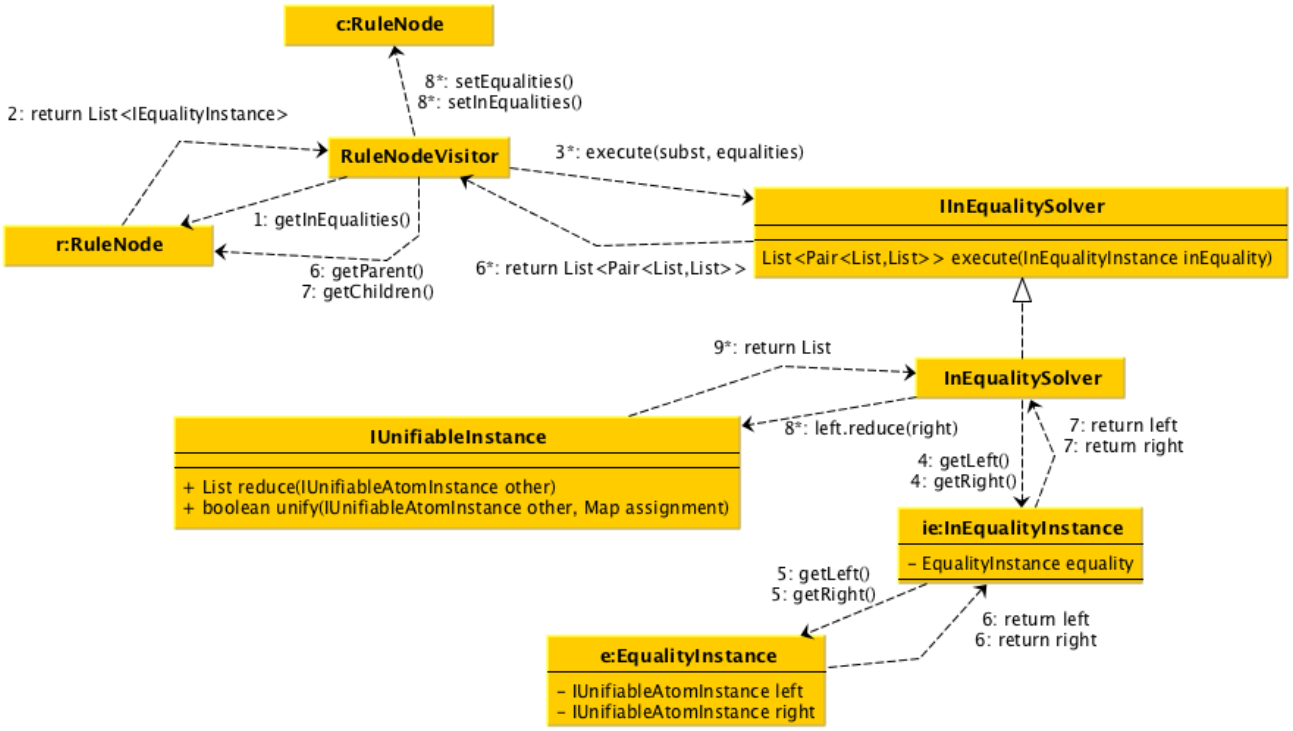


Figure 17: InEquality Solver Communication Diagram

```
p(X,Y) :- lessThan(X,Y).

lessThan(0,1).
lessThan(0,2).
lessThan(0,3).
...
```

We can write:

```
p(X,Y) :- X<Y.
```

Prolog has a native implementation of a finite constraint solver which is leveraged by prolog based implementations of ASystem. In much the same way I decided to leverage an existing java solution rather than reinventing the wheel.

Before going into implementation details I will explore two of these 'off-the-shelf' constraint solvers and their advantages and disadvantages.

19.1 JaCoP

JaCoP stands for Java Constraint Programming Solver [Jacop12]. It is an open-source java library that provides java users with an API for solving constraint satisfaction problems.

To construct a constraint satisfaction problem in JaCoP we first create a *Store* object. Into this store we add *FiniteDomainVariable* objects which be either an *IntegerVariable* or a *BooleanVariable*. Each variable can have an *IntervalDomain* such as $\{1..10\}$ or a *BoundSetDomain* of the form $\{1..10, 20..30\}$.

Into the *Store* object we also place constraints. These constraints can be *Primitive* such as ordering e.g $X < 4$ and logical e.g. $X \vee Y$ or *Global* such as $allDifferent([1, 2, 3, 4])$.

We then create a *Search* object which is used to evaluate the store and compute a solution to the constraint satisfaction problem we have specified.

The main advantage in using JaCoP would be its wide range of possible search methods and the the flexibility provided in specifying the domains of variables. For example given the following program:

```
p(X) :- X in [1,2,9,10,11].
```

JaCoP allows such 'gaps' in the domains of variables unlike other constraint programming solutions.

```
SetVar s = new SetVar(store, "X",
      new BoundSetDomain(new IntervalDomain(1,2),
                         new IntervalDomain(9,11)));
```

Figure 18 provides an example of constraint solving using the JaCoP API. A store variable is created which is essentially a collection of variables and constraints. Each constraint is 'imposed' upon the variables in the store. A search object is then instantiated. There are multiple types of search objects available each using a different method of search e.g. depth first search and then executed to provide the results.

The problem with JaCoP however is inflexibility in its set membership methods which made it difficult to integrate with JALP. This is discussed further in the implementation section.

In terms of future development of JALP, JaCoP is also rather limiting in its available constraints. For example expressions such as $X * Y - 1 < 4$ would not be supported due to a limited range of constraints that contain functions.

19.2 Choco

Choco is another java library for solving constraint satisfaction problems [Choco12]. It has a number of advantages over JaCoP making up for its limitations resulting in its integration into JALP.

Use of Choco involves the creation of a *Model* into which we place *Variable* objects and *Constraint* objects. Variable objects include types *ConstantVariable*, *IntegerVariable*, *SetVariable* and constraints exist for operating on these various different types.

Once the model has been constructed a *Solver* object is constructed through which we define search options and then use to iterate over the various solutions to the Model.

Choco's main issue with regards to integration with JALP is its inflexibility with regards to the domains that variables can take. JaCoP allowed removal of elements from the domain of variables whereas in Choco the domain must be respecified. Changes to domains of variables in

```
import JaCoP.core.*;
import JaCoP.constraints.*;
import JaCoP.search.*;

public class Main {

    static Main m = new Main ();

    public static void main (String[] args) {
        Store store = new Store();  // define FD store
        int size = 4;
        // define finite domain variables
        IntVar[] v = new IntVar[size];
        for (int i=0; i<size; i++)
            v[i] = new IntVar(store, "v"+i, 1, size);
        // define constraints
        store.impose( new XneqY(v[0], v[1]) );
        store.impose( new XneqY(v[0], v[2]) );
        store.impose( new XneqY(v[1], v[2]) );
        store.impose( new XneqY(v[1], v[3]) );
        store.impose( new XneqY(v[2], v[3]) );

        // search for a solution and print results
        Search<IntVar> search = new DepthFirstSearch<IntVar>();
        SelectChoicePoint<IntVar> select =
            new InputOrderSelect<IntVar>(store, v,
                                         new IndomainMin<IntVar>());
        boolean result = search.labeling(store, select);

        if ( result )
            System.out.println("Solution: " + v[0]+", "+v[1] +", "+
                                              v[2] +", "+v[3]);
        else
            System.out.println("*** No");
    }
}
```

Figure 18: JaCoP Example Program

```
int nbQueen = 8;
//1- Create the model
CPModel m = new CPModel();
//2- Create the variables
IntegerVariable[] queens = Choco.makeIntVarArray("Q", nbQueen, 1, nbQueen);
//3- Post constraints
for (int i = 0; i < nbQueen; i++) {
    for (int j = i + 1; j < nbQueen; j++) {
       int k = j - i;
       m.addConstraint(Choco.neq(queens[i], queens[j]));
       m.addConstraint(Choco.neq(queens[i], Choco.plus(queens[j], k))); // diagonal constraints
       m.addConstraint(Choco.neq(queens[i], Choco.minus(queens[j], k))); // diagonal constraints
} }
//4- Create the solver
CPSolver s = new CPSolver();
s.read(m);
s.solveAll();
//5- Print the number of solutions found
System.out.println("Number of solutions found:"+s.getSolutionCount());
```

Figure 19: Choco Example Program

Choco is limited to setting the upper and lower bound or reconstructing the domain entirely.

Choco's expressive set methods were one of the major reasons for choosing this package. This is discussed further in the implementation section. Another excellent feature with Choco is its IntegerExpressionVariable. This allows use of functions for example $X*2+Y<5$. Whilst JALP does not currently support the use of functions within constraints the fact that Choco supports this makes it a good choice in order to meet my design goal of extensibility for JALP.

Less important benefits in Choco's favour are its use of Maven for dependency management which meant simple integration into JALP's project structure and also excellent documentation, tutorials and extensive examples. Figure 19 shows example code for solving the 'n-queens problem'.

19.3 JALP Implementation & Integration

Out of the two choices I decided on Choco for integration into JALP. Choco has excellent documentation and a plethora of examples available as tutorial examples. . Support for functions is available and Choco itself is very extensible in that different types of variables, search methods, constraints can be easily implemented and added to the library. This provides a certain level of extensibility as per my design goals.

The main reason for the choice however is Choco's flexible implementation of finite domain variables and its possible reuse of model and solver objects. The relevance of this is explained in-context, below, as part of my explanation of the how constraint solving is handled within JALP.

Figure 20 is a UML class diagram that shows how constraint solving is handled within JALP. My design goals were to aim for modularity and loose coupling and so constraints are given abstract, JALP specific representations rather than using those provided by Choco.Parsing in-

volves constructing IConstraintInstance objects rather than constraint objects from any specific constraint solver implementation. This loose coupling promotes extensibility if for any reason a different constraint solver was chosen in the future.

The facade design pattern is used to present an interface to the chosen constraint solver and the ITermInstance and IConstraintInstance abstractions are given the responsibility to produce their Choco counter-part representations.

This abstraction also provides another advantage in that certain aspects of constraint solving can be handled natively within JALP rather than outsourcing them to the solver. This can be aimed towards implementing new features not supported by the external constraint solver or for efficiency reasons. For the sake of extensibility I believed it was important to provide this option.

For example, both the Sicstus Prolog constraint solver and hence implementations of ASystem based upon it only support enumerated variables i.e. real variables and integer variables. This is also the case with 'off-the-shelf' java constraint satisfaction libraries. I made the decision to implement list constraints on character-based constants natively. The constraint solver 'pipeline' (todo: expand on this concept) first processes any natively implemented constraints such as these character constants and then deals with any other types of constraints. For example in JALP we can specify:

```
p(X) :- X in [john, jane, mary].
```

rather than:

```
p(john).
p(jane).
p(mary).
```

which is clearly more concise.

In other ASystem implementations we would have to perform some form of mapping from the domain space $\{john, jane, mary\}$ to the integer domain e.g. $\{(john, 1), (jane, 2), (mary, 3)\}$.

