

¹ Operational semantics for Prolog with Cut in Rocq and its application to determinacy analysis

³ Jane Open Access  

⁴ Dummy University Computing Laboratory, [optional: Address], Country

⁵ My second affiliation, Country

⁶ Joan R. Public¹  

⁷ Department of Informatics, Dummy College, [optional: Address], Country

⁸ Abstract

⁹ Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis orci arcu, eu mollis dolor.
¹⁰ Aliquam eleifend suscipit lacinia. Maecenas quam mi, porta ut lacinia sed, convallis ac dui. Lorem
¹¹ ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse potenti.

¹² 2012 ACM Subject Classification Replace ccsdesc macro with valid one

¹³ Keywords and phrases Dummy keyword

¹⁴ Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

¹⁵ Funding Jane Open Access: (Optional) author-specific funding acknowledgements

¹⁶ Joan R. Public: [funding]

¹⁷ Acknowledgements I want to thank ...

¹⁸ 1 Introduction

¹⁹ ELPI is a dialect of λPROLOG (see [14, 15, 7, 12]) used as an extension language for the ROCQ
²⁰ prover (formerly the Coq proof assistant). ELPI has become an important infrastructure
²¹ component: several projects and libraries depend on it [13, 3, 4, 19, 8, 9]. Examples include
²² the Hierarchy-Builder library-structuring tool [5] and Derive [17, 18, 11], a program-and-proof
²³ synthesis framework with industrial applications at SkyLabs AI.

²⁴ Starting with version 3, ELPI gained a static analysis for determinacy [10] to help users
²⁵ tame backtracking. ROCQ users are familiar with functional programming but not necessarily
²⁶ with logic programming and uncontrolled backtracking is a common source of inefficiency
²⁷ and makes debugging harder. The determinacy checkers identifies predicates that behave
²⁸ like functions, i.e., predicates that commit to their first solution and leave no *choice points*
²⁹ (places where backtracking could resume).

³⁰ This paper reports our first steps towards a mechanization, in the ROCQ prover, of the
³¹ determinacy analysis from [10]. We focus on the control operator *cut*, which is useful to
³² restrict backtracking but makes the semantic depart from a pure logical reading.

³³ We formalize two operational semantics for PROLOG with cut. The first is a stack-
³⁴ based semantics that closely models ELPI’s implementation and is similar to the semantics
³⁵ mechanized by Pusch in ISABELLE/HOL [16] and to the model of Debray and Mishra [6,
³⁶ Sec. 4.3]. This stack-based semantics is a good starting point to study further optimizations
³⁷ used by standard PROLOG abstract machines [20, 1], but it makes reasoning about the scope
³⁸ of *cut* difficult. To address that limitation we introduce a tree-based semantics in which the
³⁹ branches pruned by *cut* are explicit and we prove the two semantics equivalent. Using the

¹ Optional footnote, e.g. to mark corresponding author



© Jane Open Access and Joan R. Public;
licensed under Creative Commons License CC-BY 4.0

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:12



Leibniz International Proceedings in Informatics

LIPIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

23:2 Operational semantics for Prolog with Cut in Rocq and its application to determinacy analysis

```

Inductive P := IP of nat. Inductive D := ID of nat. Inductive V := IV of nat.

Inductive Tm :=
| Tm_P of P      | Tm_D   of D      | Callable_P of P
| Tm_V of V      | Tm_App of Tm & Tm. | Callable_App of Callable & Tm.

```

Figure 1 Tm and Callable types

40 tree-based semantics we then show that if every rule of a predicate passes the determinacy
41 analysis, the call to a deterministic predicate does not leave any choice points.

2 Common code: the language

put unif and pros
gram in variable@
hides from types
46 the them. The smallest unit of code that we can use in the langauge is an atom. The atom
inductive (see Type 1) is either a cut or a call. A call carries a callable term (see Figure 1).
A term (Tm) is either a predicate, a datum, a variable or the binary application of a term to
another. A Callable is a term accepting predicates only predicates as functors.

```

48 Inductive A := cut | call : Callable → A.                               (1)
49 Record R := mkR { head : Callable; premises : list A }.             (2)
50 Record P := { rules : seq R; sig : sigT }.                           (3)
51 Definition Σ := { fmap V → Tm}.                                       (4)
52 Definition bc : Unif → P → FU → Callable →
Σ → FU * seq (Σ * seq A) :=                                         (5)

```

!!!: controllare il
tipo di bc nel
testo
55 atoms. A program (see Type 3) is made by a list of rules and a mapping from predicates to
their signatures. The type sigT is the classic type from the simply typed lambda calculus, i.e.
56 it is either a base type or an arrow. We decorate arrows to know the mode of the lhs type.

57 A substitution (see Type 4) is a mapping from variables to terms. It is the output of a
58 successful query and is often called the output of a query.

```

Record Unif := {
  unify : Tm → Tm → Σ → option Σ;
  matching : Tm → Tm → Σ → option Σ;
}.

```

59 The backchain function (bc, see Type 5) filters the rules in the program that can be
60 used on a given query. It takes: a unificator U which explains how to unify terms up to
61 standard unification (for output terms) or matching (for input terms); a program P to explore
62 and filter; a set S of free variable (fvS) allowing to fresh the program P by renaming the
63 its variables; a query q ; and the substitution σ in which the query q lives. The result of a
64 backchain operation is couple made of an extension of S containing the new variables that
65 have been allocated during the unification phase and a list of filtered rules r accompagnate
66 by their a substition. This substitution is the result of the unification of q with the head of
67 each rule in r .

68 In Figure 2, we have an example of a simple ELPI program which will be used in the
69 following section of the paper as an example to show how backtracking and the cut operator
70 works in the semantcis we propose. The translation of these rules in the ROCQ representation
71 is straightforward.

```
f 1 2.    f 2 3.    r 2 4.    r 2 8.
g X X.          % r1
g X Z :- r X Z, !.   % r2
g X Z :- f X Y, f Y Z.   % r3
```

Figure 2 Small ELPI program example

2.1 The cut operator

The semantics of the cut operator adopted in the ELPI language corresponds to the *hard cut* operator of standard SWI-PROLOG. This operator has two primary purposes. First, it eliminates all alternatives that are created either simultaneously with, or after, the introduction of the cut into the execution state.

