Available Projects

René Thiemann

May 2, 2023

Contents

1 Pattern-Completeness (2-3 persons)

Pattern-completeness is the question whether in a given functional program all constructor ground terms can be matched by some left-hand-side of a defining equation.

In this project you will verify an algorithm for deciding pattern-completeness on an abstract-level, refine it to an executable one, and prove termination of the latter.

The algorithm is a variant of the one presented in the "Program Verification" lecture, chapter 4. It is a simplified version: instead of using types there is just a fixed finite list of constructors; and instead of invoking an matching-algorithm, the variant in this theory integrates the matching algorithm. The latter deviation speeds up the execution time and simplifies the termination argument.

1.1 Preliminaries

We integrate some properties of the Archive of Formal Proofs, in particular first-order terms.

```
theory Project-Pattern-Completeness
  imports
    First-Order-Terms.Term
    Polynomial-Factorization.Missing-List
    Knuth-Bendix-Order.Term-Aux
    Decreasing−Diagrams−II.Decreasing-Diagrams-II
begin
```

A definition of being linear

```
fun linear-term :: ('f, 'v) term ⇒ bool where
  linear-term (Var -) = True
| linear-term (Fun - ts) = (is-partition (map vars-term ts) ∧ (∀ t∈set ts. linear-term t))
```

The linearity condition implies that several substitution $\tau\ i$ (for each argument $ts\ !\ i$) can be merged into a single substitution σ.

lemma *subst-merge*:
  assumes *part*: *is-partition* (*map vars-term ts*)
  shows $\exists\sigma.\ \forall i{<}length\ ts.\ \forall x{\in}vars\text{-}term\ (ts\ !\ i).\ \sigma\ x = \tau\ i\ x$
proof –
  let $?\tau = map\ \tau\ [0\ ..{<}\ length\ ts]$
  let $?\sigma = fun\text{-}merge\ ?\tau\ (map\ vars\text{-}term\ ts)$
  show *?thesis*
    apply (*rule exI*[*of* - $?\sigma$])
    apply (*intro allI impI ballI*)
    using *fun-merge-part*[*OF part*, *of* - - $?\tau$]
      by *auto*
qed

A measure to count the number of function symbols of the second argument that don't occur in the first argument

fun *fun-diff* :: $('f,'v)term \Rightarrow ('f,'w)term \Rightarrow nat$ **where**
  fun-diff t (*Var* -) = *0*
| *fun-diff* (*Var x*) *l* = *num-funs l*
| *fun-diff* (*Fun f ts*) (*Fun g ls*) = (*if f* = *g* ∧ *length ts* = *length ls then*
    sum-list (*map2 fun-diff ts ls*) *else 0*)

A pattern problem is a set of set of term pairs, or for the implementation it will be a list of list of term pairs. Note that in the term pairs the type of variables differ: Each left term has natural numbers as variables, so that it is easy to generate new variables, whereas each right term has arbitrary variables of type $'v$ without any further information.

type-synonym $('f,'v)pat\text{-}problem = (('f,nat)term \times ('f,'v)term)\ set\ set$
type-synonym $('f,'v)pat\text{-}problem\text{-}impl = (('f,nat)term \times ('f,'v)term)\ list\ list$

1.2 Task 1 – Soundness of the Abstract Inference System

In the sequel you find inference rules that describe a transformation of pattern problems. Fill in the missing inference rule that corresponds to a decomposition-rule in a matching-algorithm, and prove soundness of the abstract algorithm.

definition *linear-pat-problem* **where** *linear-pat-problem p* = (∀ *tl* ∈ *set p*. ∀ (*ti*,*pi*) ∈ *set tl*. *linear-term pi*)

context
  fixes *C* :: $('f \times nat)list$ — list of constructors with arities
    and *m* :: *nat* — upper bound on arities of constructors
assumes *SORT-CONSTRAINT*($'v$ :: *type*)
begin

A constructor-ground substitution for the fixed set of constructors

definition *cg-subst* :: ('*f*,*nat*,'*v*)*gsubst* ⇒ *bool* **where**
cg-subst σ = (∀ *x*. *vars-term* (σ *x*) = {} ∧ *funas-term* (σ *x*) ⊆ *set* *C*)

A definition of pattern completeness for linear pattern problems.

definition *pat-complete-linear* :: ('*f*,'*v*)*pat-problem* ⇒ *bool* **where**
pat-complete-linear *p* = (∀ σ :: ('*f*,*nat*,'*v*)*gsubst*. *cg-subst* σ ⟶ (∃ *tl* ∈ *p*. ∀ (*ti*,*li*) ∈ *tl*. ∃ μ. *ti* · σ = *li* · μ))

definition *subst-pat-problem* :: ('*f*,*nat*)*subst* ⇒ ('*f*,'*v*)*pat-problem* ⇒ ('*f*,'*v*)*pat-problem* **where**
subst-pat-problem τ *p* = (λ *tls*. (*map-prod* (λ *t*. *t* · τ) *id*) ‘ *tls*) ‘ *p*

Specify a function to compute for a variable x all substitution that instantiate x by $c(x_n, ..., x_{n+a})$ where c is an constructor of arity a and n is a parameter that determines from where to start the numbering of variables. Here, the function *subst* might be useful.

definition τ*s* :: *nat* ⇒ *nat* ⇒ ('*f*,*nat*)*subst* *list* **where**
τ*s* *x* *n* = *undefined*

Specify that a given set of numbered variables are disjoint from those that occur in a pattern problem. Note: the variables of a term can be computed via *vars-term*.

