Practical Theorem Proving with Isabelle/Isar
Lecture Notes

Jeremy Siek

May 2, 2007

Contents

Abstract

This document is the lecture notes for the course "Practical Theorem Proving with Isabelle/Isar'.

A **lemma** introduces a proposition followed by a proof. Isabelle has several automatic procedures for generating proofs, one of which is called *simp*, short for simplification. The *simp* procedure applies a set of rewrite rules that is initially seeded with a large number of rules concerning the built-in objects.

lemma *most-trivial*[*simp*]: *True* **by** *simp*

1 Isabelle's functional language

This section introduces the functional language that is embedded in Isabelle. The functional language is closely related to Standard ML.

1.1 Natural numbers, integers, and booleans

Isabelle provides Peano-style natural numbers. There are two constructors for natural numbers: '0' and 'Suc n' (where 'n' is a previously constructed natural number). Numerals such as '1' are shorthand for the appropriate Peano numeral, in this case 'Suc 0'.

lemma $\mathit{Suc}\ 0 = 1$ **by** *simp*

Isabelle also provides the usual arithmetic operations on naturals, such as '+' and '*'.

The double-colon notation ascribes a type to a term.

lemma $1 + 2 = (3{::}\mathit{nat})$ **by** *simp*
lemma $2 * 3 = (6{::}\mathit{nat})$ **by** *simp*

Isabelle provides a division function for naturals, called div, that takes the *floor* of the result (this ensures that the result is a natural number and not a real number).

lemma $3\ \mathit{div}\ 2 = (1{::}\mathit{nat})$ **by** *simp*

The mod function gives the remainder.

lemma $3\ \mathit{mod}\ 2 = (1{::}\mathit{nat})$ **by** *simp*

Isabelle also provide integers.

lemma $1 + -2 = (-1{::}\mathit{int})$ **by** *simp*

Confusingly, the numerals, such as '1', are overloaded and can be either naturals or integers, depending on the context. It is sometimes necessary to use type ascription to tell Isabelle which you want.

The following are examples of Boolean expressions.

lemma $True \wedge True = True$ **by** *simp*
lemma $True \wedge False = False$ **by** *simp*
lemma $True \vee False = True$ **by** *simp*
lemma $False \vee False = False$ **by** *simp*
lemma $\neg\ True = (False::bool)$ **by** *simp*
lemma $False \longrightarrow True$ **by** *simp*
lemma $\forall\ x.\ x = x$ **by** *simp*
lemma $\exists\ x.\ x = 1$ **by** *simp*

1.2 Definitions (non-recursive)

constdefs $xor :: bool \Rightarrow bool \Rightarrow bool$
 $xor\ A\ B \equiv (A \wedge \neg\ B) \vee (\neg\ A \wedge B)$

lemma $xor\ True\ True = False$
 by (*simp add*: *xor-def*)

Add the xor definition to the default set of simplification rules.

declare *xor-def*[*simp*]

1.3 Let expressions

A 'let' expression gives a name to value. The name can be used anywhere after the 'in', i.e., anywhere in the body of the 'let'.

lemma $(let\ x = 3\ in\ x * x) = (9::nat)$ **by** *simp*

1.4 Pairs

Pairs are created with parentheses and commas. The 'fst' function retrieves the first element of the pair and 'snd' retrieves the second.

lemma $let\ p = (2,3)::nat \times nat\ in\ fst\ p + 1 = snd\ p$ **by** *simp*

1.5 Lists

A list can be created using a comma separated sequence of items (all of the same type) enclosed in square brackets. The empty list is written []. The # operator adds an element to the front of a list (aka 'cons').

lemma *let l* = [*1*,*2*,*3*]::(*nat list*) *in hd l* = *1* ∧ *tl l* = [*2*,*3*] **by** *simp*
lemma *1*#(*2*#(*3*#[])) = [*1*,*2*,*3*] **by** *simp*
lemma *length* [*1*,*2*,*3*] = *3* **by** *simp*

Section 38 of "HOL: The basis of Higher-Order Logic" documents many useful functions on lists and lemmas concerning properties of these functions.

1.6 Records

A record is a collection of named values, similar to structs in C and records in Pascal. The following is an example declaration of a point record.

record *point* =
x-coord :: *int*
y-coord :: *int*

The following shows the creation of a record and accessing a field of the record. The Isabelle notation is somewhat unusual because the typical dot notation for field access is not used, and instead the field name is treated as a function. Some care must be taken when choosing field names because they become globally visible, and will conflict with any other uses of the names. So, for example, it would be bad to use x and y for the field names of the point record.

constdefs *pt* :: *point*
pt ≡ (|*x-coord* = *3*, *y-coord* = *7*|)

lemma *x-coord pt* = *3* **by** (*simp add*: *pt-def*)

The record update notation, shown below, creates a copy of a record except for the indicated value.

lemma *x-coord* (*pt*(|*x-coord*:=*4*|)) = *4* **by** (*simp add*: *pt-def*)

1.7 Lambdas (anonymous functions)

lemma (λ *x*. *x* + *x*) *1* = (*2*::*nat*) **by** *simp*

1.8 Conditionals: if and case

lemma (*if True then 1 else 2*) = *1* **by** *simp*

lemma (*case 1 of*
0 ⇒ *False*
| *Suc m* ⇒ *True*) **by** *simp*

1.9 Datatypes and primitive recursion

datatype $'a\ List = Nily \mid Consy\ 'a\ 'a\ List$

consts $app :: 'a\ List \Rightarrow 'a\ List \Rightarrow 'a\ List$
primrec
$app\ Nily\ ys = ys$
$app\ (Consy\ x\ xs)\ ys = Consy\ x\ (app\ xs\ ys)$

Note that one of the arguments in the recursive call must be a part of one of the parameters.

lemma $app\ (Consy\ 1\ (Consy\ 2\ Nily))\ (Consy\ 3\ Nily)$
$= (Consy\ 1\ (Consy\ 2\ (Consy\ 3\ Nily)))$
by *simp*

1.9.1 Exercises

Define a function that sums the first n natural numbers.

2 The Isar proof language

This section describes the basics of the Isar proof language.

2.1 Overview of Isar's syntax (simplified)

A lemma (or theorem) starts with a label, followed by some premises and a conclusion. The premises are introduced with the 'assumes' keyword and separated by 'and'. Each premise may be labeled so that it can be referred to in the proof. The conclusion is introduced with the 'shows' keyword. If there are no premises, then the 'assumes' and 'shows' keywords can be left out.

The following is a simplified grammar for Isar proofs.

```
proof ::= 'proof' method statement* 'qed'
        |  'by' method
```

```
statement ::= 'fix' variable+
            |  'assume' proposition+
            |  ('from' fact+)? 'have' proposition+ proof
            |  ('from' fact+)? 'show' proposition+ proof
proposition ::= (label':')? string
fact ::= label
method ::= '-' | 'this' | 'rule' fact | 'simp' | 'blast' | 'auto'
         | 'induct' variable | ...
```

The **show** statement establishes the conclusion of the proof, whereas the **have** statement is for establishing intermediate results.

2.2 Propositional reasoning

The first example will demonstrate the use of the *congI* rule to prove a conjunction (a logical 'and'). The *congI* rule is shown below. The horizontal bar is used to separate the premises from the conclusion.

$$(\mathit{conjI})\ \frac{P \qquad Q}{P \wedge Q}$$

The rule can equivalently be rendered in English as follows.

$$(\mathit{conjI}) \text{ If } P \text{ and } Q \text{ then } P \wedge Q.$$

In the following example we use the *conjI* rule twice. Each time we supply the necessary premises using the **from** clause and make sure to specify the premises in the expected order.

lemma *conj2*: **assumes** *p*: P **and** *q*: Q **shows** $P \wedge (Q \wedge P)$
proof –
  from *q p* **have** *qp*: $Q \wedge P$ **by** (*rule conjI*)
  from *p qp* **show** $P \wedge (Q \wedge P)$ **by** (*rule conjI*)
qed

The above proof is an example of *forward reasoning*. We start with basic facts, like P and Q, and work up towards proving the conclusion.

Isabelle also supports *backward reasoning*, where the focus is on decomposing the goal (the conclusion) into smaller subgoals. The following is a proof of the same proposition as above, but this time using backward reasoning. We can apply the *conjI* rule in reverse by using it as an argument to the **proof** form. The proposition you are trying to prove should match the conclusion of the rule. The resulting proof state will have a subgoal for each premise of the rule. Each subgoal is proved with a **show** statement, and the sub-proofs are separated with **next**. The *goals* window shows the list of subgoals.

thm *conjI*

lemma assumes p: P **and** q: Q **shows** $P \wedge (Q \wedge P)$
proof (*rule conjI*)
 from p **show** P **by** *this*
next
 show $Q \wedge P$
 proof (*rule conjI*)
  from q **show** Q **by** *this*
 next
  from p **show** P **by** *this*
 qed
qed

The *this* method resolves the goal using the current facts (in the **from** clause).

