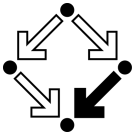


Formalising Anti-Unification in PVS

Marcos Mercandeli Rodrigues (PhD. Candidate, UnB)
Supervisor: Mauricio Ayala-Rincón (UnB)
Cosupervisor: Temur Kutsia (RISC/JKU)

Seminar Formal Methods and
Automated Reasoning
28.04.2026
JKU Linz

Outline

Anti-Unification;
Prototype Verification System (PVS);
Formalising Completeness in PVS.

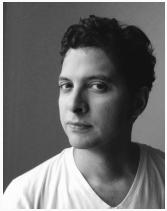
Joint Work With

Maria Júlia Dias Lima

Mauricio Ayala-Rincón

Thaynara Arielly de Lima

Temur Kutsia

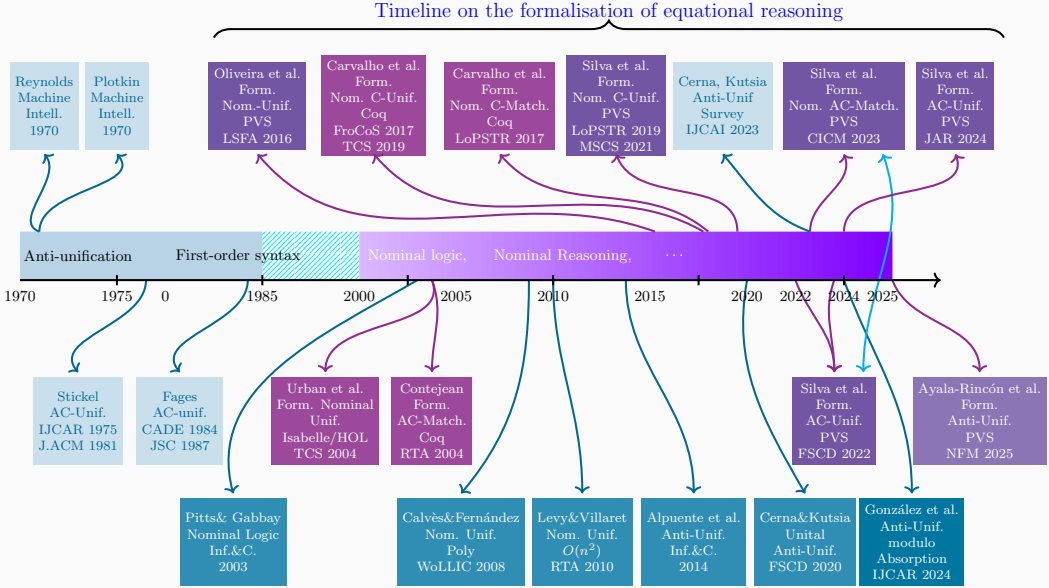


Mariano Miguel Moscato

Equational Reasoning: Anti-unification

Equational Problems

- Equality check: $s = t?$
- Matching: $\exists \sigma \colon s\sigma = t?$
- Unification: $\exists \sigma \colon s\sigma = t\sigma?$
- Anti-unification: $\exists r, \sigma, \tau \colon r\sigma = s \ \& \ r\tau = t?$