definition *tvars-disj-pp* :: *nat* *set* ⇒ ('*f*,'*v*)*pat-problem* ⇒ *bool* **where**
tvars-disj-pp *V* *p* = *undefined*

Fill in the missing parts in the decomposition rule.

inductive *pp-trans* :: ('*f*,'*v*)*pat-problem* *set* ⇒ ('*f*,'*v*)*pat-problem* *set* ⇒ *bool* **where**
pp-fail: *pp-trans* (*insert* {} *P*) {{}}
| *pp-match-solved*: *pp-trans* (*insert* (*insert* {} *p*) *P*) *P*
| *pp-match-by-var*: *pp-trans* (*insert* (*insert* (*insert* (*t*, *Var* *x*) *tls*) *p*) *P*) (*insert* (*insert* *tls* *p*) *P*)
| *pp-clash*: (*f*,*length* *ts*) ≠ (*g*,*length* *ls*) ⟹ *pp-trans* (*insert* (*insert* (*insert* (*Fun* *f* *ts*, *Fun* *g* *ls*) *tls*) *p*) *P*)
(*insert* *p* *P*)
| *pp-decomp*: (*f*,*length* *ts*) = (*g*,*length* *ls*) ⟹ *undefined* ''*further condition*'' ⟹ *pp-trans* (*insert* (*insert* (*insert* (*Fun* *f* *ts*, *Fun* *g* *ls*) *tls*) *p*) *P*)
(*undefined* ''*TODO*'')
| *pp-inst*: *tvars-disj-pp* {*n* ..< *n*+*m*} *p* ⟹ *pp-trans* (*insert* *p* *P*) (*set* (*map* (λ τ. *subst-pat-problem* τ *p*) (τ*s* *x* *n*)) ∪ *P*)

Fix a context which assumes that m is sufficiently large and that there is at least one constant constructor.

context
fixes *c* :: '*f*
assumes *c*: (*c*,*0*) ∈ *set* *C*
and *m-def*: *m* = *max-list* (*map* *snd* *C*)
begin

lemma *pp-trans*: *pp-trans* P P' $\Longrightarrow$ ($\forall$ $p \in P$. *pat-complete-linear* p) = ($\forall$ $p \in P'$. *pat-complete-linear* p)
proof (*induct* P P' *rule*: *pp-trans.induct*)
  case $*$: (*pp-clash* f *ts* g *ls* *tls* p P)
  show *?case* **sorry**
next
  case $*$: (*pp-decomp* f *ts* g *ls* *tls* p P)
  show *?case* **sorry**
next
  case $*$: (*pp-inst* n *pp* P x)
  show *?case* **sorry**
  — Further hints: there are useful things in the library such as substitution composition *subst-compose-def*, equality on terms: *term-subst-eq*, equality on lists: *nth-equalityI*.
At least one way to prove the result is considering both directions of the iff separately.
qed (*auto simp*: *pat-complete-linear-def*)

1.3 Task 2 – Termination of Implementation

The following algorithm implements the abstract inference system. Complete the definition for the decomposition rule and prove its termination via the already specified measure.

definition *subst-pat-problem-impl* :: ('*f*,*nat*)*subst* ⇒ ('*f*,'*v*)*pat-problem-impl* ⇒ ('*f*,'*v*)*pat-problem-impl*
where
  subst-pat-problem-impl τ p = *map* (*map* (*map-prod* (λ t. $t \cdot \tau$) *id*)) p

function *check-pat-complete* :: *nat* ⇒ ('*f*,'*v*)*pat-problem-impl list* ⇒ *bool* **where**
  check-pat-complete n [] = *True* — all pattern problems solved
| *check-pat-complete* n ([] # P) = *False* — no left-hand sides left
| *check-pat-complete* n (([] # *tls*) # P) = *check-pat-complete* n P — match-list empty
| *check-pat-complete* n ((((t,*Var* x) # *tls*) # *other*) # P) = *check-pat-complete* n ((*tls* # *other*) # P) — match by var
| *check-pat-complete* n ((((*Fun* f *ts*,*Fun* g *ls*) # *tls*) # *other*) # P) = (*if* $f = g \wedge$ *length ts* = *length ls*
    then check-pat-complete n *undefined* — decompose
    else check-pat-complete n (*other* # P)) — clash
| *check-pat-complete* n ((((*Var* x,*Fun* g *ls*) # *tls*) # *other*) # P) = *check-pat-complete* ($n + m$) — instantiate
    (*map* (λ τ. *subst-pat-problem-impl* τ (((*Var* x,*Fun* g *ls*) # *tls*) # *other*)) (τs x n) @ P)
  by *pat-completeness auto*

you might want to derive some additional lemmas on when two elements are in the multiset-relation which correspond to the applications in the termination proof, e.g. if you replace one element x by several smaller ones

lemma *add-many-mult*: (⋀ *y*. *y* ∈# *N* ⟹ (*y*,*x*) ∈ *R*) ⟹ (*N* + *M*, *add-mset x M*) ∈ *mult R*
  sorry

For the termination, we use a lexicographic combination: First, the multiset of function-symbol-differences is computed; second the size of terms in the right-hand sides of the pairs is measured.