As part of this abstraction it is necessary to convert between the JALP representation of constraints and variables and the representation specific to the constraint solver that is being leveraged. This was the main reason that I chose Choco over JaCoP. Choco is much more flexible in terms of methods available for comparison. JaCoP only allows set operations between SetVariables i.e. the variables had to be defined over a specific domain. Choco on the other hand allows operations such as $isMember(IntegerVariable, SetVariable)$ rather than having to specify a domain object. This flexibility allows well constructed object-oriented translations between the two representations.

Like with the equality solver one design issue was how often the constraint solver needed to be applied. Repeatedly initialising the constraint solver at each leaf node of the derivation tree could prove to be computationally expensive. The Choco Model and Solver objects do not need to be reconstructed at each stage. Variables and Constraints can be added and removed from the model as needed and the solver can be cleared and re-used.

Figure 21 is a UML communications diagram showing a typical call to the JALP constraint solver abstraction and how this is dealt with. A client object (in our case this is the JALPSystem object which is responsible for processing of queries) makes a call to a RuleNode to obtain

its constraints. It then calls the constraint solver facade with these constraints as a parameter.

The constraint solver facade provides a loosely coupled interface to a constraint solver implementation. This facade then makes calls to the various constraints that are passed to it in order to obtain Choco representations of these constraints. This stage acts somewhat like a pipeline in that substitutions are collected natively and then any remaining constraints are evaluated by the constraint solver and then combined with those substitutions already obtained. The steps are as follows:

The facade iterates through each constraint in the list that is passed to it by JALPSystem. Each abstract representation of the constraint implements a method that returns a Choco representation of that constraint. Similarly the representations of the various constants and variables that make up the constraints implement methods that return Choco representations of themselves.

The facade calls the toChoco() methods of each constraint which in turn calls the toChoco() methods of its components. These representations are stored in maps that are passed via the toChoco() method. The return value is a boolean that is used by constraint satisfaction components that are implemented natively e.g. character constants. Either a conversion to Choco representation is applied, or the constraint is evaluated natively, generating substitutions and returning a truth value based on success. For example in figure 21 the InConstantConstraint checks to see whether the given term is in the list.

Where $\bullet$ is a numerical constraint of the form $<, \leq, >, \geq$, constraints can be of the following combinations:

- $c \bullet d$
- $X \bullet c$
- $X \bullet Y$
- $X\ in\ [e_1, ..., e_n]$
- $c\ in\ [e_1, ..., e_n]$

Once all Choco representations have been obtained, the Choco solver is executed. It returns various possible substitutions which are then combined with the possible substitutions that were collected by constraint satisfaction implemented natively.

20 Solver Interaction

Figure 22 shows the order of execution of the components used in JALP to compute *one* explanation. An initial state is generated from the query and the abductive theory. The state rewriting rules are then applied followed by the equality solver and then the inequality solver in an iterative fashion, generating choice points along the way. The constraint solver is then applied at each leaf node and generates the success state.

21 State Rewriting

ASystem is a state rewriting procedure, that is, each stage involves rewriting the current state to produce a set of (possibly empty) new states. Each stage uses a single inference rule that

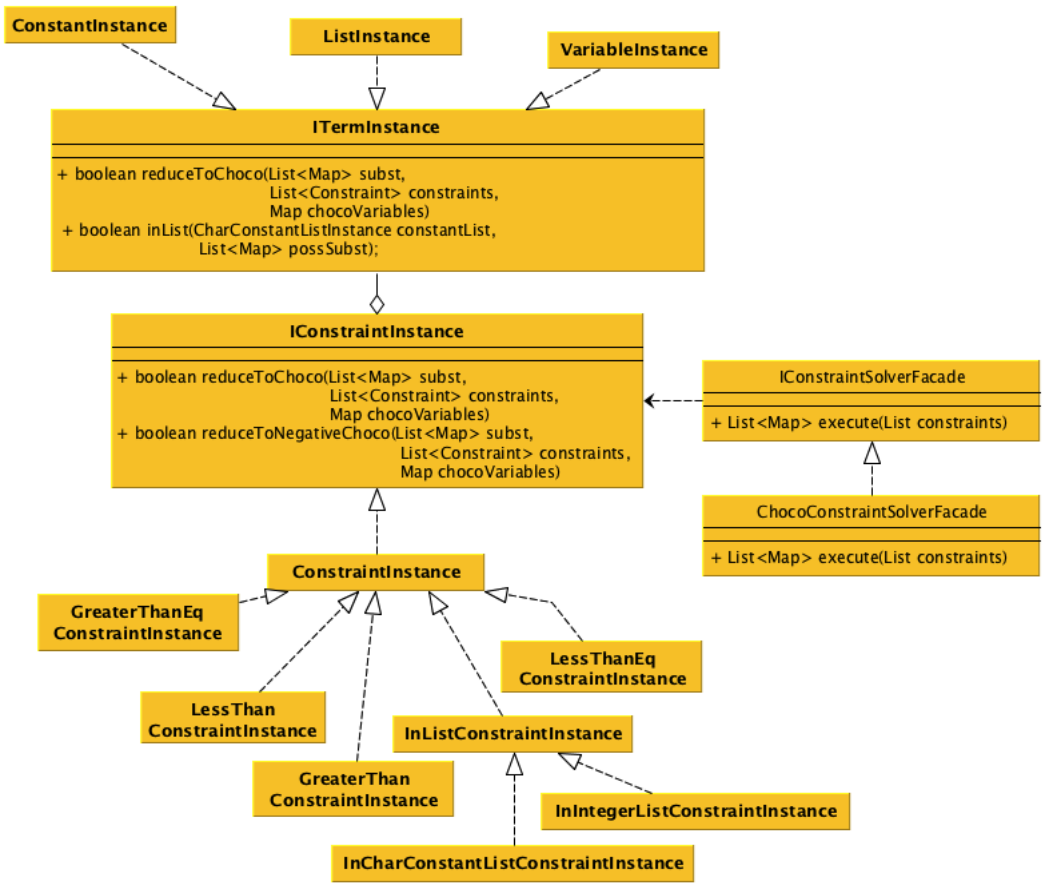


Figure 20: Constraint Solver Class Diagram

defines how to rewrite that state into a set of new states. Failure to generate any new states means that some form of backtracking procedure must be performed. i.e. we must find a way back.

Early versions of JALP used a cloning procedure to manage this state rewriting whereas new versions use a derivation tree and the visitor design pattern. This section explores these two approaches.

21.1 Cloning with Template

The first attempt at state rewriting involved the use of Java's *Cloneable* interface. Each application of a state-rewriting inference rule involved cloning the state and all objects (including predicates and variables), rules and the store within it and then performing the rewriting on the clones using the inference rules. For example an application of D1 with goal $p(X)$ and n different rules with $p(X)$ as the head would involve executing a clone on all objects in that state before adding the expansions of each rule to each new state. Clearly this is an expensive procedure.

As mentioned earlier, the equality solver generates substitutions from collected equalities. These substitutions were handled by associating a value with each variable object.

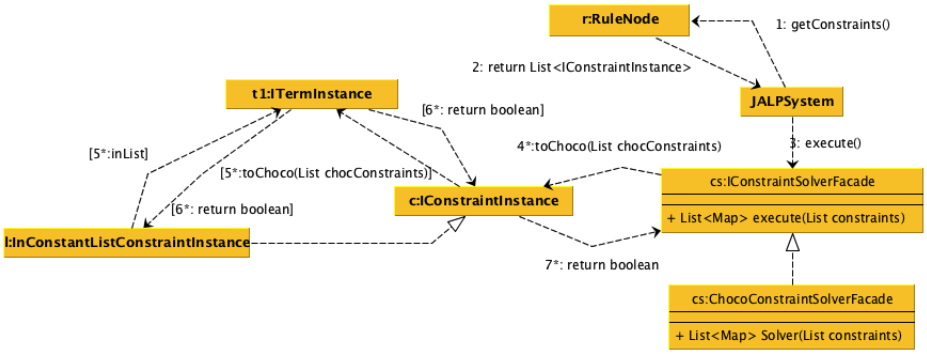


Figure 21: Constraint Solver Communication Diagram

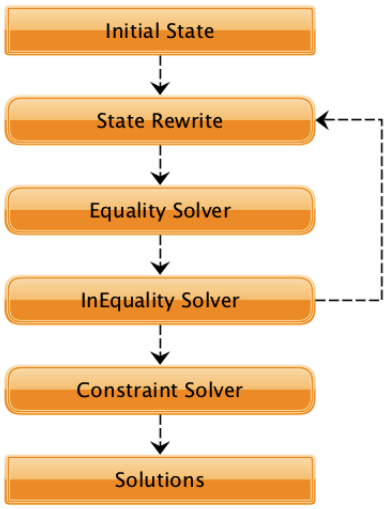


Figure 22: JALP Interaction Flow Diagram

These two methods led to a confusing implementation that was difficult to debug and did not meet my design goals. The cloning of all variables made it difficult to keep track using debugging tools. The cloning and recreation of data structures in as expensive operation. The association of values with variables and large degree of cloning also meant heap space was an issue. Figure 23 is a UML diagram showing the early structure of JALP. Inference rules were implemented as a chain of responsibility through which state objects were passed. These state objects were then cloned iteratively before being passed onto the next state. Choice points were handled through the use of a 'fringe' of unexpanded states.

21.2 Visitor with Template

The second and current approach that JALP uses places emphasis on abstraction and loose coupling between the various components and concepts involved in the abductive reasoning system. It organises the state rewriting into a derivation tree in a similar fashion to the prolog tree structure. This is a much more natural way of organising the state rewriting and meant that debugging was clearer. It also led to the development of a useful visualisation tool to display the derivation tree graphically. Cloning and creation of new data structures is kept to a minimal.

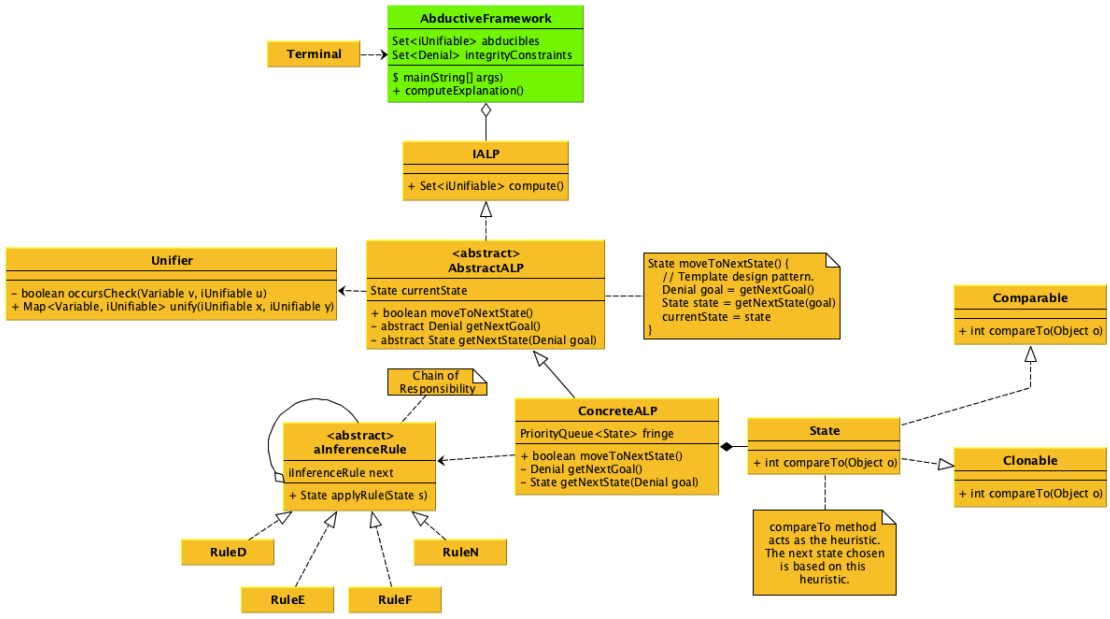


Figure 23: Cloning UML

A JALP derivation tree consists of *positive* rule nodes, *negative* rule nodes and *leaf* rule nodes.