The next example demonstrates how to prove an implication and make use of conjunctions using the following rules.

$$(impI)\ \frac{\frac{P}{Q}}{P \longrightarrow Q} \qquad (conjunct1)\ \frac{P \wedge Q}{P} \qquad (conjunct2)\ \frac{P \wedge Q}{Q}$$

The following proof uses a mixture of forward and backward reasoning. The choice between forward or backward reasoning depends on what you are trying to prove. Use whichever style seems more natural for the situation.

lemma $(0::nat) < a \wedge a < b \longrightarrow a * a < b * b$
proof (*rule impI*)
 assume x: $0 < a \wedge a < b$
 from x **have** za: $0 < a$ **by** (*rule conjunct1*)
 from x **have** ab: $a < b$ **by** (*rule conjunct2*)
 from za ab **have** aa: $a{*}a < a{*}b$ **by** *simp*
 from ab **have** bb: $a{*}b < b{*}b$ **by** *simp*
 from aa bb **show** $a{*}a < b{*}b$ **by** *arith*
qed

Modes ponens

lemma assumes ab: $A \longrightarrow B$ **and** a: A **shows** B
 by (*rule mp*)

Disjunction introduction

lemma assumes a: A **shows** $A \vee B$
 by (*rule disjI1*)

lemma assumes b: B **shows** $A \vee B$
 by (*rule disjI2*)

Reasoning by cases.

lemma assumes ab: $A \vee B$ **and** ac: $A \longrightarrow C$ **and** bc: $B \longrightarrow C$
 shows C
proof –
 note ab
 moreover {
  assume a: A
  from ac a **have** C **by** (*rule mp*)
 } moreover {
  assume b: B
  from bc b **have** C **by** (*rule mp*)
 }
 ultimately show C **by** (*rule disjE*)
qed

See the manual "Isabelle's Logics: HOL" section 2.2 for a complete list of the inference rules.

2.3 Isar shortcuts

Isar has lots of shortcuts.
 'this' refers to the fact proved by the previous statement.
 'then' = 'from this'
 'hence' = 'then have'
 'thus' = 'then show'
 'with' fact+ = 'from' fact+ 'and' 'this'
 '.' = 'by this'
 '..' = 'by' rule where Isabelle guesses the rule

A sequence of facts that will be used as premises in a statement can be grouped using 'moreover' and then fed into the statement using 'ultimately'. The order of the facts matters.

lemma $A \wedge B \longrightarrow B \wedge A$
proof (*rule impI*)
 assume ab: $A \wedge B$
 hence B **by** (*rule conjunct2*)
 moreover from ab **have** A **..**
 ultimately show $B \wedge A$ **by** (*rule conjI*)
qed

Equational reasoning is made more succinct with the combination of 'also' and 'finally'.

lemma assumes *ab*: $a = b$ **and** *bc*: $b = c$ **and** *c-d*: $c = d$
shows $a = d$
proof –
have $a = b$ **by** (*rule ab*)
also have $\ldots = c$ **by** (*rule bc*)
also have $\ldots = d$ **by** (*rule c-d*)
finally show $a = d$ **.**
qed

2.4 Universal and existential quantifiers

lemma
assumes *a*: $\forall\ x.\ P \longrightarrow Q\ x$
shows $P \longrightarrow (\forall\ x.\ Q\ x)$
proof (*rule impI*)
assume *p*: P
show $\forall\ x.\ Q\ x$
proof (*rule allI*)
fix x
from *a* **have** *pq*: $P \longrightarrow Q\ x$ **by** (*rule allE*)
from *pq p* **show** $Q\ x$ **by** (*rule mp*)
qed
qed

Isabelle's elimination rule for existentials (exE) is a little funky to understand, but Isar provides a nice 'obtain' form that makes it straightforward to use existentials.

lemma
assumes *e*: $\exists\ x.\ P \wedge Q(x)$
shows $P \wedge (\exists\ x.\ Q(x))$
proof (*rule conjI*)
from *e* **obtain** x **where** *p*: P **and** *q*: $Q(x)$ **by** *blast*
from *p* **show** P **.**
next
from *e* **obtain** x **where** *p*: P **and** *q*: $Q(x)$ **by** *blast*
from *q* **show** $\exists\ y.\ Q(y)$ **by** (*rule exI*)
qed

constdefs *divisible-by* :: *nat* ⇒ *nat* ⇒ *bool* (- | - [*80*,*80*] *80*)
$x \mid y \equiv \exists\ k.\ x = k * y$
declare *divisible-by-def*[*simp*]

lemma *divisible-by-trans*:
assumes *ab*: $a \mid (b{::}nat)$ **and** *bc*: $b \mid (c{::}nat)$
shows $a \mid c$
proof *simp*

from *ab* **obtain** m **where** m: $a = m * b$ **by** *auto*
from *bc* **obtain** n **where** n: $b = n * c$ **by** *auto*
from m n **have** $a = m * n * c$ **by** *auto*
thus $\exists\ k.\ a = k * c$ **by** (*rule exI*)
qed

lemma *divisible-by-modz*: $(a \mid b) = (a \ mod \ b = 0)$ **by** *auto*

2.4.1 Exercises

Show that division by a positive natural commutes over addition for natural numbers when the numbers being added are evenly divisible by the denominator. Hint: you may need to use a lemma from Isabelle's Nat theory.

2.5 Case analysis of datatypes

If you have a value of a datatype, it must have come from one of the constructors for the datatype. Isabelle provides a *cases* rule that generates a subgoal, replaces the value that you chose for case analysis with one of the constructors.

As an example we'll use case analysis to prove a simple property of the *drop* function from Isabelle's List theory. The *drop* function is just the tail function *tl* applied n times. For reference, the following is the definition of *drop*.

$$\begin{array}{lcl} drop\ n\ [] & = & [] \\ drop\ n\ (x \cdot xs) & = & \mathsf{case}\ n\ \mathsf{of}\ 0 \Rightarrow x \cdot xs \mid Suc\ m \Rightarrow drop\ m\ xs \end{array}$$

lemma *drop* $(n + 1)$ *xs* = *drop n* (*tl xs*)
proof (*cases xs*)
assume *xs* = []
thus *drop* $(n + 1)$ *xs* = *drop n* (*tl xs*) **by** *simp*
next
fix *a list* **assume** *xs* = *a* # *list*
thus *drop* $(n + 1)$ *xs* = *drop n* (*tl xs*) **by** *simp*
qed

2.6 Notes

The book "How to Prove It" [12] has lots of good examples and advice concerning logical reasoning and proofs. Some of the examples from this section (and later ones) were adapted from that book.

3 INDUCTION

3.1 Mathematical induction

The principle of mathematical induction says that if you want to prove some property of the natural numbers, prove the property for 0 and, assuming the property holds for an arbitrary n, prove that the property also holds for $n + 1$.

$$\frac{P\ 0 \qquad \bigwedge n.\ \dfrac{P\ n}{P\ (Suc\ n)}}{P\ n}$$

The following is a closed form equation for the sum of the first n odd numbers.

$$1 + 3 + \cdots + (2n - 1) = n^2$$

The left-hand side of the equation can be formalized as recursive function.

consts *sum-odds* :: *nat* ⇒ *nat*
primrec
  sum-odds 0 = *0*
  sum-odds (*Suc n*) = (*2* ∗ (*Suc n*) − *1*) + *sum-odds n*

We can then prove the closed form equation by mathematical induction.

lemma *sum-odds n* = *n* ∗ *n*
proof (*induct n*)
  show *sum-odds 0* = *0* ∗ *0* **by** *simp*
next
  fix *n* **assume** *IH*: *sum-odds n* = *n* ∗ *n*
  have *sum-odds* (*Suc n*) = *2* ∗ *Suc n* − *1* + *sum-odds n* **by** *simp*
  also with *IH* **have** ... = *2* ∗ *Suc n* − *1* + *n* ∗ *n* **by** *simp*
  also have ... = *n* ∗ *n* + *2* ∗ *n* + *1* **by** *simp*
  finally show *sum-odds* (*Suc n*) = *Suc n* ∗ *Suc n* **by** *simp*
qed

3.1.1 Exercises

1. Show that n * (n + 1) is even. More specifically, sow that n * (n + 1) — 2.

2. Formulate a closed form equation for the summation of the first n natural numbers. Prove that te closed form is correct using mathematical induction.

3. Formulate a closed form equation for summations of the form $1^2, 2^2 - 1^2, 3^2 - 2^2 + 1^2, 4^2 - 3^2 + 2^2 - 1^2, \ldots$ and prove by mathematical induction that the equation is true.

3.2 Structural induction

Mathematical induction is really just structural induction for natural numbers, which are created from a datatype with constructors zero and successor. In general, we can perform structural induction on any datatype.

