FARS : Functional Automated Reasoning System - v1.0

Caner Derici

January 15, 2013

Contents

Chapter 1

General Overview

FARS is a resolution-paramodulation style automated theorem prover, targeting first-order logic with equality. It's coded in Racket/PLAI language[1]. Its primary design goal is to be able to navigate very quickly in large search spaces and to cover a lot of space in a very short amount of time. Being coded in a functional style (although not purely functional) provides FARS to be highly modular in terms of heuristic usage and various compiler level optimizations that are accustomed to by many functional languages, such as tail call optimization.

This document describes the current status of FARS, as well as serves as a documentation and a reference. Chapter 2 explains the theoretical concerns, such as the inference rules and heuristics used as well as the soundness and completeness. Chapter 3 describes the implementation and usage of FARS with as much example programs as possible. Chapter 4 shows several test runs with real theorems (which are given input to and solved by OTTER). Finally, Chapter 5 includes a discussion about the further development of FARS. Additionally, a complete reference of FARS can be found in Appendix A at the end of the document.

[1]For the details of these language, refer to Section B.1

Chapter 2

Theoretical Perspective

2.1 General Characteristics

FARS is a resolution/paramodulation theorem prover which mainly employs the set of support strategy, along with hyperresolution, which is a special case of semantic resolution. It is designed to prove theorems expressed in first-order predicate logic with equality. Given the axioms and the negated conclusion as clauses[1], FARS tries to prove it by resolution-refutation method. Furthermore, all of the proofs in FARS are refutations (by contradiction).

(REWRITE HERE) The main application areas of FARS in researches are generally abstract algebra and formal logic. It has been successful in answering many open questions in subfields of this sort, such as combinatory logic and lattice theory .

2.1.1 Inference Rules and General Strategy

The inference rules in FARS are generally based on resolution (with factorization) and paramodulation. In particular, FARS employs as inference rules binary resolution and hyperresolution. Additionally, equational deduction is done with binary paramodulation. Other attributes of FARS includes also forward and backward subsumption.

[1] See Section 3.2 for more detailed discussion.

2.2 Soundness & Completeness

Generally, real automated reasoning systems (like OTTER) using resolution/paramodulation depend highly on search on huge spaces, thus they are implemented with numerous hacks and manual code-level optimizations. Therefore, it is highly impractical to prove something *about* them. However, although a formal verification to any of the parts of the implementation of FARS has not been performed, its functional implementation allows a neat architecture for the system to be easily verified by automated verification systems. At this level, the output produced by FARS need to be manually checked by a human user.

FARS's individual parts are designed and programmed to not affect the completeness of any of the theoretical aspects that is being implemented. Although some known weak points exist (see Section 2.3), FARS completeness is established at a significantly desirable level. Full resolution is achieved by the repeated application of binary resolution, in where also the factorization is plugged. Forward and backward subsumptions are not affecting completeness. However, intentional constraints which affects the completeness for the sake of efficiency can be imposed manually, such as the *max-step* parameter (which is false by default).

2.3 Known Weak Points

Due to the shortness of the implementation time, FARS include some known weak points that is to be solved in future versions to be developed.

One of the weak points is in the factorization. FARS's binary resolution adopt the idea that a clause can be inferred in four ways from its parents (C_1 & C_2):

- Resolution of C_1 and C_2
- Resolution of a factor of C_1 and C_2
- Resolution of C_1 and a factor of C_2
- Resolution of a factor of C_1 and a factor of C_2

Once FARS infer a resolvent from any of the possibilities above, it records C_1 and C_2 to what is called the *main-memory*, and never try to resolve C_1

and C_2 again. This obviously ignores the possibility that more than one resolvents can be inferred from C_1 and C_2 that involves factorization, which in turn affect the completeness.

2.4 Heuristics Used

FARS uses several heuristic approaches to different kinds of tasks, which are all governed by external heuristic functions provided as a parameter to the prover. This makes possible in future versions for heuristic functions to be provided even from the input files for individual theorems.

Several points where a heuristic approach is being employed include:

- Clause differentiation to generate nucleus and satellite for semantic resolution. Currently used built-in heuristic function implements hyperresolution by putting all the positive clauses to nucleus and all the others to satellite.
- Picking clauses from the set to be tried to resolve upon. Current heuristic chooses the clause with has the smallest number of literals.
- Predicate ordering to properly implement semantic resolution. Predicate ordering is automatically constructed from the given set of clauses by a heuristic function, again provided as input to the initiator. Current heuristic generates the ordering by putting the highest presedence to the mostly used predicate in the input. Additionally, by prove-for-all-pred-orders function, theorem can be tried to be proved by all possible permutations of the predicates one-by-one.

Chapter 3

Practical Perspective

FARS is written in Racket language using DrRacket environment. This chapter explains the general syntax and semantics of the Racket, as well as the basic usage of the DrRacket programming environment. The reason that the FARS is coded in Racket will be explained in the subseqent sections. At the most general view, FARS currently has 4 different purpose modules which have more than 1000 lines of codes in total.

3.1 Racket Language and DrRacket Environment

In Racket documentation, it is described informally as follows.

> Depending on how we look at it, Racket is
>
> - a programming language - a dialect of Lisp and a descendant of Scheme language;
> - a family of programming languages - variants of Racket, and more; or
> - a set of tools - for using a family of programming languages

In this document, Racket is simply referred as a system of a general syntax (will be explained momentaily) and DrRacket environment.

Generally, Racket is a functional mainstream programming langauge, features (most importantly for us) first-class functions and tail-call optimization. Considering functions as first-class values, Racket provides a powerful

programming framework, on which we can build theoretically the most general abstractions that could be created, such as function generating functions generating functions etc.

Furthermore, Racket as a system, provides many programming langauges with different characteristics. FARS is built with the PLAI[1] langauge inside Racket, which is mainly used for studying the design and interpretation of programming langauge characteristics. The purpose of using particularly the PLAI language, is that the abstractions such as "define-type" and "type-case" are very useful in modeling and processing the statements of any formally representable language, such as first-order predicate logic in our case.

The predominant disadvantage of using PLAI langauge is that it's evaluation is done via applicative order, which means a functions body is evaluated *after* the arguments are evaluated first. When proving theorems, in many cases we do not perform anything on the whole data set, in other words, at each step there is at least one item which we don't touch. For example, consider the clauses choosen to be resolved upon from nucleus and satellite sets. Passing these sets as arguments to functions after functions, Racket evaluates all clauses within these sets although our algorithm doesn't touch them. Using a langauge that has a normal evaluation order in which the arguments are evaluated in a need basis would drastically improve the efficiency.

For more detailed overwiev of Racket language and DrRacket environment, as well as a list of built-in function explanations that will help understanding the implementation, please refer to Appendix B.

FARS is currently a non-interactive program. It reads its input from and writes the output to standard input and output respectively. It is ran on DrRacket. However, with the help of unix operators $<$ and $>$ which helps the files to interact with the standard i/o, it is possible for FARS to be modified easily such that it takes input from directly from the console, without using DrRacket. Currently all the options and parameters are given in the source file, but it is fairly easy to implement the necessary functionality to read parameters from individual theorem input files.

[1] Programming Langauge Application and Interpretation

3.2 Input & Output

FARS accepts statements as disjuncted clauses, as all of FARS's inference rules and search algorithms operate on clauses.

