These exercises were originally included inline in the text, but my latest feeling is that I don't have the time to maintain the exercises at a sufficient quality level to match the level I'm targetting for the rest of the book. I'm including them in this file for now.

0.1 From InductiveTypes

1. Define an inductive type *truth* with three constructors, *Yes*, *No*, and *Maybe*. *Yes* stands for certain truth, *No* for certain falsehood, and *Maybe* for an unknown situation. Define "not," "and," and "or" for this replacement boolean algebra. Prove that your implementation of "and" is commutative and distributes over your implementation of "or."

2. Define an inductive type *slist* that implements lists with support for constant-time concatenation. This type should be polymorphic in a choice of type for data values in lists. The type *slist* should have three constructors, for empty lists, singleton lists, and concatenation. Define a function *flatten* that converts *slist*s to *list*s. (You will want to run `Require Import List.` to bring list definitions into scope.) Finally, prove that *flatten* distributes over concatenation, where the two sides of your quantified equality will use the *slist* and *list* versions of concatenation, as appropriate. Recall from Chapter 2 that the infix operator ++ is syntactic sugar for the *list* concatenation function *app*.

3. Modify the first example language of Chapter 2 to include variables, where variables are represented with *nat*. Extend the syntax and semantics of expressions to accommodate the change. Your new *expDenote* function should take as a new extra first argument a value of type $var \to nat$, where *var* is a synonym for naturals-as-variables, and the function assigns a value to each variable. Define a constant folding function which does a bottom-up pass over an expression, at each stage replacing every binary operation on constants with an equivalent constant. Prove that constant folding preserves the meanings of expressions.

4. Reimplement the second example language of Chapter 2 to use mutually inductive types instead of dependent types. That is, define two separate (non-dependent) inductive types *nat_exp* and *bool_exp* for expressions of the two different types, rather than a single indexed type. To keep things simple, you may consider only the binary operators that take naturals as operands. Add natural number variables to the language, as in the last exercise, and add an "if" expression form taking as arguments one boolean expression and two natural number expressions. Define semantics and constant-folding functions for this new language. Your constant folding should simplify not just binary operations (returning naturals or booleans) with known arguments, but also "if" expressions with known values for their test expressions but possibly undetermined "then" and "else" cases. Prove that constant-folding a natural number expression preserves its meaning.

5. Define mutually inductive types of even and odd natural numbers, such that any natural number is isomorphic to a value of one of the two types. (This problem does not ask you to prove that correspondence, though some interpretations of the task may be interesting exercises.) Write a function that computes the sum of two even numbers, such that the function type guarantees that the output is even as well. Prove that this function is commutative.

6. Using a reflexive inductive definition, define a type *nat_tree* of infinitary trees, with natural numbers at their leaves and a countable infinity of new trees branching out of each internal node. Define a function *increment* that increments the number in every leaf of a *nat_tree*. Define a function *leapfrog* over a natural i and a tree *nt*. *leapfrog* should recurse into the ith child of *nt*, the i+1st child of that node, the i+2nd child of the next node, and so on, until reaching a leaf, in which case *leapfrog* should return the number at that leaf. Prove that the result of any call to *leapfrog* is incremented by one by calling *increment* on the tree.

7. Define a type of trees of trees of trees of (repeat to infinity). That is, define an inductive type *trexp*, whose members are either base cases containing natural numbers or binary trees of *trexp*s. Base your definition on a parameterized binary tree type *btree* that you will also define, so that *trexp* is defined as a nested inductive type. Define a function *total* that sums all of the naturals at the leaves of a *trexp*. Define a function *increment* that increments every leaf of a *trexp* by one. Prove that, for all *tr*, *total* (*increment* *tr*) $\geq$ *total tr*. On the way to finishing this proof, you will probably want to prove a lemma and add it as a hint using the syntax `Hint Resolve` *name_of_lemma.*.

8. Prove discrimination and injectivity theorems for the *nat_btree* type defined earlier in this chapter. In particular, without using the tactics `discriminate`, `injection`, or `congruence`, prove that no leaf equals any node, and prove that two equal nodes carry the same natural number.

0.2 From Predicates

1. Prove these tautologies of propositional logic, using only the tactics `apply`, `assumption`, `constructor`, `destruct`, `intro`, `intros`, `left`, `right`, `split`, and `unfold`.

   (a) $(\mathit{True} \vee \mathit{False}) \wedge (\mathit{False} \vee \mathit{True})$

   (b) $P \rightarrow \neg \neg P$

   (c) $P \wedge (Q \vee R) \rightarrow (P \wedge Q) \vee (P \wedge R)$

2. Prove the following tautology of first-order logic, using only the tactics `apply`, `assert`, `assumption`, `destruct`, `eapply`, `eassumption`, and `exists`. You will probably find the `assert` tactic useful for stating and proving an intermediate lemma, enabling a kind of "forward reasoning," in contrast to the "backward reasoning" that is the default for

Coq tactics. The tactic `eassumption` is a version of `assumption` that will do matching of unification variables. Let some variable T of type `Set` be the set of individuals. x is a constant symbol, p is a unary predicate symbol, q is a binary predicate symbol, and f is a unary function symbol.