To illustrate this high-level description, consider the program shown in Figure 2 and the query $q = g 2 Z$. All three rules for g can be used on the query q . They are tried according to their order of appearance in the program: rule r_1 is tried first, followed by r_2 , and r_3 .

The first rule has no premises and immediately returns the assignment $Z = 2$. However, the computation does not terminate at this point, since two additional unexplored alternatives remain, corresponding to the premises of rules r_2 and r_3 .

The premises of rule r_2 are $r 2 Z, !$. At this stage, the role of the cut becomes apparent. If the premise $r 2 Z$ succeeds, the cut commits to this choice and removes the premises of rule r_3 from the alternative list, as they were generated at the same point as the cut. Moreover, if the call $r 2 Z$ itself produces multiple alternatives, only the first one is committed, while the remaining alternatives are discarded. This is because such alternatives have been created at a deeper depth in the search tree than the cut.

Concretely, the call $r 2 Z$ yields two solutions, assigning Z the values 4 and 8, respectively. The second solution is eliminated by the cut, and only the first assignment is preserved.

se metti $r1 = g A$
 $B :- f A B$. allora
 $g e f$ sono fun-
zioni, e puoi spie-
gare anche l'idea
del detcheck qui

3 Semantics intro

We propose two operational semantics for a logic program with cut. The two semantics are based on different syntaxes, the first syntax (called tree) exploits a tree-like structure and is ideal both to have a graphical view of its evolution while the state is being interpreted and to prove lemmas over it. The second syntax, called elpi, is the ELPI's syntax and has the advantage of reducing the computational cost of cutting and backtracking alternatives by using shared pointers. We aim to prove the equivalence of the two semantics together with some interesting lemmas of the cut behavior.

4 Tree semantics

```
Inductive tree :=
| KO | OK | TA of A
| Or  of option tree & Σ & tree
| And of tree & seq A & tree.
```

In the tree we distinguish 5 main cases: *KO*, *OK*, and are special meta-symbols representing, respectively, the failed and a successful terminal. These symbols are considered meta because they are internal intermediate symbols used to give structure to the tree.

The *TA* constructor (acronym for tree-atom) is the constructor of atoms in the tree. TA = Todo/
Goal?

```

Fixpoint get_end s A : Σ * tree :=
  match A with
  | TA _ | KO | OK ⇒ (s, A)
  | Or None s1 B ⇒ get_end s1 B
  | Or (Some A) _ _ ⇒ get_end s A
  | And A _ B ⇒
    let (s', pA) := get_end s A in
    if pA == OK then get_end s' B
    else (s', pA)
  end.

Definition get_subst s A := (get_end s A).1.

Definition path_end A := (get_end ε A).2. (* ~ε~ is the ~ε~ subst*)

Definition success A := path_end A == OK.

Definition failed A := path_end A == KO.

Definition path_atom A := if path_end A is TA _ then true else

```

Figure 3 Definition of *get_end*

104 The two recursive cases of a tree are the *Or* and *And* non-terminals. The *Or* non-terminal
 105 $A \vee B_\sigma$ denotes a disjunction between two trees A and B . The first branch is optional, if
 106 absent it represents a dead tree, i.e. a tree that has been entirely explored. The second
 107 branch is annotated with a suspended substitution σ so that, upon backtracking to B , σ is
 108 used as the initial substitution for the execution of B .