3.2.1 General Syntax and Naming

The general format of an input file to FARS is very simple. There can be comment lines, "negated_conclusion" and the clauses. The clauses must be given in per line basis, i.e. there can be no clause seperated by a newline (\n). Thus there is no need for the terms and statements to be terminated with periods. The propositional connectives such as "not" and "or" can be written with intuitive symbols such as "-" and "|" respectively.

The output of FARS is divided into three parts. In the first part, FARS writes the input (without the comments) with the clauses augmented with their identification number and predicate counts. If the *starter-verbose* flag is set to be true, the second part where the individual search steps (involving resolvents and factorizations) are produced as the search goes on. Finally if FARS found a proof, it produces a human readable proof which can be traced and checked easily.

Following is a simple theorem from TGTP;

```
; A.THM  See TGTP, Chapter 6, Section 6.1
P(x) | ~Q(x)

Q(a()) | Q(b())

negated_conclusion
~P(x)
```

When the above example input is given to FARS, it easily finds the solution and outputs as follows:

```
Initiating the input :
<ID> <predCount> <actualClause>
(1 2 (P(x) ~Q(x)))
(2 2 (Q(a) Q(b)))
(3 1 (~P(x)))
```

```
Initial predicate order:
(Q P)



proof-found! unit conflict.

From 1 and 2, generate -> 4 - P(b) | Q(a)

From 3 and 4, generate -> 5 - Q(a)

From 1 and 5, generate -> 6 - P(a)

From 6 and 3 ---> F
'proof-found!
```

3.2.2 Restrictions in Input

Current version of FARS impose some restrictions for the format in input files. These restrictions are:

- There can be no whitespaces among the arguments of a predicate or a function, i.e. anywhere between (and).
- There can be no predicates without a variable, so inputs such as
  A | B should be turned into A(x) | B(x).
- There can be no tab (\t) characters
- Comment symbol (;) must be at the beginning of the line. inputs like
  A | B ; axiom 6
  are not permitted.
- Variable or constant symbols should be one letter.

These restrictions are to be fixed in further versions of FARS.

3.2.3 Naming

FARS has a simple naming characteristic. If a simple term occurs in a clause, the default behavior is to look at the symbol to decide its type. If it starts with *x,y,z,g,t,u,v*, then it is classified as a variable, and if starts with *a,b,c,d,e*, it is classified as a constant.

3.3 The Main Loop

FARS's primary inference mechanism is semantic resolution. Current clause differentiator generates the nucleus and satellite sets by being positive or mixed, just as in hyperresolution, however, this can be easily modified since the differentiator function is provided as a parameter to the prover.

As mentioned before, FARS operates on mainly two different sets called nucleus and satellite, which by default contains all positive and mixed clauses respectively. The algorithm is as follows:

While (proof-nucleus-memory not contains all of nucleus clauses and no refutation has been found)

1. Let *nuc_clause* be the "best" clause in nucleus
2. Choose a suitable *sat_clause* from satellite which contains a negated version of a predicate of *nuc_clause*
3. Infer the *new_clause* using binary resolution and factorization.
4. Merge identical literals and apply factor-simplification
5. If *new_clause* has 0 literals, a refutation has been found, stop and initiate the proof printing sequence.
6. If max-literals parameter is set, check the number of literals and discard *new_clause* if there are too many
7. If *new_clause* is a unit clause, check for unit conflicts
8. Put *new_clause* to nucleus if it is a positive clause, otherwise, put it to satellite

---

Chapter 4

Test Drive

In this section, we run FARS on 4 different theorems with somewhat similar characteristics. We provide both the input and output and give (although not line-by-line) a small comment on the problem.

4.1 Pigeon Hole Problem

The first example is the famous "Pigeon Hole Problem", stated as axioms and a negated conclusion. This particular instance of the problem involves 5 pigeons and 4 holes, and FARS proves that one pigeon must have been left out. Below are the input and the output produced by FARS.

```
; %%%%%%%% PIGEON HOLE PROBLEM %%%%%%%%

; Every pigeon flies into a hole.

p00(a) | p01(a) | p02(a) | p03(a)
p04(a) | p05(a) | p06(a) | p07(a)
p08(a) | p09(a) | p10(a) | p11(a)
p12(a) | p13(a) | p14(a) | p15(a)
p16(a) | p17(a) | p18(a) | p19(a)

; Each hole holds at most one piegeon.

~p00(a) | ~p04(a)
~p00(a) | ~p08(a)
~p00(a) | ~p12(a)
~p00(a) | ~p16(a)
~p04(a) | ~p08(a)
~p04(a) | ~p12(a)
~p04(a) | ~p16(a)
~p08(a) | ~p12(a)
~p08(a) | ~p16(a)
~p12(a) | ~p16(a)
~p01(a) | ~p05(a)
```

```
~p01(a) | ~p09(a)
~p01(a) | ~p13(a)
~p01(a) | ~p17(a)
~p05(a) | ~p09(a)
~p05(a) | ~p13(a)
~p05(a) | ~p17(a)
~p09(a) | ~p13(a)
~p09(a) | ~p17(a)
~p13(a) | ~p17(a)
~p02(a) | ~p06(a)
~p02(a) | ~p10(a)
~p02(a) | ~p14(a)
~p02(a) | ~p18(a)
~p06(a) | ~p10(a)
~p06(a) | ~p14(a)
~p06(a) | ~p18(a)
~p10(a) | ~p14(a)
~p10(a) | ~p18(a)
~p14(a) | ~p18(a)
~p03(a) | ~p07(a)
~p03(a) | ~p11(a)
~p03(a) | ~p15(a)
~p03(a) | ~p19(a)
~p07(a) | ~p11(a)
~p07(a) | ~p15(a)
~p07(a) | ~p19(a)
~p11(a) | ~p15(a)
~p11(a) | ~p19(a)
~p15(a) | ~p19(a)

;; end-of-input
```