Timeline on the formalisation of equational reasoning
Reynolds Machine Intell. 1970
Plotkin Machine Intell. 1970
Oliveira et al. Form. Nom.-Unif. PVS LSFA 2016
Carvalho et al. Form. Nom. C-Unif. Coq FroCoS 2017 TCS 2019
Carvalho et al. Form. Nom. C-Match. Coq LoPSTR 2017
Silva et al. Form. Nom. C-Unif. PVS LoPSTR 2019 MSCS 2021
Cerna, Kutsia Anti-Unif Survey IJCAI 2023
Silva et al. Form. Nom. AC-Match. PVS CICM 2023
Silva et al. Form. AC-Unif. PVS JAR 2024
Anti-unification
First-order syntax
Nominal logic,
Nominal Reasoning,
· · ·
1970
1975
0
1985
2000
2005
2010
2015
2020
2022
2024
2025
Stickel AC-Unif. IJCAR 1975 J.ACM 1981
Fages AC-unif. CADE 1984 JSC 1987
Urban et al. Form. Nominal Unif. Isabelle/HOL TCS 2004
Contejean Form. AC-Match. Coq RTA 2004
Silva et al. Form. AC-Unif. PVS FSCD 2022
Ayala-Rincón et al. Form. Anti-Unif. PVS NFM 2025
Pitts& Gabbay Nominal Logic Inf.&C. 2003
Calvès&Fernández Nom. Unif. Poly WoLLIC 2008
Levy&Villaret Nom. Unif. $O(n^2)$ RTA 2010
Alpuente et al. Anti-Unif. Inf.&C. 2014
Cerna&Kutsia Unital Anti-Unif. FSCD 2020
González et al. Anti-Unif. modulo Absorption IJCAR 2024

Anti-Unification

Generalisation as a Guiding Scientific Principle

Generalisation

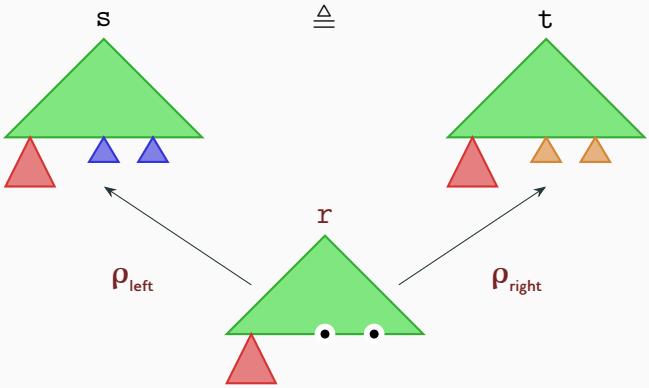
- Finding commonalities between objects while uniformly abstracting their differences;
- Inductive reasoning;
- In Logic: **Anti-unification**.

Generalisation as a Guiding Scientific Principle

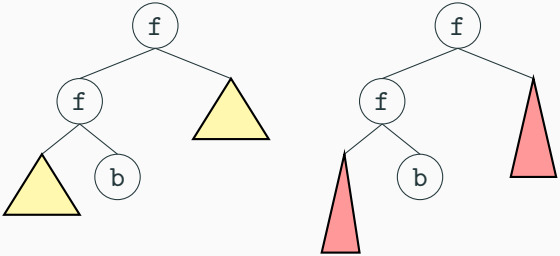
Usefulness?

- In Sciences, inferring general laws from finite amount of data (Plotkin [1970]):
  - **This** sample of pure water boils at 373.3 K;
  - **That** sample of pure water boils at 373.3 K;
  - **Any** sample of pure water boils at 373.3 K.
- In Computer Science:
  - Detecting code clones, preventing bugs, finding duplicate codes (Cerna and Kutsia [2023]);
  - Automated bug-fixing (Bader et al. [2019]);
  - Program repair (Winter et al. [2022]).