For example, the induction rule for the list datatype is

$$\frac{P\ [] \qquad \bigwedge a\ list.\ \dfrac{P\ list}{P\ (a{\cdot}list)}}{P\ list}$$

To prove some property about lists, we prove that the property is true of the empty list and we prove that, assuming the property is true for an arbitrary list, we prove that the property is true of the list with an element added to the front.

thm *append-Nil*
thm *append-Cons*

lemma *append-assoc*: $xs\ @\ (ys\ @\ zs) = (xs\ @\ ys)\ @\ zs$
proof (*induct xs*)
  show $[]\ @\ (ys\ @\ zs) = ([]\ @\ ys)\ @\ zs$
  proof –
    have $[]\ @\ (ys\ @\ zs) = ys\ @\ zs$ **by** *simp*
    also have $\ldots = ([]\ @\ ys)\ @\ zs$ **by** *simp*
    finally show *?thesis* **.**
  qed
next
  fix *x xs*
  assume *IH*: $xs\ @\ (ys\ @\ zs) = (xs\ @\ ys)\ @\ zs$
  show $(x\#xs)\ @\ (ys\ @\ zs) = ((x\#xs)\ @\ ys)\ @\ zs$
  proof –
    have $(x\#xs)\ @\ (ys\ @\ zs) = x\#(xs\ @\ (ys\ @\ zs))$ **by** *simp*
    also have $\ldots = x\#((xs\ @\ ys)\ @\ zs)$ **using** *IH* **by** *simp*
    also have $\ldots = (x\#(xs\ @\ ys))\ @\ zs$ **by** *simp*
    also have $\ldots = ((x\#xs)\ @\ ys)\ @\ zs$ **by** *simp*
    finally show *?thesis* **.**
  qed

qed

Homework: Exercise 2.4.1 from the Isabelle/HOL tutorial concerning binary trees and the relationship between the flatten, mirror, and list reversal functions.

4 MORE LOGICAL REASONING

4.1 Negation, contradiction, and false

To prove a negation, assume the un-negated proposition and then try to reach a contradiction (prove False).

If you've proved both A and not A, then you've proved False.

lemma assumes *xx*: $x * x + y = 13$ **and** *y*: $y \neq 4$
  shows $x \neq (3::nat)$
proof (*rule notI*)
  assume $x = 3$
  with *xx* **have** $y = 4$ **by** *simp*
  with *y* **show** *False* **by** (*rule notE*)
qed

You can prove anything from False.

lemma $1 = (2::nat) \longrightarrow 3 = (4::nat)$
proof (*rule impI*)
  assume $1 = (2::nat)$
  hence *False* **by** *simp*
  thus $3 = (4::nat)$ **by** (*rule FalseE*)
qed

To prove an if and only if (written =), prove that the left-hand-side implies the right-hand-side and vice versa.

lemma $((R \longrightarrow C) \land (S \longrightarrow C)) = ((R \lor S) \longrightarrow C)$
proof (*rule iffI*)
  assume *a*: $((R \longrightarrow C) \land (S \longrightarrow C))$
  from *a* **show** $((R \lor S) \longrightarrow C)$ **by** *blast*
next
  assume *a*: $R \lor S \longrightarrow C$
  thus $(R \longrightarrow C) \land (S \longrightarrow C)$ **by** *blast*
qed

If and only if elimination

lemma assumes $A = B$ **and** A **shows** B
  by (*rule iffD1*)

lemma assumes $A = B$ **and** B **shows** A
  by (*rule iffD2*)

5 GENERALIZING FOR INDUCTION

consts *reverse* :: *'a list* ⇒ *'a list*
primrec
reverse [] = []
reverse (*x#xs*) = (*reverse xs*) @ [*x*]

Here's a more efficient version of reverse.

consts *itrev* :: *'a list* ⇒ *'a list* ⇒ *'a list*
primrec
itrev [] *ys* = *ys*
itrev (*x#xs*) *ys* = *itrev xs* (*x#ys*)

We try to prove that itrev produces the same output as reverse

lemma *itrev xs* [] = *reverse xs*
proof (*induct xs*)
  show *itrev* [] [] = *reverse* [] **by** *simp*
next
  fix *a xs* **assume** *IH*: *itrev xs* [] = *reverse xs*
  have *itrev* (*a#xs*) [] = *itrev xs* [*a*] **by** *simp*
  — Problem: the induction hypothesis does not apply.
  show *itrev* (*a#xs*) [] = *reverse* (*a#xs*) **oops**

Often times generalizing (strengthening) what you want to prove will allow the induction to go through.

Why does generalizing help instead of make it harder? In a proof by induction, in the induction step you get to assume what you are trying to prove for the sub-problem. Now, the stronger the thing you are proving, the more you get to assume about the sub-problem. So often times, when doing proofs by induction, proving a stronger statement is easier than proving a weaker statement.

When using structural induction, universally quantify all variables other than the induction variable.

lemma ∀ *ys*. *itrev xs ys* = (*reverse xs*) @ *ys*
proof (*induct xs*)
  show ∀ *ys*. *itrev* [] *ys* = (*reverse* []) @ *ys* **by** *simp*
next
  fix *a xs* **assume** *IH*: ∀ *ys*. *itrev xs ys* = *reverse xs* @ *ys*
  show ∀ *ys*. *itrev* (*a#xs*) *ys* = *reverse* (*a#xs*) @ *ys*

proof (*rule allI*)
  fix *ys*
  have *itrev* (*a*#*xs*) *ys* = *itrev xs* (*a*#*ys*) **by** *simp*
  also from *IH* **have** ... = *reverse xs* @ (*a*#*ys*) **by** (*rule allE*)
  also have ... = *reverse* (*a* # *xs*) @ *ys* **by** *simp*
  finally show *itrev* (*a* # *xs*) *ys* = *reverse* (*a* # *xs*) @ *ys* **by** *simp*
qed
qed

constdefs *divides* :: *nat* ⇒ *nat* ⇒ *bool* (- | - [*80*,*80*] *80*)
  $x \mid y \equiv \exists\ k.\ k * x = y$
declare *divides-def*[*simp*]

constdefs *isGCD* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* (- *is gcd of* - *and* - [*40*,*40*,*40*] *39*)
  k is gcd of m and n ≡ ($k|m \wedge k|n \wedge (\forall\ q.\ q|m \wedge q|n \longrightarrow q|k)$)

consts *compute-gcd* :: *nat* × *nat* ⇒ *nat*
recdef *compute-gcd measure*(λ (*m*,*n*). *n*)
  compute-gcd(*m*, *n*) = (*if n* = *0 then m else compute-gcd*(*n*, *m mod n*))

lemma *divides-add*:
  assumes *km*: $k|m$ **and** *kn*: $k|n$ **shows** $k|(m+n)$
proof −
  from *km kn* **obtain** *q r* **where** $m = k*q$ **and** $n=k*r$ **apply** *auto* **by** *blast*
  hence $m + n = k*(q + r)$ **by** (*blast intro*: *add-mult-distrib2*[*symmetric*])
  thus $k|(m+n)$ **by** *simp*
qed

lemma *divides-diff*:
  assumes *km*: $k|m$ **and** *kn*: $k|n$ **shows** $k|(m - n)$
proof −
  from *km kn* **obtain** *q r* **where** $m = k*q$ **and** $n=k*r$ **apply** *auto* **by** *blast*
  hence $m - n = k*q - k*r$ **by** *simp*

```
  also have ... = k*(q - r) by (blast intro: diff-mult-distrib2[symmetric])
  finally have m - n = k*(q - r) by simp
  thus k|(m-n) by simp
qed

lemma gcd-preserved:
  assumes M: m = q*n + r
  shows (x is gcd of m and n) = (x is gcd of n and r)
proof -
  { fix k assume k|m and k|n
    hence k|(m - q*n) using divides-diff by auto
    hence k|r using M by simp
  } moreover {
    fix k assume k|n and k|r
    hence k|(q*n + r) using divides-add by auto
    hence k|m using M by simp
  } ultimately show ?thesis using M
    by (simp add: isGCD-def, blast)
qed

theorem compute-gcd-computes-gcd:
  compute-gcd(m,n) is gcd of m and n
proof (induct rule: compute-gcd.induct)
  fix m n
  assume IH: n ≠ 0 ⟶ compute-gcd(n, m mod n) is gcd of n and (m mod n)
  show compute-gcd(m,n) is gcd of m and n
  proof (case-tac n = 0)
    assume n = 0 thus ?thesis using isGCD-def by simp
  next
    assume N: n ≠ 0
    have m = (m div n)*n + (m mod n) by auto
    with N IH gcd-preserved
    have compute-gcd(n, m mod n) is gcd of m and n by blast
    with N show ?thesis by simp
  qed
qed
```