FARS output:

```
Initiating the input :
<ID> <predCount> <actualClause>
(1 4 (p00(a) p01(a) p02(a) p03(a)))
(2 4 (p04(a) p05(a) p06(a) p07(a)))
(3 4 (p08(a) p09(a) p10(a) p11(a)))
(4 4 (p12(a) p13(a) p14(a) p15(a)))
(5 4 (p16(a) p17(a) p18(a) p19(a)))
(6 2 (~p00(a) ~p04(a)))
(7 2 (~p00(a) ~p08(a)))
(8 2 (~p00(a) ~p12(a)))
(9 2 (~p00(a) ~p16(a)))
(10 2 (~p04(a) ~p08(a)))
(11 2 (~p04(a) ~p12(a)))
(12 2 (~p04(a) ~p16(a)))
(13 2 (~p08(a) ~p12(a)))
(14 2 (~p08(a) ~p16(a)))
(15 2 (~p12(a) ~p16(a)))
(16 2 (~p01(a) ~p05(a)))
(17 2 (~p01(a) ~p09(a)))
(18 2 (~p01(a) ~p13(a)))
(19 2 (~p01(a) ~p17(a)))
```

```
(20 2 (~p05(a) ~p09(a)))
(21 2 (~p05(a) ~p13(a)))
(22 2 (~p05(a) ~p17(a)))
(23 2 (~p09(a) ~p13(a)))
(24 2 (~p09(a) ~p17(a)))
(25 2 (~p13(a) ~p17(a)))
(26 2 (~p02(a) ~p06(a)))
(27 2 (~p02(a) ~p10(a)))
(28 2 (~p02(a) ~p14(a)))
(29 2 (~p02(a) ~p18(a)))
(30 2 (~p06(a) ~p10(a)))
(31 2 (~p06(a) ~p14(a)))
(32 2 (~p06(a) ~p18(a)))
(33 2 (~p10(a) ~p14(a)))
(34 2 (~p10(a) ~p18(a)))
(35 2 (~p14(a) ~p18(a)))
(36 2 (~p03(a) ~p07(a)))
(37 2 (~p03(a) ~p11(a)))
(38 2 (~p03(a) ~p15(a)))
(39 2 (~p03(a) ~p19(a)))
(40 2 (~p07(a) ~p11(a)))
(41 2 (~p07(a) ~p15(a)))
(42 2 (~p07(a) ~p19(a)))
(43 2 (~p11(a) ~p15(a)))
(44 2 (~p11(a) ~p19(a)))
(45 2 (~p15(a) ~p19(a)))


Initial predicate order:
(p00 p04 p08 p12 p16 p01 p05 p09 p13 p17 p02 p06 p10 p14 p18 p03 p07 p11 p15 p19)


Trying a clash with: 6 and 1 :
Using factor of both :  ->> (46 4 (~p04(a) p01(a) p02(a) p03(a)))

Trying a clash with: 6 and 2 :  failed.

Trying a clash with: 7 and 1 :
Using factor of both :  ->> (47 4 (~p08(a) p01(a) p02(a) p03(a)))

Trying a clash with: 7 and 3 :  failed.

...

Trying a clash with: 14 and 5 :  failed.

Trying a clash with: 15 and 4 :
Using factor of both :  ->> (55 4 (~p16(a) p13(a) p14(a) p15(a)))

Trying a clash with: 15 and 5 :  failed.

Trying a clash with: 16 and 1 :
Using factor of both :  ->> (56 4 (~p05(a) p00(a) p02(a) p03(a)))

...
```

```
Trying a clash with: 35 and 5 :  failed.

Trying a clash with: 36 and 1 :
Using factor of both :  ->> (76 4 (~p07(a) p00(a) p01(a) p02(a)))

Trying a clash with: 36 and 2 :  failed.

Trying a clash with: 37 and 1 :
Using factor of both :  ->> (77 4 (~p11(a) p00(a) p01(a) p02(a)))

Trying a clash with: 37 and 3 :  failed.

...

Trying a clash with: 45 and 4 :
Using factor of both :  ->> (85 4 (~p19(a) p12(a) p13(a) p14(a)))

...

Trying a clash with: 57 and 3 :  failed.

Trying a clash with: 56 and 2 :  failed.

Trying a clash with: 55 and 5 :
Using factor of both :  ->> (86 6 (p13(a) p14(a) p15(a) p17(a) p18(a) p19(a)))

...

Trying a clash with: 35 and 87 :
Using factor of both :
Using factor of both :  ->> (88 2 (p15(a) p19(a)))

Trying a clash with: 35 and 86 :
Using factor of both :
Using factor of both :  ->> (89 4 (p13(a) p15(a) p17(a) p19(a)))

...

Trying a clash with: 44 and 86 :  failed.

Trying a clash with: 45 and 90 :
Using factor of both :

proof-found! F is derived.

From 15 and 4, generate -> 55 - ~p16(a) | p13(a) | p14(a) | p15(a)

From 55 and 5, generate -> 86 - p13(a) | p14(a) | p15(a) | p17(a) | p18(a) | p19(a)

From 35 and 86, generate -> 89 - p13(a) | p15(a) | p17(a) | p19(a)

From 25 and 89, generate -> 90 - p15(a) | p19(a)

From 45 and 90 ---> F
'proof-found!
```

4.2 Lifschitz's theorem

This example demonstrates the use of prove-file-preds which tries to prove the theorem with all possible permutations of the predicate order. Note that starter-verbose flag is set to be false, thus the intermediate search steps are not included in the output.

The proof is as follows:

```
; LIFSCHTZ.THM  (Lifschitz's theorem)

; See Wos, Automated Reasoning, 33 Basic Research Problems, pages 150

negated_conclusion

P(f(x,y),f(x,y)) | ~S(x,f(x,y)) | ~Q(y,f(x,y))
P(f(x,y),f(x,y)) | ~S(x,f(x,y)) | Q(f(x,y),z)
P(f(x,y),f(x,y)) | ~S(x,f(x,y)) | ~S(y,y)

P(f(x,y),f(x,y)) | S(f(x,y),z) | ~Q(y,f(x,y))
P(f(x,y),f(x,y)) | S(f(x,y),z) | Q(f(x,y),w)
P(f(x,y),f(x,y)) | S(f(x,y),z) | ~S(y,y)

P(f(x,y),f(x,y)) | ~Q(z,z) | ~Q(y,f(x,y))
P(f(x,y),f(x,y)) | ~Q(z,z) | Q(f(x,y),z)
P(f(x,y),f(x,y)) | ~Q(z,z) | ~S(y,y)

~P(x,x) | ~S(x,f(x,y)) | ~Q(y,f(x,y))
~P(x,x) | ~S(x,f(x,y)) | Q(f(x,y),z)
~P(x,x) | ~S(x,f(x,y)) | ~S(y,y)

~P(x,x) | S(f(x,y),z) | ~Q(y,f(x,y))
~P(x,x) | S(f(x,y),z) | Q(f(x,y),w)
~P(x,x) | S(f(x,y),z) | ~S(y,y)

~P(x,x) | ~Q(y,y) | ~Q(z,f(x,z))
~P(x,x) | ~Q(y,y) | Q(f(x,z),y)
~P(x,x) | ~Q(y,y) | ~S(z,z)

S(x,y) | ~S(y,f(y,z)) | ~Q(z,f(y,z))
S(x,y) | ~S(y,f(y,z)) | Q(f(y,z),w)
S(x,y) | ~S(y,f(y,z)) | ~S(z,z)

S(x,y) | S(f(y,z),x) | ~Q(z,f(y,z))
S(x,y) | S(f(y,z),x) | Q(f(y,z),w)
S(x,y) | S(f(y,z),x) | ~S(z,z)

S(x,y) | ~Q(z,z) | ~Q(w,f(y,w))
S(x,y) | ~Q(z,z) | Q(f(y,w),z)
S(x,y) | ~Q(z,z) | ~S(w,w)

;; end-of-input
```