Anti-unification

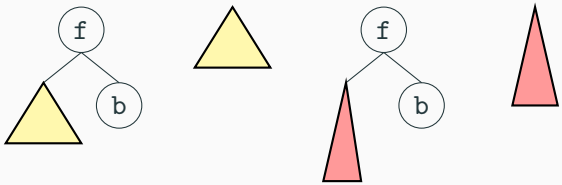


Example

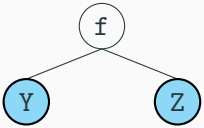


Generaliser

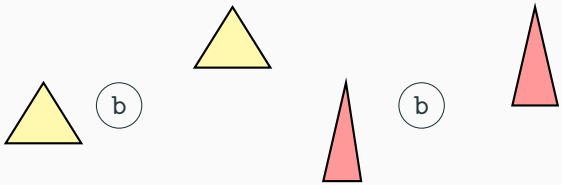
Example



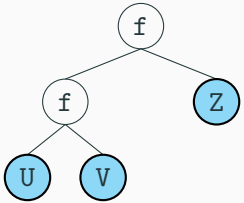
A less general generaliser



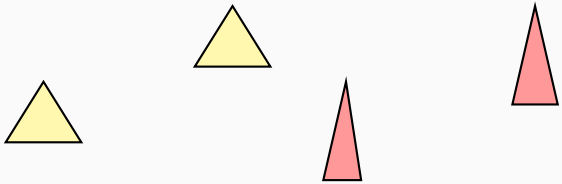
Example



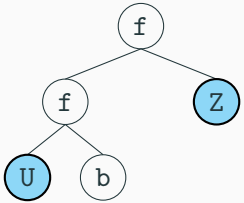
A less general generaliser



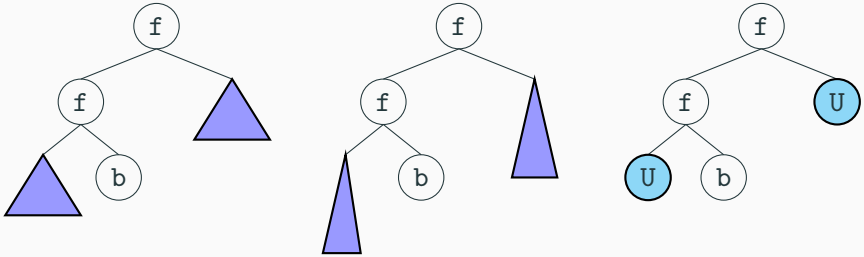
Example



A less general generaliser



Example



What do We Need?

Important Concepts to Formalise

- Terms, Substitutions, and Actions.
- Equalities, Configurations, and Rules;
- Freshness and Generalisers.

Inference Rules [Ayala-Rincón et al., 2025]

$$\text{(Decompose-Function)}\ \frac{\langle f\, s \triangleq_X f\, t,\ U \mid S \mid \sigma\rangle}{\langle s \triangleq_Y t,\ U \mid S \mid \sigma\{X \mapsto f\,Y\}\rangle}$$

$$\text{(Decompose-Pair)}\ \frac{\langle (s,u) \triangleq_X (t,v),\ U \mid S \mid \sigma\rangle}{\langle s \triangleq_Y t,\ u \triangleq_Z v,\ U \mid S \mid \sigma\{X \mapsto (Y,Z)\}\rangle}$$

$$\text{(Solve-Repeated)}\ \frac{\langle s \triangleq_X t,\ U \mid S \mid \sigma\rangle}{\langle U \mid S \mid \sigma\{X \mapsto X'\}\rangle}\ \text{if } s \triangleq_X t \text{ is solved and } s \triangleq_{X'} t \in S$$

$$\text{(Solve-Non-Repeated)}\ \frac{\langle s \triangleq_X t,\ U \mid S \mid \sigma\rangle}{\langle U \mid s \triangleq_X t,\ S \mid \sigma\rangle}\ \text{if } s \triangleq_X t \text{ is solved and non-repeated in } S$$

$$\text{(Syntactic)}\ \frac{\langle s \triangleq_X s,\ U \mid S \mid \sigma\rangle}{\langle U \mid S \mid \sigma\{X \mapsto s\}\rangle}\ \text{if } s \triangleq_X s \text{ is trivial}$$

PVS

A Prototype Verification System

PVS

- An interactive and automated theorem prover;
- Based on a strongly typed higher-order language;
- Specifying and proving mathematical theories as well as for hardware verification (Owre et al. [1995]).

A Prototype Verification System

PVS

- SRI International and NASA;
- Current Version: PVS 8.0;
- Common Lisp;
- Linux/Mac;
- Emacs and VSCode (https://github.com/nasa/vscode-pvs).

NASALib

- NASALib (https://github.com/nasa/pvslib);
- Maintained by the Formal Methods Team at NASA LaRC;
- 62 libraries, 38 000 proven formulas.

