Available Projects

René Thiemann

April 26, 2022

Contents

1 Matching First-Order Terms (1 person)

A matching algorithm for first-order terms computes tries to compute a substitution that witnesses that one (list of) term(s) is an instance of another. In this project you will implement and verify a matching algorithm.

theory *Project-Matching*
  imports *Main*
begin

1.1 Terms and Substitutions

First-order terms are either variables or function symbols applied to a list of argument terms:

type-synonym *id* = *string*
datatype *term* = *Var id* | *Fun id term list*

A substitution is a mapping from variables to terms.

type-synonym *subst* = *id* ⇒ *term option*

Define a function *subst* that applies a substitution to a given term. Variables that are not part of the substitution should be left untouched.

fun *subst* :: *subst* ⇒ *term* ⇒ *term*
  where
    subst σ *t* = *undefined*

Define a function *vars* that computes the set of variables occurring in a given term.

fun *vars* :: *term* ⇒ *id set*
  where
    vars t = *undefined*

Define a matching algorithm that, given two lists of terms *ts* and *us*, and an initial substitution σ, computes (if possible) a substitution τ that makes the *ts* equal to *us*:

map (*subst* τ) *ts* = *us*

fun *match* :: *term list* ⇒ *term list* ⇒ *subst* ⇒ *subst option*
  where
    match ts us σ = *undefined*

Show that the domain *dom* of a substitution computed by *match* is contained in the union of the domain of the initial substitution σ with the variables occurring in *ts*.

lemma *match-Some-dom*:
  assumes $match\ ts\ us\ \sigma = Some\ \tau$
  shows $dom\ \tau \subseteq dom\ \sigma \cup \bigcup (set\ (map\ vars\ ts))$
  sorry

1.2 Soundness

Prove that *match* is sound.

You will need an auxiliary result on the relationship between the initial substitution σ and the resulting substitution τ. To this end the order $(\subseteq_m)$ should be useful.

lemma *match-sound*:
  assumes $match\ ts\ us\ \sigma = Some\ \tau$
  shows $map\ (subst\ \tau)\ ts = us$
  sorry

1.3 Matchers

Define the set of matchers of two given lists of terms *ts* and *us*.

Here, a matcher for *ts* and *us* is any substitution σ that makes *ts* equal to *us* and is defined at least on all variables of *ts*.

definition $matchers :: term\ list \Rightarrow term\ list \Rightarrow subst\ set$
  where
    $matchers\ ts\ us = undefined$

Prove the following equations for *matchers* (after suitably replacing *whatever*).

lemma *matchers-simp* [*simp*]:
  $matchers\ []\ (u\ \#\ us) = whatever$
  $matchers\ (t\ \#\ ts)\ [] = whatever$
  $matchers\ (Var\ x\ \#\ ts)\ (u\ \#\ us) = whatever$
  $matchers\ (Fun\ f\ ts\ \#\ tss)\ (Var\ y\ \#\ us) = whatever$
  $matchers\ (Fun\ f\ ts\ \#\ tss)\ (Fun\ g\ us\ \#\ uss) = whatever$
  $length\ ts \neq length\ us \Longrightarrow matchers\ ts\ us = whatever$
  $length\ ss = length\ us \Longrightarrow matchers\ (ss\ @\ ts)\ (us\ @\ vs) = whatever$
  sorry

1.4 Completeness

Prove that *match* is complete, that is, if $match\ ts\ us\ \sigma = None$ then there is no extension of σ that is a matcher of *ts* and *us*.

lemma *match-complete*:

```
  assumes match ts us σ = None
  shows matchers ts us ∩ {τ. σ ⊆m τ} = {}
  sorry

end
```

2 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
```

2.1 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
```

2.2 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

2.3 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

3 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*: $(a,\ b,\ g) \in gcd \Longrightarrow g\ dvd\ a \wedge g\ dvd\ b$
  sorry

3.1 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 $0 < a \vee 0 < b$
    and $d\ dvd\ a$
    and $d\ dvd\ b$
  shows $d \leq g$
  sorry

3.2 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

end

4 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

4.1 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 α :: *'a* ⇒ *bool*.

fun *eval* :: (*'a* ⇒ *bool*) ⇒ *'a form* ⇒ *bool*
  where
    eval α φ = *undefined*

Define a predicate *sat* that captures satisfiable formulas.