termination
proof −
  define *rel1-inner* **where** *rel1-inner* = *size-list* (λ*xs*. ∑(*t* :: (′*f*,*nat*)*term*, *l* :: (′*f*,′*v*)*term*)←*xs*. *fun-diff t l*)
  define *rel2* :: ((′*f*,′*v*)*pat-problem-impl list*)*rel* **where** *rel2* = *measure* (*size-list* (*size-list* (*size-list* (*size o snd*))))
  define *rel1* **where** *rel1* = *mult* {(*x*, *y* :: *nat*). *x* < *y*}
  let *?R* = *inv-image* (*rel1* <*lex*> *rel2*) (λ (*n*,*p*). (*mset* (*map rel1-inner p*),*p*)) :: (*nat* × (′*f*,′*v*)*pat-problem-impl list*)*rel*
  have *wf*: *wf ?R* **unfolding** *rel2-def rel1-def* **by** (*auto intro*: *wf-mult wf-less*)
  note *defs* = *rel1-inner-def rel1-def rel2-def*
  show *?thesis*
  proof (*standard*, *rule wf*, *goal-cases*)
    case (*1 n tls P*)
    show *?case* **sorry**
  next
    case (*2 n t x tls other P*)
      show *?case* **unfolding** *defs* **by** (*auto simp*: *termination-simp o-def intro*: *mult-singleton*)
  next
    case (*3 n f ts g ls tls other P*)
    show *?case* **sorry**
  next
    case (*4 n f ts g ls tls other P*)
    show *?case* **unfolding** *defs* **by** *simp*
  next
    case (*5 n x g ls tls other P*)
    show *?case* **unfolding** *defs in-inv-image split in-lex-prod*
      apply (*rule disjI1*)
      apply *simp*
      apply (*rule add-many-mult*)
      apply *clarsimp*
    proof *goal-cases*
      case (*1 τ*)
      thus *?case* **sorry**

      thm *sum-list-mono*
      thm *sum-list-mono2*
      thm *size-list-pointwise*

    qed
  qed

qed

1.4 Task 3 – Prove that the algorithm implements the abstract inference system.

definition *pp-of-impl* :: $('f,'v)$*pat-problem-impl* ⇒ $('f,'v)$*pat-problem* **where**
  pp-of-impl p = *set* ‘ *set p*

abbreviation *pat-complete-linear-impl* ≡ (λ *p*. *pat-complete-linear* (*pp-of-impl p*))

This is the a nice easy lemma to perform the upcoming proof: to show that we can switch in the implementation from one state to another, we just apply the corresponding abstract inference rule via *local.pp-trans*.

lemma *pp-trans-impl*: *pp-trans P P′* ⟹ *pp-of-impl* ‘ *set PI* = *P* ⟹ *pp-of-impl* ‘ *set PI′* = *P′* ⟹
  Ball (*set PI′*) *pat-complete-linear-impl* = *Ball* (*set PI*) *pat-complete-linear-impl*

  using *pp-trans*[*of P P′*] **by** *auto*

lemma *pp-of-impl-subst*[*simp*]: *pp-of-impl* (*subst-pat-problem-impl* τ *p*) = *subst-pat-problem* τ (*pp-of-impl p*)
  sorry

In the lemma we require linearity of the pattern problem and we also need a condition that the parameter n is chosen correctly, so that all variables will be fresh enough.

lemma *check-pat-complete-linear-impl*: **assumes** *Ball* (*set P*) *linear-pat-problem*
  and *undefined ″Condition on n being fresh for P″*
  shows *check-pat-complete n P* = *Ball* (*set P*) *pat-complete-linear-impl*
proof –
  note *def* = *pp-of-impl-def linear-pat-problem-def*
  show *?thesis* **using** *assms*
  proof (*induction P rule*: *check-pat-complete.induct*)
    case *1*
    show *?case* **sorry**
  next
    case (*2 n P*)
    show *?case* **sorry**
  next
    case (*3 n tls P*)
   from *3* **have** *IH*: *check-pat-complete n P* = (*Ball* (*set P*) *pat-complete-linear-impl*)

      by *auto*
    show *?case* **unfolding** *check-pat-complete.simps IH*
      by (*rule pp-trans-impl*[*OF pp-match-solved - refl*], *auto simp*: *def*)
  next
    case (*4 n t x tls other P*)

```
    show ?case sorry
  next
    case (5 n f ts g ls tls other P)
    show ?case
    proof (cases f = g ∧ length ts = length ls)
      case False
      show ?thesis sorry
    next
      case True
      show ?thesis sorry
      thm set-zip
      thm in-set-zipE
    qed
  next
    case (6 n x g ls tls other P)
    define pp where pp = ((Var x, Fun g ls) # tls) # other
    show ?case sorry
    thm max-list
    thm set-map
    thm vars-term-subst
  qed
qed

end
end
end
```

2 Congruence Closure (2-3 persons)

We consider a set ground equations GE such as

- f(g(a)) = h(b)
- f(b) = b
- g(a) = b

  and are interested in the question whether a particular equation is implied GE. For instance the sequence of equality-steps

- f(h(b)) = f(f(g(a))) = f(f(b)) = f(b)

  proves that f(h(b)) = f(b) follows from E.

  Whereas it is easy to validate a given sequence of equality-steps, the problem is to detect whether such a sequence exists for a given equation. To this end, the congruence closure algorithm has been developed which should be partially verified in this project.