Each rule node represents a possible state. 'RuleNode' refers to the name of the rule that will be applied in order to expand and rewrite that particular node. A positive rule node will have an inference rule that operates in the positive mode of reasoning. A negative rule node is an inference rule that operates in the negative mode of reasoning i.e. the current goal is nested within denials. As an implementation of an ASystem state, each rule node has with it an associated list of goals and a store. It also has a list of child states which allows construction of the derivation tree and a *node mark*.

When an inference rule is applied to a RuleNode, the node is marked as either *failed*, *succeeded* or *expanded*. Any node that has not yet had an inference rule applied to it is marked as *unexpanded*.

Inference rules are iteratively applied to the derivation tree in a depth first search manner until all leaf nodes have been marked with success or failure by the equality solver, inequality solver or the constraint solver.

JALP makes use of the visitor design pattern, a method of separating a data structure from the algorithms that operate on it. Figure 24 is a UML diagram explaining the visitor pattern. Inference rules are iteratively applied to the derivation tree by a RuleNodeVisitor which calls the RuleNode's acceptVisitor() method and receives a callback. Upon callback the RuleNodeVisitor then performs the state rewriting associated with each inference rule as well as the application of the equality and inequality solvers. Child nodes are generated and the RuleNode is marked. This interaction is controlled by the JALPSystem class.

The use of the visitor pattern meets my design goals of extensibility and modularity in that other visitors can be written to interact with the data structure with minimal effort and that

the state rewriting algorithm is completely separated from the derivation tree itself.

State rewriting often involves substitutions and creation of *choice points*. The equality solver deals with collected equalities through the generation of substitutions which are then applied to the entire ASystem state, or in our case, the RuleNode. Choice points are generated by certain inference rules and also by both the inequality solver and the constraint solver. On reaching a failed node we must *backtrack* to the choice points which are held on a stack data structure. This is not backtracking in the same sense as in prolog, but in that the next rule node is taken from a *last in, first out* stack. This process is effectively a depth first generation of the derivation tree. In order to generate the tree we must generate new child nodes which involves cloning of information held by the parent state. Unlike the previous implementation, however, cloning is *shallow* i.e. only the data structures themselves are cloned rather than the individual components such as variables and constants. Even this is avoided when possible and is discussed in detail in the profiling section of the evaluation.

Substitution is handled through a recursive mechanism involving a *HashMap* that is passed to the denial, predicate, rule or other logical structure that is involved. The *HashMap* represents the mapping of variable onto term and references are updated by passing this structure down to the various VariableInstances, which look themselves up in the *HashMap* and return their new mapping. For example $\leftarrow p(q(X)))\theta$ where $\theta = \{X/c\}$. The denial passes the substitution to p which then passes the substitution to q, which then passes the substitution to X. X then returns c which is assigned by q to be its new parameter. This is more memory efficient than assigning a value to a variable object as in the previous implementation.

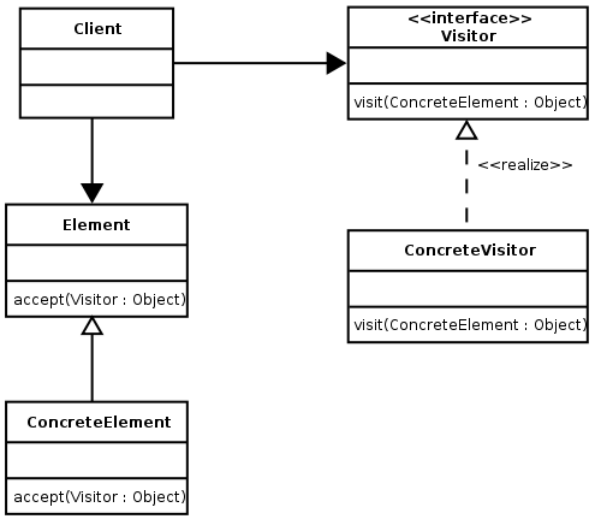


Figure 24: Visitor Design Pattern

22 Edge Cases

This section discusses various exceptional situations that can arise in terms of abductive theories and input to the system and how this is handled.

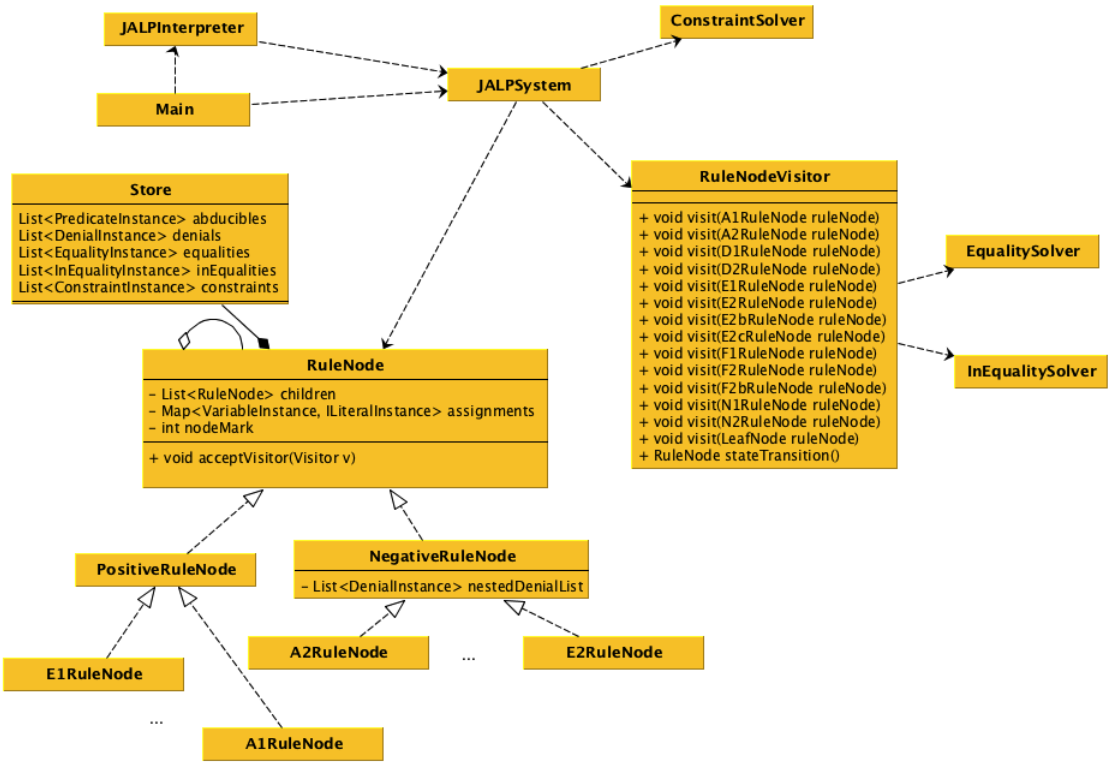


Figure 25: State Rewriter UML Diagram

As mentioned earlier the situation could arise whereby two terms X and u are unified where u contain X for example $X = p(X)$. In this case we'd end up with an infinitely repeating structure. Rather than performing an occurs check which is inefficient, JALP counts the depth at which it is performing a substitution. An exception is thrown if the limit is exceeded.

Another exceptional case is that of evaluation of universally quantified variables with an infinite domain for example $\forall X.p(X) \leftarrow X < 4$ or $\leftarrow not\ X\ in\ [1, 2, 3]$. This is unsafe and will cause the constraint solver to search for an infinite number of solutions. This effect is known as floundering and is handled in JALP by using a timeout on the constraint solver. At the moment this is fixed but future versions of JALP should allow this to be manipulated by the user.

23 Unit Testing

One of my design goals was that the system must be *robust*, as well as remain so in the future. One way to make this more likely is through the use of unit testing.

23.1 Individual Components

Unit tests on oft-used operations such as substitution and cloning as well as each individual component in JALP have been implemented.

Substitution and cloning are critical parts of JALP's execution. New instances of denials must be created in rules such as A2 whereby existing collected constraints are checked. Substitutions are applied to each state, and choice points must be saved and have their own instances of each collected abducible so that if we need to backtrack at some point, their values have not been changed. Therefore there are a variety of unit tests against all structures that can be substituted or cloned.

The unification examples presented in figure 3 are also implemented as unit tests. Unification is used at nearly every stage in the ASystem state rewriting process and so confirmation that unification works flawlessly is important.

Each ASystem inference rule has one or more unit tests designed to ensure that they are working correctly. For example, Figure 26 shows an example unit test (inference rule D1) that has been executed and output as a visualisation in the form of HTML and JavaScript.

The constraint solver, equality solver and inequality solvers each have their own unit test suites. Each solver has been implemented in such a way that they are independent from the abductive reasoning system. This was for both testing purposes and as a way of promoting loose coupling and modularity.

23.2 Example Programs

Additionally a large array of more complex unit tests have been created using various abductive theories as examples. Credit must be given to my 2nd supervisor Jiefei Mai as most of these examples have been supplied by him. These examples include the circuit analysis examples, flies examples and more that are available in the appendix.

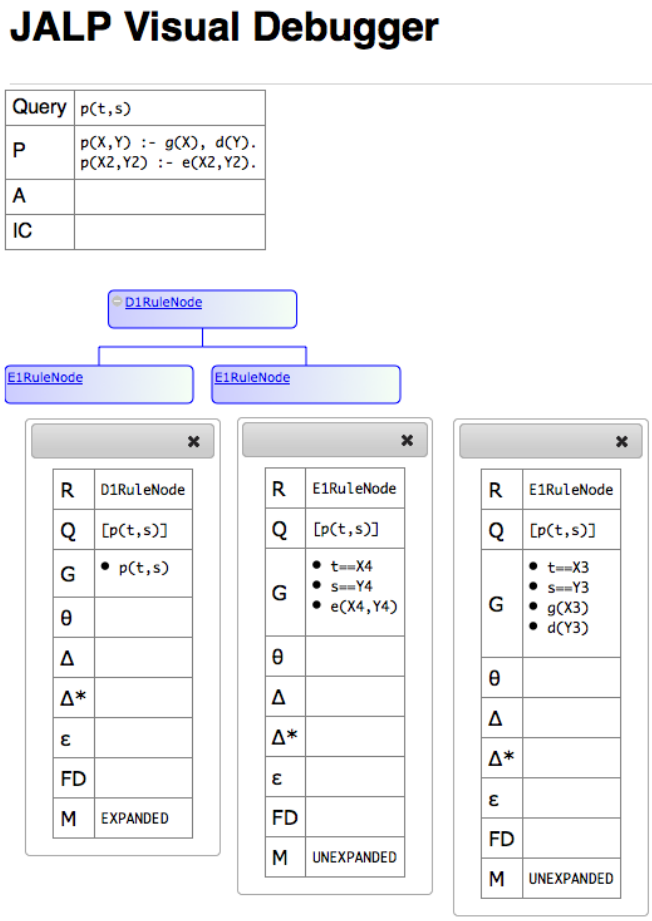


Figure 26: D1 Unit Test

Part V
Evaluation

In this section a critical analysis is performed on JALP through both qualitative and quantitative means.