109 The *And* non-terminal $A \wedge_{B_0} B$ represents a conjunction of two trees A and B . We call B_0
 110 the reset point for B ; it is used to restore the state of B to its initial form if a backtracking
 111 operation occurs on A . Intuitively, let $t2l$ be the function flattening a tree in a list of sequents
 112 disjunction, in PROLOG-like syntax the tree $A \wedge_{B_0} B$ becomes $(A_1, t2l B); (A_2, B_0); \dots; (A_n, B_0)$
 t2l nope, mettiti
 un r3 = g
 X Z :- r .., nf
 .., !, e rifatti
 all'esempio della
 sezione prima
 (fai in modo che
 f funzioni solo
 con la seconda
 regola per r)
 associate to
 the...
 as much as
 needed, indee
 prolog programs
 do not necessar
 ily terminate
 122
 123
 124
 125
 126
 127
 128
 129
 130
 131
 132
 133
 134
 135
 136
 137
 138
 139
 140
 141
 142
 143
 144
 145
 146
 147
 148
 149
 150
 151
 152
 153
 154
 155
 156
 157
 158
 159
 160
 161
 162
 163
 164
 165
 166
 167
 168
 169
 170
 171
 172
 173
 174
 175
 176
 177
 178
 179
 180
 181
 182
 183
 184
 185
 186
 187
 188
 189
 190
 191
 192
 193
 194
 195
 196
 197
 198
 199
 200
 201
 202
 203
 204
 205
 206
 207
 208
 209
 210
 211
 212
 213
 214
 215
 216
 217
 218
 219
 220
 221
 222
 223
 224
 225
 226
 227
 228
 229
 230
 231
 232
 233
 234
 235
 236
 237
 238
 239
 240
 241
 242
 243
 244
 245
 246
 247
 248
 249
 250
 251
 252
 253
 254
 255
 256
 257
 258
 259
 260
 261
 262
 263
 264
 265
 266
 267
 268
 269
 270
 271
 272
 273
 274
 275
 276
 277
 278
 279
 280
 281
 282
 283
 284
 285
 286
 287
 288
 289
 290
 291
 292
 293
 294
 295
 296
 297
 298
 299
 300
 301
 302
 303
 304
 305
 306
 307
 308
 309
 310
 311
 312
 313
 314
 315
 316
 317
 318
 319
 320
 321
 322
 323
 324
 325
 326
 327
 328
 329
 330
 331
 332
 333
 334
 335
 336
 337
 338
 339
 340
 341
 342
 343
 344
 345
 346
 347
 348
 349
 350
 351
 352
 353
 354
 355
 356
 357
 358
 359
 360
 361
 362
 363
 364
 365
 366
 367
 368
 369
 370
 371
 372
 373
 374
 375
 376
 377
 378
 379
 380
 381
 382
 383
 384
 385
 386
 387
 388
 389
 390
 391
 392
 393
 394
 395
 396
 397
 398
 399
 400
 401
 402
 403
 404
 405
 406
 407
 408
 409
 410
 411
 412
 413
 414
 415
 416
 417
 418
 419
 420
 421
 422
 423
 424
 425
 426
 427
 428
 429
 430
 431
 432
 433
 434
 435
 436
 437
 438
 439
 440
 441
 442
 443
 444
 445
 446
 447
 448
 449
 450
 451
 452
 453
 454
 455
 456
 457
 458
 459
 460
 461
 462
 463
 464
 465
 466
 467
 468
 469
 470
 471
 472
 473
 474
 475
 476
 477
 478
 479
 480
 481
 482
 483
 484
 485
 486
 487
 488
 489
 490
 491
 492
 493
 494
 495
 496
 497
 498
 499
 500
 501
 502
 503
 504
 505
 506
 507
 508
 509
 510
 511
 512
 513
 514
 515
 516
 517
 518
 519
 520
 521
 522
 523
 524
 525
 526
 527
 528
 529
 530
 531
 532
 533
 534
 535
 536
 537
 538
 539
 540
 541
 542
 543
 544
 545
 546
 547
 548
 549
 550
 551
 552
 553
 554
 555
 556
 557
 558
 559
 560
 561
 562
 563
 564
 565
 566
 567
 568
 569
 570
 571
 572
 573
 574
 575
 576
 577
 578
 579
 580
 581
 582
 583
 584
 585
 586
 587
 588
 589
 590
 591
 592
 593
 594
 595
 596
 597
 598
 599
 600
 601
 602
 603
 604
 605
 606
 607
 608
 609
 610
 611
 612
 613
 614
 615
 616
 617
 618
 619
 620
 621
 622
 623
 624
 625
 626
 627
 628
 629
 630
 631
 632
 633
 634
 635
 636
 637
 638
 639
 640
 641
 642
 643
 644
 645
 646
 647
 648
 649
 650
 651
 652
 653
 654
 655
 656
 657
 658
 659
 660
 661
 662
 663
 664
 665
 666
 667
 668
 669
 670
 671
 672
 673
 674
 675
 676
 677
 678
 679
 680
 681
 682
 683
 684
 685
 686
 687
 688
 689
 690
 691
 692
 693
 694
 695
 696
 697
 698
 699
 700
 701
 702
 703
 704
 705
 706
 707
 708
 709
 710
 711
 712
 713
 714
 715
 716
 717
 718
 719
 720
 721
 722
 723
 724
 725
 726
 727
 728
 729
 730
 731
 732
 733
 734
 735
 736
 737
 738
 739
 740
 741
 742
 743
 744
 745
 746
 747
 748
 749
 750
 751
 752
 753
 754
 755
 756
 757
 758
 759
 760
 761
 762
 763
 764
 765
 766
 767
 768
 769
 770
 771
 772
 773
 774
 775
 776
 777
 778
 779
 780
 781
 782
 783
 784
 785
 786
 787
 788
 789
 790
 791
 792
 793
 794
 795
 796
 797
 798
 799
 800
 801
 802
 803
 804
 805
 806
 807
 808
 809
 8010
 8011
 8012
 8013
 8014
 8015
 8016
 8017
 8018
 8019
 8020
 8021
 8022
 8023
 8024
 8025
 8026
 8027
 8028
 8029
 8030
 8031
 8032
 8033
 8034
 8035
 8036
 8037
 8038
 8039
 8040
 8041
 8042
 8043
 8044
 8045
 8046
 8047
 8048
 8049
 8050
 8051
 8052
 8053
 8054
 8055
 8056
 8057
 8058
 8059
 8060
 8061
 8062
 8063
 8064
 8065
 8066
 8067
 8068
 8069
 8070
 8071
 8072
 8073
 8074
 8075
 8076
 8077
 8078
 8079
 8080
 8081
 8082
 8083
 8084
 8085
 8086
 8087
 8088
 8089
 8090
 8091
 8092
 8093
 8094
 8095
 8096
 8097
 8098
 8099
 80100
 80101
 80102
 80103
 80104
 80105
 80106
 80107
 80108
 80109
 80110
 80111
 80112
 80113
 80114
 80115
 80116
 80117
 80118
 80119
 80120
 80121
 80122
 80123
 80124
 80125
 80126
 80127
 80128
 80129
 80130
 80131
 80132
 80133
 80134
 80135
 80136
 80137
 80138
 80139
 80140
 80141
 80142
 80143
 80144
 80145
 80146
 80147
 80148
 80149
 80150
 80151
 80152
 80153
 80154
 80155
 80156
 80157
 80158
 80159
 80160
 80161
 80162
 80163
 80164
 80165
 80166
 80167
 80168
 80169
 80170
 80171
 80172
 80173
 80174
 80175
 80176
 80177
 80178
 80179
 80180
 80181
 80182
 80183
 80184
 80185
 80186
 80187
 80188
 80189
 80190
 80191
 80192
 80193
 80194
 80195
 80196
 80197
 80198
 80199
 80200
 80201
 80202
 80203
 80204
 80205
 80206
 80207
 80208
 80209
 80210
 80211
 80212
 80213
 80214
 80215
 80216
 80217
 80218
 80219
 80220
 80221
 80222
 80223
 80224
 80225
 80226
 80227
 80228
 80229
 80230
 80231
 80232
 80233
 80234
 80235
 80236
 80237
 80238
 80239
 80240
 80241
 80242
 80243
 80244
 80245
 80246
 80247
 80248
 80249
 80250
 80251
 80252
 80253
 80254
 80255
 80256
 80257
 80258
 80259
 80260
 80261
 80262
 80263
 80264
 80265
 80266
 80267
 80268
 80269
 80270
 80271
 80272
 80273
 80274
 80275
 80276
 80277
 80278
 80279
 80280
 80281
 80282
 80283
 80284
 80285
 80286
 80287
 80288
 80289
 80290
 80291
 80292
 80293
 80294
 80295
 80296
 80297
 80298
 80299
 80300
 80301
 80302
 80303
 80304
 80305
 80306
 80307
 80308
 80309
 80310
 80311
 80312
 80313
 80314
 80315
 80316
 80317
 80318
 80319
 80320
 80321
 80322
 80323
 80324
 80325
 80326
 80327
 80328
 80329
 80330
 80331
 80332
 80333
 80334
 80335
 80336
 80337
 80338
 80339
 80340
 80341
 80342
 80343
 80344
 80345
 80346
 80347
 80348
 80349
 80350
 80351
 80352
 80353
 80354
 80355
 80356
 80357
 80358
 80359
 80360
 80361
 80362
 80363
 80364
 80365
 80366
 80367
 80368
 80369
 80370
 80371
 80372
 80373
 80374
 80375
 80376
 80377
 80378
 80379
 80380
 80381
 80382
 80383
 80384
 80385
 80386
 80387
 80388
 80389
 80390
 80391
 80392
 80393
 80394
 80395
 80396
 80397
 80398
 80399
 80400
 80401
 80402
 80403
 80404
 80405
 80406
 80407
 80408
 80409
 80410
 80411
 80412
 80413
 80414
 80415
 80416
 80417
 80418
 80419
 80420
 80421
 80422
 80423
 80424
 80425
 80426
 80427
 80428
 80429
 80430
 80431
 80432
 80433
 80434
 80435
 80436
 80437
 80438
 80439
 80440
 80441
 80442
 80443
 80444
 80445
 80446
 80447
 80448
 80449
 80450
 80451
 80452
 80453
 80454
 80455
 80456
 80457
 80458
 80459
 80460
 80461
 80462
 80463
 80464
 80465
 80466
 80467
 80468
 80469
 80470
 80471
 80472
 80473
 80474
 80475
 80476
 80477
 80478
 80479
 80480
 80481
 80482
 80483
 80484
 80485
 80486
 80487
 80488
 80489
 80490
 80491
 80492
 80493
 80494
 80495
 80496
 80497
 80498
 80499
 80500
 80501
 80502
 80503
 80504
 80505
 80506
 80507
 80508
 80509
 80510
 80511
 80512
 80513
 80514
 80515
 80516
 80517
 80518
 80519
 80520
 80521
 80522
 80523
 80524
 80525
 80526
 80527
 80528
 80529
 80530
 80531
 80532
 80533
 80534
 80535
 80536
 80537
 80538
 80539
 80540
 80541
 80542
 80543
 80544
 80545
 80546
 80547
 80548
 80549
 80550
 80551
 80552
 80553
 80554
 80555
 80556
 80557
 80558
 80559
 80560
 80561
 80562
 80563
 80564
 80565
 80566
 80567
 80568
 80569
 80570
 80571
 80572
 80573
 80574
 80575
 80576
 80577
 80578
 80579
 80580
 80581
 80582
 80583
 80584
 80585
 80586
 80587
 80588
 80589
 80590
 80591
 80592
 80593
 80594
 80595
 80596
 80597
 80598
 80599
 80600
 80601
 80602
 80603
 80604
 80605
 80606
 80607
 80608
 80609
 80610
 80611
 80612
 80613
 80614
 80615
 80616
 80617
 80618
 80619
 80620
 80621
 80622
 80623
 80624
 80625
 806