(a) $p\ x \to (\forall\ x,\ p\ x \to \exists\ y,\ q\ x\ y) \to (\forall\ x\ y,\ q\ x\ y \to q\ y\ (f\ y)) \to \exists\ z,\ q\ z\ (f\ z)$

3. Define an inductive predicate capturing when a natural number is an integer multiple of either 6 or 10. Prove that 13 does not satisfy your predicate, and prove that any number satisfying the predicate is not odd. It is probably easiest to prove the second theorem by indicating "odd-ness" as equality to $2 \times n + 1$ for some n.

4. Define a simple programming language, its semantics, and its typing rules, and then prove that well-typed programs cannot go wrong. Specifically:

   (a) Define *var* as a synonym for the natural numbers.

   (b) Define an inductive type *exp* of expressions, containing natural number constants, natural number addition, pairing of two other expressions, extraction of the first component of a pair, extraction of the second component of a pair, and variables (based on the *var* type you defined).

   (c) Define an inductive type *cmd* of commands, containing expressions and variable assignments. A variable assignment node should contain the variable being assigned, the expression being assigned to it, and the command to run afterward.

   (d) Define an inductive type *val* of values, containing natural number constants and pairings of values.

   (e) Define a type of variable assignments, which assign a value to each variable.

   (f) Define a big-step evaluation relation `eval`, capturing what it means for an expression to evaluate to a value under a particular variable assignment. "Big step" means that the evaluation of every expression should be proved with a single instance of the inductive predicate you will define. For instance, "1 + 1 evaluates to 2 under assignment *va*" should be derivable for any assignment *va*.

   (g) Define a big-step evaluation relation *run*, capturing what it means for a command to run to a value under a particular variable assignment. The value of a command is the result of evaluating its final expression.

   (h) Define a type of variable typings, which are like variable assignments, but map variables to types instead of values. You might use polymorphism to share some code with your variable assignments.

   (i) Define typing judgments for expressions, values, and commands. The expression and command cases will be in terms of a typing assignment.

   (j) Define a predicate *varsType* to express when a variable assignment and a variable typing agree on the types of variables.

(k) Prove that any expression that has type t under variable typing vt evaluates under variable assignment va to some value that also has type t in vt, as long as va and vt agree.

(l) Prove that any command that has type t under variable typing vt evaluates under variable assignment va to some value that also has type t in vt, as long as va and vt agree.

A few hints that may be helpful:

(a) One easy way of defining variable assignments and typings is to define both as instances of a polymorphic map type. The map type at parameter T can be defined to be the type of arbitrary functions from variables to T. A helpful function for implementing insertion into such a functional map is *eq_nat_dec*, which you can make available with `Require Import` *Arith.*. *eq_nat_dec* has a dependent type that tells you that it makes accurate decisions on whether two natural numbers are equal, but you can use it as if it returned a boolean, e.g., `if` *eq_nat_dec n m* `then` *E1* `else` *E2*.

(b) If you follow the last hint, you may find yourself writing a proof that involves an expression with *eq_nat_dec* that you would like to simplify. Running `destruct` on the particular call to *eq_nat_dec* should do the trick. You can automate this advice with a piece of Ltac:

```
match goal with
  | [ ⊢ context[eq_nat_dec ?X ?Y] ] ⇒ destruct (eq_nat_dec X Y)
end
```

(c) You probably do not want to use an inductive definition for compatibility of variable assignments and typings.

(d) The *CpdtTactics* module from this book contains a variant *crush'* of *crush*. *crush'* takes two arguments. The first argument is a list of lemmas and other functions to be tried automatically in "forward reasoning" style, where we add new facts without being sure yet that they link into a proof of the conclusion. The second argument is a list of predicates on which inversion should be attempted automatically. For instance, running *crush'* (*lemma1*, *lemma2*) *pred* will search for chances to apply *lemma1* and *lemma2* to hypotheses that are already available, adding the new concluded fact if suitable hypotheses can be found. Inversion will be attempted on any hypothesis using *pred*, but only those inversions that narrow the field of possibilities to one possible rule will be kept. The format of the list arguments to *crush'* is that you can pass an empty list as *tt*, a singleton list as the unadorned single element, and a multiple-element list as a tuple of the elements.

(e) If you want *crush'* to apply polymorphic lemmas, you may have to do a little extra work, if the type parameter is not a free variable of your proof context (so that *crush'* does not know to try it). For instance, if you define a polymorphic map insert function *assign* of some type $\forall\ T$: `Set`, ..., and you want particular

applications of *assign* added automatically with type parameter U, you would need to include *assign* in the lemma list as *assign* U (if you have implicit arguments off) or *assign* $(T := U)$ or @*assign* U (if you have implicit arguments on).

0.3 From Coinductive

1. (a) Define a co-inductive type of infinite trees carrying data of a fixed parameter type. Each node should contain a data value and two child trees.

   (b) Define a function *everywhere* for building a tree with the same data value at every node.