The qualitative analysis involves comparisons between the interface presented by JALP and that presented by Prolog. The quantitative analysis performed involves the use of profiling tools to collect CPU usage and memory allocation information. Additionally, quantitative comparisons are made against an existing implementation in Prolog. [Abduction]

24 Profiling

In order to assess efficiency I performed per-method CPU profiling and per-method profiling of memory allocations. JProfiler was leveraged for this and was chosen due to simple integration into the IntelliJ IDE that was used to develop JALP. JProfiler provides computation of method hotspots and heap space allocation hotspots and is useful for performance tuning. [JProfiler12]

Profiling was performed using an abductive theory for 'circuit analysis' [Abduction]. This example involves many thousands of applications of the state rewriting rules and so gives plenty of scope for profiling of both CPU usage and memory usage.

```
%Query: output(g2,off)

output(Gate, Value) :-
        inverter(Gate),
        input(Gate,InValue),
        opposite(InValue,Value),
        not broken(Gate).

output(Gate, InValue) :-
        inverter(Gate),
        input(Gate,InValue),
        broken(Gate).

input(g2,Value):- output(g1,Value).
input(g1,on).

inverter(g1).
inverter(g2).

opposite(on,off).
opposite(off,on).

ic :- output(Gate, on),output(Gate, off).
ic :- input(Gate,on),input(Gate,off).

abducible(broken/1).
```

After profiling of both CPU hotspots and Memory hotspots I then made various improvements to JALP and performed a runtime analysis of the circuits example using query $Q = \{output(g2, off)\}$. This involved repeating the query 1000's of times before and after the improvements were made and comparing the average time taken to perform the query.

24.1 CPU

The initial results for CPU profiling are shown in figure 27. It is clear that a large amount of time is spent shallow cloning and performing substitutions. A shallow clone of a rule node is the duplication of its various data structures including the store, assignments and goals. This is often performed when executing the inequality solver and other areas where choice points are generated. This cloning of data structures is also performed when substitutions are applied to the entire state.

The cloning of data structures for the child of each rule node is only actually necessary when we want to preserve the derivation tree structure for the visualizer. If this tree is not necessary for output then cloning of data structures only has to take place at choice points i.e. when there are multiple child rule nodes. Figure 28 shows method hotspots after these changes. Notice a significant decline in the time spent executing each method. Also notice the shallow cloning of CharInstanceConstants has also disappeared. Cloning of character constants was an unnecessary step as they are never substituted in the way that a variable is.

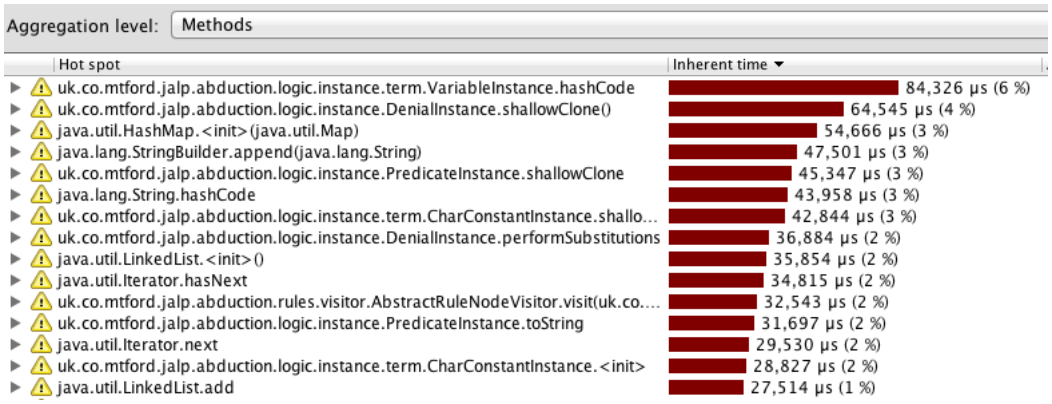


Figure 27: Method CPU Time

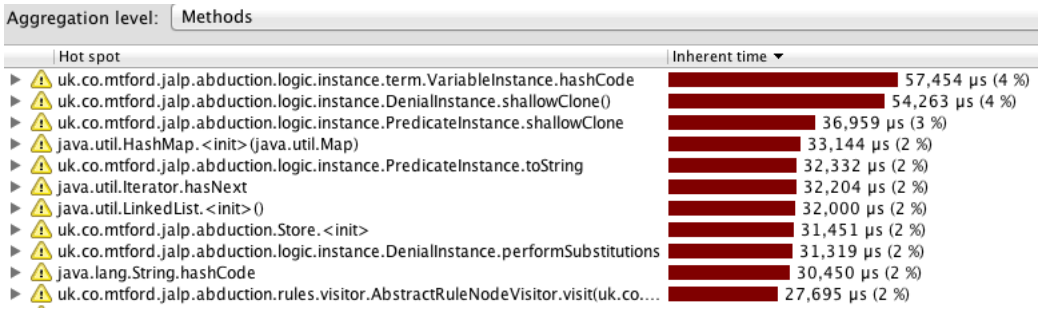


Figure 28: Post-Improvement Method CPU Time

24.2 Memory

In a similar vein to the CPU profiling a large amount of heap space is taken up due to maintaining the derivation tree in memory. Maintenance of references also means that garbage collection never kicks in. Figure 29 shows garbage collected objects before improvements are made to JALP. Figure 30 shows garbage collection post-improvements. Improvements involved simply removing references to nodes that do not represent choice points and setting any references to the root node of the derivation tree to null in order to encourage garbage collection.

24.3 Runtime Analysis

Running the query $Q = \{output(g2, off)\}$ both before and after the previously discussed improvements shows a significant improvement in runtime. This was done many thousands of time in order to cater for variations in heap space allocation, garbage collection and also changes in the system.

Before improvements were made the average time taken to run the query was 132 ms and post improvements the average time taken was 28ms. This is obviously a significant improvement.

25 Benchmarking

In this section I perform a quantitative comparison between Dr. Jiefei Ma's implementation of ASystem [Abduction] and JALP using abductive theories describing various graph colouring

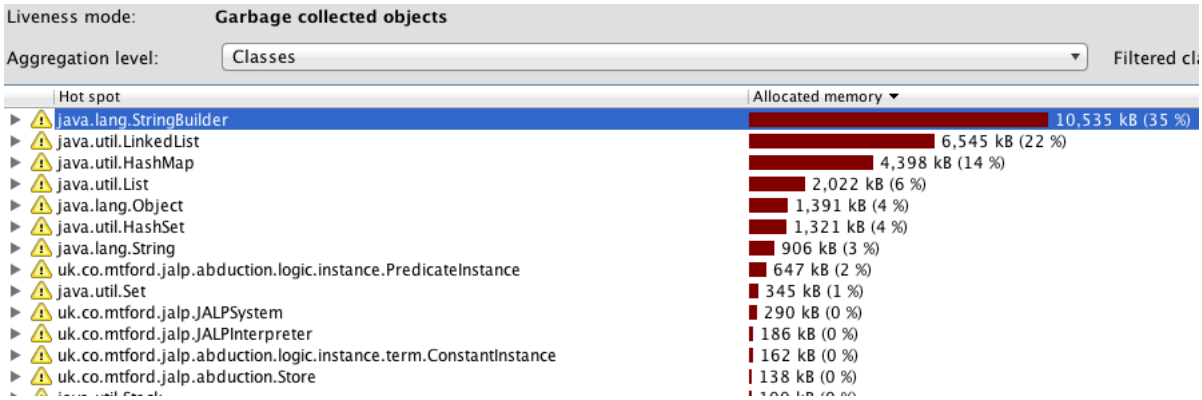


Figure 29: Pre-Improvement Garbage Collection

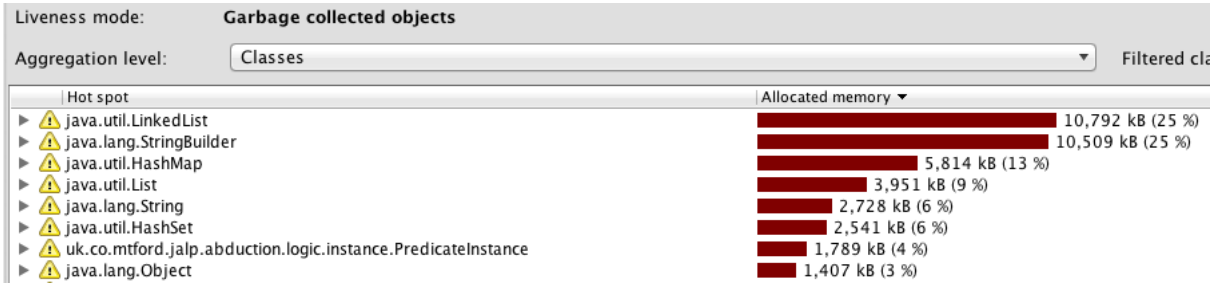


Figure 30: Post-Improvement Garbage Collection

problems.
Graph colouring is the assignment of labels traditionally called 'colours' to the the elements of a graph subject to certain constraints. In it's simplest form, which is used here, it is a way of colouring the vertices of a graph such that no adjacent vertices share the same colour. The following is the abductive theory for a graph colouring problem with two nodes and two colours.

```
edge(node1,node2).
edge(node2,node1).

colour(red).
colour(blue).

ic :- colour(C), has_colour(N,C), edge(N,M), has_colour(M,C).

abducible(has_colour/2).
```

The idea is to scale up the number of colours and number of nodes being used and perform an analysis of the amount of time needed for each implementation to calculate the explanations. Figure 31 is a graphical depiction of solutions to several graph colouring problems.
Benchmarking of both systems was performed at the code level. In JALP's case this involved using *System.nanoTime()* both before and after query execution and then computing the difference. In the prolog implementations case this involved using the Sicstus statistics/2 predicate

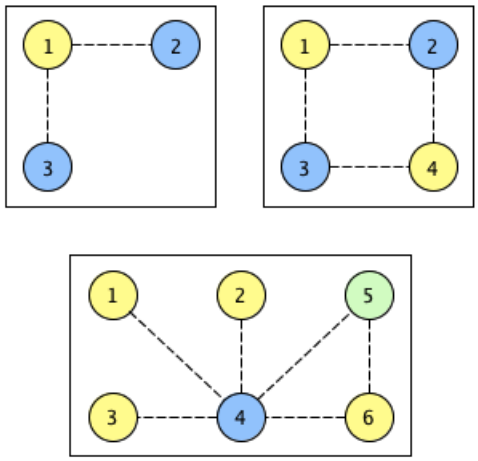


Figure 31: Graph Colouring

which provides runtime execution statistics at the millisecond level.