Basic knowledge of term rewriting is helpful for this project. The describtion of the algorithm is based on *Franz Baader and Tobias Nipkow*, *Term Rewriting and All That*, *Chapter 4.3*.

theory *Project-Congruence-Closure*
  imports
    Main
begin

2.1 Definition of Algorithm

We start by definining ground terms where the type of symbols are just strings.

type-synonym *symbol* = *string*

datatype *trm* = *Fun symbol trm list*

type-synonym *eqs* = (*trm* × *trm*)*set*

Define the set of subterms of a term, e.g., the subterms of f(g(a),b) would be $\{f(g(a),b),\ g(a),\ a,\ b\}$.

fun *subt* :: *trm* ⇒ *trm set* **where**
  subt (*Fun f ts*) = *undefined*

Prove two useful lemmas about subterms.

lemma *self-subt*: *u* ∈ *subt u* **sorry**

lemma *subt-trans*: *s* ∈ *subt t* ⟹ *t* ∈ *subt u* ⟹ *s* ∈ *subt u* **sorry**

For a set of ground-equalities, the congruence closure algorithm is in particular interested in all subterms that occur in the equalities.

definition *subt-eqs* **where** *subt-eqs GE* = ⋃ ((λ (*l*,*r*). *subt l* ∪ *subt r*) ‘ *GE*)

From now on fix a specific set of ground-equalities GE.

context
  fixes *GE* :: *eqs*
begin

Define an equality step where one can either replace one side of an equation in GE by the other side (a root-step), or where one can apply a step in a context.

inductive-set *estep* :: *trm rel* **where**
  root: *undefined* ⟹ *undefined* ∈ *estep*
| *ctxt*: (*s*,*t*) ∈ *estep* ⟹ (*Fun f* (*before* @ *s* # *after*), *Fun f* (*before* @ *t* # *after*)) ∈ *estep*

The other important definition is the Cong-operation which given a set of equalities derives new equalities of these by reflexivity, symmetry, transitivity or context.

inductive-set *Cong* :: *eqs* ⇒ *eqs* **for** *E* **where**
C-keep: $eq \in E \Longrightarrow eq \in Cong\ E$
| *C-refl*: $(t,t) \in Cong\ E$
| *C-sym*: $(s,t) \in E \Longrightarrow (t,s) \in Cong\ E$
| *C-trans*: $(s,t) \in E \Longrightarrow (t,u) \in E \Longrightarrow (s,u) \in Cong\ E$
| *C-cong*: $length\ ss = length\ ts \Longrightarrow (\forall\ i < length\ ts.\ (ss\ !\ i,\ ts\ !\ i) \in E) \Longrightarrow (Fun\ f\ ss,\ Fun\ f\ ts) \in Cong\ E$

Let us now fix to terms s and t where we are interested in whether GE implies s = t.

context
fixes *s t* :: *trm*
begin

In the congruence closure algorithm one only is interested in equalities of terms in S.

definition *S* **where** $S = subt\ s \cup subt\ t \cup subt\text{-}eqs\ GE$

definition *CongS* **where** $CongS\ E = Cong\ E \cap (S \times S)$

CCA defines the equalities that are obtained in the i-th iteration of the congruence closure algorithm, which iteratively applies the *local.CongS* operation starting from *GE*.

definition *CCA* **where** $CCA\ i = (CongS\ \widehat{}\widehat{}\ i)\ GE$

Prove the following simple inclusions.

lemma *GE-S*: $GE \subseteq S \times S$ **sorry**

lemma *GE-CCA*: $GE \subseteq CCA\ i$ **sorry**

2.2 Completeness of CCA

The crucial result of the congruence closure algorithm is given in the following lemma on the completeness of the algorithm: if the algorithm has stabilized in the i-th iteration, then all equations in *local.S* × *local.S* that can be derived with arbitrary many steps are also contained in the equalities of CCA.

lemma *esteps-imp-CCA*: **assumes** $CongS\ (CCA\ i) = CCA\ i$
shows $(u,v) \in estep\widehat{}* \cap (S \times S) \longrightarrow (u,v) \in CCA\ i$
proof

The proof is by induction on the number of steps and then by the size of the starting term *u*. This is expressed as follows in Isabelle.

assume (*u*,*v*) ∈ *estep*^* ∩ (*S* × *S*)
then obtain *n* **where** ∗: *u* ∈ *S* *v* ∈ *S* (*u*,*v*) ∈ *estep*^^*n*
by (*auto simp*: *rtrancl-power*)
obtain *m* **where** *m* = (*n*,*size u*) **by** *auto*
with ∗ **show** (*u*,*v*) ∈ *CCA i*
proof (*induction m arbitrary*: *u v n rule*: *wf-induct*[*OF wf-measures*[*of* [*fst*,*snd*]]])
case (*1 m u v n*)

For handling the induction, we first convert the derivation into a function which gives us all intermediate terms via function w.

from *1*(*4*)[*unfolded relpow-fun-conv*] **obtain** *w*
where *w*: *w 0* = *u* *w n* = *v* (∀ *i*<*n*. (*w i*, *w* (*Suc i*)) ∈ *estep*) **by** *auto*

And the proof now proceeds by case-analysis on whether any of these steps was a root step or whether all steps are non-root.

show *?case* **sorry**
qed
qed

Next, completeness of CCA is easily established

lemma *esteps-imp-CCA-st*: **assumes** *CongS* (*CCA i*) = *CCA i*
shows (*s*,*t*) ∈ *estep*^* ⟶ (*s*,*t*) ∈ *CCA i*
sorry

2.3 Soundness of CCA

The crucial step to prove soundness is the following lemma, which might require some further auxiliary lemmas.

lemma *Cong-esteps*: *E* ⊆ *estep*^* ⟹ *Cong E* ⊆ *estep*^* **sorry**

But you can easily verify that *?E* ⊆ $estep^*$ ⟹ *Cong ?E* ⊆ $estep^*$ is the key to prove soundness of CCA.