   (c) Define a function *map* for building an output tree out of two input trees by traversing them in parallel and applying a two-argument function to their corresponding data values.

   (d) Define a tree *falses* where every node has the value *false*.

   (e) Define a tree *true_false* where the root node has value *true*, its children have value *false*, all nodes at the next have the value *true*, and so on, alternating boolean values from level to level.

   (f) Prove that *true_false* is equal to the result of mapping the boolean "or" function *orb* over *true_false* and *falses*. You can make *orb* available with `Require Import` *Bool.*. You may find the lemma *orb_false_r* from the same module helpful. Your proof here should not be about the standard equality $=$, but rather about some new equality relation that you define.

0.4 From Subset

All of the notations defined in this chapter, plus some extras, are available for import from the module *MoreSpecif* of the book source.

1. Write a function of type $\forall\ n\ m :\ nat,\ \{n \leq m\} + \{n > m\}$. That is, this function decides whether one natural is less than another, and its dependent type guarantees that its results are accurate.

2. (a) Define *var*, a type of propositional variables, as a synonym for *nat*.

   (b) Define an inductive type *prop* of propositional logic formulas, consisting of variables, negation, and binary conjunction and disjunction.

   (c) Define a function *propDenote* from variable truth assignments and *prop*s to `Prop`, based on the usual meanings of the connectives. Represent truth assignments as functions from *var* to *bool*.

(d) Define a function *bool_true_dec* that checks whether a boolean is true, with a maximally expressive dependent type. That is, the function should have type $\forall\ b, \{b = true\} + \{b = true \rightarrow False\}$.

(e) Define a function `decide` that determines whether a particular *prop* is true under a particular truth assignment. That is, the function should have type $\forall\ (truth : var \rightarrow bool)\ (p : prop), \{propDenote\ truth\ p\} + \{\tilde{}\ propDenote\ truth\ p\}$. This function is probably easiest to write in the usual tactical style, instead of programming with `refine`. The function *bool_true_dec* may come in handy as a hint.

(f) Define a function *negate* that returns a simplified version of the negation of a *prop*. That is, the function should have type $\forall\ p : prop, \{p' : prop \mid \forall\ truth, propDenote\ truth\ p \leftrightarrow \neg\ propDenote\ truth\ p'\}$. To simplify a variable, just negate it. Simplify a negation by returning its argument. Simplify conjunctions and disjunctions using De Morgan's laws, negating the arguments recursively and switching the kind of connective. Your `decide` function may be useful in some of the proof obligations, even if you do not use it in the computational part of *negate*'s definition. Lemmas like `decide` allow us to compensate for the lack of a general Law of the Excluded Middle in CIC.

3. Implement the DPLL satisfiability decision procedure for boolean formulas in conjunctive normal form, with a dependent type that guarantees its correctness. An example of a reasonable type for this function would be $\forall\ f : formula, \{truth : tvals \mid formulaTrue\ truth\ f\} + \{\forall\ truth, \neg\ formulaTrue\ truth\ f\}$. Implement at least "the basic backtracking algorithm" as defined here:

   http://en.wikipedia.org/wiki/DPLL_algorithm

   It might also be instructive to implement the unit propagation and pure literal elimination optimizations described there or some other optimizations that have been used in modern SAT solvers.

0.5 From MoreDep

1. Define a kind of dependently typed lists, where a list's type index gives a lower bound on how many of its elements satisfy a particular predicate. In particular, for an arbitrary set A and a predicate P over it:

   (a) Define a type $plist : nat \rightarrow$ `Set`. Each *plist* n should be a list of As, where it is guaranteed that at least n distinct elements satisfy P. There is wide latitude in choosing how to encode this. You should try to avoid using subset types or any other mechanism based on annotating non-dependent types with propositions after-the-fact.

(b) Define a version of list concatenation that works on *plist*s. The type of this new function should express as much information as possible about the output *plist*.

(c) Define a function *plistOut* for translating *plist*s to normal *list*s.

(d) Define a function *plistIn* for translating *list*s to *plist*s. The type of *plistIn* should make it clear that the best bound on P-matching elements is chosen. You may assume that you are given a dependently typed function for deciding instances of P.

(e) Prove that, for any list *ls*, *plistOut* (*plistIn ls*) = *ls*. This should be the only part of the exercise where you use tactic-based proving.

(f) Define a function *grab* : $\forall\ n\ (ls : plist\ (S\ n))$, *sig P*. That is, when given a *plist* guaranteed to contain at least one element satisfying P, *grab* produces such an element. The type family *sig* is the one we met earlier for sigma types (i.e., dependent pairs of programs and proofs), and *sig P* is extensionally equivalent to $\{x : A \mid P\ x\}$, though the latter form uses an eta-expansion of P instead of P itself as the predicate.

0.6 From DataStruct

remove printing *

Some of the type family definitions and associated functions from this chapter are duplicated in the *DepList* module of the book source. Some of their names have been changed to be more sensible in a general context.