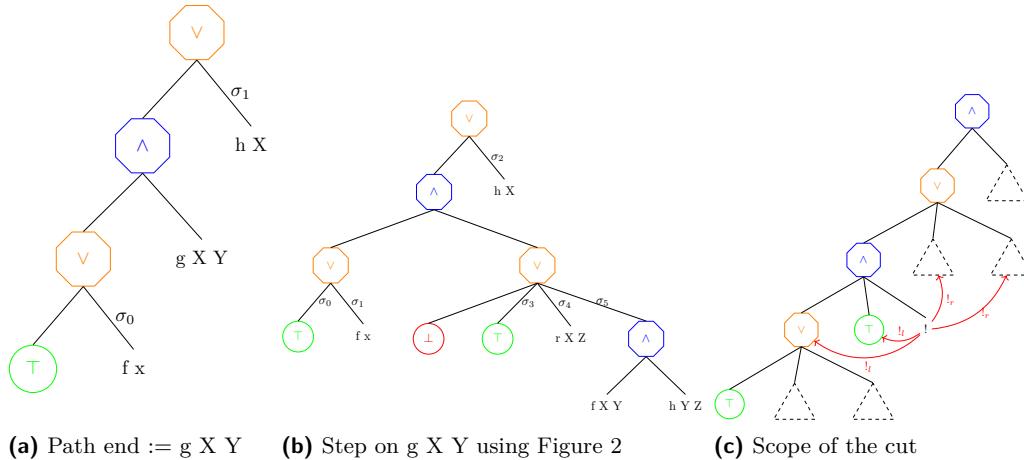


Figure 4 Some tree representations

138 In Figure 4a the *path_end* of the tree is `g X Y`.

139 Below we define three special kinds of trees depending on their *path_end*.

140 **Definition** `success A := path_end A == OK.` (3)

141 **Definition** `failed A := path_end A == KO.` (4)

142 **Definition** `path_atom A := if path_end A is TA _ then true else false.` (5)

143 The latter definition identifies path ending in an atom.

144 4.1 The *step* procedure

145 The *step* procedure takes as input a program, a set of free variables (fv), a substitution, and
146 a tree, and returns an updated set of free variables, a *step_tag*, and an updated tree.