Figure 32 shows execution time in milliseconds for both systems to execute queries involving increasing number of nodes and possible colours. Queries were executed multiple times on both systems in order to cater to system level changes such as memory allocation and scheduling.

Num. Nodes	Repetitions	JALP (ms)	Abduction (ms)
2	100000	0.657	0
3	10000	4.482	7.5
4	10000	13.094	10
5	100	5671.487	105
6	100	10974.95	2410

Figure 32: Time taken for JALP to compute all explanations.

The Prolog example is more efficient than JALP. JALP suffers a large increase in time taken when number of nodes are increased from 4 to 5 and this is something that warrants further investigation.

26 User Interface

One of the motivations for this project was to provide a user-friendly implementation in a popular language to encourage adoption of ALP in software projects. The issue so far has been Prolog's interface which is confusing for those with limited experience with logic programming. Not only does there lack a native Java API for abduction but users are forced to use a Prolog interpreter as a platform for the abductive system onto which theories are loaded and queries are executed. Figure 33 is a comparison between output from JALP and output from Jiefei Ma's implementation of ASystem [Abduction]. This demonstrates much more clarity to the average

user than in the prolog based system.

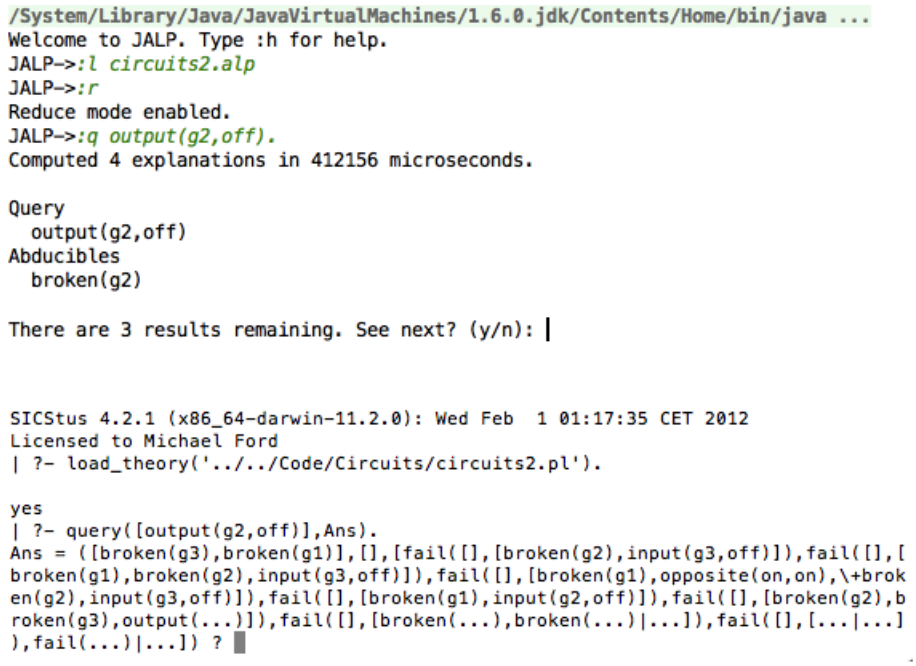


Figure 33: Interface Comparison

27 Summary

To summarise, it is clear that JALP has several weaknesses. Whilst improvements were made, the amount of cloning that takes place at choice points and during substitutions is still quite large. There are likely less obvious optimisations that can be made in this area and this is something that should be looked at in the future. This weakness is more obvious when compared to the prolog implementation. Another weakness of JALP is the lack of heuristics. Certain examples, such as the block example in the appendix, can not be executed due to out of memory exceptions. This is not an issue with the prolog implementation due to the use of clever heuristics. Again this is an area of future development for JALP.

Extra care has been taken to make JALP both extensible and modular. Given these improvements then, with it's user-friendly interface and accessible API, I believe JALP will be a useful alternative to the logic reasoning systems implemented in prolog.

Part VI
Conclusions and Future Work

28 Objectives

The core objectives as stated at the beginning of the project were as follows:

- Implementation of Logic Programming components necessary for abductive logic programming.
  - Unification.
  - Equality Solver.
  - Inequality Solver.
- Implementation of the ASystem state-rewriting algorithm for abduction.
- Development of a Java API for abductive logic reasoning.
- Development of command line tools for dealing with abductive theories and queries over those theories.

All objectives were completed successfully. Unification was implemented in an object-oriented fashion based on the robinson unification algorithm. The equality solver and inequality solver were implemented as separate machinery and integrated with the implementation of the ASystem state rewriting mechanism. Both the Java API and and command-line tools have been implemented and documented.

29 Extensions

A number of possible extensions were also decided upon, to be completed given time. These include:

- Finite constraint solver.
  - $<, <=, >, >=$
  - List constraints.
  - Character constants.
  - Functions.
  - Real numbers.
- Execution visualizer.
- State rewriting heuristics.

Constraint solving is supported in JALP via a combination of native evaluation and use of an external constraint solver. Numerical constraints such as $<$ and $>$ are solved through the use of the Choco constraint solving API, whereas constraints involving character-based constants are handled natively. Functions and real numbers are left for future developers to integrate, however due to the extensible way in which the constraint solver was implemented and also the choice of the constraint solver itself, this shouldn't be too difficult given time.

A visualizer was constructed initially as a debugging tool but is now available as a command-line option and via the Java API.

30 Future Work

30.1 Finite Domain Constraint Solver

Future versions of JALP could handle constraints such as $X + Y < 2$. This could be either handled natively or by the external constraint solver. Choco has a large library of functions including *abs, cos, div, max* and many others which are available in the documentation at [Choco12].

At the moment JALP only handles constraints using integers. Combined with an implementation of functions, it would be useful to be able to reason over real number domains. Again, Choco features an implementation of *RealVariables* which could be leveraged to this end.

Choco also features various search options with the default being a depth first search. Future work could include an investigation into which methods are more appropriate for JALP's case, or to provide the user with the ability to pick and choose what kind of search methods are used in constraint solving.

30.2 Heuristics

One of JALP's critical issues is heap space. For example, take the block world example taken from [NU04].

```
initially(1,2).
initially(2,3).
initially(3,0).

ablock(X) :- X in [1,2,3].
location(X) :- X in [0,1,2,3].
time(X) :- X in [1,2,3,4,5,6].

on(X,Y,T) :- initially(X,Y), not moved(X,Y,0,T).
on(X,Y,T) :- move(X,Y,E), E<T, not moved(X,Y,E,T).
moved(X,Y,E,T) :- move(X,Z,C), Z!=Y, between(C,E,T).
between(C,E,T) :- E <= C, C < T.

succeeds_move(X,Y,E) :- ablock(X), location(Y), time(E), X!=Y, clear_block(X,E), clear_location(Y,

clear_block(X,E) :- not something_on(X,E).
something_on(X,E) :- on(Y,X,E).

clear_location(0,E).
clear_location(Y,E) :- Y!=0, clear_block(Y,E).

ic :- move(X,Y,E), not succeeds_move(X,Y,E).

ic :- move(X,Y1,T), move(X,Y2,T), Y1!=Y2.
ic :- move(X1,Y,T), move(X2,Y,T), X1!=X2.

abducible(move/3).
```

JALP is unable to execute queries over this example due to out of memory exceptions even with the improvements to garbage collection and heap allocation mentioned in the profiling section.

There are a number of heuristics available that can be used to trim the derivation tree and hence avoid unnecessary exploration and reduce memory usage.

30.3 Semantic Analysis

At the moment only syntax analysis is performed on theory files i.e. JALP only examines the individual tokens rather than any meaning. Semantic analysis could be used to check for such issues as floundering rather than performing these checks at runtime.

Part VII
Bibliography

References

[KKT92] Antonis C. Kakas, Robert A. Kowalski, and Francesca Toni . Abductive logic programming. *Journal of Logic and Computation*, 2(6):719âĂŞ770, 1992.

[Sha05] Murray Shanahan. Perception as abduction: Turning sensor data into meaningful representation. *Cognitive Science*, 29(1):103 to 134, 2005.

[Esh88] Kave Eshghi. Abductive planning with event calculus. In *International Conference on Logic Programming/Symposium on Logic Programming*, pages 562 to 579, 1988.

[Sha00] Murray Shanahan. An abductive event calculus planner. *Journal of Logic Pro- gramming*, 44(1-3):207 to 240, 2000.

[Rob65] John Alan Robinson *A machine-oriented logic based on the resolution principle*. 1965

[LL87] Lloyd, J.W. *Foundations of Logic Programming*. Springer-Verlag 1987.

[HV2009] Krystof Hoder and Andrei Voronkov, *Comparing Unification Algorithms in First-Order Thereom Proving*, University of Manchester, 2009

[Wam99] Hassan Ait-Kaci, *WarrenâĂŹs Abstract Machine, A Tutorial Reconstruction*. February 18, 1999.

[JM11] Jiefei Ma, *Distributed Abductive Reasoning: Theory, Implementation and Application*. September 2011.

[NU04] Bert Van Nuffelen, *Abductive Constraint Logic Programming: Implementation and Applications*. June 2004.

[KM90] A.C. Kakas and P. Macarella, *Database Updates Through Abduction*. 1990.

[Pei31] Charles S. Peirce. *Collected Papers of Charles Sanders Peirce*. Harvard University Press, 1931.

[FT12] Francesca Toni, *Knowledge Representation and Reasoning Lecture Slides*. Imperial College London, 2012.

[KR11] Krzysztof Kuchcinski and RadosÅĆaw Szymanek *JaCoP Library User's Guide: http://jacopguide.osolpro.com/guideJaCoP.html*. Version 3.1, February 3, 2011.

[Res12] Resolution, http://en.wikipedia.org/wiki/Resolution_(logic), 15/5/2012.

[JG2003] Jean Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Chapter 9, 2003

[Bar99] Roman Bartak, Constraint programming: In pursuit of the holy grail *In Proceed-ings of the 8th Annual Conference of Doctoral Students (Invited Lecture)*, pages 555âĂŞ564, 1999.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503 to 581, 1994.