1. Define a tree analogue of *hlist*. That is, define a parameterized type of binary trees with data at their leaves, and define a type family *htree* indexed by trees. The structure of an *htree* mirrors its index tree, with the type of each data element (which only occur at leaves) determined by applying a type function to the corresponding element of the index tree. Define a type standing for all possible paths from the root of a tree to leaves and use it to implement a function *tget* for extracting an element of an *htree* by path. Define a function *htmap2* for "mapping over two trees in parallel." That is, *htmap2* takes in two *htree*s with the same index tree, and it forms a new *htree* with the same index by applying a binary function pointwise.

   Repeat this process so that you implement each definition for each of the three definition styles covered in this chapter: inductive, recursive, and index function.

2. Write a dependently typed interpreter for a simple programming language with ML-style pattern-matching, using one of the encodings of heterogeneous lists to represent the different branches of a `case` expression. (There are other ways to represent the same thing, but the point of this exercise is to practice using those heterogeneous list types.) The object language is defined informally by this grammar:

   $t ::= bool \mid t + t$

$p ::= x \mid b \mid inl\ p \mid inr\ p$
$e ::= x \mid b \mid inl\ e \mid inr\ e \mid \texttt{case}\ e\ \texttt{of}\ [p \Rightarrow e]^* \mid \_ \Rightarrow e$

The non-terminal x stands for a variable, and b stands for a boolean constant. The production for `case` expressions means that a pattern-match includes zero or more pairs of patterns and expressions, along with a default case.
Your interpreter should be implemented in the style demonstrated in this chapter. That is, your definition of expressions should use dependent types and de Bruijn indices to combine syntax and typing rules, such that the type of an expression tells the types of variables that are in scope. You should implement a simple recursive function translating types t to `Set`, and your interpreter should produce values in the image of this translation.

0.7 From Equality

1. Implement and prove correct a substitution function for simply typed lambda calculus. In particular:

   (a) Define a datatype `type` of lambda types, including just booleans and function types.

   (b) Define a type family $exp : list\ \texttt{type} \to \texttt{type} \to \texttt{Type}$ of lambda expressions, including boolean constants, variables, and function application and abstraction.

   (c) Implement a definitional interpreter for *exp*s, by way of a recursive function over expressions and substitutions for free variables, like in the related example from the last chapter.

   (d) Implement a function `subst` : $\forall\ t'\ ts\ t,\ exp\ (t' :: ts)\ t \to exp\ ts\ t' \to exp\ ts\ t$. The type of the first expression indicates that its most recently bound free variable has type t'. The second expression also has type t', and the job of `subst` is to substitute the second expression for every occurrence of the "first" variable of the first expression.

   (e) Prove that `subst` preserves program meanings. That is, prove

   $\forall\ t'\ ts\ t\ (e : exp\ (t' :: ts)\ t)\ (e' : exp\ ts\ t')\ (s : hlist\ typeDenote\ ts),$
   $expDenote\ (\texttt{subst}\ e\ e')\ s = expDenote\ e\ (expDenote\ e'\ s ::: s)$

   where ::: is an infix operator for heterogeneous "cons" that is defined in the book's *DepList* module.

The material presented up to this point should be sufficient to enable a good solution of this exercise, with enough ingenuity. If you get stuck, it may be helpful to use the following structure. None of these elements need to appear in your solution, but we can at least guarantee that there is a reasonable solution based on them.

(a) The *DepList* module will be useful. You can get the standard dependent list definitions there, instead of copying-and-pasting from the last chapter. It is worth reading the source for that module over, since it defines some new helpful functions and notations that we did not use last chapter.

(b) Define a recursive function *liftVar* : ∀ *ts1 ts2 t t'*, *member t* (*ts1* ++ *ts2*) → *member t* (*ts1* ++ *t'* :: *ts2*). This function should "lift" a de Bruijn variable so that its type refers to a new variable inserted somewhere in the index list.

(c) Define a recursive function *lift'* : ∀ *ts t* (*e* : *exp ts t*) *ts1 ts2 t'*, *ts* = *ts1* ++ *ts2* → *exp* (*ts1* ++ *t'* :: *ts2*) *t* which performs a similar lifting on an *exp*. The convoluted type is to get around restrictions on `match` annotations. We delay "realizing" that the first index of *e* is built with list concatenation until after a dependent `match`, and the new explicit proof argument must be used to cast some terms that come up in the `match` body.

(d) Define a function *lift* : ∀ *ts t t'*, *exp ts t* → *exp* (*t'* :: *ts*) *t*, which handles simpler top-level lifts. This should be an easy one-liner based on *lift'*.

(e) Define a recursive function *substVar* : ∀ *ts1 ts2 t t'*, *member t* (*ts1* ++ *t'* :: *ts2*) → (*t'* = *t*) + *member t* (*ts1* ++ *ts2*). This function is the workhorse behind substitution applied to a variable. It returns *inl* to indicate that the variable we pass to it is the variable that we are substituting for, and it returns *inr* to indicate that the variable we are examining is *not* the one we are substituting for. In the first case, we get a proof that the necessary typing relationship holds, and, in the second case, we get the original variable modified to reflect the removal of the substitutee from the typing context.