FARS output:

```
Initiating the input :
<ID> <predCount> <actualClause>
(1 3 (P(f(x,y),f(x,y)) ~S(x,f(x,y)) ~Q(y,f(x,y))))
(2 3 (P(f(x,y),f(x,y)) ~S(x,f(x,y)) Q(f(x,y),z)))
(3 3 (P(f(x,y),f(x,y)) ~S(x,f(x,y)) ~S(y,y)))
(4 3 (P(f(x,y),f(x,y)) S(f(x,y),z) ~Q(y,f(x,y))))
(5 3 (P(f(x,y),f(x,y)) S(f(x,y),z) Q(f(x,y),w)))
(6 3 (P(f(x,y),f(x,y)) S(f(x,y),z) ~S(y,y)))
(7 3 (P(f(x,y),f(x,y)) ~Q(z,z) ~Q(y,f(x,y))))
(8 3 (P(f(x,y),f(x,y)) ~Q(z,z) Q(f(x,y),z)))
(9 3 (P(f(x,y),f(x,y)) ~Q(z,z) ~S(y,y)))
(10 3 (~P(x,x) ~S(x,f(x,y)) ~Q(y,f(x,y))))
(11 3 (~P(x,x) ~S(x,f(x,y)) Q(f(x,y),z)))
(12 3 (~P(x,x) ~S(x,f(x,y)) ~S(y,y)))
(13 3 (~P(x,x) S(f(x,y),z) ~Q(y,f(x,y))))
(14 3 (~P(x,x) S(f(x,y),z) Q(f(x,y),w)))
(15 3 (~P(x,x) S(f(x,y),z) ~S(y,y)))
(16 3 (~P(x,x) ~Q(y,y) ~Q(z,f(x,z))))
(17 3 (~P(x,x) ~Q(y,y) Q(f(x,z),y)))
(18 3 (~P(x,x) ~Q(y,y) ~S(z,z)))
(19 3 (S(x,y) ~S(y,f(y,z)) ~Q(z,f(y,z))))
(20 3 (S(x,y) ~S(y,f(y,z)) Q(f(y,z),w)))
(21 3 (S(x,y) ~S(y,f(y,z)) ~S(z,z)))
(22 3 (S(x,y) S(f(y,z),x) ~Q(z,f(y,z))))
(23 3 (S(x,y) S(f(y,z),x) Q(f(y,z),w)))
(24 3 (S(x,y) S(f(y,z),x) ~S(z,z)))
(25 3 (S(x,y) ~Q(z,z) ~Q(w,f(y,w))))
(26 3 (S(x,y) ~Q(z,z) Q(f(y,w),z)))
(27 3 (S(x,y) ~Q(z,z) ~S(w,w)))


Initial predicate order:
(S Q P)


Trying order : (Q P S)

Stopping for max-step

Failed.. Memories are reset.. Going on...

Trying order : (P S Q)


proof-found! F is derived.

From 18 and 5 ---> F
'QED
```

4.3 LINEAR

Again, using the prove-file-preds to try all the predicate orders.

```
; LINEAR.THM      See TGTP   Section 5.2.5

A(a) | B(a) | C(a) | D(a)
A(a) | B(a) | C(a) | ~D(a)
A(a) | B(a) | ~C(a)
A(a) | ~B(a) | C(a)
A(a) | ~B(a) | ~C(a)
~A(a) | B(a)
~A(a) | ~B(a) | C(a)

negated_conclusion
~A(a) | ~B(a) | ~C(a)
```

FARS output:

```
Initiating the input :
<ID> <predCount> <actualClause>
(1 4 (A(a) B(a) C(a) D(a)))
(2 4 (A(a) B(a) C(a) ~D(a)))
(3 3 (A(a) B(a) ~C(a)))
(4 3 (A(a) ~B(a) C(a)))
(5 3 (A(a) ~B(a) ~C(a)))
(6 2 (~A(a) B(a)))
(7 3 (~A(a) ~B(a) C(a)))
(8 3 (~A(a) ~B(a) ~C(a)))


Initial predicate order:
(A B C D)


Trying order : (B D C A)

Stopping : No useful clause in nucleus

Failed.. Memories are reset.. Going on...

Trying order : (D C A B)

Stopping for max-step

Failed.. Memories are reset.. Going on...

Trying order : (D A B C)

proof-found! F is derived.

From 8 and 1, generate -> 11 - D(a)
From 2 and 11, generate -> 12 - A(a) | B(a) | C(a)
From 8 and 12 ---> F
'QED
```

4.4 The Lion and the Unicorn Puzzle

This example basically shows that FARS is not only a theorem prover, but an automated reasoning system, which can be used to solve mathematical puzzles, stated as axioms and conclusion. This puzzle is from Alice in Wonderland as follows:

> When Alice entered the forest of forgetfulness, she did not forget everything, only certain things. She often forgot her name, and the most likely thing for her to forget was the day of the week. Now, the lion and the unicorn were frequent visitors to this forest. These two are strange creatures. The lion lies on Mondays, Tuesdays, and Wednesdays, and tell the truth on the other days of the week. The unicorn, on the other hand, lies on Thursdays, Fridays, and Saturdays, but tells the truth on the other days of the week.
>
> One day Alice met the lion and the unicorn resting under a tree. They made the following statements:
>
> Lion: Yesterday was one of my lying days.
>
> Unicorn: Yesterday was one of my lying days.
>
> From these statements, Alice, who was a bright girl, was able to deduce the day of the week. What was it?

The proof that today is actually Thursday, is as follows (again the verbosity is disabled due to space concerns):

```
; LION.THM

;The Lion and the Unicorn found from JAR , pp 327-332, 1985
; Authors: H. J. Ohlbach and M. Schmidt-Schauss

; a day is either monday or tuesday or ... or sunday

MO(x)|TU(x)|WE(x)|TH(x)|FR(x)|SA(x)|SU(x)

; the days are all distinct, ie, if x is monday, then x is not tuesday

~MO(x)|~TU(x)
~MO(x)|~WE(x)
~MO(x)|~TH(x)
~MO(x)|~FR(x)
~MO(x)|~SA(x)
~MO(x)|~SU(x)
~TU(x)|~WE(x)
~TU(x)|~TH(x)
```

```
~TU(x)|~FR(x)
~TU(x)|~SA(x)
~TU(x)|~SU(x)
~WE(x)|~TH(x)
~WE(x)|~FR(x)
~WE(x)|~SA(x)
~WE(x)|~SU(x)
~TH(x)|~FR(x)
~TH(x)|~SA(x)
~TH(x)|~SU(x)
~FR(x)|~SA(x)
~FR(x)|~SU(x)
~SA(x)|~SU(x)

; relating yesterday and today

~MO(ystday(x)) | TU(x)
~TU(x) | MO(ystday(x))
~TU(ystday(x)) | WE(x)
~WE(x) | TU(ystday(x))
~WE(ystday(x)) | TH(x)
~TH(x) | WE(ystday(x))
~TH(ystday(x)) | FR(x)
~FR(x) | TH(ystday(x))
~FR(ystday(x)) | SA(x)
~SA(x) | FR(ystday(x))
~SA(ystday(x)) | SU(x)
~SU(x) | SA(ystday(x))
~SU(ystday(x)) | MO(x)
~MO(x) | SU(ystday(x))

; if x is one of the days the lion lies, then x is monday, tuesday, or wednesday

~Mem(x,lydays(lion())) | MO(x) | TU(x) | WE(x)
~Mem(x,lydays(unicorn())) | TH(x) | FR(x) | SA(x)

; if x is monday, then x is a lying day for the lion

~MO(x) | Mem(x,lydays(lion()))
~TU(x) | Mem(x,lydays(lion()))
~WE(x) | Mem(x,lydays(lion()))
~TH(x) | Mem(x,lydays(unicorn()))
~FR(x) | Mem(x,lydays(unicorn()))
~SA(x) | Mem(x,lydays(unicorn()))

; if x is not a member of the lying days of z  and  z says on day x that he lies on day y,
;   then y is a member of the lying days of z.

Mem(x,lydays(z)) | ~LA(z,x,y) | Mem(y,lydays(z))
Mem(x,lydays(z)) | LA(z,x,y) | ~Mem(y,lydays(z))
~Mem(x,lydays(z)) | ~LA(z,x,y) | ~Mem(y,lydays(z))
~Mem(x,lydays(z)) | LA(z,x,y) | Mem(y,lydays(z))

; the lion says today that he lied yesterday

LA(lion(),today(),ystday(today()))
```

```
LA(unicorn(),today(),ystday(today()))

negated_conclusion

; today is not thursday

~TH(today())
```