147 Free variables are those variables that appear in a tree; they are used and updated when
148 a backchaining operation takes place.

149 The *step_tag* (see Type 6) indicates the kind of an internal tree step: `CutBrothers` denotes
150 the interpretation of a superficial cut, i.e., a cut whose parent nodes are all *And*-nodes.
151 `Expanded` denotes the interpretation of non-superficial cuts or predicate calls. `Failure` and
152 `Success` are returned for, respectively, `failed` and `success` trees.

153 The step procedure is intended to interpretate atoms, that is, it transforms the tree iff its
154 *path_end* is an atom, otherwsise, it returns the identity.

155 **Lemma** `succ_step_iff u p fv s A: success A ↔ step u p fv s A = (fv, Success, A).` (1)

156 **Lemma** `fail_step_iff u p fv s A: failed A ↔ step u p fv s A = (fv, Failed, A).` (2)

157 *Call step* The interpretation of a call *c* stars by calling the *bc* function on *c*. The output
158 list *l* is taken to represent build the new subtree. If *l* is empty then *KO* tree is returned,
159 otherwise the subtree is a right-skewed tree made of *n* inner *Or* nodes, where *n* is the length
160 of *l*. The root has *KO* as left child. The lhs of the other nodes is a right-skewed tree of *And*
161 nodes. The *And* nodes are again a right-seked tree containing premises of the selected rule .

162 A step in the tree of Figure 4a makes a backchain operation over the query `g X Y` and, in
163 the program defined in Figure 2, the new tree would be the one in Figure 4b. We have put a
164 red border aroung the new generated subtree. It is a disjunction of four subtrees: the first
165 node is the *KO* node (by default), the second is *OK*, since *r1* has no premises, the third and
166 the fourth contains the premises of respectively *r2* and *r3*.

dire dei reset
point

dire che le sostituzioni del backchain sono importanti

CVIT 2016

167 *Cut step* The cut case is delicate, since interpreting a cut in a tree has three main impacts:
 168 at first the cut is replaced by the *OK* node, then some special subtrees, in the scope of the
 169 *Cut*, are cut away: in particular we need to soft-kill the left-siblings of the *Cut* and hard-kill
 170 the right-uncles of the the *Cut*.

171 ► **Definition 1** (Left-siblings (resp. right-sibling)). *Given a node A, the left-siblings (resp.*
 172 *right-sibling) of A are the list of subtrees sharing the same parent of A and that appear on*
 173 *its left (resp. right).*

174 ► **Definition 2** (Right-uncles). *Given a node A, the right-uncles of A are the list of right-sibling*
 175 *of the father of A.*

176 ► **Definition 3** (Hard-kill, $!_r$). *Given a tree t, hard-kill replaces the given subtree with the*
 177 *KO node*

178 ► **Definition 4** (Soft-kill, $!_l$). *Given a successfull tree t, soft-kill replaces with KO all subtrees*
 179 *that are not part of the path in t leading to the OK node.*

180 An example of the impact of the cut is show in Figure 4c, the dashed triangles represent
 181 generic trees. The step routine interprets the cut since it is the node in its path-end: we pass
 182 through a and and all trees on the left of the cut are successful. In the example we have 4
 183 arrow tagged with the $!_l$ or $!_r$ symbols. The $!_l$ arrows go left and soft-kill the pointed subtree,
 184 it keeps *OK* nodes since they are part of the tree leading to the cut, and replaces the other
 185 subtrees with *KO*. The $!_r$ procedure replaces the nodes pointed by the arrows with *KO*.

186 4.2 The *next_alt* procedure

187 It is evident that the *step* alone is not sufficient to reproduce entirely the behavior of the full
 188 expected prolog interpreter. In particular, we need to bracktrack on failures. Moreover, in
 189 case of success, we should return a state where the state in cleaned of the success itself, this
 190 is essential to, non deterministically, find all the solution of a given query. By Lemmas 1
 191 and 2, we know that *step* returns the identity on successful and failed states. In order to
 192 continue the computation on these particular trees, we need the *next_alt* procedure aiming
 193 to especially work with failed and successful trees: and its implementation in Figure 5.

194 The *next_alt* procedure takes a boolean and a tree, clean it from failures or success and
 195 returns a new tree if this tree still contains a non explored path. The idea behind *next_alt* is
 196 to clean recursively every subtree in DFS order if its *path_end* is a failure. Moreover, if the
 197 boolean passed to *next_alt* is true, then it erases the first successful path in the tree.

198 The base cases of *next_alt* are immidiate. The *Or* case is rathere intuitive: if the lhs
 199 of the *Or* does not exist we look for the *next_alt* in the rhs. Otherwise, we look for the
 200 *next_alt* in the lhs, if this *next_alt* does not exists, we look for the *next_alt* in the rhs.