(f) Define a recursive function *subst'* : ∀ *ts t* (*e* : *exp ts t*) *ts1 t' ts2*, *ts* = *ts1* ++ *t'* :: *ts2* → *exp* (*ts1* ++ *ts2*) *t'* → *exp* (*ts1* ++ *ts2*) *t*. This is the workhorse of substitution in expressions, employing the same proof-passing trick as for *lift'*. You will probably want to use *lift* somewhere in the definition of *subst'*.

(g) Now `subst` should be a one-liner, defined in terms of *subst'*.

(h) Prove a correctness theorem for each auxiliary function, leading up to the proof of `subst` correctness.

(i) All of the reasoning about equality proofs in these theorems follows a regular pattern. If you have an equality proof that you want to replace with *eq_refl* somehow, run `generalize` on that proof variable. Your goal is to get to the point where you can `rewrite` with the original proof to change the type of the generalized version. To avoid type errors (the infamous "second-order unification" failure messages), it will be helpful to run `generalize` on other pieces of the proof context that mention the equality's lefthand side. You might also want to use `generalize dependent`, which generalizes not just one variable but also all variables whose types depend on it. `generalize dependent` has the sometimes-helpful property of removing from the context all variables that it generalizes.

Once you do manage the mind-bending trick of using the equality proof to rewrite its own type, you will be able to rewrite with *UIP_refl*.

(j) The *ext_eq* axiom from the end of this chapter is available in the Coq standard library as *functional_extensionality* in module *FunctionalExtensionality*, and you will probably want to use it in the *lift'* and *subst'* correctness proofs.

(k) The `change` tactic should come in handy in the proofs about *lift* and `subst`, where you want to introduce "extraneous" list concatenations with *nil* to match the forms of earlier theorems.

(l) Be careful about `destruct`ing a term "too early." You can use `generalize` on proof terms to bring into the proof context any important propositions about the term. Then, when you `destruct` the term, it is updated in the extra propositions, too. The *case_eq* tactic is another alternative to this approach, based on saving an equality between the original term and its new form.

0.8 From LogicProg

printing * ·

1. I did a Google search for group theory and found a page that proves some standard theorems[1]. This exercise is about proving all of the theorems on that page automatically.

   For the purposes of this exercise, a group is a set G, a binary function f over G, an identity element e of G, and a unary inverse function i for G. The following laws define correct choices of these parameters. We follow standard practice in algebra, where all variables that we mention are quantified universally implicitly at the start of a fact. We write infix $\times$ for f, and you can set up the same sort of notation in your code with a command like `Infix "*" :=` f.

   - **Associativity**: $(a \times b) \times c = a \times (b \times c)$
   - **Right Identity**: $a \times e = a$
   - **Right Inverse**: $a \times i\ a = e$

   The task in this exercise is to prove each of the following theorems for all groups, where we define a group exactly as above. There is a wrinkle: every theorem or lemma must be proved by either a single call to *crush* or a single call to `eauto`! It is allowed to pass numeric arguments to `eauto`, where appropriate. Recall that a numeric argument sets the depth of proof search, where 5 is the default. Lower values can speed up execution when a proof exists within the bound. Higher values may be necessary to find more involved proofs.

[1] http://dogschool.tripod.com/housekeeping.html

- **Characterizing Identity**: $a \times a = a \rightarrow a = e$
- **Left Inverse**: $i\ a \times a = e$
- **Left Identity**: $e \times a = a$
- **Uniqueness of Left Identity**: $p \times a = a \rightarrow p = e$
- **Uniqueness of Right Inverse**: $a \times b = e \rightarrow b = i\ a$
- **Uniqueness of Left Inverse**: $a \times b = e \rightarrow a = i\ b$
- **Right Cancellation**: $a \times x = b \times x \rightarrow a = b$
- **Left Cancellation**: $x \times a = x \times b \rightarrow a = b$
- **Distributivity of Inverse**: $i\ (a \times b) = i\ b \times i\ a$
- **Double Inverse**: $i\ (i\ a) = a$
- **Identity Inverse**: $i\ e = e$

One more use of tactics is allowed in this problem. The following lemma captures one common pattern of reasoning in algebra proofs:

```
Lemma mult_both : ∀ a b c d1 d2,
  a × c = d1
  → b × c = d2
  → a = b
  → d1 = d2.
  crush.
Qed.
```

That is, we know some equality $a = b$, which is the third hypothesis above. We derive a further equality by multiplying both sides by c, to yield $a \times c = b \times c$. Next, we do algebraic simplification on both sides of this new equality, represented by the first two hypotheses above. The final result is a new theorem of algebra.

The next chapter introduces more details of programming in Ltac, but here is a quick teaser that will be useful in this problem. Include the following hint command before you start proving the main theorems of this exercise:

```
Hint Extern 100 (_ = _) ⇒
  match goal with
    | [ _ : True ⊢ _ ] ⇒ fail 1
    | _ ⇒ assert True by constructor; eapply mult_both
  end.
```

This hint has the effect of applying *mult_both at most once* during a proof. After the next chapter, it should be clear why the hint has that effect, but for now treat it as a useful black box. Simply using `Hint Resolve` *mult_both* would increase proof search

time unacceptably, because there are just too many ways to use *mult_both* repeatedly within a proof.