lemma *CCA-imp-esteps*: *CCA i* ⊆ *estep*^* **sorry**

2.4 Correctness of CCA

Having soundness and completeness, correctness is simple.

theorem *congruence-closure-correct*: **assumes** *CongS* (*CCA i*) = *CCA i*
shows (*s*,*t*) ∈ *estep*^* ⟷ (*s*, *t*) ∈ *CCA i*
sorry

2.5 Termination of CCA

The precondition *local.CongS* (*local.CCA i*) = *local.CCA i* can be discharged proving termination of the congruence closure algorithm which just computes the least i such that the precondition is satisfied. The existence of such an i follows from the fact that CCA i is increasing with increasing i and CCA i is bounded by the finite set of terms S x S, assuming finiteness of GE.

Formulating and proving these facts in Isabelle is another task of this project, if it is conducted as a 3-person project.

context
  assumes *finite GE*
begin

lemma *i-exists*: ∃ *i. CongS (CCA i) = CCA i* **sorry**

definition *fixpointI* = (*LEAST i. CongS (CCA i) = CCA i*)

lemma *fixpointI*: *CongS (CCA fixpointI) = CCA fixpointI*
  sorry

Design an algorithm to compute *local.fixpointI* and prove its termination. The algorithm itself of course must not use *local.fixpointI*, but the measure for proving termination might very well depend on this unknown constant.

end
end
end
end

3 Tseitin Transformation (2 persons)

Since most SAT solvers insist on formulas in conjunctive normal form (CNF) as input, but in general the CNF of a given formula may be exponentially larger, there is interest in efficient transformations that produce a small equisatisfiable CNF for a given formula. Probably the earliest and most well-known of these transformation is due to Tseitin.

In this project you will implement a two-step transformation of propositional formulas into equisatisfiable CNFs and formally prove results about the complexity and that the resulting CNFs are indeed equisatisfiable to the original formula.

theory *Project-Tseitin-Fresh*
  imports *Main*
begin

3.1 Syntax and Semantics

For the purposes of this project propositional formulas (with atoms of an arbitrary type) are restricted to the following (functionally complete) connectives:

datatype $'a$ *form* =
  Bot — the "always false" formula
  | *Top* — the "always true" formula
  | *Var* $'a$ — propositional variables
  | *Neg* $'a$ *form* — negation
  | *Disj* $'a$ *form* $'a$ *form* — disjunction
  | *Conj* $'a$ *form* $'a$ *form* — conjunction

Define a function *eval* that evaluates the truth value of a formula with respect to a given truth assignment $\alpha :: 'a \Rightarrow bool$.

fun *eval* :: $('a \Rightarrow bool) \Rightarrow 'a\ form \Rightarrow bool$
 where
  eval $\alpha\ \varphi$ = *undefined*

Define a predicate *sat* that captures satisfiable formulas.

definition *sat* :: $'a\ form \Rightarrow bool$
 where
  sat $\varphi \longleftrightarrow$ *undefined*

3.2 Conjunctive Normal Forms

Literals are positive or negative variables.

datatype $'a$ *literal* = $P\ 'a \mid N\ 'a$

A clause is a disjunction of literals, represented as a list of literals.

type-synonym $'a$ *clause* = $'a$ *literal list*

A CNF is a conjunction of clauses, represented as list of clauses.

type-synonym $'a$ *cnf* = $'a$ *clause list*

Implement a function *of-cnf* that, given a CNF (of $'a$ *cnf*, computes a logically equivalent formula (of $'a$ *form*).

fun *of-cnf* :: $'a\ cnf \Rightarrow 'a\ form$
 where
  of-cnf cs = *undefined*

3.3 Tseitin Transformation

The idea of Tseitin's transformation is to assign to each subformula φ a label a_φ and use the following definitions

- $a_\bot \longleftrightarrow \bot$
- $a_\top \longleftrightarrow \top$
- $a_{\neg\varphi} \longleftrightarrow \neg\ \varphi$

- $a_{\varphi \vee \psi} \longleftrightarrow (\varphi \vee \psi)$
- $a_{\varphi \wedge \psi} \longleftrightarrow (\varphi \wedge \psi)$

  to recursively compute clauses *tseitin* φ such that $a_\varphi \wedge$ *tseitin* φ and φ are equisatisfiable (that is, the former is satisfiable iff the latter is).

  Define a function *tseitin* that computes the clauses corresponding to the above idea.

```
fun tseitin :: 'a form ⇒ ('a form) cnf
  where
    tseitin φ = undefined
```

Prove that $a_\varphi \wedge$ *tseitin* φ are equisatisfiable.

```
lemma tseitin-equisat:
  sat (of-cnf ([P φ] # tseitin φ)) ⟷ sat φ
  sorry
```

Prove linear bounds on the number of clauses and literals by suitably replacing *n* and *num-literals* below:

```
lemma tseitin-num-clauses:
  length (tseitin φ) ≤ n * size φ
  sorry

lemma tseitin-num-literals:
  num-literals (tseitin φ) ≤ n * size φ
  sorry
```

3.4 Fresh Variables

One of the problems in the tseitin transformation above is that the type of propositional variables is changed from *'a* to *'a form*.