201 We want to spend few words about the *And* case, since the reset point *B0* for *B* plays an
 202 important role. The *next_alt* in an *And* tree should consider two cases: if the lhs succeeds,
 203 then the *next_alt* should be retrived in the rhs. If this alternative does not exists it means
 204 that the rhs has entirely been explored. We need to erase the success in the lhs and try to
 205 find if a non-explored alternative exists. If so, we return a new tree with the new lhs and the
 206 rhs is built from the reset point. *big_and* is a trivial function build a right-skewed tree of
 207 and nodes where the leaves are the atoms written in the reset point. We need to reuse the
 208 reset point since, the step procedure in *And* trees evaluates the rhs of a *And* tree if the lhs
 209 succeeds. This evaluation is dependent on the subsitution in the lhs tree. Therefore, if we
 210 need to backtrack in the lhs, we need to reset the rhs.

```

Definition next_alt :  $\mathbb{B} \rightarrow \text{tree} \rightarrow \text{option tree} :=$ 
fix next_alt b A :=
match A with
| KO => None
| OK => if b then None else Some OK
| TA _ => Some A
| And A B0 B =>
let build_B0 A := And A B0 (big_and B0) in
if success A then
  match next_alt b B with
  | None => omap build_B0 (next_alt true A)
  | Some B' => Some (And A B0 B')
  end
else if failed A then omap build_B0 (next_alt false A)
else Some (And A B0 B)
| Or None sB B => omap (fun x => Or None sB x) (next_alt b B)
| Or (Some A) sB B =>
  match next_alt b A with
  | None => omap (fun x => Or None sB x) (next_alt false B)
  | Some A' => Some (Or (Some A') sB B)
end
end.

```

■ **Figure 5** *next_alt* implementation

Some interesting property of *next_alt* are shown below and allow to see how *next_alt* complements *step*.

```

213 Lemma path_atom_next_alt_id b A: path_atom A → next_alt b A = Some A. (3)
214 Lemma next_alt_failedF b A A': next_alt b A = Some A' → failed A' = false. (4)

```

For example, in Figure 4b the step procedure has created a failed state: its path-end ends in *KO*. The expected behavior of *next_alt* is to take this *KO* node and make it a This allows *step* to continue the exploration of the tree. In particular, the path-end of this new state end in *OK*. The step leaves the state unchanged producing the new substitution. This solution however is not unique, we should be able to backtrack on this successful state. To do so we can call *next_alt* and it will deadify the *OK* node allowing *step* to proceed on r X Z.

subst taken form
the or

221 4.3 The *run* inductive

The inductive procedure *run* is modeled as a function: it takes as input a program, a set of free variables, an initial substitution σ_0 , and a tree t_0 , and returns a substitution σ_1 together with an optional updated tree t_1 . The substitution σ_1 represents the most-general unifier that makes the execution of the tree t_0 succeed starting from the initial substitution σ_0 , σ_1 is an extension of σ_0 . The tree t_1 is the updated tree containing the alternatives that have not yet been explored. If the tree contains no solution, then *None* is returned.

The procedure *run* is based on three main derivation rules, shown in Figure 6. If the *path_end* of the tree *t* is a success, the input substitution is returned and the input tree is cleaned of its successful path. If the *path_end* of the tree is an atom, then *step* is invoked to evaluate this atom, and *run* is recursively called on the new tree. Finally, if the *path_end* of the tree is a failure, *next_alt* is called to clear the failed path; if the resulting cleaned tree exists, *run* is recursively called on it.

$$\begin{array}{c}
 \frac{\text{success } A \quad \text{get_subst } s1 A = s2 \quad (\text{next_alt true } A) = B}{\text{run fv } s1 A s2 B} \text{ RUN_DONE} \\
 \\
 \frac{\text{path_atom } A \quad \text{step } u p \text{ fv0 } s1 A = (\text{fv1}, \text{st}, B) \quad \text{run fv1 } s1 B s2 r}{\text{run fv0 } s1 A s2 r} \text{ RUN_STEP} \\
 \\
 \frac{\text{failed } A \quad \text{next_alt false } A = \text{Some } B \quad \text{run fv0 } s1 B s2 r}{\text{run fv0 } s1 A s2 r} \text{ RUN_FAIL}
 \end{array}$$

Figure 6 Rule system for *run***4.4 Valid trees**

235 The inductive tree allows one to generate a large number of trees, some of which are not
 236 valid, in the sense that they cannot be produced starting from a given query. The class of
 237 valid trees is characterized by the following function.

```

Fixpoint valid_tree s :=
  match s with
  | TA _ | OK | KO => true
  | Or None _ B => valid_tree B
  | Or (Some A) _ B => valid_tree A && ((B == KO) || B.base_or B)
  | And A B0 B => valid_tree A &&
    if success A then valid_tree B
    else B == big_and B0
  end.
  
```

238 Once again, the most interesting cases to analyze are *Or* and *And*.

239 For the *Or* constructor, we distinguish two cases depending on whether the left-hand
 240 side (lhs) exists. If it does not exist, then the right-hand side (rhs) must be a valid tree.
 241 Otherwise, the lhs must itself be a valid tree, and the rhs is either the *KO* tree, since it may
 242 have been removed by the evaluation of a superficial cut in the lhs, or it has not yet been
 243 explored. In the latter case, it is a *base_or* tree, namely the right-skewed tree formed by a
 244 disjunction of conjunctions.

245 For the *And* constructor, the lhs is required to be a valid tree. The shape of the rhs
 246 depends on whether the lhs represents a success. If the lhs is not successful, then the rhs
 247 has never been explored: the procedures *step* and *next_alt* modify the rhs only when the
 248 lhs succeeds. In this case, the lhs must be the right-skewed tree containing the conjunctions
 249 of the atoms present in the reset point *B*₀. In other words, the rhs coincides with the reset
 250 point. If the lhs is a success tree, then the rhs must be a valid tree.

5 Elpi semantics

252 We now want to introduce the elpi semantics. The interpreter we show reflects the interpreter
 253 of the ELPI language and is an operational semantics close to the one picked by Pusch in
 254 [16], in turn closely related to the one given by Debray and Mishra in [6, Section 4.3]. Pusch
 255 mechanized the semantics in Isabelle/HOL together with some optimizations that are present
 256 in the Warren Abstract Machine [20, 1].

257 The inductive representing a state of the ELPI language is shown below.

```

Inductive alts := no_alt | more_alt of (Σ * goals) & alts
with goals := no_goals | more_goals of (A * alts) & goals .
  