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

4.2 Conjunctive Normal Forms

Literals are positive or negative variables.

datatype *'a literal* = *P 'a* | *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* ⇒ *'a form*
  where
    of-cnf cs = *undefined*

4.3 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_φ ∧ *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_φ ∧ *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*:
  $length\ (tseitin\ \varphi) \leq n * size\ \varphi$
  sorry

lemma *tseitin-num-literals*:
  $num\text{-}literals\ (tseitin\ \varphi) \leq n * size\ \varphi$
  sorry

4.4 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* :: $('a \Rightarrow 'b) \Rightarrow 'a\ cnf \Rightarrow 'b\ cnf$
  where
    $rename\text{-}cnf\ f\ cs = undefined$

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

lemma $property\ f\ cs \Longrightarrow sat\ (of\text{-}cnf\ (rename\text{-}cnf\ f\ cs)) \longleftrightarrow sat\ (of\text{-}cnf\ cs)$ **sorry**

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

definition *tseitin-fresh* :: $'your\text{-}type\ form \Rightarrow 'your\text{-}type\ cnf$ **where**
  $tseitin\text{-}fresh\ \varphi = (let$
    $cs = [P\ \varphi]\ \#\ tseitin\ \varphi;$
    $renaming = undefined$
  $in\ rename\text{-}cnf\ renaming\ cs)$

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*: $sat\ \varphi \longleftrightarrow sat\ (of\text{-}cnf\ (tseitin\text{-}fresh\ \varphi))$ **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**
  $test\text{-}form = Neg\ (Conj\ (Disj\ (Neg\ (Var\ X))\ (Var\ Z))\ (Neg\ (Var\ Y)))$

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

end

5 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 four 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 $R_0 := R_0 + R_{n+1}$

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

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*

5.1 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}.

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

5.2 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
```

6 Congruence Closure (2 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
```

6.1 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*: $u \in subt\ u$ **sorry**

lemma *subt-trans*: $s \in subt\ t \Longrightarrow t \in subt\ u \Longrightarrow s \in subt\ u$ **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* = $\bigcup$ ((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* ∈ *E* ⟹ *eq* ∈ *Cong E*
| *C-refl*: (*t*,*t*) ∈ *Cong E*
| *C-sym*: (*s*,*t*) ∈ *E* ⟹ (*t*,*s*) ∈ *Cong E*
| *C-trans*: (*s*,*t*) ∈ *E* ⟹ (*t*,*u*) ∈ *E* ⟹ (*s*,*u*) ∈ *Cong E*
| *C-cong*: *length ss* = *length ts* ⟹ (∀ *i* < *length ts*. (*ss* ! *i*, *ts* ! *i*) ∈ *E*) ⟹ (*Fun f ss*, *Fun f ts*) ∈ *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* ∪ *subt t* ∪ *subt-eqs GE*

definition *CongS* **where** *CongS E* = *Cong E* ∩ (*S* × *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* ^^ *i*) *GE*

Prove the following simple inclusions.

lemma *GE-S*: *GE* ⊆ *S* × *S* **sorry**

lemma *GE-CCA*: *GE* ⊆ *CCA i* **sorry**

6.2 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*) ∈ *estep*^∗ ∩ (*S* × *S*) ⟶ (*u*,*v*) ∈ *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 deriviation 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

6.3 Soundness of CCA

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

lemma *Cong-esteps*: $E \subseteq estep\hat{}{*} \Longrightarrow Cong\ E \subseteq estep\hat{}{*}$ **sorry**

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

lemma *CCA-imp-esteps*: $CCA\ i \subseteq estep\hat{}{*}$ **sorry**

6.4 Correctness of CCA

Having soundness and completeness, correctness is simple.

theorem *congruence-closure-correct*: **assumes** $CongS\ (CCA\ i) = CCA\ i$
  shows $(s,t) \in estep\hat{}{*} \longleftrightarrow (s,\ t) \in CCA\ i$
  sorry

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. Proving termination formally is not part of this project.

end
end
end

7 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

7.1 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** $\to_p$ *66*)

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

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

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

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

7.2 Natural Deduction

The natural deduction rules we consider are captured by the following inductive predicate *proves P* φ, 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*: $\varphi \# P \vdash \psi \Longrightarrow P \vdash (\varphi \to_p \psi)$
| *impE*: $P \vdash \varphi \Longrightarrow P \vdash \varphi \to_p \psi \Longrightarrow P \vdash \psi$
| *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

7.3 Soundness

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

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

7.4 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