6 Mutual recursion and induction

```
datatype 'a tree = EmptyT | NodeT 'a 'a forest
and      'a forest = NilF | ConsF 'a tree 'a forest
```

consts
flatten-tree :: *'a tree* ⇒ *'a list*
flatten-forest :: *'a forest* ⇒ *'a list*
primrec
flatten-tree EmptyT = []
flatten-tree (*NodeT x f*) = *x*#(*flatten-forest f*)

flatten-forest NilF = []
flatten-forest (*ConsF t f*) = (*flatten-tree t*) @ (*flatten-forest f*)

consts
map-tree :: *'a tree* ⇒ (*'a* ⇒ *'b*) ⇒ *'b tree*
map-forest :: *'a forest* ⇒ (*'a* ⇒ *'b*) ⇒ *'b forest*
primrec
map-tree EmptyT h = *EmptyT*
map-tree (*NodeT x f*) *h* = *NodeT* (*h x*) (*map-forest f h*)

map-forest NilF h = *NilF*
map-forest (*ConsF t f*) *h* = *ConsF* (*map-tree t h*) (*map-forest f h*)

The following is the induction rule for trees and forests.

$$\frac{\bigwedge a\ forest.\ \dfrac{P2\ forest}{P1\ (NodeT\ a\ forest)} \qquad \begin{matrix} P1\ EmptyT \\ P2\ NilF \end{matrix} \qquad \bigwedge tree\ forest.\ \dfrac{P1\ tree \qquad P2\ forest}{P2\ (ConsF\ tree\ forest)}}{P1\ tree \wedge P2\ forest}$$

thm *tree-forest.induct*

lemma *flatten-tree* (*map-tree t h*) = *map h* (*flatten-tree t*)
  ∧ *flatten-forest* (*map-forest f h*) = *map h* (*flatten-forest f*)
proof (*induct-tac t* **and** *f*)
  show *flatten-tree* (*map-tree EmptyT h*) = *map h* (*flatten-tree EmptyT*) **by** *simp*
next
  fix *a f*
  assume *IH*: *flatten-forest* (*map-forest f h*) = *map h* (*flatten-forest f*)
  have *flatten-tree* (*map-tree* (*NodeT a f*) *h*)
    = *flatten-tree* (*NodeT* (*h a*) (*map-forest f h*)) **by** *simp*
  also have ... = (*h a*)#(*flatten-forest* (*map-forest f h*)) **by** *simp*
  also have ... = (*h a*)#(*map h* (*flatten-forest f*)) **using** *IH* **by** *simp*
  also have ... = *map h* (*flatten-tree* (*NodeT a f*)) **by** *simp*
  finally show *flatten-tree* (*map-tree* (*NodeT a f*) *h*)
    = *map h* (*flatten-tree* (*NodeT a f*)) **.**

next
  show *flatten-forest* (*map-forest NilF h*) = *map h* (*flatten-forest NilF*) **by** *simp*
next
  fix *t f*
  assume *IH1*: *flatten-tree* (*map-tree t h*) = *map h* (*flatten-tree t*)
    and *IH2*: *flatten-forest* (*map-forest f h*) = *map h* (*flatten-forest f*)
  from *IH1 IH2*
  show *flatten-forest* (*map-forest* (*ConsF t f*) *h*) =
        map h (*flatten-forest* (*ConsF t f*)) **by** *simp*
qed

7 Case study: compiling to a stack machine

types *'v binop* = *'v* ⇒ *'v* ⇒ *'v*

Isabelle does not have built-in support for LISP-style 'symbols', so the typically approach for representing variables is to use natural numbers.

datatype *'v expr* = *Const 'v* | *Var nat* | *App 'v binop 'v expr 'v expr*

The following *eval* function is an interpreter for this simple language.

consts *eval* :: *'v expr* ⇒ (*nat* ⇒ *'v*) ⇒ *'v*
primrec
  eval (*Const b*) *env* = *b*
  eval (*Var x*) *env* = *env x*
  eval (*App f e1 e2*) *env* = (*f* (*eval e1 env*) (*eval e2 env*))

We compile this language to instructions for a stack machine. Here is the datatype for instructions. The ILoad instruction looks up a variable and puts it on the stack and the IApply instruction applies the binary operation to the top two elements of the stack.

datatype *'v instr* = *IConst 'v* | *ILoad nat* | *IApp 'v binop*

The exec function implements the stack machine, executing a list of instructions.

consts *exec* :: *'v instr list* ⇒ (*nat*⇒*'v*) ⇒ *'v list* ⇒ *'v list*
primrec
  exec [] *env vs* = *vs*
  exec (*i#is*) *env vs* =
    (*case i of*
      IConst v ⇒ *exec is env* (*v#vs*)
     | *ILoad x* ⇒ *exec is env* ((*env x*)#*vs*)
     | *IApp f* ⇒ *exec is env* ((*f* (*hd vs*) (*hd* (*tl vs*)))#(*tl*(*tl vs*))))

TODO: explain arbitrary stuff from partially defined functions, like hd of an empty list.

The compiler translates an expression to a list of instructions.

consts *comp* :: $'v$ *expr* $\Rightarrow$ $'v$ *instr list*
primrec
comp (*Const v*) = [*IConst v*]
comp (*Var x*) = [*ILoad x*]
comp (*App f e1 e2*) = (*comp e2*) @ (*comp e1*) @ [*IApp f*]

7.1 The compiler is correct

To check that the compiler is correct, we prove that the result of compiling and then executing is the same as interpreting.

theorem *exec* (*comp e*) *env* [] = [*eval e s*] **oops**

We're going to prove this by induction on 'e', but first need to generalize the theorem a bit.

theorem ∀ *vs*. *exec* (*comp e*) *env vs* = (*eval e env*)#*vs*
proof (*induct e*)
 fix *v*
 show ∀*vs*. *exec* (*comp* (*Const v*)) *env vs* = (*eval* (*Const v*) *env*)#*vs* **by** *simp*
next
 fix *x*
 show ∀*vs*. *exec* (*comp* (*Var x*)) *env vs* = *eval* (*Var x*) *env* # *vs* **by** *simp*
next
 fix *f e1 e2*
 assume *IH1*: ∀*vs*. *exec* (*comp e1*) *env vs* = *eval e1 env* # *vs*
  and *IH2*: ∀*vs*. *exec* (*comp e2*) *env vs* = *eval e2 env* # *vs*
 show ∀*vs*. *exec* (*comp* (*App f e1 e2*)) *env vs* = *eval* (*App f e1 e2*) *env* # *vs*
 proof
  fix *vs*
  have *A*: (*comp* (*App f e1 e2*)) = (*comp e2*) @ (*comp e1*) @ [*IApp f*] **by** *simp*

  have *eval* (*App f e1 e2*) *env* = (*f* (*eval e1 env*) (*eval e2 env*)) **by** *simp*
  have (*f* (*eval e1 env*) (*eval e2 env*))#*vs*
    = *exec* [*IApp f*] *env* ((*eval e1 env*) # (*eval e2 env* # *vs*)) **by** *simp*
  also have ... = *exec* [*IApp f*] *env* (*exec* (*comp e1*) *env* (*eval e2 env* # *vs*))
   using *IH1* **by** *simp*
  also have ... = *exec* [*IApp f*] *env* (*exec* (*comp e1*) *env* (*exec* (*comp e2*) *env vs*))
   using *IH2* **by** *simp*
  — At this point we need a lemma about exec and append
  oops

lemma *exec-append*[*rule-format*]:
 ∀ *vs*. *exec* (*xs*@*ys*) *env vs* = *exec ys env* (*exec xs env vs*)
 apply (*induct xs*) **apply** *simp* **apply** *auto*
 apply (*case-tac a*) **apply** *auto* **done**

```
theorem ∀ vs. exec (comp e) env vs = (eval e env)#vs
proof (induct e)
 fix v
 show ∀vs. exec (comp (Const v)) env vs = (eval (Const v) env)#vs by simp
next
 fix x
 show ∀vs. exec (comp (Var x)) env vs = eval (Var x) env # vs by simp
next
 fix f e1 e2
 assume IH1: ∀vs. exec (comp e1) env vs = eval e1 env # vs
  and IH2: ∀vs. exec (comp e2) env vs = eval e2 env # vs
 show ∀vs. exec (comp (App f e1 e2)) env vs = eval (App f e1 e2) env # vs
 proof
  fix vs
  have exec (comp (App f e1 e2)) env vs
       = exec ((comp e2) @ (comp e1) @ [IApp f]) env vs by simp
  also have ... = exec ((comp e1) @ [IApp f]) env (exec (comp e2) env vs)
   using exec-append by blast
  also have ... = exec [IApp f] env (exec (comp e1) env (exec (comp e2) env vs))
   using exec-append by blast
  also have ... =  exec [IApp f] env (exec (comp e1) env (eval e2 env # vs))
   using IH2 by simp
  also have ... = exec [IApp f] env ((eval e1 env) # (eval e2 env # vs))
   using IH1 by simp
  also have ... = (f (eval e1 env) (eval e2 env))#vs by simp
  also have ... = eval (App f e1 e2) env # vs by simp
  finally
  show exec (comp (App f e1 e2)) env vs = eval (App f e1 e2) env # vs
   by blast
 qed
qed
```