Using PVS

- Theories are specified in a `.pvs` file;
- Theories may import other theories;
- Proofs are kept in a `.prf` file;
- One can edit and copy proofs.

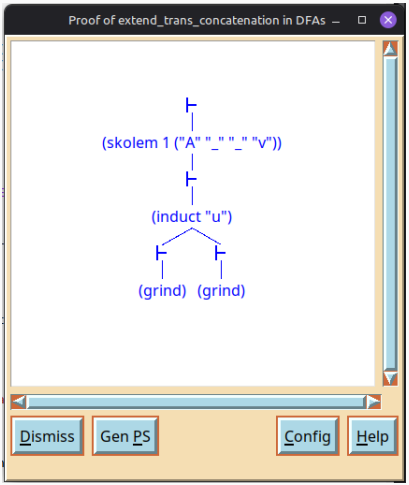
Example: DFAs

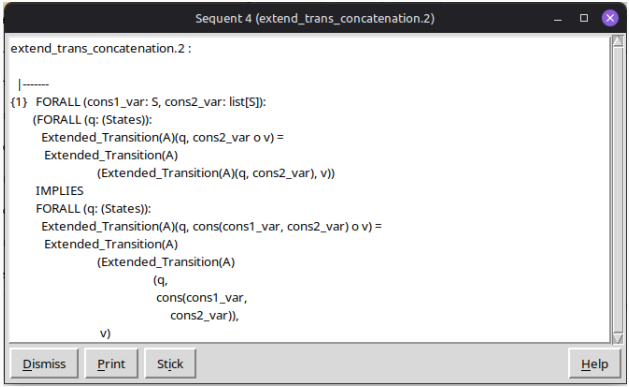
```
DFAs[Q: TYPE+, S: TYPE]: THEORY
BEGIN
        ASSUMING
           Finiteness_Q: ASSUMPTION
              is_finite[Q](fullset[Q])
           Finiteness_S: ASSUMPTION
              is_finite[S](fullset[S])
        ENDASSUMING

        States: set[Q] = fullset[Q]
        Input_Symbols: set[S] = fullset[S]

        DFA: TYPE = [#
                      Start_State: (States),
                      Final_States: { F: set[(States)] | subset?(F, States) },
                      Transition: [(States), (Input_Symbols) -> (States)]
                    #]
END DFAs
```

Proving in PVS

- Proof: Sequent-like proof-tree;
- Proof-goal: A node;
- Root: Formula to be proved;
- Task: Expand the tree and close its branches;
- Strategies (https://sri-fm.github.io/pvs/documentation.html).

PVS


Proof of extend_trans_concatenation in DFAs
(skolem 1 ("A" "_" "_" "v"))
(induct "u")
(grind)
(grind)
Dismiss
Gen PS
Config
Help


Sequent 4 (extend_trans_concatenation.2)
extend_trans_concatenation.2 :
|-------
{1} FORALL (cons1_var: S, cons2_var: list[S]):
(FORALL (q: (States)):
Extended_Transition(A)(q, cons2_var o v) =
Extended_Transition(A)
(Extended_Transition(A)(q, cons2_var), v))
IMPLIES
FORALL (q: (States)):
Extended_Transition(A)(q, cons(cons1_var, cons2_var) o v) =
Extended_Transition(A)
(Extended_Transition(A)
(q,
cons(cons1_var,
cons2_var)),
v)
Dismiss
Print
Stick
Help

TCCs

- The type system of PVS is not algorithmically decidable;
- Theorem proving may be required to establish the type-consistency of a PVS specification;
- Type-correctness conditions;
- Built-in prover strategies or human-guided proofs.