```

```

Definition stepE fv t s a gl :=
  let (fv', rs) := bc u p fv t s in
  let rs_ca := save_alts a gl (r2a rs) in
  (fv', rs_ca).

Inductive nur :  $\mathcal{F}_v \rightarrow \Sigma \rightarrow \text{goals} \rightarrow \text{alts} \rightarrow \Sigma \rightarrow \text{alts} \rightarrow \text{Prop}$  :=

$$\frac{}{\text{nur fv s } [::] \text{ a s a}} \text{STOPE}$$


$$\frac{\text{nur fv s gl ca s1 r}}{\text{nur fv s } [:: (\text{cut}, \text{ca}) \& \text{ gl}] \text{ a s1 r}} \text{CUTE}$$


$$\frac{\text{stepE fv t s al gl} = (\text{fv}', [:: b \& bs]) \quad \text{nur fv' b.1 b.2 (bs++al) s1 r}}{\text{nur fv s } [:: (\text{call t}, \text{ca}) \& \text{ gl}] \text{ al s1 r}} \text{CALLE}$$


$$\frac{\text{stepE fv t s al gl} = (\text{fv}', [::]) \quad \text{nur fv' s1 a al s2 r}}{\text{nur fv s } [:: (\text{call t}, \text{ca}) \& \text{ gl}] \text{ [:: (s1, a) \& al] s2 r.}} \text{FAILE}$$


```

Figure 7 Rule system for *nur*

258 An elpi state is an enhanced two-dimension list. The outermost list represents the list
 259 of alternatives in disjunction accompany with the substitution that should be used to
 260 for their interpretation. The innermost list is a list of atom, representing a list of goals in
 261 conjunctions. These goals are decorated with a pointer to an elpi state, and are used to keep
 262 trace of the alternatives that should be kept when a cut is interpreted. We call these, special,
 263 alternatives the cut-to alternatives.

264 The idea of the ELPI interpreter is to receive a list of alternatives. The first alternative
 265 consists of a list of goals. Four cases must be taken into account; they are shown in Figure 7.
 266 In order to simplify goal retrieval, we split the head of the alternatives from the tail, so
 267 that it can be immediately matched in the inductive definition. Note that an empty list of
 268 alternatives represents, by definition, a failing state. If the goal list is empty (STOPE), then
 269 we have, by definition, a success, and the input solution together with the list of alternatives
 270 is returned. If the goal list starts with a cut (CUTE), then the current alternatives are erased
 271 in favour of the cut alternatives, and a recursive call is made on the remaining goal list.

272 Finally, we must consider the case in which the goal list starts with a call. The call
 273 can either fail (FAILE) or succeed (CALLE). We distinguish the two cases by looking if the
 274 backtracking operation returns zero or more rules. We have wrapped this task in the *stepE*
 275 procedure, which also updates the goal and cut-alternative list. The fail case, is relatively
 276 easy: the first goal does not succeed, we need to take the head of the alternatives, and make
 277 it the new list of goals to be explored.

278 The case in which backtracking produces a non empty list, the *save_alts* routine is in
 279 charge of: taking the list of premises and add to each atom the the list of alternatives *a* as
 280 their new cut-alternatives, then it append the list of goals *gl* to each of these new lists.

281

6

Semantic equivalence

²⁸² The equivalence between the two semantics is possible under two conditions: we need to work with "valid states", i.e. state that can be generated from a query of type call.

284 6.1 From trees to lists

285 The translation of a tree to a list is as follows:

```

Fixpoint t2l (A: tree) s (bt : alts) : alts :=

  match A with
  | OK          => [:: (s, [::]) ]
  | KO          => [::]
  | TA a        => [:: (s, [:: (a,[::]) ]) ]
  | Or None s1 B => add_ca_deep bt (t2l B s1 [::])
  | Or (Some A) s1 B   =>
    let lB := t2l B s1 [::] in
    let lA := t2l A s lB in
    add_ca_deep bt (lA ++ lB)
  | And A B0 B   =>
    let lA := t2l A s bt in
    if lA is [:: (s1A, x) & xs] then
      let lB0 := a2g B0 in
      let xz := add_deepG bt lB0 x in
      let xs := add_deep bt lB0 xs in
      let xs := map (catr lB0) xs in
      let lB := t2l B s1A (xs ++ bt) in
      (map (catl xz) lB) ++ xs
    else [::]
  end.

```

► Theorem 5 (tree_to_elpi).

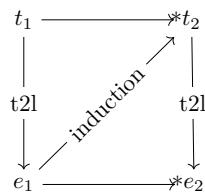
286 $\forall A \sigma_1 B \sigma_2 b \sigma_0, \text{vt } A \rightarrow$
 287 $\text{run}_u \sigma_1 A (\text{Some } \sigma_2) B b \rightarrow$
 288 $\exists x xs, t2l A \sigma_1 \emptyset = x ::: xs \wedge \text{nur}_u x.1 x.2 xs \sigma_2 (t2l B \sigma_0 \emptyset).$

► Theorem 6 (elpi_to_tree).

289 $\forall \sigma_1 \sigma_2 a \; na \; g,$
 290 $\text{nur}_u \sigma_1 g \; a \; \sigma_2 \; na \rightarrow$
 291 $\forall \sigma_0 t, \text{vt} \; t \rightarrow (\text{t2l} \; t \; \sigma_0 \; \emptyset) = ((\sigma_1, g) :: a) \rightarrow$
 292 $\exists t' \; n, \text{run}_u \sigma_0 \; t \; (\text{Some} \; \sigma_2) \; t' \; n \wedge \text{t2l} \; t' \; \sigma_0 \; \emptyset = na.$

The proof of Theorem 6 is based on the idea explained in [2, Section 3.3]. An ideal statement for this lemma would be: given a function $12t$ transforming an elpi state to a tree, we would have that the execution of an elpi state e is the same as executing run on the tree resulting from $12t(e)$. However, it is difficult to retrieve the structure of an elpi state and create a tree from it. This is because, in an elpi state, we have no clear information about the scope of an atom inside the list and, therefore, no evident clue about where this atom should be placed in the tree.