7.2 Notes

This section is based on section 3.3 of the Isabelle/HOL tutorial

8 Sets

One of the nice aspects of Isabelle is that it provides good support for reasoning with sets. For reference, see section 2.3 in "Isabelle's Logics: HOL".

```
constdefs Evens :: nat set
 Evens ≡ { n. ∃ m. n = 2*m }
```

lemma $2 \in Evens$ **by** (*simp add*: *Evens-def*)
lemma $34 \in Evens$ **by** (*simp add*: *Evens-def*)

constdefs *Odds* :: *nat set*
  $Odds \equiv \{ n.\ \exists\ m.\ n = 2{*}m + 1 \}$

In the following proof we use the rules for intersection introduction and elimination and the mem_Collect_eq rule that can be used to introduce and eliminate membership in a set that is formed by comprehension.

lemma $x \notin (Evens \cap Odds)$
proof (*induct x*)
  show $0 \notin (Evens \cap Odds)$ **by** (*simp add*: *Odds-def*)
next
  fix *x* **assume** *xneo*: $x \notin (Evens \cap Odds)$
  show $Suc\ x \notin (Evens \cap Odds)$
  proof
    assume *towards-contra*: $Suc\ x \in (Evens \cap Odds)$

    from *towards-contra* **have** *sxo*: $Suc\ x \in Odds$ **by** (*rule IntD2*)
    from *sxo* **have** *xm*: $\exists\ m.\ Suc\ x = 2 * m + 1$
      by (*simp only*: *Odds-def mem-Collect-eq*)
    from *xm* **obtain** *m* **where** *M*: $Suc\ x = 2{*}m + 1$ **..**
    from *M* **have** $\exists\ m.\ x = 2 * m$ **by** *simp*
    hence *xe*: $x \in Evens$ **by** (*simp only*: *Evens-def mem-Collect-eq*)

    from *towards-contra* **have** *sxe*: $Suc\ x \in Evens$ **by** (*rule IntD1*)
    from *sxe* **obtain** *n* **where** *M*: $Suc\ x = 2{*}n$
      apply (*simp only*: *Evens-def mem-Collect-eq*) **by** *blast*
    from *M* **have** $x = 2 * (n - 1) + 1$ **by** *arith*
    hence *xo*: $x \in Odds$ **by** (*simp add*: *Odds-def*)

    from *xe xo* **have** $x \in (Evens \cap Odds)$ **by** (*rule IntI*)
    with *xneo* **show** *False* **by** *simp*
  qed
qed

Note how the rules for intersection are similar to the rules for conjunction. That is because the two notions are equivalent in the following sense.

lemma $(x \in A \wedge x \in B) = (x \in A \cap B)$ **by** *simp*

Union is equivalent to disjunction and has similar introduction and elimination rules.

lemma $(x \in A \vee x \in B) = (x \in A \cup B)$ **by** *simp*

Subset is equivalent to implication.

lemma $(\forall\ x.\ x \in A \longrightarrow x \in B) = (A \subseteq B)$ **by** *auto*

Complement is equivalent to not.

lemma $(x \in -A) = (x \notin A)$ **by** *simp*

9 FINITE SETS

Finite sets can be formed using insert and also set notation.

lemma $\mathit{insert}\ 1\ \{0\} = \{0,1\}$ **by** *auto*

The size of a finite set, its cardinality, is given by the *card* function.

lemma $\mathit{card}\ \{\} = 0$ **by** *simp*
lemma $\mathit{card}\ \{4{::}\mathit{nat}\} = 1$ **by** *simp*
lemma $\mathit{card}\ \{4{::}\mathit{nat},1\} = 2$ **by** *simp*
lemma $x \neq y \Longrightarrow \mathit{card}\ \{x,y\} = 2$ **by** *simp*

You can define functions over finite sets using the 'fold' function.

constdefs $\mathit{setsum} :: \mathit{nat\ set} \Rightarrow \mathit{nat}$
 $\mathit{setsum}\ S \equiv \mathit{fold}\ (\lambda\, x\, y.\ x + y)\ (\lambda\, x.\ x)\ 0\ S$
declare *setsum-def* [*simp*]

lemma $\mathit{setsum}\ \{1,2,3\} = 6$ **by** *simp*

You can perform induction on finite sets.

(This is also the first example of proof by case analysis. Perhaps we should introduce proof by cases earlier.)

lemma *setsum-ge*: $\mathit{finite}\ S \Longrightarrow \forall\ x \in S.\ x \leq \mathit{setsum}\ S$
proof (*induct rule*: *finite-induct*)
 show $\forall x{\in}\{\}.\ x \leq \mathit{setsum}\ \{\}$ **by** *simp*
next
 fix x **and** $F{::}\mathit{nat\ set}$
 assume *fF*: $\mathit{finite}\ F$ **and** *xF*: $x \notin F$
  and *IH*: $\forall\ x{\in}F.\ x \leq \mathit{setsum}\ F$
 show $\forall y{\in}\mathit{insert}\ x\ F.\ y \leq \mathit{setsum}\ (\mathit{insert}\ x\ F)$
 proof
  fix y **assume** *yxF*: $y \in \mathit{insert}\ x\ F$
  show $y \leq \mathit{setsum}\ (\mathit{insert}\ x\ F)$
  proof (*cases* $y = x$)
   assume *yx*: $y = x$
   from *fF xF* **have**
    mc: $\mathit{setsum}\ (\mathit{insert}\ x\ F) = x + (\mathit{setsum}\ F)$ **by** *auto*
   with *yx* **show** $y \leq \mathit{setsum}\ (\mathit{insert}\ x\ F)$ **by** *simp*

next
assume *yx*: $y \neq x$
from *yx yxF* **have** *yF*: $y \in F$ **by** *auto*
with *IH* **have** *ysF*: $y \leq$ *setsum F* **by** *blast*
from *fF xF* **have**
mc: *setsum* (*insert x F*) $= x +$ (*setsum F*) **by** *auto*
with *ysF* **show** $y \leq$ *setsum* (*insert x F*) **by** *auto*
qed
qed
qed

10 Case study: automata and the Pumping Lemma

In this section we model deterministic finite automata (DFA) and prove the Pumping Lemma.

We define a *DFA* in Isabelle to be a record consisting of the set of states, the starting state, the set of final states, and the transition function δ. The transition function says which state the DFA goes to given an input character (we're using natural numbers for characters here) and the current state.

types *state* = *nat*

record *DFA* =
DFA-states :: *state set* (Q)
DFA-start :: *state* (q_0)
DFA-finals :: *state set* (F)
DFA-delta :: *nat* ⇒ *state* ⇒ *state* (δ)

A DFA can be used to define a regular language: if the DFA accepts a string, then it is in the language, otherwise the string is not in the language. A DFA accepts a string if feeding the string into the DFA causes the DFA to transition to a final (i.e. accepting) state.

types
string = *nat list*
lang = *string set*

The set consisting of the natural numbers up to n, called *iota*, will be used in several places in the definitions and proofs. We collect some useful properties of *iota* here.

constdefs *iota* :: *nat* ⇒ *nat set*
iota n ≡ $\{ i.\ i \leq n \}$

lemma *iota-z*: *iota 0* = {*0*} **by** (*simp add*: *iota-def*)
lemma *iota-s*: *iota* (*Suc n*) = *insert* (*Suc n*) (*iota n*)
apply (*simp add*: *iota-def*) **by** *auto*

lemma *not-in-iota*: $Suc\ n \notin iota\ n$
apply (*induct n*) **by** (*auto simp add*: *iota-def*)
lemma *iota-finite*: *finite* (*iota n*)
apply (*induct n*) **by** (*auto simp add*: *iota-z iota-s*)
lemma *card-iota*: $card\ (iota\ n) = n + 1$
apply (*induct n*) **using** *not-in-iota iota-finite*
by (*auto simp add*: *iota-z iota-s*)

We define the predicate *good-DFA* to make explicit some assumptions about DFAs. For example, we assume that the range of the transition function is a subset of the states of the DFA. Also, we assume that the states are numbered $0 \ldots n-1$.

constdefs *good-DFA* :: $DFA \Rightarrow bool$
$good\text{-}DFA\ M \equiv finite\ (Q\ M) \wedge (Q\ M) = iota\ (card\ (Q\ M) - 1)$
$\wedge\ (q_0\ M \in Q\ M) \wedge (F\ M \subseteq Q\ M)$
$\wedge\ (\forall\ a.\ \forall\ q \in Q\ M.\ \delta\ M\ a\ q \in Q\ M)$

We use semicolons for function composition, and read the composition from left to right (instead of the usual right to left).

syntax *comp-fwd* :: $('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c)$ (**infixl** ; *70*)
translations $f;g == g \circ f$