The order of the theorems above is itself a meta-level hint, since I found that order to work well for allowing the use of earlier theorems as hints in the proofs of later theorems.

The key to this problem is coming up with further lemmas like *mult_both* that formalize common patterns of reasoning in algebraic proofs. These lemmas need to be more than sound: they must also fit well with the way that `eauto` does proof search. For instance, if we had given *mult_both* a traditional statement, we probably would have avoided "pointless" equalities like $a = b$, which could be avoided simply by replacing all occurrences of b with a. However, the resulting theorem would not work as well with automated proof search! Every additional hint you come up with should be registered with `Hint Resolve`, so that the lemma statement needs to be in a form that `eauto` understands "natively."

I recommend testing a few simple rules corresponding to common steps in algebraic proofs. You can apply them manually with any tactics you like (e.g., `apply` or `eapply`) to figure out what approaches work, and then switch to `eauto` once you have the full set of hints.

I also proved a few hint lemmas tailored to particular theorems, but which do not give common algebraic simplification rules. You will probably want to use some, too, in cases where `eauto` does not find a proof within a reasonable amount of time. In total, beside the main theorems to be proved, my sample solution includes 6 lemmas, with a mix of the two kinds of lemmas. You may use more in your solution, but I suggest trying to minimize the number.

0.9 From Match

1. An anonymous Coq fan from the Internet was excited to come up with this tactic definition shortly after getting started learning Ltac:

```
Ltac deSome :=
  match goal with
    | [ H : Some _ = Some _ ⊢ _ ] ⇒ injection H; clear H; intros; subst; deSome
    | _ ⇒ reflexivity
  end.
```

Without lifting a finger, exciting theorems can be proved:

```
Theorem test : ∀ (a b c d e f g : nat),
  Some a = Some b
  → Some b = Some c
  → Some e = Some c
```

```
  → Some f = Some g
  → c = a.
  intros; deSome.
Qed.
```

Unfortunately, this tactic exhibits some degenerate behavior. Consider the following example:

```
Theorem test2 : ∀ (a x1 y1 x2 y2 x3 y3 x4 y4 x5 y5 x6 y6 : nat),
  Some x1 = Some y1
  → Some x2 = Some y2
  → Some x3 = Some y3
  → Some x4 = Some y4
  → Some x5 = Some y5
  → Some x6 = Some y6
  → Some a = Some a
  → x1 = x2.
  intros.
  Time try deSome.
Abort.
```

This (failed) proof already takes about one second on my workstation. I hope a pattern in the theorem statement is clear; this is a representative of a class of theorems, where we may add more matched pairs of x and y variables, with equality hypotheses between them. The running time of *deSome* is exponential in the number of such hypotheses.

The task in this exercise is twofold. First, figure out why *deSome* exhibits exponential behavior for this class of examples and record your explanation in a comment. Second, write an improved version of *deSome* that runs in polynomial time.

2. Sometimes it can be convenient to know that a proof attempt is doomed because the theorem is false. For instance, here are three non-theorems about lists:

```
Theorem test1 : ∀ A (ls1 ls2 : list A), ls1 ++ ls2 = ls2 ++ ls1.
Theorem test2 : ∀ A (ls1 ls2 : list A), length (ls1 ++ ls2) = length ls1 - length ls2.
Theorem test3 : ∀ A (ls : list A), length (rev ls) - 3 = 0.
```

The task in this exercise is to write a tactic that disproves these and many other related "theorems" about lists. Your tactic should follow a simple brute-force enumeration strategy, considering all *list bool* values with length up to some bound given by the user, as a *nat* argument to the tactic. A successful invocation should add a new hypothesis of the negation of the theorem (guaranteeing that the tactic has made a sound decision about falsehood).

A few hints: A good starting point is to pattern-match the conclusion formula and use the `assert` tactic on its negation. An `assert` invocation may include a `by` clause to specify a tactic to use to prove the assertion.

The idea in this exercise is to disprove a quantified formula by finding instantiations for the quantifiers that make it manifestly false. Recall the `specialize` tactic for specializing a hypothesis to particular quantifier instantiations. When you have instantiated quantifiers fully, `discriminate` is a good choice to derive a contradiction. (It at least works for the three examples above and is smart enough for this exercise's purposes.) The `type of` Ltac construct may be useful to analyze the type of a hypothesis to choose how to instantiate its quantifiers.

To enumerate all boolean lists up to a certain length, it will be helpful to write a recursive tactic in continuation-passing style, where the continuation is meant to be called on each candidate list.

Remember that arguments to Ltac functions may not be type-checked in contexts large enough to allow usual implicit argument inference, so instead of *nil* it will be useful to write @*nil bool*, which specifies the usually implicit argument explicitly.