FARS output:

```
Initiating the input :
<ID> <predCount> <actualClause>
(1 7 (MO(x) TU(x) WE(x) TH(x) FR(x) SA(x) SU(x)))
(2 2 (~MO(x) ~TU(x)))
(3 2 (~MO(x) ~WE(x)))
(4 2 (~MO(x) ~TH(x)))
(5 2 (~MO(x) ~FR(x)))
(6 2 (~MO(x) ~SA(x)))
(7 2 (~MO(x) ~SU(x)))
(8 2 (~TU(x) ~WE(x)))
(9 2 (~TU(x) ~TH(x)))
(10 2 (~TU(x) ~FR(x)))
(11 2 (~TU(x) ~SA(x)))
(12 2 (~TU(x) ~SU(x)))
(13 2 (~WE(x) ~TH(x)))
(14 2 (~WE(x) ~FR(x)))
(15 2 (~WE(x) ~SA(x)))
(16 2 (~WE(x) ~SU(x)))
(17 2 (~TH(x) ~FR(x)))
(18 2 (~TH(x) ~SA(x)))
(19 2 (~TH(x) ~SU(x)))
(20 2 (~FR(x) ~SA(x)))
(21 2 (~FR(x) ~SU(x)))
(22 2 (~SA(x) ~SU(x)))
(23 2 (~MO(ystday(x)) TU(x)))
(24 2 (~TU(x) MO(ystday(x))))
(25 2 (~TU(ystday(x)) WE(x)))
(26 2 (~WE(x) TU(ystday(x))))
(27 2 (~WE(ystday(x)) TH(x)))
(28 2 (~TH(x) WE(ystday(x))))
(29 2 (~TH(ystday(x)) FR(x)))
(30 2 (~FR(x) TH(ystday(x))))
(31 2 (~FR(ystday(x)) SA(x)))
(32 2 (~SA(x) FR(ystday(x))))
(33 2 (~SA(ystday(x)) SU(x)))
(34 2 (~SU(x) SA(ystday(x))))
(35 2 (~SU(ystday(x)) MO(x)))
(36 2 (~MO(x) SU(ystday(x))))
(37 4 (~Mem(x,lydays(lion)) MO(x) TU(x) WE(x)))
(38 4 (~Mem(x,lydays(unicorn)) TH(x) FR(x) SA(x)))
(39 2 (~MO(x) Mem(x,lydays(lion))))
(40 2 (~TU(x) Mem(x,lydays(lion))))
(41 2 (~WE(x) Mem(x,lydays(lion))))
(42 2 (~TH(x) Mem(x,lydays(unicorn))))
(43 2 (~FR(x) Mem(x,lydays(unicorn))))
```

```
(44 2 (~SA(x) Mem(x,lydays(unicorn))))
(45 3 (Mem(x,lydays(z)) ~LA(z,x,y) Mem(y,lydays(z))))
(46 3 (Mem(x,lydays(z)) LA(z,x,y) ~Mem(y,lydays(z))))
(47 3 (~Mem(x,lydays(z)) ~LA(z,x,y) ~Mem(y,lydays(z))))
(48 3 (~Mem(x,lydays(z)) LA(z,x,y) Mem(y,lydays(z))))
(49 1 (LA(lion,today,ystday(today))))
(50 1 (LA(unicorn,today,ystday(today))))
(51 1 (~TH(today)))


Initial predicate order:
(Mem TH MO TU WE FR SA SU LA)



proof-found! F is derived.

From 51 and 1, generate -> 52 - MO(today) | TU(today) | WE(today) | FR(today) | SA(today) | SU(today)

From 12 and 52, generate -> 200 - MO(today) | WE(today) | FR(today) | SA(today)

From 6 and 200, generate -> 204 - WE(today) | FR(today)

From 14 and 204 ---> F
'proof-found!
```

Chapter 5

Discussion and Future Work

There are currently lots of deficiencies and inefficiencies in FARS due to time concerns. These are planned to be improved or corrected in the future versions. Below are explanations for each planned future work:

- The known full completeness issue due to the factorization is to be corrected.
- The efficiency of the whole system can be drastically improved by incorporating FARS to a platform with normal order evaluation strategy (such as lazy racket).
- Unit testing as well as an overall performance testing is required. Possibly a helper module implementing the necessary functionality for test functions to be generated and applied for other modules to be able to test themselves easily. This way, comprehensive test suite can be automatically generated, which is a huge step in validating the soundness and correctness of the whole system.
- FARS is currently working in a non-interactive mode. Proof searching should be interruptable, as well as continuable after certain parameters are modified in between. This functionality obviously require FARS to be able to run in interactive mode.
- The restrictions in the format of the input files should be released.
- FARS is highly generalized, as all the heuristics and even some parts of its inference strategy can be provided as a parameter. This can be further improved by supporting these parametizations at the input file

level, where for example heuristic functions to be used are directly fed from the input file.

- An advanced weighting system and also the "hints strategy" can be adopted from OTTER, which are very useful for the core prover to navigate in the search space very efficiently.

- A better improvement for heuristic usage is to implement the necessary framework for supporting the learning heuristics, which employ machine learning techniques to learn and improve themselves for the similar proof tasks.

At the current stage, FARS can not perform well on a very broad area of theorems, however, its efficiency in proving is clearly observable despite the inefficient parts of its implementation. It's functional and generalized characteristics on the other hand, provides that FARS can be made to be very capable prover with a very little effort.

Appendices

Appendix A

Complete Reference

Below are the sections involving explanations for each and every part of the implementation. This reference is designed to be a direct informal reflection of the implementation. Thus the reference is **not** alphabetical, but spatial with respect to the implementation. It can be referred by using the context of the item or the part in consideration. For example, section names are actually source file names, and each section contains all global parameters as well as functions in the order that they are appearing in the actual source.

Functions include contracts beside their names. For example

- **factor** : clause -> clause

means that the function named *factor* consumes a single clause and produces a single clause. Higher arities are indicated by a single space break between input parameters in the contract.

- **resolve** : clause clause (listof symbol?) -> clause

meaning that the function named *resolve* consumes two clauses and a list of symbol (indicating the predicate order) and produce a single clause (resolvent).

Function contracts are not an explanation of what a function does, but only an indicator of input-output relationships in order to ease the functional development process.

A.1 Main

A.1.1 Global Parameters

- **max-step**

  A run-time constraint parameter. Limits the maximum steps taken by the prover. Default to false.

- **max-literal**

  A simple heuristic for resolution. Any inferred clause that is bigger than the *max-literal* parameter is discarded.

- **verbose**

  Verbosity flag. Can either be true or false.