Our theorem states that, starting from a valid state t which translates to a list of alternatives $(\sigma_1, q) :: a$. If we run in elpi the list of alternatives, then the execution of the



■ **Figure 8** Induction scheme for Theorem 6

302 tree t returns the same result as the execution in elpi. The proof is performed by induction
 303 on the derivations of the elpi execution. We have 4 derivations.
 304 We have 4 case to analyse:

305 7 Case study: determinacy analysis

306 we mechanize the first order part of xxx.
 307 snippet det, main thm, invariant det tree (valid tree prev section?)
 308 proof induciton on exec, step/next alt preserving invariant proved by induction on the
 309 tree.
 310 with list semantics cut and next alt requires to express a link btween the ca or next alts
 311 and the current goal, which is non trivial without an intermediate data strature like the tree

312 8 Related work

313 prolog semantics, King lost
 314 yves for the proof technique

315 9 Conclusion

316 References

- 317 1 Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, 08
 318 1991. doi:10.7551/mitpress/7160.001.0001.
- 319 2 Yves Bertot. A certified compiler for an imperative language. Technical Report RR-3488,
 320 INRIA, September 1998. URL: <https://inria.hal.science/inria-00073199v1>.
- 321 3 Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller,
 322 Assia Mahboubi, and Pierre Vial. Compositional pre-processing for automated reasoning in
 323 dependent type theory. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve
 324 Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on
 325 Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages
 326 63–77. ACM, 2023. doi:10.1145/3573105.3575676.
- 327 4 Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: Proof transfer for free, with or
 328 without univalence. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages
 329 239–268, Cham, 2024. Springer Nature Switzerland.
- 330 5 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: Algebraic hierarchies
 331 Made Easy in Coq with Elpi. In *Proceedings of FSCD*, volume 167 of *LIPICS*, pages 34:1–34:21,
 332 2020. URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2020.34>.
- 333 6 Saumya K. Debray and Prateek Mishra. Denotational and operational semantics for prolog. *J.
 334 Log. Program.*, 5(1):61–91, March 1988. doi:10.1016/0743-1066(88)90007-6.

23:12 Operational semantics for Prolog with Cut in Rocq and its application to determinacy analysis

- 336 7 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast,
337 embeddable, λ Prolog interpreter. In *Proceedings of LPAR*, volume 9450 of *LNCS*, pages
338 460–468. Springer, 2015. URL: <https://inria.hal.science/hal-01176856v1>, doi:10.1007/978-3-662-48899-7_32.
- 340 8 Davide Fissore and Enrico Tassi. A new Type-Class solver for Coq in Elpi. In *The Coq
341 Workshop*, July 2023. URL: <https://inria.hal.science/hal-04467855>.
- 342 9 Davide Fissore and Enrico Tassi. Higher-order unification for free!: Reusing the meta-
343 language unification for the object language. In *Proceedings of PPDP*, pages 1–13. ACM, 2024.
344 doi:10.1145/3678232.3678233.
- 345 10 Davide Fissore and Enrico Tassi. Determinacy checking for elpi: an higher-order logic program-
346 ming language with cut. In *Practical Aspects of Declarative Languages: 28th International
347 Symposium, PADL 2026, Rennes, France, January 12–13, 2026, Proceedings*, pages 77–95,
348 Berlin, Heidelberg, 2026. Springer-Verlag. doi:10.1007/978-3-032-15981-6_5.
- 349 11 Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. Practical and sound equality
350 tests, automatically. In *Proceedings of CPP*, page 167–181. Association for Computing
351 Machinery, 2023. doi:10.1145/3573105.3575683.
- 352 12 Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing type theory
353 in higher order constraint logic programming. In *Mathematical Structures in Computer
354 Science*, volume 29, pages 1125–1150. Cambridge University Press, 2019. doi:10.1017/
355 S0960129518000427.
- 356 13 Robbert Krebbers, Luko van der Maas, and Enrico Tassi. Inductive Predicates via Least
357 Fixpoints in Higher-Order Separation Logic. In Yannick Forster and Chantal Keller, editors,
358 *16th International Conference on Interactive Theorem Proving (ITP 2025)*, volume 352 of *Leib-
359 niz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:21, Dagstuhl, Germany,
360 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs. ITP.2025.27>, doi:10.4230/LIPIcs. ITP.2025.27.
- 362 14 Dale Miller. A logic programming language with lambda-abstraction, function variables, and
363 simple unification. In *Extensions of Logic Programming*, pages 253–281. Springer, 1991.
- 364 15 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge
365 University Press, 2012.
- 366 16 Cornelia Pusch. Verification of compiler correctness for the wam. In Gerhard Goos, Juris
367 Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors,
368 *Theorem Proving in Higher Order Logics*, pages 347–361, Berlin, Heidelberg, 1996. Springer
369 Berlin Heidelberg.
- 370 17 Enrico Tassi. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λ Prolog
371 dialect). In *The Fourth International Workshop on Coq for Programming Languages*, January
372 2018. URL: <https://inria.hal.science/hal-01637063>.
- 373 18 Enrico Tassi. Deriving proved equality tests in Coq-Elpi. In *Proceedings of ITP*, volume 141 of
374 *LIPIcs*, pages 29:1–29:18, September 2019. URL: <https://inria.hal.science/hal-01897468>,
375 doi:10.4230/LIPIcs.CVIT.2016.23.
- 376 19 Luko van der Maas. Extending the Iris Proof Mode with inductive predicates using Elpi.
377 Master’s thesis, Radboud University Nijmegen, 2024. doi:10.5281/zenodo.12568604.
- 378 20 David H.D. Warren. An Abstract Prolog Instruction Set. Technical Report Technical Note 309,
379 SRI International, Artificial Intelligence Center, Computer Science and Technology Division,
380 Menlo Park, CA, USA, October 1983. URL: <https://www.sri.com/wp-content/uploads/2021/12/641.pdf>.