Example: Extended Transition Function

```
IMPORTING Words

Input(A: DFA): TYPE = Word[S]

Extended_Transition(A: DFA)(q: (States), w: Input(A)): RECURSIVE (States) =
   CASES w OF
      null: q,
      cons(head, tail): Extended_Transition(A)(A`Transition(q, head), tail)
   ENDCASES
MEASURE w BY <<

accepted?(A: DFA)(w: Input(A)): bool =
   A`Final_States(Extended_Transition(A)(A`Start_State, w))

Lang(A: DFA): Language[(Input_Symbols)] = { w: Input(A) | accepted?(A)(w) }
```

Example: TCCs

```
% Subtype TCC generated (at line 33, column 64) for  head
    % expected type  (Input_Symbols)
  % proved - complete
Extended_Transition_TCC1: OBLIGATION
  FORALL (A: DFA, q: (States), w: Input(A), head: S, tail: list[S]):
    w = cons(head, tail) IMPLIES Input_Symbols(head);

% Termination TCC generated (at line 33, column 25) for
    % Extended_Transition(A)(A`Transition(q, head), tail)
  % proved - complete
Extended_Transition_TCC2: OBLIGATION
  FORALL (A: DFA, q: (States), w: Input(A), head: S, tail: list[S]):
    w = cons(head, tail) IMPLIES <<[S](tail, w);
```

Demo

Demo

PVS Demo.

Formalising Anti-Unification in PVS

Terms

```
first_order_term[constant: TYPE, variable: TYPE+, f_symbol: TYPE]: DATATYPE
BEGIN
    const (a: constant): const?
    variable (V: variable): var?
    unit: unit?
    pair(term1: first_order_term, term2: first_order_term): pair?
    app(f_sym: f_symbol, arg: first_order_term): app?
END first_order_term
```

first_order_term_adt.pvs

Basic Substitutions

```
basic_sub: TYPE = [variable, first_order_term]
t, arg: VAR first_order_term
sigma_basic: VAR basic_sub

subs(sigma_basic, t): RECURSIVE first_order_term =
    LET X = sigma_basic`1,
        s = sigma_basic`2
        IN CASES t OF
           const(a): const(a),
           variable(Y): IF X = Y THEN s
                        ELSE variable(Y)
                        ENDIF,
           unit: unit,
           pair(t1,t2): pair(subs(sigma_basic,t1),subs(sigma_basic,t2)),
           app(f, arg): app(f, subs(sigma_basic, arg))
    ENDCASES
MEASURE t by <<
```

Substitutions

```
sub: TYPE = list[basic_sub]
sigma: VAR sub

subs(sigma)(t): RECURSIVE first_order_term =
    CASES sigma OF
          null: t,
          cons(head, tail): subs(head, subs(tail)(t))
    ENDCASES
MEASURE sigma BY <<

subs_app: LEMMA
    subs(sigma)(app(f,arg)) = app(f,subs(sigma)(arg))
```

Nice Substitutions

```
nice?(sigma): RECURSIVE bool =
    IF
        null?(sigma)
    THEN
        TRUE
    ELSE
        LET
            (X, t) = car(sigma)
        IN
            (NOT member(X, vars(t)) AND
            NOT member(X, supset_dom(cdr(sigma))) AND
            disjoint?(vars(t), supset_dom(cdr(sigma)))
            AND nice?(cdr(sigma)))
    ENDIF
MEASURE sigma BY <<
```

Antiunify

```
AUT: TYPE = [# lhs, rhs : Term , label : variable #]

Configuration: TYPE = [# unsolved,solved:List_AUT,substitution:(nice?) #]

validConfiguration?(c: Configuration): bool
  = LET allAUTs = append(c`unsolved, c`solved)
    IN
        valid_AUTs?(allAUTs) AND
        disjoint?(supset_dom(c`substitution), labels(allAUTs)) AND
        validSolvedAUTs?(c`solved) AND
        disjoint?(supset_dom(c`substitution), vars(allAUTs))

  freshLabel(V : setof[variable]) : variable =
    epsilon((lambda(v:variable): NOT member(v,V)))

  freshLabel(c:(validConfiguration?)): variable =
    freshLabel(vars(c))
```