- **starter-verbose**

  Sometimes the prover can set its verbosity level itself, *starter-verbose* is used to reset the *verbose* flag. Mostly used in subsequent automatic runs of the FARS.

- **init**

  An internal flag controlling the repeated application of *binary-resolution*. When *binary-resolution* returns false, *resolve* understands from *init* whether the given clauses are originally not resolvable or not further resolvable. Defaults to true.

- **last-input-index**

  A mutable parameter that is set by the *initiate* function, indicating the ID of the last clause given in input. Used by *print-proof*.

- **main-memory**

  A list containing pairs of clause ID's (e.g. (4 12)) that are used by the resolution. Each time a new clause is inferred, its parents are recorded to *main-memory* in order to avoid duplicate resolutions.

- **proof-nucleus-memory**

  A list containing the ID's of the clauses from nucleus that is failed to be of any use with that particular stage of the prover. Each time a new clause inferred, *proof-nucleus-memory* is reset to be empty, so that previous nucleus clauses can be tried with new clauses.

- **proof-memory**

  A list containing a lot of information about each semantic clash, helping the proofs to be printed easily. Each time a proof is found, *proof-memory* is scanned backwards from the top of the *main-memory* to construct a step-by-step human readable proof.

A.1.2 Functions

- **factor** : clause -> clause

  Implements full factorization by repeatedly applying itself, corresponding to a repeated application of binary factorization.

- **resolve** : clause clause (listof symbol?) -> clause

  Just a wrapper function to initiate correctly the resolution parameters before the resolution begins.

- **resolve-inner** : clause clause (listof symbol?) -> clause

  Like *factor*, it implements full resolution by repeatedly calls itself, which is actually performing binary resolution. Note that, *resolve-inner* tries to resolve the factorizations of the given clauses as well as the clauses themselves. Whenever it finds a single resolvent, it stops to try others and goes on with that resolvent.

- **binary-resolve** : clause clause (listof symbol?) -> (listof clause clause)

  Implements binary resolution on given two clauses, considering the given predicate ordering. It does not return a single resolvent clause, but returns two clauses that are ready to form the resolvent by simply appending together. The reason is to easily repeat the binary-resolution on them to perform full resolution.

- **initiate** : (listof clause) pred-order-constructor clause-differentiator clause-picker -> prover-input-object

  Constructs a *pi* object to be consumed by the prover. Indexes the given clauses, construct the predicate order by the given pred-order-constructor, differentiate nucleus and satellite sets by the given clause-differentiator and puts all of the initial information that is needed by the prover in the *pi* object.

- **struct:pi**

  PI (prover-input) is a structure containing all the necessary parameters to initiate the prover, such as nucleus and satellite sets, unit clauses, clause picker function and predicate order.

- **start** pi-object -> proof

  Just a wrapper function to set the global parameters and run the prover with the given pi-object.

- **easy-init** : (listof clauses) -> proof

  We have built-in predicate ordering, nucleus/satellite differentiator (interpretation creator) and clause picker functions. *Easy-init* is a wrapper for these built-in functions to call *initiate* easily with them.

- **prove-file** : string? -> proof

  Just a wrapper function to parse the given file-name, run *easy-init* and then *start*.

- **prove** : nucleus satellite nuc-units sat-units clause-picker pred-order next-index step -> proof

  Implements the main prover loop. *next-index* is used to index the newly inferred clause. *step* counts the number of steps the prover takes, mainly for using *max-step* control. Detailed discussion about the implemented algorithm can be found in Section 3.3.

- **sieve-clause-set** : (listof siever-functions) (listof clause) chosen-nuc-clause-ID -> (listof clause)

  Mainly used in satellite clause selection. *sieve-clause-set* eliminates the irrelevant clauses in satellite set according to the given list of eliminator functions constructed using the chosen nucleus predicates. These functions together decide whether a clause in satellite is suitable or not for the chosen clause in nucleus, by looking at its literals.

- **unit-conflict-check** : clause (listof clause) (listof symbol?) -> proof

  Performs the widely-known unit conflict check. Each time a unit clause is inferred in *prove*, *unit-conflict-check* is run to find an easy proof quickly.

- **start-with-order** : (listof symbol?) pi-object -> proof

Just another helper function for *prove*. Essential purpose is to make possible to override the predicate order constructor function and provide a particular order manually.

- **start-for-all-preds** : pi-object -> proof

  Like *start* for *prove*, *start-for-all-preds* is just a wrapper function to set the global parameters and run the *prove-for-all-pred-orders* with the given pi-object.

- **prove-for-all-pred-orders** : pi-object (listof (listof symbol?)) -> proof

  When a suitable predicate order constructor function can not be found, *prove-for-all-pred-orders* generates all possible permutations of the predicates and tries to prove them one by one normally.

- **prove-file-preds** : string? -> proof

  Like prove-file, a wrapper function to parse the given file-name, run *easy-init* and then *start-for-all-preds*.

- **print-proof** : (listof clause?) proof-memory -> (void)

  Just prints the proof sequence constructed by the construct-proof-sequence using proof-memory. Returns nothing.

- **print-real-proof** : proof-sequence -> (void)

  This consumes the proof-sequence generated by the construct-proof-sequence and displays them in a human readable format one by one. Returns nothing.

A.2 Util

A.2.1 Functions

- **extract-args** : string? -> (listof string?)

  A wrapper function for *extract-arguments*.

- **comma-after-paren-count0** : (listof char?) number number -> number

  A helper function for extract arguments, to handle situtations like f(x(),g(h(),p())). Returns the comma index after the balanced paranthesis.

- **extract-arguments** : string? output-list -> (listof string?)

  Extracts the comma seperated strings as a list from the given string. output-list is just for implementing tail-recursion.

- **remove-identicals** : clause -> clause

  Removes all the identical literals in the given clause.

- **disagreement-dive** : clause -> clause

  A helper function for *disagreement-set*. Essentially a hack to handle the cases such as f(g(k)) f(u) –> {g(k),u}.

- **disagreement-set** : (listof (listof FOP?)) -> (listof FOP?)

  Extracts the disagreement set from the given argument set of literals

- **preliminary-check** : (listof clause?) -> boolean

  A helper for *unify* to check if the given list of clauses all contain the same predicate names and parameter numbers. If not, unification can stop immediately without computing anything.

- **find-term** : var? (listof FOP?) -> FOP?

  Finds the first term in the disagreement set that not includes the given variable. Essentially a helper for *find-var-term* to find a term.

- **find-var-term** : (listof FOP?) -> (listof var? FOP?)

  Extracts a {v/t} pair from the given disagreement set. Essentially a helper for *unify*.

- **apply-subst** : (listof var? FOP?) clause -> clause

  A helper function for *apply-mgu*. Applies a single substitution to the given clause.

- **apply-mgu** : (listof (listof var? FOP?)) clause -> clause

  Applies the given set of substitutions to the given clause, using *apply-subst* repeatedly.

- **unify** : (listof clause?) (listof (listof var? FOP?)) -> (listof (listof var? FOP?))

  Implements the unification algorithm on the given set of clauses to produce a most general unifier (set of substitutions). The second parameter is merely to implement tail-recursion.

- **find-unifiable-pair-mgu** : clause clause -> (listof mgu clause clause)

  Finds a unifiable pair of terms from the given clauses and produce their mgu's along with the chosen clauses.