Define a function to rename variables in a CNF.

```
fun rename-cnf :: ('a ⇒ 'b) ⇒ 'a cnf ⇒ 'b cnf
  where
    rename-cnf f cs = undefined
```

Think of a property such that renaming preserves satisfiability. Note that injectivity is already defined in Isabelle (*inj* or *inj-on*.)

```
lemma property f cs ⟹ sat (of-cnf (rename-cnf f cs)) ⟷ sat (of-cnf cs) sorry
```

Next, we define a tseitin transformation which does not change the type of propositional variables.

```
definition tseitin-fresh :: 'your-type form ⇒ 'your-type cnf where
  tseitin-fresh φ = (let
      cs = [P φ] # tseitin φ;
```

renaming = *undefined*
in rename-cnf renaming cs)

Implement a corresponding renaming function such that the following soundness property can be proved. Here, you also need to change the type-variable *'your-type*, where for this project it is perfectly fine to use a concrete type which has infinitely many elements, e.g., *nat* or *int* or *string*.

lemma *tseitin-fresh*: *sat* $\varphi \longleftrightarrow$ *sat* (*of-cnf* (*tseitin-fresh* φ)) **sorry**

Your function definitions should be executable.

definition *X* :: *'your-type* **where** *X* = *undefined*
definition *Y* :: *'your-type* **where** *Y* = *undefined*
definition *Z* :: *'your-type* **where** *Z* = *undefined*

definition *test-form* :: *'your-type form* **where**
test-form = *Neg* (*Conj* (*Disj* (*Neg* (*Var X*)) (*Var Z*)) (*Neg* (*Var Y*)))

The Isabelle command *value* (*code*) *tseitin-fresh test-form* should succeed.

end

4 A Compiler for the Register Machine from Hell (2 persons)

Processors from Hell has released its next-generation RISC processor RMfH. It features an infinite bank of registers R_0, R_1, ... holding unbounded integers. Register R_0 plays the role of the accumulator and is the implicit source or destination register of all instructions. Any other register involved in an instruction must be distinct from R_0, which is enforced by implicitly incrementing its index.
There are five instructions

LDI i has the effect $R_0 := i$

LD n has the effect $R_0 := R_{n+1}$

ST n has the effect $R_{n+1} := R_0$

ADD n has the effect $R_0 := R_0 + R_{n+1}$

MUL n has the effect $R_0 := R_0 * R_{n+1}$

were i is an integer and n a natural number.
In this project you will implement and verify a compiler for the Register Machine from Hell (RMfH).

(Adapted from https://isabelle.in.tum.de/exercises/advanced/regmachine/ex.pdf)

theory *Project-Register-Machine-from-Hell*
imports *Main*
begin

Define a data type of instructions and an execution function *exec* that takes an instruction and a state and returns the new state.

type-synonym *state* = *nat* ⇒ *int*
datatype *instr* = *Undefined*

fun *exec* :: *instr* ⇒ *state* ⇒ *state*
where
exec i s = *undefined*

Extend *exec* to lists of instructions:

fun *execute* :: *instr list* ⇒ *state* ⇒ *state*
where
execute is s = *undefined*

The engineers of *PfH* soon got tired of writing assembly language code and designed their own high-level programming language of arithmetic expressions. An expression can be

- an integer constant,
- one of the variables v_0, v_1, . . . , or
- the sum of two expressions
- the product of two expressions
- the difference of two expressions
- exponentiation of an expression with a fixed exponent, i.e., a natural number constant

Define a data type of expressions and an evaluation function that takes an expression and a state and returns the resulting value. Because this is a clean language, there is no implicit increment going on: the value of v_n in state s is simply $s\ n$.

datatype *expr* = *Undefined*

fun *value* :: *expr* ⇒ *state* ⇒ *int*
where
value e s = *undefined*

4.1 A Compiler

You have been recruited to write a compiler from *expr* to *instr list*. You remember your compiler course and decide to emulate a stack machine using free registers, that is, registers not used by the expression you are compiling.

Implement a compiler *compile* :: *expr* ⇒ *nat* ⇒ *instr list* where the second argument is the index of the first free register that can be used to store intermediate results. The result of an expression should be returned in R_0.

Because R_0 is the accumulator, you decide on the following compilation scheme: v_i will be held in R_{i+1}.

Hint: perhaps you first treat a simplified version of expressions without the difference- and exponentiation-operations, since these operations are not directly supported by the RMfH architecture.

Challenge: Can you do better than compiling exponentation x^n into $O(n)$ multiplications?

fun *compile* :: *expr* ⇒ *nat* ⇒ *instr list*
  where
    compile e k = *undefined*

4.2 Compiler Verification

Although you are convinced about the correctness of your compiler, the boss of *PfH* (which coincides with the lecturer of interactive theorem proving) actually wants you to verify the compiler. Below is a sketch of the correctness statement.

However, there is definitely a precondition missing because *k* should be large enough not to interfere with any of the variables in *e*. Moreover, you have some lingering doubts about having the same *s* on both sides despite the index shift between variables and registers. But because all your definitions are executable, you hope that Isabelle will spot any incorrect propositions before you even start its proofs. What worries you most is the number of auxiliary lemmas it may take to prove your proposition.

lemma
  execute (*compile e k*) *s 0* = *value e s*
  sorry

end

5 Propositional Logic (2 persons)

Soundness and completeness of a logic establish that the syntactic notion of provability is equivalent to the semantic notation of logical entailment.