Antiunify

```
match_DecF?(eq: AUT): bool =
    LET lhs = eq`lhs, rhs = eq`rhs
    IN (app?(lhs) AND app?(rhs)) AND sym(lhs) = sym(rhs)

match_DecF_conf?(c:(validConfiguration?)): bool = match_DecF?(c`unsolved)

DecF(c: (match_DecF_conf?)): {cp : (validConfiguration?)|cons?(cp`unsolved)
     AND cdr(c`unsolved) = cdr(cp`unsolved)
     AND subs(cp`substitution)(variable(car(c`unsolved)`label)) =
            app(sym((car(c`unsolved))`lhs),variable(car(cp`unsolved)`label))
     AND size(cp`unsolved) < size(c`unsolved)}
 = LET eq = car(c`unsolved),
         lhs = eq`lhs,
         rhs = eq`rhs,
         lbl = freshLabel(c)
   IN c with [unsolved:=cons(makeAUT(arg(lhs),arg(rhs),lbl),cdr(c`unsolved)),
    substitution:=cons((eq`label,app(sym(eq`lhs),
                                variable(lbl))), c`substitution)]
```

Antiunify

```
antiunify(c: (validConfiguration?)): RECURSIVE (validConfiguration?)
  =  IF cons?(c`unsolved)
     THEN
       IF match_DecF_conf?(c)
       THEN
            antiunify(DecF(c))
       ELSIF match_DecP_conf?(c)
       THEN
            antiunify(DecP(c))
       ELSIF match_Synt_conf?(c)
       THEN
            antiunify(Synt(c))
       ELSE
            antiunify(Solve(c))
       ENDIF
      ELSE c
      ENDIF
  MEASURE size(c`unsolved)
```

Completeness

```
generalizer?(c: (validConfiguration?))(gamma: (nice?)) : bool =
    labels(c`unsolved) = dom(gamma) AND
    EXISTS(tau_l, tau_r : sub) :
        FORALL (eq : AUT | member(eq,c`unsolved)) :
           subs(append(tau_l, gamma))(eq`label) = eq`lhs AND
           subs(append(tau_r, gamma))(eq`label) = eq`rhs

r_generalizer?(c: (validConfiguration?))(gamma: (nice?)) : bool =
    subset?(labels(c`unsolved), dom(gamma)) AND
    disjoint?(vars(img(gamma)), vars(antiunify(c))) AND
    disjoint?(difference(dom(gamma), labels(c`unsolved)), vars(antiunify(c))) AND
    EXISTS(tau_l, tau_r : sub) :
        FORALL (eq : AUT | member(eq,c`unsolved)) :
           subs(append(tau_l, gamma))(eq`label) = eq`lhs AND
           subs(append(tau_r, gamma))(eq`label) = eq`rhs

antiunif_completeness: LEMMA % It depends on previous conjectures
    FORALL (c:(validConfiguration?), (gamma : (nice?) | r_generalizer?(c)(gamma))) :
        more_general?(gamma, antiunify(c)`substitution, labels(c`unsolved))

antiunif_is_complete: COROLLARY % It depends on previous conjectures
    FORALL (c:(validConfiguration?), (gamma: (nice?) | generalizer?(c)(gamma))) :
        more_general?(gamma, antiunify(c)`substitution, labels(c`unsolved))
```

Completeness

```
DecF_MakeSub_nice_app_case: LEMMA
   FORALL (c: (match_DecF_conf?), gamma: (r_generalizer?(c))):
      app?(subs(gamma)(car(c`unsolved)`label)) IMPLIES
         LET
            y = car(DecF(c)`unsolved)`label,
            t = subs(gamma)(car(c`unsolved)`label)
         IN
            nice?(MakeSubs(c)(gamma)(cons[basic_sub]((y, arg(t)),  null[basic_sub])))

DecF_MakeSub_r_gen_app_case: LEMMA
   FORALL (c: (match_DecF_conf?), gamma: (r_generalizer?(c))):
      app?(subs(gamma)(car(c`unsolved)`label)) IMPLIES
         LET
            y = car(DecF(c)`unsolved)`label,
            t = subs(gamma)(car(c`unsolved)`label)
         IN
            r_generalizer?(DecF(c))(MakeSubs(c)(gamma)
            (cons[basic_sub]((y, arg(t)),  null[basic_sub])))

matchingFuns_r_generalizer_classification_general: LEMMA
    FORALL(c: (validConfiguration?), gamma:(r_generalizer?(c))):
      FORALL(eq: AUT | member(eq, c`unsolved)):
         match_DecF?(eq) IMPLIES var?(subs(gamma)(eq`label)) OR app?(subs(gamma)(eq`label))
```