The Δ function is the extension of the transition function δ to strings. This define what it means to feed a string into a DFA.

consts *ext-delta* :: $DFA \Rightarrow string \Rightarrow state \Rightarrow state$ (Δ)
primrec
$\Delta\ M\ [] = id$
$\Delta\ M\ (a\#w) = \delta\ M\ a;\ \Delta\ M\ w$

We can now formally define the language of a DFA as the set of strings that take the DFA to a final state via the extended transition function.

constdefs *lang-of* :: $DFA \Rightarrow lang$
$lang\text{-}of\ M \equiv \{\ w.\ \Delta\ M\ w\ (q_0\ M) \in F\ M\ \}$

consts *strpow* :: $string \Rightarrow nat \Rightarrow string$ (-$^{-}$ [*80*,*80*] *80*)
primrec
$w^{0} = w$
$w^{Suc\ n} = w\ @\ w^{n}$

10.1 Properties of the extended transition function

lemma *ext-delta-append*:
$\Delta\ M\ (x@y) = \Delta\ M\ x;\ \Delta\ M\ y$ **by** (*induct x*, *auto*)

lemma *ext-delta-idempotent*:

$\forall\ M\ p.\ good\text{-}DFA\ M \wedge p \in Q\ M \wedge p = \Delta\ M\ y\ p \longrightarrow p = \Delta\ M\ (y^k)\ p$
apply (*induct k*) **using** *ext-delta-append* **by** *auto*

lemma *ext-delta-good*:
$\forall\ M\ q.\ good\text{-}DFA\ M \wedge q \in Q\ M \longrightarrow \Delta\ M\ w\ q \in Q\ M$
apply (*induct w*) **by** (*auto simp add*: *good-DFA-def*)

10.2 Some properties of the take and drop string functions

lemma *take-eq-take-app-drop-take*: **assumes** *ilj*: $i < j$
shows $take\ j\ w = (take\ i\ w)\ @\ (drop\ i\ (take\ j\ w))$
proof –
from *ilj* **have** *B*: $take\ i\ (take\ j\ w) = take\ i\ w$
by (*simp add*: *min-def*)
have *C*: $(take\ i\ (take\ j\ w))\ @\ (drop\ i\ (take\ j\ w)) = take\ j\ w$
by (*simp only*: *append-take-drop-id*)
from *B C* **show** $take\ j\ w = take\ i\ w\ @\ drop\ i\ (take\ j\ w)$ **by** *simp*
qed

lemma *w-equals-xyz*: **assumes** *ij*: $i < j$ **and** *jw*: $j \leq length\ w$
shows $w = (take\ i\ w)\ @\ (drop\ i\ (take\ j\ w))\ @\ (drop\ j\ w)$
proof –
have *A*: $(take\ j\ w)\ @\ (drop\ j\ w) = w$ **by** *simp*
obtain *t* **where** *T*: $t = take\ j\ w$ **by** *simp*
from *A T* **have** *X*: $t\ @\ drop\ j\ w = w$ **by** *simp*
from *ij* **have** *D*: $take\ j\ w = take\ i\ w\ @\ drop\ i\ (take\ j\ w)$
by (*rule take-eq-take-app-drop-take*)
from *D T* **have** *D2*: $t = take\ i\ w\ @\ drop\ i\ (take\ j\ w)$ **by** *simp*
from *X D2* **show** *?thesis* **by** *simp*
qed

10.3 The Pumping Lemma

The pumping lemma relies on the pigeonhole principle, which we state without proof here.

lemma *pigeonhole*:
assumes $card\ B < card\ A$ **and** $(\forall\ x \in A.\ f\ x \in B)$
shows $\exists\ x\ y.\ x \neq y \wedge x \in A \wedge y \in A \wedge f\ x = f\ y$ **sorry**

constdefs *steps* :: $DFA \Rightarrow string \Rightarrow nat \Rightarrow state$
$steps\ M\ w\ n \equiv \Delta\ M\ (take\ n\ w)\ (q_0\ M)$

The Pumping Lemma is best described by the diagram in Figure 1. Given a string w that is longer than the number of states in the DFA, at some point the DFA must loop back on itself and revisit some state p (this is by the pigeonhole principle). Let x be the first portion of w that gets the DFA to p, y the next portion that gets w back to p, and let z be the remainder of w. If w is in the language of the DFA (takes it to a final state), then so is xy^kz, because the DFA can take the y loop any number of times and then proceed via z to a final state.

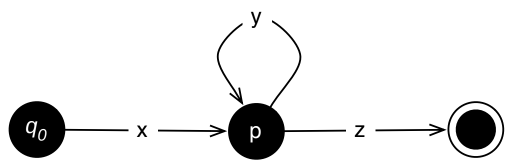

Figure 1: The Pumping Lemma

```
lemma pumping-regular: assumes g: good-DFA M
  shows ∃ n. ∀ w. w ∈ lang-of M ∧ n ≤ length w ⟶
        (∃ x y z. w = x@y@z ∧ y ≠ [] ∧ length (x@y) ≤ n
         ∧ (∀ k. x@y^k@z ∈ lang-of M))
proof –
  let ?n = (card (Q M)) — Choosing n is an important decision!
  { fix w assume wl: w ∈ lang-of M and nw: ?n ≤ length w
    from wl have wd: Δ M w (q0 M) ∈ F M
      by (simp add: lang-of-def)
    — Setting up to use the Pigeonhole Principle
    let ?A = iota (card (Q M))
      and ?B = iota (card (Q M) − 1)
    have F: ∀ x ∈ ?A. (steps M w) x ∈ ?B
      using g ext-delta-good steps-def good-DFA-def by auto
    from g have C: card ?B < card ?A
      using good-DFA-def card-iota by auto
    from F C obtain i j where ij: i ≠ j and iA: i ∈ ?A
      and jA: j ∈ ?A
      and sab: steps M w i = steps M w j
      using pigeonhole by blast
    — without loss of generality, assume i ¡ j
    { fix i j assume ilj: i < j and iA: i ∈ ?A
      and jA: j ∈ ?A
      and sab: steps M w i = steps M w j
      obtain x y z where x: x = take i w
        and y: y = drop i (take j w)
        and z: z = drop j w by simp
      from jA nw have jlw: j ≤ length w
        by (simp add: iota-def)
      from ilj jlw have w: w = x@y@z
        using x y z w-equals-xyz by blast
      from jlw ilj y have ly: length y = j − i
        by (simp add: min-def)
      with ilj have ynil: y ≠ [] by auto
```

```
    from iA jA nw x ilj ly
    have lxyn: length (x@y) ≤ ?n
      by (simp add: iota-def min-def)
    have ∀ k. x@y^k@z ∈ lang-of M
    proof
      fix k let ?p = steps M w i
      from g have pq: ?p ∈ Q M
        using ext-delta-good steps-def good-DFA-def by simp
      from ilj x have zyp: ?p = Δ M x (q_0 M)
        by (simp add: steps-def)
      from x y ilj have take j w = x @ y
        apply simp by (rule take-eq-take-app-drop-take)
      with sab zyp have pyp: ?p = Δ M y ?p
        by (simp add: steps-def ext-delta-append)
      from g pq pyp have pykp: ?p = Δ M (y^k) ?p
        using ext-delta-idempotent by blast
      from w zyp pyp pykp
      have Δ M w (q_0 M) = Δ M (x@y^k@z) (q_0 M)
        by (simp add: ext-delta-append o-assoc)
      with wl show x@y^k@z ∈ lang-of M
        by (simp add: lang-of-def)
    qed
    with w ynil lxyn
    have ∃ x y z. w = x@y@z ∧ y ≠ []
          ∧ length (x@y) ≤ ?n ∧ (∀ k. x@y^k@z ∈ lang-of M)
      by blast
  } with ij iA jA sab
  have ∃ x y z. w = x@y@z ∧ y ≠ []
          ∧ length (x@y) ≤ ?n ∧ (∀ k. x@y^k@z ∈ lang-of M)
    apply (case-tac i < j) apply force
    apply (case-tac j < i) by auto
 } thus ?thesis by blast
qed
```

11 INDUCTIVELY DEFINED SETS AND A GRAPH EXAMPLE

In this section we will explore the use of inductively defined sets by modeling some basic graph theory in Isabelle. We start by defining a type for directed graphs.

types *'vertex digraph* $=$ *'vertex set* $\times$ (*'vertex* $\times$ *'vertex*) *set*

To match the notation of the CLR [4], we provide the $V[G]$ and $E[G]$ syntax for the vertex and edge sets of a graph.

syntax
vertices- :: *'v digraph* $\Rightarrow$ *'v set* (V[-] *100*)
edges- :: *'v digraph* $\Rightarrow$ *'e set* (E[-] *100*)
translations
$V[G] \rightleftharpoons$ *fst G*
$E[G] \rightleftharpoons$ *snd G*