- **fup-mgu/factor** : clause clause -> (listof mgu clause clause)

  Like find-unifiable-pair-mgu, but ignores that the choosen terms are exactly the same. Mostly used in factorization for finding a unifiable pair of terms in the same clause.

- **detach-literal** : FOP? clause -> clause

  Detaches the given literal from the given clause. Mostly used in resolution, to detach the resolved literals.

- **pred-or-neg-with-name** : symbol? -> (FOP? -> boolean?)

  Constructs a function which consumes a literal (pred or negated pred) and checks if it has the name which is originally a parameter to *pred-or-neg-with-name*.

- **pred-with-name** : symbol? -> (FOP? -> boolean?)

  Like *pred-or-neg-with-name*, but only checks for non-negated predicates.

- **neg-with-name** : symbol? -> (FOP? -> boolean?)

  Converse of *pred-with-name*, only checks for the negated predicates.

- **contains-pred-or-neg** : clause symbol? -> boolean?

  Checks if the given clause contains the predicate or its negation with the given name.

- **highest-pred-in** : clause (listof symbol?) -> symbol?

  Produces the name of the predicate that has the highest order in the given clause.

- **rename-vars** : clause clause -> (listof clause clause)

  If the given clauses have variables with the same names, it systematically renames the shared variables in one of the clauses with automatically generated unique variable names.

- **process-same-lits** : clause clause -> (listof clause clause)

  This is essentially a hack. It constructs an artificial mgu and applies it to one of the clauses that will make possible EXACTLY SAME (negated) literals to be resolved by the normal resolution procedure. Consider c1:P(x)VQ(x) and c2:$\tilde{P}$(x)V$\tilde{Q}$(x). Normally c1 will be renamed like P(gsymb1)VQ(gsymb1), thus the resolution does not resolve these literals with literals in c2 (since P(x) and $\tilde{P}$(y) are not unifiable). However, *process-same-lits* turn c1 into P(a)VQ(a) and do not touch c2. Therefore, resolution will be forced to resolve these.

- **construct-inner-mgu** : clause clause (listof (listof var? FOP?)) -> (listof (listof var? FOP?))

  Actual worker function in *process-same-lits*, constructing the necessary mgu.

- **choose-unifiable-literals** : clause clause (listof symbol?) boolean -> (list mgu FOP? FOP?)

  Chooses two literals L1 and L2 from C1 and C2 respectively, according to the given predicate order, such that L1 and $\tilde{L}$2 are unifiable. Note that hack-flag (the last parameter) is used for ignoring the predicate order, generalizing the function further.

- **count-preds** : FOP? number -> number

  Counts the number of predicates in the given literal. Second parameter is only for tail-recursion.

- **count-preds-clause** : clause -> number

  Counts the number of predicates in the given clause, using *count-preds*.

- **construct-pred-order** : (listof clause) -> (listof symbol?)

  This is an example predicate order constructor function. It consumes a set of clauses and produces a predicate order such that the first predicate in the output has the highest number of occurance in the set.

  A lot of predicate order constructor functions can be created and provided to the prover as input. *construct-pred-order* is only an example.

- **picker-example** : (listof (listof number number clause)) -> (listof number number clause)

This is an example clause picker for tasks like resolution where a clause need to be picked from the set of clauses. *picker-example* chooses the clause with the least predicate count.

- **differentiator-example** : (listof (listof clauses)) -> (listof (listof clauses) (listof clauses))

  This is an example differentiator function, extracts the nucleus and satellite sets of clauses by putting all positive clauses to nucleus, and mixed ones into the satellite.

- **insert-everywhere** : any (listof any) (listof any) (listof any) -> (listof (listof any))

  A helper function for *generate-all-permutations-of*. Inserts the given element into every possible places among the elements in the third argument. The second and fourth arguments are for tail-recursion. Example usage: (insert-everywhere 2 '() '(1 3) '()) will produce '((2 1 3) (1 2 3) (1 3 2))

- **generate-all-permutations-of** : (listof any) -> (listof (listof any))

  Produce all possible permutations of the given list of elements. Mostly used by *prove-for-all-pred-orders*.

- **construct-proof-sequence** : proof-memory (list id id) id -> (listof (listof id id (id clause)))

  A helper for *print-proof*. Constructs the proof-sequence of the found proof by backtracking in proof-memory, starting from the given end-point (second parameter).

A.3 Language

A.3.1 Global Parameters

- **constant-symbols**

  This list constitute the symbols mostly used for constants in input theorems. By default, it contains {a,b,c,d,e}.

- **var-symbols**

  Like constant-symbols, but for variable symbols. By default, it contains {x,y,z,g,t,u,v}.

A.3.2 Functions

- **constant?** : symbol? -> boolean?

  Checks if the given symbol is among constant symbols or not in variable symbols.

- **variable?** : symbol? -> boolean?

  Checks if the given symbol is among variable symbols or not in constant symbols.

- **listof?** : (any -> boolean?) -> ((listof any) -> boolean?)

  Constructs a function which will consume a list of elements and apply the given predicate function to every element of the list, checking every element satisfies the predicate.

- **FOP**

  This is not a function, but a data type, representing a single term.

- **FOP=?** : FOP? FOP? -> boolean?

  Checks if the given two terms are exactly the same.

- **var-contain?** : var? FOP? -> boolean?

  Checks if the given term contains the given variable.

- **rewrite-term** : var? FOP? FOP? -> FOP?

  Substitutes the given variable with the given term(second parameter) in the third parameter.

- **var-names** : clause (listof symbol?) -> (listof symbol?)

  Extracts the variable names from the given clause. Second parameter is for tail-recursion.

- **pred-names** : clause (listof symbol?) -> (listof symbol?)

  Like *var-names*, but for predicate names.

- **neg-pred-names** : clause (listof symbol?) -> (listof symbol?)

  Like *var-names*, but for negated predicate names.

- **extract-names** : (clause->(listof symbol?)) clause -> (listof symbol?)

  Extracts the names using the given function from the given clause.

- **extract-names-from-S** : (clause->(listof symbol?)) (listof clause) -> (listof symbol?)

  Extracts the names using the given function and *extract-names* from the given set of clauses.

- **process-args** : (listof FOP?) string? string -> string

  Helper function for *unparse* to place seperators between the unparsed arguments.

- **unparse** : FOP? -> string?

  Unparses the given term.

- **unparse-clause** : clause -> string?

  Unparses the given clause using *unparse*.

- **unparse-S** : (listof clause?) -> string?

  Unparses the given set of clauses using *unparse-clause*.

A.4 Parser

A.4.1 Global Parameters

- **comment**

  Comment symbol used to indicate a comment line in the given input. Defaults to ";".

- **negline**

  String used to indicate the negated conclusion in the input. Defaults to "negated_conclusion".

- **or-sym**

  String used to indicate the disjunction in the given input. Defaults to "|".

- **or-char**

  Like or-sym, but in type char. Defaults to "#\|".

A.4.2 Functions

- **pre-process** : (listof (listof string?)) -> (listof (listof string?))

  Performs the pre-processing tasks like empty line, comment line and "negated_conclusion" deletion before parsing begins.

- **remove-or-symbols** : (listof string?) -> (listof string?)

  Removes the *or-sym*'s from the unparsed lines.

- **parse-term** : string? -> FOP?