[JavaCC12] http://javacc.java.net/, 5/5/2012

[Choco12] http://www.emn.fr/z-info/choco-solver/, 10/5/2012

[Jacop12] http://www.jacop.eu/, 1/5/2012

[JProfiler12] http://www.ej-technologies.com/products/jprofiler/overview.html, 14/6/2012

[Abduction] http://www-dse.doc.ic.ac.uk/cgi-bin/moin.cgi/abduction 10/5/2012

Part VIII
Appendix

31 ASystem Example Derivations

31.1 Graph Colouring

The following is a simple graph colouring program with two nodes. The aim is to find two colours such that any node connected to another node has a different colour.

```
edge(node1,node2).
edge(node2,node1).

colour(red).
colour(blue).

ic :- colour(C), has_colour(N,C), edge(N,M), has_colour(M,C).

abducible(has_colour(N,C)).
```

Given a query $Q = \{has\_colour(node_1, C_1), has\_colour(node_2, C_2)\} \cup IC$ we are attempting to find an explanation, that is, an assignment to C and D such that node1 and node2 have different colours to any node that they share an edge with.
The initial state is:

$$G = \{has\_colour(node_1, C_1), has\_colour(node_2, C_2)\} \cup IC, ST = \{\emptyset, \emptyset, \emptyset\}$$

We apply A1 twice to get:

$$G = \{\forall C, N, M. \leftarrow colour(C), has\_colour(N, C), edge(N, M), has\_colour(M, C)\}$$

$$\Delta = \{has\_colour(node_1, C_1), has\_colour(node_2, C_2)\}$$

We apply D2 to get:

$$\begin{aligned} G &= \{\forall C, N, M. \leftarrow C = red, has\_colour(N, C), edge(N, M), has\_colour(M, C)\} \\ &\cup \{\forall C', N', M'. \leftarrow C' = blue, has\_colour(N', C'), edge(N', M'), has\_colour(M', C')\} \end{aligned}$$

$$\Delta = \{has\_colour(node_1, C_1), has\_colour(node_2, C_2)\}$$

We apply the equality solver to obtain:

$$\begin{aligned} G &= \{\forall N, M. \leftarrow has\_colour(N, red), edge(N, M), has\_colour(M, red)\} \\ &\cup \{\forall C', N', M'. \leftarrow C' = blue, has\_colour(N', C'), edge(N', M'), has\_colour(M', C')\} \end{aligned}$$

$$\Delta = \{has\_colour(node_1, C_1), has\_colour(node_2, C_2)\}$$

$$\theta = \{C/red\}$$

We apply A2 to obtain:

$$
\begin{array}{rcl}
G & = & \{\forall N, M. \leftarrow node_1 = N, C_1 = red, edge(N, M), has\_colour(M, red)\} \\
& \cup & \{\forall C, N, M. \leftarrow node_2 = N, C_2 = red, edge(N, M), has\_colour(M, red)\} \\
& \cup & \{\forall C', N', M'. \leftarrow C' = blue, has\_colour(N', C'), edge(N', M'), has\_colour(M', C')\}
\end{array}
$$

$$\Delta = \{has\_colour(node_1, C_1), has\_colour(node_2, C_2)\}$$

$$\Delta^* = \{\forall N, M. \leftarrow has\_colour(N, red), edge(N, M), has\_colour(M, red)\}$$

$$\theta = \{C/red\}$$

We apply the equality solver to obtain:

$$
\begin{array}{rcl}
G & = & \{\forall M. \leftarrow edge(node_1, M), has\_colour(M, red)\} \\
& \cup & \{\forall C, N, M. \leftarrow node_2 = N, C_2 = red, edge(N, M), has\_colour(M, red)\} \\
& \cup & \{\forall C', N', M'. \leftarrow C' = blue, has\_colour(N', C'), edge(N', M'), has\_colour(M', C')\}
\end{array}
$$

$$\Delta = \{has\_colour(node_1, C_1), has\_colour(node_2, C_2)\}$$

$$\Delta^* = \{\forall N, M. \leftarrow has\_colour(N, red), edge(N, M), has\_colour(M, red)\}$$

$$\theta = \{C/red, N/node_1, C1/red\}$$

We apply D2 to obtain:

$$
\begin{array}{rcl}
G & = & \{\forall M. \leftarrow node_1 = node_2, M = node_1 has\_colour(M, red)\} \\
& \cup & \{\forall M. \leftarrow node_1 = node_1, M = node_2 has\_colour(M, red)\} \\
& \cup & \{\forall C, N, M. \leftarrow node_2 = N, C_2 = red, edge(N, M), has\_colour(M, red)\} \\
& \cup & \{\forall C', N', M'. \leftarrow C' = blue, has\_colour(N', C'), edge(N', M'), has\_colour(M', C')\}
\end{array}
$$

$$\Delta = \{has\_colour(node_1, C_1), has\_colour(node_2, C_2)\}$$

$$\Delta^* = \{\forall N, M. \leftarrow has\_colour(N, red), edge(N, M), has\_colour(M, red)\}$$

$$\theta = \{C/red, N/node_1, C1/red\}$$

We apply the equality solver to obtain:

$$
\begin{array}{rcl}
G & = & \{\leftarrow has\_colour(node_2, red)\} \\
& \cup & \{\forall C, N, M. \leftarrow node_2 = N, C_2 = red, edge(N, M), has\_colour(M, red)\} \\
& \cup & \{\forall C', N', M'. \leftarrow C' = blue, has\_colour(N', C'), edge(N', M'), has\_colour(M', C')\}
\end{array}
$$

$$\Delta = \{has\_colour(node_1, C_1), has\_colour(node_2, C_2)\}$$

$$\Delta^* = \{\forall N, M. \leftarrow has\_colour(N, red), edge(N, M), has\_colour(M, red)\}$$

$$\theta = \{C/red, N/node_1, C1/red\}$$

We apply A2 and the equality solver to get:

$$
\begin{array}{rcl}
G & = & \{\leftarrow C_2 = red\} \\
& \cup & \{\forall C, N, M. \leftarrow node_2 = N, C_2 = red, edge(N, M), has\_colour(M, red)\} \\
& \cup & \{\forall C', N', M'. \leftarrow C' = blue, has\_colour(N', C'), edge(N', M'), has\_colour(M', C')\}
\end{array}
$$

$$\Delta = \{has\_colour(node_1, C_1), has\_colour(node_2, C_2)\}$$

$$\begin{aligned}\Delta^* &= \{\forall N, M. \leftarrow has\_colour(N, red), edge(N, M), has\_colour(M, red)\} \\ &\cup \{\leftarrow has\_colour(node_2, red)\}\end{aligned}$$

$$\theta = \{C/red, N/node_1, C1/red\}$$

E2 gives us two branches. We can have C_2/red which would then fail the next integrity constraint as C_1 must not equal C_2. Therefore we can add the inequality to the store and continue.

$$\begin{aligned}G &= \{\forall C, N, M. \leftarrow node_2 = N, C_2 = red, edge(N, M), has\_colour(M, red)\} \\ &\cup \{\forall C', N', M'. \leftarrow C' = blue, has\_colour(N', C'), edge(N', M'), has\_colour(M', C')\}\end{aligned}$$

$$\Delta = \{has\_colour(node_1, C_1), has\_colour(node_2, C_2)\}$$

$$\begin{aligned}\Delta^* &= \{\forall N, M. \leftarrow has\_colour(N, red), edge(N, M), has\_colour(M, red)\} \\ &\cup \{\leftarrow has\_colour(node_2, red)\}\end{aligned}$$

$$\varepsilon = \{C_2 \neq red\}$$

$$\theta = \{C/red, N/node_1, C_1/red\}$$

As mentioned before this results in the integrity constraint failing:

$$G = \{\forall C', N', M'. \leftarrow C' = blue, has\_colour(N', C'), edge(N', M'), has\_colour(M', C')\}$$

$$\Delta = \{has\_colour(node_1, C_1), has\_colour(node_2, C_2)\}$$

$$\begin{aligned}\Delta^* &= \{\forall N, M. \leftarrow has\_colour(N, red), edge(N, M), has\_colour(M, red)\} \\ &\cup \{\leftarrow has\_colour(node_2, red)\}\end{aligned}$$

$$\varepsilon = \{C_2 \neq red\}$$

$$\theta = \{C/red, N/node_1, C_1/red\}$$

And then if we follow the same process as before we end up with:

$$\Delta = \{has\_colour(node_1, C_1), has\_colour(node_2, C_2)\}$$

$$\theta = \{C_1/red\, C_2/blue\}$$

31.2 Clp-Ex1

This example was taken from [Abduction].

```
abducible(a/1).
abducible(b/1).

p(X, Y) :- a(X), q(Y).

q(Y) :- Y in [1,2,3,4], b(Y).

ic :- a(X), not X in [2,3].
ic :- a(X), b(Y), X >= Y.
```

Given a query $Q = \{p(X, Y)\}$ we are attempting to find an explanation, that is, an assignment to X and Y. This example makes use of the constraint solver.

The initial state is:

$$G = \{p(X, Y)\} \cup IC, ST = \{\emptyset, \emptyset, \emptyset\}$$

We apply D1 to get:

$$G = \{a(X), q(Y)\} \cup IC, ST = \{\emptyset, \emptyset, \emptyset\}$$

We apply A1 to get:

$$G = \{q(Y)\} \cup IC, ST = \{\{a(X)\}, \emptyset, \emptyset\}$$

We apply D1 to get:

$$G = \{Yin[1, 2, 3, 4], b(Y)\} \cup IC, ST = \{\{a(X)\}, \emptyset, \emptyset\}$$

We apply F1 to get:

$$G = \{b(Y)\} \cup IC, ST = \{\{a(X)\}, \emptyset, \{Y\ in\ [1, 2, 3, 4]\}\}$$

We apply A1 to get:

$$G = \{\leftarrow a(X'), \neg X'in[2, 3]. \cup IC*, ST = \{\{a(X), b(Y)\}, \emptyset, \{Y\ in\ [1, 2, 3, 4]\}\}$$

We apply A2 to get:

$$G = \{\leftarrow \neg Xin[2, 3]. \cup IC*, ST = \{\{a(X), b(Y)\}, \{\leftarrow a(X'), \neg X'in[2, 3]\}, \{Y\ in\ [1, 2, 3, 4]\}\}$$

We apply N2 to get:

$$G = \{Xin[2, 3]. \cup IC*, ST = \{\{a(X), b(Y)\}, \{\leftarrow a(X'), \neg X'in[2, 3]\}, \{Y\ in\ [1, 2, 3, 4]\}\}$$

And F1 to get:

$$G = IC*, ST = \{\{a(X), b(Y)\}, \{\leftarrow a(X'), \neg X'in[2, 3]\}, \{Y\ in\ [1, 2, 3, 4], X\ in\ [2, 3]\}\}$$

After several more steps we end up with $\{a(X), b(y)\}$ in the abducible store and $Yin[1, 2, 3, 4], Xin[2, 3], notX >= Y$ in the constraint store. Execution of the constraint solver leaves us with the following solutions:

- $a(2), b(3)$
- $a(2), b(4)$
- $a(3), b(4)$

32 JALPS Example Code

32.1 Insane

```
boy(bob).
boy(peter).
boy(max).

girl(jane).
girl(maria).

insane(maria).

likes(X,Y) :- boy(X), girl(Y), not insane(Y).
likes(X,Y) :- girl(X), boy(Y).
```

32.2 Circuits Analysis

This example was taken from [Abduction]

```
output(Gate, Value) :-
inverter(Gate),
input(Gate,InValue),
opposite(InValue,Value),
not broken(Gate).

output(Gate, InValue) :-
inverter(Gate),
input(Gate,InValue),
broken(Gate).

input(g2,Value):- output(g1,Value).
input(g3,Value):- output(g2,Value).
input(g1,on).

inverter(g1).
inverter(g2).
inverter(g3).

opposite(on,off).
opposite(off,on).

ic :- output(Gate, on),output(Gate, off).
ic :- input(Gate,on),input(Gate,off).

abducible(broken/1).
```