Completeness

```
antiunif_compl_DecF: LEMMA
 FORALL (c:(validConfiguration?), (gamma : (nice?) | r_generalizer?(c)(gamma))) :
     (   match_DecF_conf?(c) AND
        FORALL (cp: (validConfiguration?)):
         FORALL ((gammaP: (nice?) | r_generalizer?(cp)(gammaP))):
          size(cp`unsolved) < size(c`unsolved) IMPLIES
           more_general?(gammaP, antiunify(cp)`substitution,
                         labels(cp`unsolved))  )
IMPLIES more_general?(gamma, antiunify(c)`substitution, labels(c`unsolved))
```

Completeness

Inference Rule	Proof size (# lines)	Dependencies (# lines)	Lemma	Proof size (# lines)
(DecF)	373	2237	matchingFuns_r_generalizer_classification	33
			DecF_MakeSub_r_gen_var_case	1146
			MakeSubs_preserves_fresh_vars	44
			antiunify_r_gen_members_img_vars_disjoint	74
			DecF_MakeSub_r_gen_app_case	593
			matchingFuns_r_generalizer_f_symbol	41
			DecF_MakeSub_nice_app_case	188
(DecP)	521	3133	matchingPairs_r_generalizer_classification	34
			DecP_MakeSub_r_gen_var_case	1945
			MakeSubs_preserves_fresh_vars	44
			antiunify_r_gen_members_img_vars_disjoint	74
			DecP_MakeSub_r_gen_pair_case	701
			DecP_MakeSub_nice_pair_case	217

Challenges

Some Remarks

Current

- Current proof strategy: Induction from final configuration to current configuration.
- The idea for proving completeness for DecF and DecP worked just fine;
- The same idea was not sufficient to prove completeness for Solve and Synt;
- Problem: We need access to the computational trace up until that point;
- **Idea**: Use the reflexive-transitive closure of single-step anti-unify and do induction on the number of steps, starting from an initial configuration.

Antiunify Derivation

```
DecF?(c, c_p: (validConfiguration?)): bool =
   match_DecF_conf?(c) AND DecF(c) = c_p

DecP?(c, c_p: (validConfiguration?)): bool =
   match_DecP_conf?(c) AND DecP(c) = c_p

Synt?(c, c_p: (validConfiguration?)): bool =
   match_Synt_conf?(c) AND Synt(c) = c_p

Solve?(c, c_p: (validConfiguration?)): bool =
   match_Sol_conf?(c) AND Solve(c) = c_p

Antiunify?(c, c_p: (validConfiguration?)): bool =
   DecF?(c, c_p) OR DecP?(c, c_p) OR Synt?(c, c_p) OR Solve?(c, c_p)
```

Antiunify Derivation

```
labels(c: (validConfiguration?)): finite_set[variable] =
   union(labels(append(c`unsolved, c`solved)), dom(c`substitution))

protected_variables(c: (validConfiguration?)): finite_set[variable] =
   union(vars(append(c`unsolved, c`solved)),
      difference(vars(img(c`substitution)),
         labels(append(c`unsolved, c`solved))))

left_substitution(c: (validConfiguration?)): (nice?) =
   build_subs_left(append(c`unsolved, c`solved))

right_substitution(c: (validConfiguration?)): (nice?) =
   build_subs_right(append(c`unsolved, c`solved))
```