11.1 Inductive definition of a path through a graph

When recursion is required to define a set, use the **inductive** command. Here we define the set of all paths in a directed graph. The introduction rules define what one must show to prove a path is in *paths G*. The conclusion of each introduction must be of the form $x \in$ *paths G*.

consts *paths* :: *'v digraph* $\Rightarrow$ (*'v* $\times$ *'v list* $\times$ *'v*) *set*
inductive *paths G* **intros**
paths-basis: $u \in V[G] \Longrightarrow (u,[],u) \in$ *paths G*
paths-step: $[\![\ (v,p,w) \in$ *paths G*; $(u,v) \in E[G]$; $u \in V[G]\ ]\!]$
$\Longrightarrow (u,\, v\#p,\, w) \in$ *paths G*

Next we define nice syntax for writing path expressions.

syntax
paths- :: [*'v*, *'v list*, *'v*, *'v digraph*] $\Rightarrow$ *bool* (- $\hookrightarrow$- - *in* - *100*)
translations
$u \hookrightarrow_p v$ *in G* $\rightleftharpoons (u,p,v) \in$ *paths G*

Isabelle automatically creates a rule for performing inductive proofs over inductively defined sets. The generated rule *paths.induct* is

$$\frac{xc \hookrightarrow_{xb} xa \text{ in } G \qquad \bigwedge u.\ \dfrac{u \in V[G]}{P\ u\ []\ u} \qquad \bigwedge p\ u\ v\ w.\ \dfrac{v \hookrightarrow_p w \text{ in } G \qquad P\ v\ p\ w \qquad (u,\, v) \in E[G] \qquad u \in V[G]}{P\ u\ (v{\cdot}p)\ w}}{P\ xc\ xb\ xa}$$

Here is an example of performing induction on paths. Each inductive intro gives rise to a subgoal that must be proved. The parenthesis provide scoping for the **fix** and **assume** commands.

lemma *last-is-in-V*: $u \hookrightarrow_p v$ *in* $G \Longrightarrow v \in V[G]$
proof (*induct rule*: *paths.induct*)
 fix u **assume** $u \in V[G]$ **thus** $u \in V[G]$ **.**
next
 fix $p\ u\ v\ w$ **assume** $w \in V[G]$ **thus** $w \in V[G]$ **.**
qed

If you already know that a set is in paths G, then you know that the set must satisfy the conditions given by the intro rules. Isabelle will generate this inverse rule for you automatically if you ask nicely using the **inductive_cases** command.

inductive-cases *paths-inv*:
 $(u,p,w) \in$ *paths G*

lemma $(u,p,v) \in$ *paths* $G \Longrightarrow u \in V[G]$
 apply (*erule paths-inv*)
 apply *simp*
 apply *simp*
 done

The inverse rule is

$$\llbracket u \hookrightarrow_p w \text{ in } G;\ \llbracket u \in V[G];\ p = [];\ w = u \rrbracket \Longrightarrow P;$$
$$\bigwedge pa\ v.\ \llbracket v \hookrightarrow_{pa} w \text{ in } G;\ (u, v) \in E[G];\ u \in V[G];\ p = v{\cdot}pa \rrbracket \Longrightarrow P \rrbracket$$
$$\Longrightarrow P$$

11.2 The strongly connected relation is an equivalence

We are going to show that the strongly connected pairs relation is an equivalence relation. A pair of vertices (u, v) are strongly connected if there is a path from u to v and from v to u.

constdefs
 strongly-connected-pairs :: *'v digraph* $\Rightarrow$ $('v \times 'v)$ *set*
 strongly-connected-pairs G $\equiv$
  $\{(u,v).\ \exists p\ q.\ (u \hookrightarrow_p v \text{ in } G) \wedge (v \hookrightarrow_q u \text{ in } G)\}$

The Isabelle Relation theory contains definitions for reflexive, symmetric, and transitive relations, which we use here to define an equivalence relation

constdefs
 equivalence-relation :: $['a\ set, ('a \times 'a)\ set] \Rightarrow bool$

equivalence-relation S R $\equiv$ *refl S R* $\wedge$ *sym R* $\wedge$ *trans R*

To show the transitivity property we will need to be able to join two paths. The following lemma proves that the result of appending one path to another is a valid path. The way in which this lemma is stated is a bit strange so as to fit what the *paths.induct* method is expecting. First, the only thing to the left of the $\Longrightarrow$ is a *paths* expression. Next, all other premises appear to the right of the $\Longrightarrow$ but to the left of the $\longrightarrow$. The conclusion of the lemma appears to the right of the $\longrightarrow$. Finally, note the use of $\forall$. The variables to the left of the $\Longrightarrow$ are automatically universally quantified, but we need to make sure the rest of the variables are also universally quantified. The use of *[rule-format]* tells Isabelle to transform the statement of the lemma (after it has been proved) into format that is easier to use.

lemma *append-path* [*rule-format*]:
  $a \hookrightarrow_p b \; in \; G \Longrightarrow (\forall \; q \; c. \; (b \hookrightarrow_q c \; in \; G) \longrightarrow (a \hookrightarrow_{p@q} c \; in \; G))$
proof (*induct rule*: *paths.induct*)
  fix u **assume** $u \in V[G]$
  thus $\forall q \; c. \; u \hookrightarrow_q c \; in \; G \longrightarrow u \hookrightarrow_{[]@q} c \; in \; G$ **by** *simp*
next
  fix $p \; u \; v \; w$
  assume *vw*: $v \hookrightarrow_p w \; in \; G$ **and** *uv-inE*: $(u,v) \in E[G]$
    and *u-inV*: $u \in V[G]$
    and *IH*: $\forall q \; c. \; w \hookrightarrow_q c \; in \; G \longrightarrow v \hookrightarrow_{p@q} c \; in \; G$
  show $\forall q \; c. \; w \hookrightarrow_q c \; in \; G \longrightarrow u \hookrightarrow_{(v\#p)@q} c \; in \; G$
  proof *clarify*
      — clarify removed forall, changed single arrow to double
    fix $q \; r \; c$ **assume** *wc*: $w \hookrightarrow_q c \; in \; G$
    from *wc* **and** *IH* **have** $v \hookrightarrow_{p@q} c \; in \; G$ **by** *simp*
    with *u-inV* **and** *uv-inE* **have** $u \hookrightarrow_{v\#(p@q)} c \; in \; G$
      by (*simp add*: *paths.intros*)
    thus $u \hookrightarrow_{(v\#p)@q} c \; in \; G$ **by** *simp*
  qed
qed

The resulting lemma *append-path* is the following:

$[\![a \hookrightarrow_p b \; in \; G; b \hookrightarrow_q c \; in \; G]\!] \Longrightarrow a \hookrightarrow_{p \; @ \; q} c \; in \; G$

lemma *strongly-connected-is-an-equivalence-relation*:
  equivalence-relation ($V[G]$) (*strongly-connected-pairs G*)
  — Going into the proof, we apply the def. of equivalence
  — relation and then the perform induction on the path
proof (*simp add*: *equivalence-relation-def*, *auto*)
  show *refl* ($V[G]$) (*strongly-connected-pairs G*)
  proof (*simp add*: *refl-def strongly-connected-pairs-def*, *auto*,
    erule paths.induct)

fix u **assume** $u \in V[G]$ **thus** $u \in V[G]$ **.**
next — next clears out any fixed variables or assumptions
fix u **assume** $u \in V[G]$ **thus** $u \in V[G]$ **.**
next
fix $a\ p\ b$ **assume** $(a, p, b) \in \mathit{paths}\ G$
thus $b \in V[G]$ **by** $(\mathit{rule\ last\text{-}is\text{-}in\text{-}V})$
next
fix x **assume** $x \in V[G]$
from *prems* **have** $x \hookrightarrow_{[]} x\ \mathit{in}\ G$ **by** $(\mathit{simp\ add}\colon \mathit{paths\text{-}basis})$
thus $\exists\ p.\ x \hookrightarrow_p x\ \mathit{in}\ G$ **by** *auto*
qed
next
show $\mathit{sym}\ (\mathit{strongly\text{-}connected\text{-}pairs}\ G)$
proof $(\mathit{simp\ only}\colon \mathit{sym\text{-}def\ strongly\text{-}connected\text{-}pairs\text{-}def}, \mathit{clarify})$
fix $x\ y\ p\ q$ **assume** $x \hookrightarrow_p y\ \mathit{in}\ G$ **and** $y \hookrightarrow_q x\ \mathit{in}\ G$
thus $\exists p\ q.\ y \hookrightarrow_p x\ \mathit{in}\ G \wedge x \hookrightarrow_q y\ \mathit{in}\ G$ **by** *auto*
qed
next
show $\mathit{trans}\ (\mathit{strongly\text{-}connected\text{-}pairs}\ G)$
proof $(\mathit{simp\ only}\colon \mathit{trans\text{-}def\ strongly\text{-}connected\text{-}pairs\text{-}def}, \mathit{clarify}, \mathit{rename\text{-}tac}\ r\ s)$
fix $x\ y\ z\ p\ q\ r\ s$
assume $\mathit{xy}\colon x \hookrightarrow_p y\ \mathit{in}\ G$ **and** $\mathit{yx}\colon y \hookrightarrow_q x\ \mathit{in}\ G$
and $\mathit{yz}\colon y \hookrightarrow_r z\ \mathit{in}\ G$ **and** $\mathit{zy}\colon z \hookrightarrow_s y\ \mathit{in}\ G$
from *xy* **and** *yz* **have** $\mathit{xz}\colon x \hookrightarrow_{p@r} z\ \mathit{in}\ G$ **by** $(\mathit{rule\ append\text{-}path})$
from *zy* **and** *yx* **have** $\mathit{zx}\colon z \hookrightarrow_{s@q} x\ \mathit{in}\ G$ **by** $(\mathit{rule\ append\text{-}path})$
from *xz* **and** *zx* **show** $\exists p\ q.\ x \hookrightarrow_p z\ \mathit{in}\ G \wedge z \hookrightarrow_q x\ \mathit{in}\ G$ **by** *auto*
qed
qed