  Parses a single unit of the given input theorem. Assumptions about the input file include (-) function-like symbols with no arguments are actually constants (like a()), and (-) there are no nested predicates.

- **parse-term-inner** : boolean? string -> FOP?

  Actual working function withing *parse-term*. The first argument indicates whether we dove into the parameters of a predicate, or we are at the top level (false means we are looking for a predicate, true means a function).

- **parse-clause** : (listof string?) -> clause

  Parses the given clause line using *parse-term*.

- **parse-file** : symbol? -> (listof clauses)

  Parses the given file using *read-words/line*(from Racket library), *pre-process* and *parse-clause*, producing the raw set of clauses to be initialized and proved.

Appendix B

Racket as a System

As briefly explained in Section 3.1, Racket is a rather big system containing a lot of programming languages, an advanced programming environment, as well as lots of different kinds of tools in it. In this chapter, we try to provide a general idea about the syntax and semantics of the Racket language and DrRacket environment, which will hopefully be helpful in understanding the implementation of FARS.

B.1 Overview of Syntax and Semantics

Almost all of the dialects of LISP including Racket uses s-expressions as a concrete syntax. The self referential definition of s-expressions is as follows:

An s-expression is:

- an atom (mostly indicated by a symbol), or
- a list of of s-expressions

For instance, a, (), (a), ((a)) or (a (b (c) (d)) (((e)))), are all s-expressions. Note that however, all Racket expressions are s-expressions, but not all s-expressions are Racket expressions.

In particular, Racket uses s-expressions in a prefix form, in which the functions and operators are written *before* arguments, such as

(+ 1 2)

or

(prove-file "LION.THM")

Every definition in Racket uses (define ...) structure. Because the functions are treated as first-class values (like numbers), they are also defined in this way.

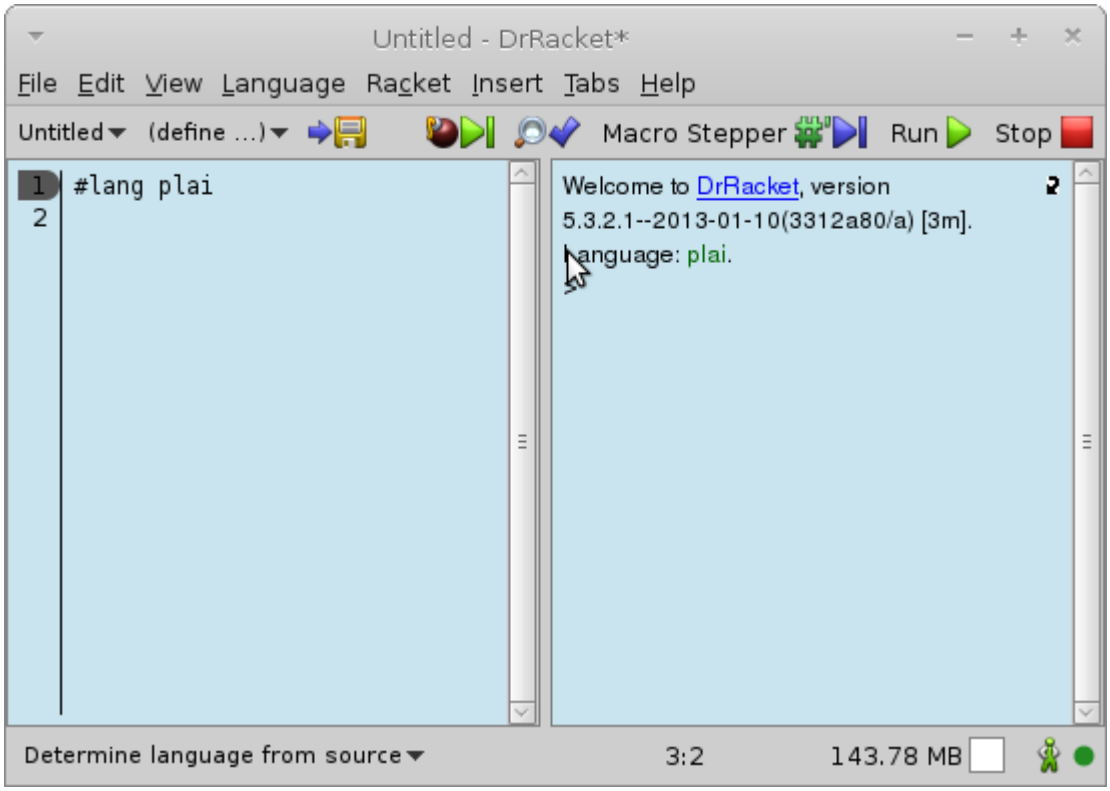
```
(define a 5)
(define (sqr x) (* x x))
```

As explained in the previous paragraphs, a defined function is applied in again the prefix way, such as

```
(sqr 5)
```

B.2 DrRacket

DrRacket is an interactive programming environment, which involves two main frames as can be seen in the following figure.



Two main frames are called the "definitions" (left in the figure) and "interactions" (right in the figure) windows. The program definitions are written in the definitions window and when the "Run" button is pressed, the definitions are parsed and compiled automatically, at which point the interactions window can be used to run the programs manually or automatically.

The interactions window implements what is called the REPL (Read-Eval-Print Loop), in which the programs are read, evaulated, the corresponding outputs are written and then the reading starts again and so on. It can be considered as a very complicated calculator, on which we can run arithmetic expressions as well as prove theorems, given that the necessary definitions are provided. At default settings, the interaction window also serves as the standard output which is exactly the same thing that we know from various other contexts such as Linux console. In fact, the programs written in DrRacket can also be computed through the console program "racket", passing the outputs to the real standard output.

DrRacket can be run on all mainstream platforms such as Linux, Mac OS and Windows. It can be downloaded from:

http://download.racket-lang.org/

The complete documentation of the Racket system can be found at:

http://docs.racket-lang.org/

B.3 Some Built-In Racket Functions

In this section, we try to explain several built-in functions of Racket that is used to implement FARS, along with examples where necessary. Most of the functions described below are list processing functions, because Racket is a not so long cousin of LISP (list-processing) language. Thus, it would be appropriate to give a formal definition of a list of elements.

A listof elements is either:

- empty, or
- (cons e ls), where e is an element and ls is a listof elements.

Here are the built-in Racket functions which are most frequently used in the implementation of FARS.

- **cons** : constructs a list from the given two arguments.

  (cons 4 empty) -> (4)

  (cons 4 (cons 6 empty)) -> (4 6)

- **car** : extracts the first element of the given list.

  (car (cons 3 empty)) -> 3

- **cdr** : extracts the rest of the given list.

  (cdr (cons 2 (cons 3 empty))) -> (3)

- **null?** : checks if the given list is empty.

- **map** : applies the given function to every element of the given list

- **filter** : filters the given list by testing each element with the given predicate function

- **begin** : sequentially evaluates its sub-expressions one-by-one, its return value is the return value of the last sub-expression.

- **set!** : mutation operator. Mutates the value of the variable identified by the given variable name.

  (define a 5) (begin (set! a 6) a) -> 6

- **require** : includes the module given as either a filename (in the same directory) or a built-in library name.

- **lambda** : an anonymous function creator.

  ((lambda (x) (+ x x)) 4) -> 8