In this project you will formally prove soundness and completeness of a specific set of natural deduction rules for propositional logic.

theory *Project-Logic*
  imports *Main*
begin

5.1 Syntax and Semantics

Propositional formulas are defined by the following data type (that comes with some syntactic sugar):

type-synonym *id* = *string*
datatype *form* =
    Atom id
  | *Bot* ($\bot_p$)
  | *Neg form* ($\neg_p$ - [*68*] *68*)
  | *Conj form form* (**infixr** $\wedge_p$ *67*)
  | *Disj form form* (**infixr** $\vee_p$ *67*)
  | *Impl form form* (**infixr** $\rightarrow_p$ *66*)

Define a function *eval* that evaluates the truth value of a formula with respect to a given truth assignment.

fun *eval* :: $(id \Rightarrow bool) \Rightarrow form \Rightarrow bool$
  where
    $eval\ v\ \varphi \longleftrightarrow undefined$

Using *eval*, define semantic entailment of a formula from a list of formulas.

definition *entails* :: $form\ list \Rightarrow form \Rightarrow bool$ (**infix** $\models$ *51*)
  where
    $\Gamma \models \varphi \longleftrightarrow undefined$

5.2 Natural Deduction

The natural deduction rules we consider are captured by the following inductive predicate *proves* $P\ \varphi$, with infix syntax $P \vdash \varphi$, that holds whenever a formula φ is provable from a list of premises P.

inductive *proves* (**infix** $\vdash$ *58*)
  where
    premise: $\varphi \in set\ P \Longrightarrow P \vdash \varphi$
  | *conjI*: $P \vdash \varphi \Longrightarrow P \vdash \psi \Longrightarrow P \vdash \varphi \wedge_p \psi$
  | *conjE1*: $P \vdash \varphi \wedge_p \psi \Longrightarrow P \vdash \varphi$
  | *conjE2*: $P \vdash \varphi \wedge_p \psi \Longrightarrow P \vdash \psi$
  | *impI*: $\varphi \# P \vdash \psi \Longrightarrow P \vdash (\varphi \rightarrow_p \psi)$
  | *impE*: $P \vdash \varphi \Longrightarrow P \vdash \varphi \rightarrow_p \psi \Longrightarrow P \vdash \psi$
  | *disjI1*: $P \vdash \varphi \Longrightarrow P \vdash \varphi \vee_p \psi$
  | *disjI2*: $P \vdash \psi \Longrightarrow P \vdash \varphi \vee_p \psi$
  | *disjE*: $P \vdash \varphi \vee_p \psi \Longrightarrow \varphi \# P \vdash \chi \Longrightarrow \psi \# P \vdash \chi \Longrightarrow P \vdash \chi$

| *negI*: $\varphi \# P \vdash \bot_p \Longrightarrow P \vdash \neg_p \varphi$
| *negE*: $P \vdash \varphi \Longrightarrow P \vdash \neg_p \varphi \Longrightarrow P \vdash \bot_p$
| *botE*: $P \vdash \bot_p \Longrightarrow P \vdash \varphi$
| *dnegE*: $P \vdash \neg_p \neg_p \varphi \Longrightarrow P \vdash \varphi$

Prove that $\vdash$ is monotone with respect to premises, that is, we can arbitrarily extend the list of premises in a valid prove.

lemma *proves-mono*:
assumes $P \vdash \varphi$ **and** $set\ P \subseteq set\ Q$
shows $Q \vdash \varphi$
sorry

Prove the following derived natural deduction rules that might be useful later on:

lemma *dnegI*:
assumes $P \vdash \varphi$
shows $P \vdash \neg_p \neg_p \varphi$
sorry

lemma *pbc*:
assumes $\neg_p \varphi \# P \vdash \bot_p$
shows $P \vdash \varphi$
sorry

lemma *lem*:
$P \vdash \varphi \vee_p \neg_p \varphi$
sorry

lemma *neg-conj*:
assumes $\chi \in \{\varphi, \psi\}$ **and** $P \vdash \neg_p \chi$
shows $P \vdash \neg_p (\varphi \wedge_p \psi)$
sorry

lemma *neg-disj*:
assumes $P \vdash \neg_p \varphi$ **and** $P \vdash \neg_p \psi$
shows $P \vdash \neg_p (\varphi \vee_p \psi)$
sorry

lemma *trivial-imp*:
assumes $P \vdash \psi$
shows $P \vdash \varphi \rightarrow_p \psi$
sorry

lemma *vacuous-imp*:
assumes $P \vdash \neg_p \varphi$
shows $P \vdash \varphi \rightarrow_p \psi$
sorry

lemma *neg-imp*:

assumes $P \vdash \varphi$ **and** $P \vdash \neg_p\ \psi$
shows $P \vdash \neg_p\ (\varphi \rightarrow_p \psi)$
sorry

5.3 Soundness

Prove soundness of $\vdash$ with respect to $\models$.

lemma *proves-sound*:
assumes $P \vdash \varphi$
shows $P \models \varphi$
sorry

5.4 Completeness

Prove completeness of $\vdash$ with respect to $\models$ in absence of premises.

lemma *prove-complete-Nil*:
assumes $[] \models \varphi$
shows $[] \vdash \varphi$
sorry

Now extend the above result to also incorporate premises.

lemma *proves-complete*:
assumes $P \models \varphi$
shows $P \vdash \varphi$
sorry

Conclude that semantic entailment is equivalent to provability.

lemma *entails-proves-conv*:
$P \models \varphi \longleftrightarrow P \vdash \varphi$
sorry

end

6 BIGNAT - Natural Numbers of Arbitrary Size (1 person)

Hardware platforms have a limit on the largest number they can represent. This is usually fixed by the bit lengths of registers and ALUs used.