32.3 Genes

This example was taken from [Abduction]

```
produce(Enz) :- code(Gene,Enz), express(Gene).

express(Gene) :- on(Gene,Operon),bind(polymerase,Operon), not bind(repressor,Operon).

bind(polymerase,operon) :- bind(activator,operon).
bind(activator,operon) :- bind(cAMP,activator).
bind(cAMP,activator) :- highconcentration(cAMP), not present(sugar).
bind(repressor,operon) :- not highconcentration(allolactose), present(sugar).

highconcentration(cAMP) :- present(lactose), absent(sugar).
highconcentration(allolactose) :- absent(glucose).
highconcentration(allolactose) :- not present(allosugar).

code(lac(z),galactosidase).
code(lac(y),permease).

on(lac(X),operon).
```

```
abducible(present/1).
abducible(absent/1).

ic:- present(X), absent(X).

% ic:- absent(sugar), absent(glucose).
```

32.4 Block World

This example was taken from [NU04]

```
initially(1,2).
initially(2,3).
initially(3,0).

ablock(X) :- X in [1,2,3].
location(X) :- X in [0,1,2,3].
time(X) :- X in [1,2,3,4,5,6].

on(X,Y,T) :- initially(X,Y), not moved(X,Y,0,T).
on(X,Y,T) :- move(X,Y,E), E<T, not moved (X,Y,E,T).
moved(X,Y,E,T) :- move(X,Z,C), not Z=Y, between (C,E,T).
between(C,E,T) :- E <= C, C < T.

succeeds_move(X,Y,E) :- ablock(X), location(Y), time(E), not X=Y, clear_block(X,E),
                        clear_location(Y,E).

clear_block(X,E) :- not something_on(X,E).
something_on(X,E) :- on(Y,X,E).

clear_location(0,E).
clear_location(Y,E) :- not Y = 0, clear_block(Y,E).

ic :- move(X,Y,E), not succeeds_move(X,Y,E).

ic :- move(X,Y1,T), move(X,Y2,T), not Y1=Y2.
ic :- move(X1,Y,T), move(X2,Y,T), not X1=X2.

abducible(move).
```

Part IX
User Guide

33 Java API

33.1 Abductive Logic Programming

The JALP API is provided for integration of ALP into java software projects. Logical elements such as predicates and variables are constructed from a factory-like static class and used to construct the rules, denials and abducibles that make up an abductive framework. This abductive framework is then passed to a solver which then generates a list of Result objects which act as abductive explanations. This API is presented below:

The following objects represent truth and falsity ($\top$ and $\bot$):

- TrueInstance makeTrueInstance()
- FalseInstance makeFalseInstance()

Constants can either be character based or have an integer value. This affects the way in which they are evaluated. Constraints consisting of character constants are handled natively whereas constraints involving integers are handled by the Choco constraint solver:

- CharConstantInstance makeCharConstantInstance(String string)
- IntegerConstantInstance makeIntegerConstantInstance(int n)

Variables can be instantiated as follows:

- VariableInstance makeVariableInstance(String name)

Predicates can be instantiated using either a list or an array to represent parameters:

- PredicateInstance makePredicateInstance(String name, List <IUnifiableAtomInstance> parameters)
- PredicateInstance makePredicateInstance(String name, IUnifiableAtomInstance ... parameters)

Equalities and inequalities:

- EqualityInstance makeEqualityInstance(IUnifiableAtomInstance left, IUnifiableAtomInstance right)
- InEqualityInstance makeInEqualityInstance(IUnifiableAtomInstance left, IUnifiableAtomInstance right)
- InEqualityInstance makeInEqualityInstance(EqualityInstance equalityInstance)

Lists can be instantiated using various different data structures. Lists can either consist of IntegerConstantInstances or CharConstantInstances:

- CharConstantListInstance makeCharConstantListInstance(Collection<CharConstantInstance> constants)

- CharConstantListInstance makeCharConstantListInstance(CharConstantInstance[] constants)
- CharConstantListInstance makeCharConstantListInstance(String ... strings)
- CharConstantListInstance makeCharConstantListInstance()
- IntegerConstantListInstance makeIntegerConstantListInstance(Collection<IntegerConstantInstance> integers)
- IntegerConstantListInstance makeIntegerConstantListInstance(IntegerConstantInstance[] integers)
- IntegerConstantListInstance makeIntegerConstantListInstance(int ... integers)
- IntegerConstantListInstance makeIntegerConstantListInstance()

Negation or 'not' is applied to inferebles through the use of a wrapper object:

- NegationInstance makeNegationInstance(IInferableInstance subformula)

Constraints can be constructed using the following methods:

- InIntegerListConstraintInstance makeInIntegerListConstraint(ITermInstance left, IntegerConstantListInstance right)
- InConstantListConstraintInstance makeInConstantListConstraint(ITermInstance left, CharConstantListInstance right)
- GreaterThanConstraintInstance makeGreaterThanConstraintInstance(ITermInstance left, ITermInstance right)
- GreaterThanEqConstraintInstance makeGreaterThanEqConstraintInstance(ITermInstance left, ITermInstance right)
- LessThanConstraintInstance makeLessThanConstraintInstance(ITermInstance left, ITermInstance right)
- LessThanEqConstraintInstance makeLessThanEqConstraintInstance(ITermInstance left, ITermInstance right)
- NegativeConstraintInstance makeNegativeConstraintInstance(ConstraintInstance constraintInstance)

There are multiple ways of constructing denials:

- DenialInstance makeDenialInstance(List<VariableInstance> universalVariables,List<IInferableInstance> body)
- DenialInstance makeDenialInstance(List<VariableInstance> universalVariables,IInferableInstance ... body)
- DenialInstance makeDenialInstance(List<IInferableInstance> body)
- DenialInstance makeDenialInstance(IInferableInstance ... body)
- DenialInstance makeDenialInstance()

Rules are constructed in much the same way as denials however can be defined as 'facts' i.e. rules without bodies.

- Definition makeFact(String headPredicateName, IUnifiableAtomInstance[] headParameters)
- Definition makeFact(String headPredicateName, List<IUnifiableAtomInstance> headParameters)
- Definition makeRule(String headPredicateName, IUnifiableAtomInstance[] headParameters, IInferableInstance[] body)
- Definition makeRule(String headPredicateName, List<IUnifiableAtomInstance> headParameters, IInferableInstance[] body)
- Definition makeRule(String headPredicateName, IUnifiableAtomInstance[] headParameters, List<IInferableInstance> body)
- Definition makeRule(String headPredicateName, List<IUnifiableAtomInstance> headParameters, List<IInferableInstance> body)

Usage of the system involves construction of the abductive theory using the above methods. Definitions, denials and abducibles are added to an AbductiveFramework object. A query is then constructed which takes the form of a String, a list of PredicateInstances or a single PredicateInstance. A JALPSystem object is then instantiated using the AbductiveFramework. which is responsible for processing queries. The query object is passed to the processQuery method of JALPSystem which then returns a list of Result objects. The Result object contains an ASystem store and a mapping of assignments that act as an abductive explanation.

The following is a simple example operating on an abductive theory that only contains a single fact.

```
import static uk.co.mtford.jalp.JALP;
import uk.co.mtford.jalp.*;
import uk.co.mtford.jalp.abduction.JALPSystem;

public static void main(String[] args) {
    /* Setup logic theory. */
    CharConstantInstance john = JALP.makeCharConstantInstance("john");
    CharConstantInstance jane = JALP.makeCharConstantInstance("jane");
    Definition def = makeFact(likes,john,jane);

    AbductiveFramework framework = new AbductiveFramework();
    framework.addDefinition(def);

    /* Setup system. */
    JALPSystem system = new JALPSystem(framework);

    /* Setup query. */
    VariableInstance X = JALP.makeVariableInstance("X");
    VariableInstance Y = JALP.makeVariableInstance("Y");
    PredicateInstance query = makePredicateInstance("likes",X,Y);

    /* Execute query. */
    List<Result> results = system.processQuery(query);
    // List<Result> results = system.processQuery("likes(X,Y)");
```

```
    for (Result r:results) {
r.reduce() // Get rid of irrelevant items from the store.
    }
}
```

The following is an example showing usage of constraints and rules.

```
import static uk.co.mtford.jalp.JALP;
import uk.co.mtford.jalp.*;
import uk.co.mtford.jalp.abduction.JALPSystem;

public static void main(String[] args) {

    /* Setup logic theory. */
    VariableInstance X = JALP.makeVariableInstance("X");
    VariableInstance Y = JALP.makeVariableInstnace("Y");

    IntegerConstantListInstance list1 = JALP.makeIntegerConstantListInstance(1,2,3);
    IntegerConstantListInstance list2 = JALP.makeIntegerConstantListInstance(1,2);

    InIntegerListConstraintInstance in1 = JALP.makeInIntegerListConstraint(X,list1);
    InIntegerListConstraintInstance in2 = JALP.makeInIntegerListConstraint(Y,list2);
    LessThanConstraintInstance lessThan = JALP.makeLessThanConstraintInstance(X<Y);

    VariableInstance X2 = JALP.makeVariableInstance("X");
    Definition p = makeRule("p", {X2}, in1, in2, lessThan);

    AbductiveFramework framework = new AbductiveFramework();
    framework.addDefinition(p);

    /* Setup system. */
    JALPSystem system = new JALPSystem(framework);

    /* Execute query. */
    try {
        List<Result> results = system.processQuery("p(X)");
        for (Result r:results) {
            System.out.println(r.toString());
        }
    }
    catch (ParseException e) {
        e.printStackTrace();
    }

}
```

33.2 Unification

The unification implementation can be leveraged by users who are not interested in using the abductive system. This simply involves constructing two UnifiableInstances and then calling the unify the method to obtain a boolean representing whether or not unification was successful and also the *Map* that represents the substitutions made during unification.

```
import static uk.co.mtford.jalp.JALP;

public static void main(String[] args) {

    VariableInstance X = JALP.makeVariableInstance("X");
    ConstantInstance jane = JALP.makeConstantInstance("jane");

    PredicateInstance p1 = JALP.makePredicateInstance("p", X);
    PredicateInstance p2 = JALP.makePredicateInstance("p", jane);

    Map<VariableInstance,IUnifiableInstance> assignment
            = new HashMap<VariableInstance, IUnifiableInstance>();
    boolean success = p1.unify(p2,assignments);

    if (success) {
        System.out.println(assignment.toString());
    }
    else {
        System.out.println("failed");
    }

}
```

33.3 Equality Solver

The equality solver implementation can also be leveraged by users who are not interested in using the abductive system. This involves construction of an *EqualitySolver* object to which we then pass a list of equalities. Success is returned by the method and substitutions are added to the Map that is passed to the solver.

```
import static uk.co.mtford.jalp.JALP;

public static void main(String[] args) {

    VariableInstance X = JALP.makeVariableInstance("X");
    ConstantInstance jane = JALP.makeConstantInstance("jane");

    EqualityInstance e = JALP.makeEqualityInstance(X,jane);

    Map<VariableInstance, IUnifiableAtomInstance> assignments
        = new HashMap<VariableInstance, IUnifiableAtomInstance>();
    boolean success = e.equalitySolve(assignments);

    System.out.println(assignments);

}
```

33.4 Inequality Solver

The inequality solver implementation can also be leveraged by users who are not interested in using the abductive system. This is performed in a similar fashion to the equality solver. An *InEqualitySolver* object is constructed to which we then pass an InEqualityInstance. This is

then broken down to pairs of EqualityInstances and InEqualityInstances all of which are possible combinations that would make the original inequality true.

```
import static uk.co.mtford.jalp.JALP;

public static void main(String[] args) {

    VariableInstance X = JALP.makeVariableInstance("X");
    ConstantInstance jane = JALP.makeConstantInstance("jane");

    InEqualityInstance e = JALP.makeInEqualityInstance(john,jane);

    InEqualitySolver solver = new InEqualitySolver();

    List<Pair<List<EqualityInstance>,List<InEqualityInstance>> possibleEqualities = solver.execute

    assert(!list.isEmpty()); // Should produce an empty list.

}
```