Antiunify Derivation

```
antiunify_derivation_equivalence: THEOREM
   FORALL(c: (validConfiguration?)):
      RTC(Antiunify?)(c, antiunify(c))

antiunify_derivation_labels: COROLLARY
   FORALL(c: (validConfiguration?),
   c_p: (validConfiguration?) | RTC(Antiunify?)(c, c_p)):
      subset?(labels(c), labels(c_p))

antiunify_protected_variables: THEOREM
   FORALL(c: (validConfiguration?),
   c_p: (validConfiguration?) | RTC(Antiunify?)(c, c_p)):
      protected_variables(c_p) = protected_variables(c)
```

Antiunify Derivation

```
lateral_substitutions_computed_substitution_antiunify: THEOREM
   FORALL(c: (validConfiguration?), c_p: (validConfiguration?) |
      RTC(Antiunify?)(c, c_p), X: (dom(c`substitution))):
      subs(append(left_substitution(c_p), c_p`substitution))(X) =
         subs(append(left_substitution(c), c`substitution))(X) AND
            subs(append(right_substitution(c_p), c_p`substitution))(X) =
                  subs(append(right_substitution(c), c`substitution))(X)

antiunify_solved_history: LEMMA
   FORALL(c_i: (initial_configuration?), c: (validConfiguration?) |
   RTC(Antiunify?)(c_i, c), eq: AUT | member(eq, c`solved)):
      EXISTS(c_p: (match_Sol_conf?) | Intermediate?(c_i, c)(c_p)):
         eq = car(c_p`unsolved) AND
            NOT AUT_repeated_in?(eq, c_p`solved)

action_computed_substitution: THEOREM (see PVS)
```

Some Remarks

Future

- Finish the proof of completeness for `antiunify`;
- Extend `antiunify` to other theories (C, A, Abs) and their combinations;
- In particular: Extend `antiunify` to the absorptive case (Abs) [Ayala-Rincón et al., 2024].

Thank You!

References

M. Ayala-Rincón, T. A. de Lima, M. J. Dias Lima, M. M. Moscato, and T. Kutsia. Verification of an anti-unification algorithm in PVS. In A. Dutle, L. R. Humphrey, and L. Titolo, editors, *NASA Formal Methods - 17th International Symposium, NFM 2025, Williamsburg, VA, USA, June 11-13, 2025, Proceedings*, Lecture Notes in Computer Science, pages 54–71. Springer, 2025. doi: 10.1007/978-3-031-93706-4_4. URL https://doi.org/10.1007/978-3-031-93706-4_4.

Mauricio Ayala-Rincón, David M. Cerna, Andres Felipe Gonzalez Barragan, and Temur Kutsia. Equational anti-unification over absorption theories. In *Automated Reasoning - 12th Int. Joint Conference, IJCAR, Proc., Part II*, volume 14740 of *Lecture Notes in Computer Science*, pages 317–337. Springer, 2024. doi: 10.1007/978-3-031-63501-4_17. URL https://doi.org/10.1007/978-3-031-63501-4_17.

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA):159:1–159:27, 2019. URL `https://doi.org/10.1145/3360585`.

D. M. Cerna and T. Kutsia. Anti-unification and generalization: A survey. In *Proceedings of the 32nd Int. Joint Conference on Artificial Intelligence, IJCAI*, pages 6563–6573. ijcai.org, 2023. URL `https://doi.org/10.24963/ijcai.2023/736`.

S. Owre, J. M. Rushby, N. Shankar, and M. K. Srivas. A tutorial on using PVS for hardware verification. In R. Kumar and T. Kropf, editors, *Theorem Provers in Circuit Design*, pages 258–279, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-540-49177-4.

G. D. Plotkin. A note on inductive generalization. *Machine Intelligence 5*, 5:153–163, 1970.

Emily Rowan Winter, Vesna Nowack, David Bowes, Steve Counsell, Tracy Hall, Sæmundur Óskar Haraldsson, John R. Woodward, Serkan Kirbas, Etienne Windels, Olayori McBello, Abdurahman Atakishiyev, Kevin Kells, and Matthew W. Pagano. Towards developer-centered automatic program repair: findings from bloomberg. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 1578–1588. ACM, 2022. doi: 10.1145/3540250.3558953. URL https://doi.org/10.1145/3540250.3558953.