In order to be able to perform calculations that require arbitrarily large numbers, the provided arithmetic operations need to be extended in order for them to work on an abstract data type representing numbers of arbitrary size.

In this project you will build and verify an implementation for BIGNAT, an abstract data type representing natural numbers of arbitrary size.

(Adapted from http://isabelle.in.tum.de/exercises/proj/bignat/ex.pdf)

theory *Project-BIGNAT*
  imports *Main*
begin

6.1 Representation

A BIGNAT is represented by a list of natural numbers in a range supported by the target machine. In our case, this will be all natural numbers smaller than a given base b.

Note: Natural numbers in Isabelle are of arbitrary size.

type-synonym *bignat* = *nat list*

Define a function *valid* that takes a base and checks if a given BIGNAT is valid.

fun *valid* :: *nat* ⇒ *bignat* ⇒ *bool*
  where
    valid b n = *undefined*

Define a function *val* that takes a BIGNAT and its corresponding base, and returns the natural number represented by the BIGNAT.

fun *val* :: *nat* ⇒ *bignat* ⇒ *nat*
  where
    val b n = *undefined*

6.2 Addition

Define a function *add* that adds two BIGNATs with the same base. Make sure that your algorithm preserves the validity of the BIGNAT representation.

fun *add* :: *nat* ⇒ *bignat* ⇒ *bignat* ⇒ *bignat*
  where
    add b m n = *undefined*

Using *val*, verify formally that your *add* function computes the sum of two BIGNATs correctly.

lemma *val-add*: *val b* (*add b m n*) = *val b m* + *val b n*
  sorry

Using *valid*, verify formally that your function *add* preserves the validity of the BIGNAT representation.

lemma *valid-add*:
  assumes *valid b m* **and** *valid b n*
  shows *valid b* (*add b m n*)
  sorry

6.3 Multiplication

Define a function *mult* that multiplies two BIGNATs with the same base. You may use *add*, but not so often as to make the solution trivial. Make sure that your algorithm preserves the validity of the BIGNAT representation.

fun *mult* :: *nat* ⇒ *bignat* ⇒ *bignat* ⇒ *bignat*
  where
    mult b m n = *undefined*

Using *val*, verify formally that your *mult* function computes the product of two BIGNATs correctly.

lemma *val-mult*: *val b* (*mult b m n*) = *val b m* ∗ *val b n*
  sorry

Using *valid*, verify formally that your *mult* function preserves the validity of the BIGNAT representation.

lemma *valid-mult*:
  assumes *valid b m* **and** *valid b n*
  shows *valid b* (*mult b m n*)
  sorry

end

7 The Euclidean Algorithm - Inductively (1 person)

In this project you will develop and verify an inductive specification of the Euclidean algorithm.

(Adapted from http://isabelle.in.tum.de/exercises/proj/euclid/ex.pdf)

theory *Project-GCD*
  imports *Main*
begin

Define the set *gcd* of triples (*a*,*b*,*g*) such that *g* is the greatest common divisor of *a* and *b* inductively.

Your definition should closely follow the Euclidean algorithm, which repeatedly subtracts the smaller from the larger number, until one of them is zero (at this point, the other number is the greatest common divisor).

inductive-set *gcd* :: (*nat* × *nat* × *nat*) *set*

Show that the greatest common divisor as given by *gcd* is indeed a divisor.

lemma *gcd-divides*: (*a*, *b*, *g*) ∈ *gcd* ⟹ *g dvd a* ∧ *g dvd b*
  sorry

7.1 Soundness

Show that the greatest common divisor as given by *gcd* is greater than or equal to any other common divisor.

lemma *gcd-greatest*:
  assumes $(a,\ b,\ g) \in gcd$
    and $0 < a \lor 0 < b$
    and $d\ dvd\ a$
    and $d\ dvd\ b$
  shows $d \leq g$
  sorry

7.2 Completeness

So far, you have only shown that *gcd* is correct, but there might still be values a and b such that there is no g with $(a,b,g) \in gcd$.

Thus, show completeness of your specification. First prove the following result by course-of-value recursion, that is, using $(\bigwedge n.\ \forall m{<}n.\ ?P\ m \Longrightarrow ?P\ n) \Longrightarrow ?P\ ?n$. (Inside the induction make a case analysis corresponding to the different clauses of the algorithm.)

lemma *gcd-defined-aux*:
  $a + b \leq n \Longrightarrow \exists g.\ (a,\ b,\ g) \in gcd$
  sorry

lemma *gcd-defined*: $\exists g.\ (a,\ b,\ g) \in gcd$
  sorry

7.3 Uniqueness

Show that the gcd is uniquely determined.

lemma *gcd-unique*: $(a,b,g) \in gcd \Longrightarrow (a,b,g') \in gcd \Longrightarrow g = g'$
  sorry

7.4 Code

Finally use the above results to generate code for computing gcds.

Gcd as function.

definition $Gcd :: nat \Rightarrow nat \Rightarrow nat$ **where**
  $Gcd\ a\ b = (THE\ g.\ (a,b,g) \in gcd)$

lemma *gcd-to-Gcd*: $(a,b,g) \in gcd \Longrightarrow Gcd\ a\ b = g$
  sorry

lemma *Gcd-to-gcd*: $(a,\ b,\ Gcd\ a\ b) \in gcd$
  sorry

lemma *Gcd-code*[*code*]:
  Gcd a b = *undefined ''some recursive equation''*
  sorry

This value-command should succeed.

Congratulations, you have just defined the recursive Gcd-function without using the function package.

end