3. Some theorems involving existential quantifiers are easy to prove with `eauto`.

```
Theorem test1 : ∃ x, x = 0.
  eauto.
Qed.
```

Others are harder. The problem with the next theorem is that the existentially quantified variable does not appear in the rest of the theorem, so `eauto` has no way to deduce its value. However, we know that we had might as well instantiate that variable to *tt*, the only value of type *unit*.

```
Theorem test2 : ∃ x : unit, 0 = 0.
```

We also run into trouble in the next theorem, because `eauto` does not understand the *fst* and *snd* projection functions for pairs.

```
Theorem test3 : ∃ x : nat × nat, fst x = 7 ∧ snd x = 2 + fst x.
```

Both problems show up in this monster example.

```
Theorem test4 : ∃ x : (unit × nat) × (nat × bool),
  snd (fst x) = 7 ∧ fst (snd x) = 2 + snd (fst x) ∧ snd (snd x) = true.
```

The task in this problem is to write a tactic that preprocesses such goals so that `eauto` can finish them. Your tactic should serve as a complete proof of each of the above examples, along with the wide class of similar examples. The key smarts that your tactic will bring are: first, it introduces separate unification variables for all the "leaf

types" of compound types built out of pairs; and second, leaf unification variables of type *unit* are simply replaced by *tt*.

A few hints: The following tactic is more convenient than direct use of the built-in tactic `evar`, for generation of new unification variables:

```
Ltac makeEvar T k := let x := fresh in
  evar (x : T); let y := eval unfold x in x in clear x; k y.
```

This is a continuation-passing style tactic. For instance, when the goal begins with existential quantification over a type T, the following tactic invocation will create a new unification variable to use as the quantifier instantiation:

makeEvar T `ltac:(fun` $x \Rightarrow$ `exists` x`)`

printing exists $\exists$

Recall that `exists` formulas are desugared to uses of the *ex* inductive family. In particular, a pattern like the following can be used to extract the domain of an `exists` quantifier into variable T:

| [$\vdash$ *ex* (A := ?T) _] $\Rightarrow$ ...

The *equate* tactic used as an example in this chapter will probably be useful, to unify two terms, for instance if the first is a unification variable whose value you want to set.

```
Ltac equate E1 E2 := let H := fresh in
  assert (H : E1 = E2) by reflexivity; clear H.
```

Finally, there are some minor complications surrounding overloading of the $\times$ operator for both numeric multiplication and Cartesian product for sets (i.e., pair types). To ensure that an Ltac pattern is using the type version, write it like this:
| (?*T1* $\times$?*T2*)%`type` $\Rightarrow$ ...

4. An exercise in the last chapter dealt with automating proofs about rings using `eauto`, where we must prove some odd-looking theorems to push proof search in a direction where unification does all the work. Algebraic proofs consist mostly of rewriting in equations, so we might hope that the `autorewrite` tactic would yield more natural automated proofs. Indeed, consider this example within the same formulation of ring theory that we dealt with last chapter, where each of the three axioms has been added to the rewrite hint database *cpdt* using `Hint Rewrite`:

```
Theorem test1 : ∀ a b, a × b × i b = a.
  intros; autorewrite with cpdt; reflexivity.
Qed.
```

So far so good. However, consider this further example:

```
Theorem test2 : ∀ a, a × e × i a × i e = e.
  intros; autorewrite with cpdt.
```

The goal is merely reduced to $a \times (i\ a \times i\ e) = e$, which of course `reflexivity` cannot prove. The essential problem is that `autorewrite` does not do backtracking search. Instead, it follows a "greedy" approach, at each stage choosing a rewrite to perform and then never allowing that rewrite to be undone. An early mistake can doom the whole process.

The task in this problem is to use Ltac to implement a backtracking version of `autorewrite` that works much like `eauto`, in that its inputs are a database of hint lemmas and a bound on search depth. Here our search trees will have uses of `rewrite` at their nodes, rather than uses of `eapply` as in the case of `eauto`, and proofs must be finished by `reflexivity`.

An invocation to the tactic to prove *test2* might look like this:

rewriter (*right_identity*, (*right_inverse*, *tt*)) 3.

The first argument gives the set of lemmas to consider, as a kind of list encoded with pair types. Such a format cannot be analyzed directly by Gallina programs, but Ltac allows us much more freedom to deconstruct syntax. For example, to case analyze such a list found in a variable x, we need only write:

```
match x with
  | (?lemma, ?more) ⇒ ...
end
```

In the body of the case analysis, *lemma* will be bound to the first lemma, and *more* will be bound to the remaining lemmas. There is no need to consider a case for *tt*, our stand-in for *nil*. This is because lack of any matching pattern will trigger failure, which is exactly the outcome we would like upon reaching the end of the lemma list without finding one that applies. The tactic will fail, triggering backtracking to some previous `match`.

There are different kinds of backtracking, corresponding to different sorts of decisions to be made. The examples considered above can be handled with backtracking that only reconsiders decisions about the order in which to apply rewriting lemmas. A full-credit solution need only handle that kind of backtracking, considering all rewriting sequences up to the length bound passed to your tactic. A good test of this level of applicability is to prove both *test1* and *test2* above. However, some theorems could only be proved using a smarter tactic that considers not only order of rewriting lemma uses, but also choice of arguments to the lemmas. That is, at some points in a proof, the same lemma may apply at multiple places within the goal formula, and some choices may lead to stuck proof states while others lead to success. For an extra challenge (without any impact on the grade for the problem), you might try beefing up your tactic to do backtracking on argument choice, too.