33.5 Constraint Solver

The constraint solver implementation can also be leveraged by users who are not interested in using the abductive system. This involves the construction of constraints and terms using the JALP Java API followed by construction of a *ChocoConstraintSolverFacade* object. We then pass the constructed constraints as arguments to this object and receive the possible substitutions to the variables within the constraints, if any exist.

```
import static uk.co.mtford.jalp.JALP;

public static void main(String[] args) {

    VariableInstance X = JALP.makeVariableInstance("X");
    VariableInstance Y = JALP.makeVariableInstnace("Y");

    IntegerConstantListInstance list1 = JALP.makeIntegerConstantListInstance(1,2,3);
    IntegerConstantListInstance list2 = JALP.makeIntegerConstantListInstance(1,2);

    InIntegerListConstraintInstance in1 = JALP.makeInIntegerListConstraint(X,list1);
    InIntegerListConstraintInstance in2 = JALP.makeInIntegerListConstraint(Y,list2);
    LessThanConstraintInstance lessThan = JALP.makeLessThanConstraintInstance(X<Y);

    LinkedList<IConstraintInstance> constraintList = new LinkedList<IConstraintInstance>();
    constraintList.add(in1);
    constraintList.add(in2);
    constraintList.add(lessThan);

    Map<VariableInstance, IUnifiableAtomInstance> assignments
        = new HashMap<VariableInstance, IUnifiableAtomInstance>();

    ChocoConstraintSolverFacade cs = new ChocoConstraintSolverFacade();
```

```
    List<Map<VariableInstance, IUnifiableAtomInstance>> results
        = cs.executeSolver(assignments, constraintList);

    for (Map<VariableInstance, IUnifiableAtomInstance> r:results) {
        System.out.println(r.toString());
    }

}
```

34 Syntax

JALP can parse prolog style abductive theories via a command line interface as an alternative to using the Java API. This section provides a brief guide to the syntax involved.

34.1 Data Types

The following data types are supported by JALP:

- Variables e.g. X, Y
- Integer Constants e.g. $1, 2, 3$
- Character Constants e.g. $john, jane, bob$
- Lists e.g. $[1, 2, 3], [john, jane, bob]$

34.2 Definitions

A definition is also known as a rule, horn clause or clause. A definition without a body is known as a fact. Various examples are displayed below:

```
likes(john,jane).
boring(X) :- likes(X,jane).
normal(X) :- likes(X,Y).

p(Y,Z) :- q(Y), not r(Z).
```

34.3 Integrity Constraints

Integrity constraints constrain the explanations that can be collected.

```
ic :- migraine(X), not headache(X).
```

This means that for X to have a migraine, they must have a headache. The following would not be valid however and is considered unsafe:

```
ic :- not headache(X), migraine(X).
```

34.4 Equalities

Equalities are used to perform unification and take the following form:

```
p(X,Y) :- X=Y.
q(X,Y) :- X=jane, Y=john.
```

34.5 Inequalities

JALP has an inequality solver and can be used to constrain the value of variables.

```
p(X,Y) :- X!=Y.
q(X,Y) :- X!=jane, Y!=john.
```

34.6 Finite Domain Constraints

The constraints available are as follows:

- List membership: X in [1...3], john in [john,jane,bob].
- Greater than/equal to: $X > 2$, $X \geq 2$.
- Less than/equal to: $X < 2$, $X \leq 2$.

Before a constraint between two variables e.g. X<Y can be evaluated the domain of each variable must be constructed i.e. they must be finite domain variables. For example the following is valid:

```
p(X) :- X in [1,2,3,4], Y < 2, Y > 5, X > Y.
```

But the follow is not as Y is in the domain $-\inf \rightarrow 1$

```
p(X) :- X in [1..4], Y < 2, X > Y.
```

34.7 Abducibles

Abducibles are predicates that can be used in an abductive explanation. They are established using a special fact predicate *abducible*/1 e.g.

```
abducible(girl/1).
```

An example program using abducibles is:

```
boy(john).
likes(X,Y) :- boy(X), girl(Y).

abducible(girl/1).
```

This means that boys only like girls, and girls like nobody. If a query (observation) suggests that somebody likes someone else, then the abductive solver will compute an explanation involving the abducible.

35 Command Line

The JALP command line interface is used to parse theory files return in the syntax specified in the previous section. Any number of theory files can be parsed and merged into one abductive theory. The command line executable takes the following form:

```
$ java -jar jalp (filename)+ -q <query> [-r | -d]
```

The options are as follows:

- -q: Signifies the query that should be executed on the theory.

- -r: Performs a reduction on the results i.e. remove irrelevant collectables, reduce assignments and perform substitutions.
- -d: Debug mode. Generates a HTML visualisation of the derivation tree and a log file at the DEBUG level.

Figure 34 is a screenshot of the command line tool in action.

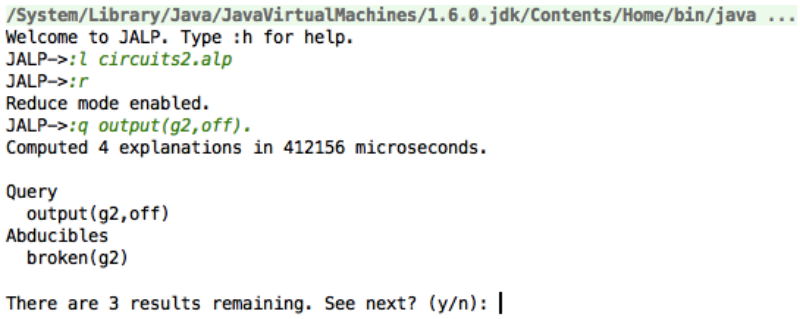


Figure 34: Command Line

36 Interpreter

The interpreter presents a prolog like interface for manual specification of abductive theories and execution of queries on those theories. The interpreter can be started as follows:

```
$ java -jar jalp
Welcome to JALP. Enter :h for help.
JALP->
```

or with files:

```
$ java -jar jalp file1.alp file2.alp
Loading file1.alp...
Loading file2.alp...
Welcome to JALP. Enter :h for help.
JALP->
```

There are multiple options available for interaction with the interpreter:

- :l <filename> - Load a file.
- :q <query> - Execute a query.
- :f - View framework.
- :c - Clear theory.
- :r - Enable reduce mode.
- :q - Quit.

:l can be used with multiple file names. The abductive theory from all files will be merged into one.

Reduce mode is enabled with the command :r and means that results from queries are tidied up with irrelevant constraints, equalities etc from the store being removed to present a clearer explanation. A typical usage is as follows:

```
JALP->likes(john,jane).
JALP->:f
Program
  likes(john,jane).
JALP->:q likes(X,Y).

Substitutions
  Y=jane
  X=john
JALP->
```

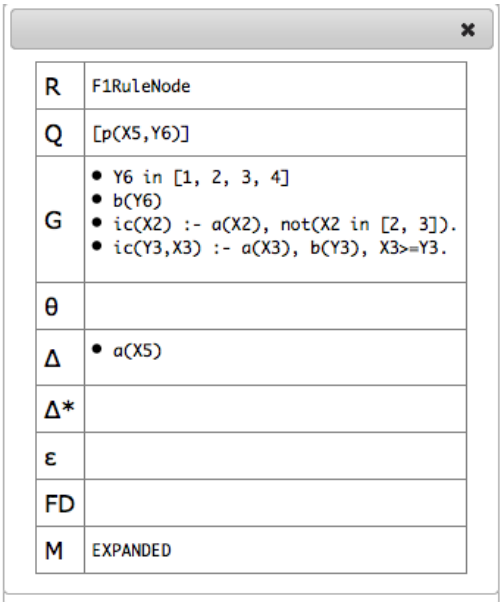
37 Visualizer

The visualizer can be accessed via the java API or via the command line in debug mode as mentioned earlier.

```
$ java -jar jalp file.alp -d <filepath>
```

The above command will generate a log file and the visualizer html at the folder specified.

The visualizer is a simple javascript tree structure. Clicking on the node will bring up a popup window with details about that node including the state of the store, assignments, the next rule that will be applied etc. The tree can also be collapsed to make it easier to move through. Figure 35 shows a screenshot of the information window, figure 36 shows a screenshot of a derivation tree.



R	F1RuleNode
Q	[p(X5,Y6)]
G	• Y6 in [1, 2, 3, 4] • b(Y6) • ic(X2) :- a(X2), not(X2 in [2, 3]). • ic(Y3,X3) :- a(X3), b(Y3), X3>=Y3.
θ	
Δ	• a(X5)
Δ*	
ε	
FD	
M	EXPANDED

Figure 35: Visualizer Info Pane

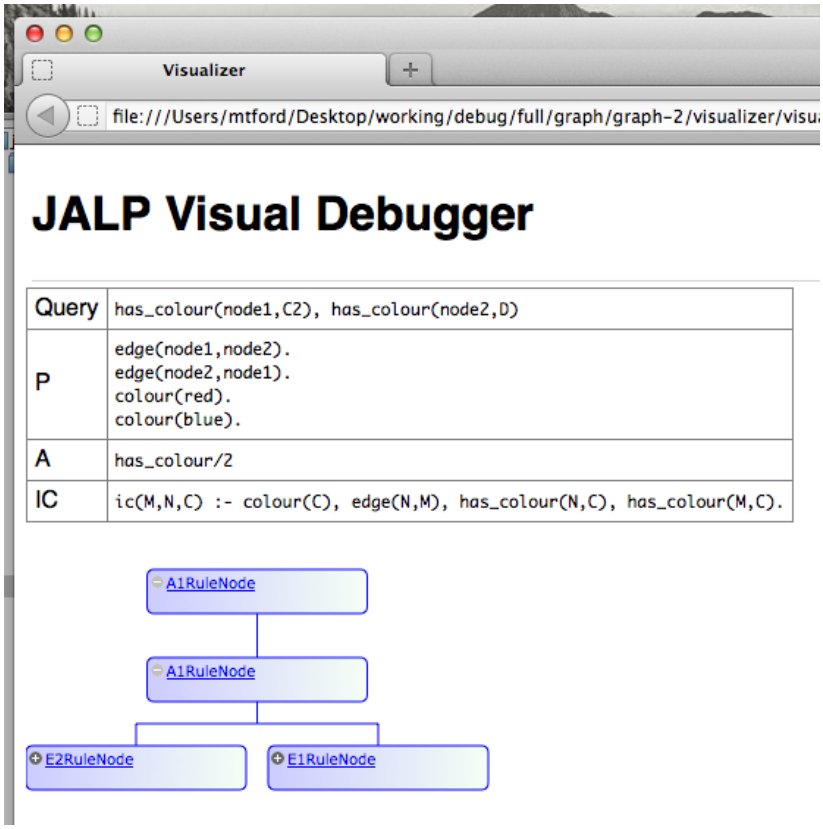


Figure 36: Visualizer Derivation Tree