0.10 Exercises

remove printing *

1. Implement a reflective procedure for normalizing systems of linear equations over rational numbers. In particular, the tactic should identify all hypotheses that are linear equations over rationals where the equation righthand sides are constants. It should normalize each hypothesis to have a lefthand side that is a sum of products of constants and variables, with no variable appearing multiple times. Then, your tactic should add together all of these equations to form a single new equation, possibly clearing the original equations. Some coefficients may cancel in the addition, reducing the number of variables that appear.

   To work with rational numbers, import module *QArith* and use `Local Open Scope` *Q_scope*. All of the usual arithmetic operator notations will then work with rationals, and there are shorthands for constants 0 and 1. Other rationals must be written as *num* # *den* for numerator *num* and denominator *den*. Use the infix operator == in place of =, to deal with different ways of expressing the same number as a fraction. For instance, a theorem and proof like this one should work with your tactic:

   ```
   Theorem t2 : ∀ x y z, (2 # 1) × (x - (3 # 2) × y) == 15 # 1
     → z + (8 # 1) × x == 20 # 1
     → (-6 # 2) × y + (10 # 1) × x + z == 35 # 1.
     intros; reifyContext; assumption.
   Qed.
   ```

   Your solution can work in any way that involves reifying syntax and doing most calculation with a Gallina function. These hints outline a particular possible solution. Throughout, the `ring` tactic will be helpful for proving many simple facts about rationals, and tactics like `rewrite` are correctly overloaded to work with rational equality ==.

   (a) Define an inductive type *exp* of expressions over rationals (which inhabit the Coq type *Q*). Include variables (represented as natural numbers), constants, addition, subtraction, and multiplication.

   (b) Define a function *lookup* for reading an element out of a list of rationals, by its position in the list.

   (c) Define a function *expDenote* that translates *exp*s, along with lists of rationals representing variable values, to *Q*.

   (d) Define a recursive function *eqsDenote* over *list* (*exp* × *Q*), characterizing when all of the equations are true.

   (e) Fix a representation *lhs* of flattened expressions. Where *len* is the number of variables, represent a flattened equation as *ilist Q len*. Each position of the list gives the coefficient of the corresponding variable.

   (f) Write a recursive function *linearize* that takes a constant k and an expression e and optionally returns an *lhs* equivalent to $k \times e$. This function returns *None* when it discovers that the input expression is not linear. The parameter *len* of *lhs* should be a parameter of *linearize*, too. The functions *singleton*, *everywhere*,

and *map2* from *DepList* will probably be helpful. It is also helpful to know that *Qplus* is the identifier for rational addition.

(g) Write a recursive function *linearizeEqs* : *list* (*exp* × *Q*) → *option* (*lhs* × *Q*). This function linearizes all of the equations in the list in turn, building up the sum of the equations. It returns *None* if the linearization of any constituent equation fails.

(h) Define a denotation function for *lhs*.

(i) Prove that, when *exp* linearization succeeds on constant k and expression e, the linearized version has the same meaning as $k \times e$.

(j) Prove that, when *linearizeEqs* succeeds on an equation list *eqs*, then the final summed-up equation is true whenever the original equation list is true.

(k) Write a tactic *findVarsHyps* to search through all equalities on rationals in the context, recursing through addition, subtraction, and multiplication to find the list of expressions that should be treated as variables. This list should be suitable as an argument to *expDenote* and *eqsDenote*, associating a Q value to each natural number that stands for a variable.

(l) Write a tactic *reify* to reify a Q expression into *exp*, with respect to a given list of variable values.

(m) Write a tactic *reifyEqs* to reify a formula that begins with a sequence of implications from linear equalities whose lefthand sides are expressed with *expDenote*. This tactic should build a *list* (*exp* × *Q*) representing the equations. Remember to give an explicit type annotation when returning a nil list, as in `constr:`(@*nil* (*exp* × *Q*)).

(n) Now this final tactic should do the job:

```
Ltac reifyContext :=
  let ls := findVarsHyps in
    repeat match goal with
             | [ H : ?e == ?num # ?den ⊢ _ ] ⇒
               let r := reify ls e in
                 change (expDenote ls r == num # den) in H;
                 generalize H
           end;
    match goal with
      | [ ⊢ ?g ] ⇒ let re := reifyEqs g in
          intros;
            let H := fresh "H" in
            assert (H : eqsDenote ls re); [ simpl in *; tauto
              | repeat match goal with
                         | [ H : expDenote _ _ == _ ⊢ _ ] ⇒ clear H
                       end;
```

```
    generalize (linearizeEqsCorrect ls re H); clear H; simpl;
      match goal with
        | [ ⊢ ?X == ?Y → _ ] ⇒
          ring_simplify X Y; intro
      end ]
end.
```