12 Case study: the simply typed lambda calculus

We formalize an operational semantics for the simply typed lambda calculus in the evaluation context style [6, 13] and prove type safety.

We use a relatively new approach for representing variables called "locally nameless" [3, 7, 11]. In the locally nameless approach, bound variables are represented with de Bruijn indices whereas free variables are represented with symbols. This approach enjoys the benefits of the de Bruijn indices (α-equivalent terms are syntactically identical) while avoiding much of the complication (normally caused by representing free variables with de Bruijn indices). Separate functions are used to substitution for free and bound variables.

12.1 Syntax of the simply typed lambda calculus

datatype *ty* = *IntT* | *BoolT* | *ArrowT ty ty* (**infixr** → *200*)

datatype *const* = *IntC int* | *BoolC bool* | *Succ* | *IsZero*

datatype *expr* =
  BVar nat | *FVar nat* | *Const const*
 | *Lam ty expr* (λ:-. - [*52*,*52*] *51*)
 | *App expr expr*

Free variables

consts *FV* :: *expr* ⇒ *nat set*
primrec
 FV (*BVar i*) = {}
 FV (*FVar x*) = {*x*}
 FV (*Const c*) = {}
 FV (λ:σ. *e*) = *FV e*
 FV (*App e1 e2*) = *FV e1* ∪ *FV e2*

lemma *finite-FV*: *finite* (*FV e*) **apply** (*induct e*) **by** *auto*

Substitution for free variables

consts *fsubst* :: *nat* ⇒ *expr* ⇒ *expr* ⇒ *expr* ([-→-]- [*54*,*54*,*54*] *53*)
primrec
 [*z*→*e*](*BVar i*) = *BVar i*
 [*z*→*e*](*FVar x*) = (*if z* = *x then e else* (*FVar x*))
 [*z*→*e*](*Const c*) = *Const c*
 [*z*→*e*](λ:σ. *e*′) = (λ:σ. [*z*→*e*]*e*′)
 [*z*→*e*](*App e1 e2*) = *App* ([*z*→*e*]*e1*) ([*z*→*e*]*e2*)

Substitution for bound variables

consts *bsubst* :: *nat* ⇒ *expr* ⇒ *expr* ⇒ *expr* ({-→-}- [*54*,*54*,*54*] *53*)
primrec
 {*k*→*e*}(*BVar i*) = (*if k* = *i then e else* (*BVar i*))
 {*k*→*e*}(*FVar x*) = *FVar x*
 {*k*→*e*}(*Const c*) = *Const c*
 {*k*→*e*}(λ:σ. *e*′) = (λ:σ. {*Suc k*→*e*}*e*′)
 {*k*→*e*}(*App e1 e2*) = *App* ({*k*→*e*}*e1*) ({*k*→*e*}*e2*)

12.2 Operational semantics with evaluation contexts

A utility function for casting an arbitrary expression to an integer.

consts *to-int* :: *expr* ⇒ *int option*
primrec

$to\text{-}int\ (BVar\ x) = None$
$to\text{-}int\ (FVar\ x) = None$
$to\text{-}int\ (Const\ c) =$
$\quad (case\ c\ of$
$\quad\quad IntC\ n \Rightarrow Some\ n$
$\quad\quad |\ BoolC\ b \Rightarrow None$
$\quad\quad |\ Succ \Rightarrow None$
$\quad\quad |\ IsZero \Rightarrow None)$
$to\text{-}int\ (Lam\ \tau\ e) = None$
$to\text{-}int\ (App\ e1\ e2) = None$

The δ function evaluates the primitive operators.

consts $delta :: const \Rightarrow expr \Rightarrow expr\ option\ (\delta)$
primrec
$delta\ (IntC\ n)\ e = None$
$delta\ (BoolC\ b)\ e = None$
$delta\ Succ\ e =$
$\quad (case\ to\text{-}int\ e\ of$
$\quad\quad None \Rightarrow None$
$\quad\quad |\ Some\ n \Rightarrow Some\ (Const\ (IntC\ (n + 1))))$
$delta\ IsZero\ e =$
$\quad (case\ to\text{-}int\ e\ of$
$\quad\quad None \Rightarrow None$
$\quad\quad |\ Some\ n \Rightarrow Some\ (Const\ (BoolC\ (n = 0))))$

Evaluation reduces expressions to values. The following is the definition of which expressions are values.

consts $Values :: expr \Rightarrow bool$
primrec
$Values\ (BVar\ i) = True$
$Values\ (FVar\ x) = True$
$Values\ (Const\ c) = True$
$Values\ (\lambda{:}\sigma.\ e) = True$
$Values\ (App\ e1\ e2) = False$

The call-by-value notion of reduction is defined as follows.

consts $reduces :: (expr \times expr)\ set$
syntax $reduces :: expr \Rightarrow expr \Rightarrow bool$ (**infixl** $\longrightarrow$ 51)
translations $e \longrightarrow e' == (e,e') \in reduces$
inductive $reduces$ **intros**
$Beta: Values\ v \Longrightarrow App\ (\lambda{:}\tau.\ e)\ v \longrightarrow \{0{\to}v\}e$
$Delta: [\![\ \delta\ c\ v = Some\ v';\ Values\ v\ ]\!] \Longrightarrow App\ (Const\ c)\ v \longrightarrow v'$

constdefs $redex :: expr \Rightarrow bool$
$redex\ r \equiv (\exists\ r'.\ r \longrightarrow r')$

We use contexts to specify where reduction can take place within an expression.

datatype *ctx* = *Hole* | *AppL ctx expr* | *AppR expr ctx*

consts *wf-ctx* :: *ctx set*
inductive *wf-ctx* **intros**
WFHole: *Hole* ∈ *wf-ctx*
WFAppL: *E* ∈ *wf-ctx* ⟹ *AppL E e* ∈ *wf-ctx*
WFAppR: ⟦ *Values v*; *E* ∈ *wf-ctx* ⟧ ⟹ *AppR v E* ∈ *wf-ctx*

consts *fill* :: *ctx* ⇒ *expr* ⇒ *expr*   (-[-] [*82*,*82*] *81*)
primrec
Hole[*e*] = *e*
(*AppL E e2*)[*e*] = *App* (*E*[*e*]) *e2*
(*AppR e1 E*)[*e*] = *App e1* (*E*[*e*])

consts *eval-step* :: (*expr* × *expr*) *set*
syntax *eval-step* :: *expr* ⇒ *expr* ⇒ *bool* (**infixl** ⟼ *51*)
translations *e* ⟼ *e′* == (*e*,*e′*) ∈ *eval-step*
inductive *eval-step* **intros**
Step: ⟦ *E* ∈ *wf-ctx*; *r* −→ *r′* ⟧ ⟹ *E*[*r*] ⟼ *E*[*r′*]

12.3 Creating fresh variables

constdefs *max* :: *nat* ⇒ *nat* ⇒ *nat*
max x y ≡ (*if x* < *y then y else x*)
declare *max-def*[*simp*]

interpretation *AC-max*: *ACe* [*max 0*::*nat*]
by (*auto intro*: *ACf.intro ACe-axioms.intro*)

constdefs *setmax* :: *nat set* ⇒ *nat*
setmax S ≡ *fold max* (λ *x*. *x*) *0 S*

lemma *max-ge*: *finite L* ⟹ ∀ *x* ∈ *L*. *x* ≤ *setmax L*
apply (*induct rule*: *finite-induct*)
apply *simp*
apply *clarify*
apply (*case-tac xa* = *x*)
proof −
fix *x* **and** *F*::*nat set* **and** *xa*
assume *fF*: *finite F* **and** *xF*: *x* ∉ *F* **and** *xax*: *xa* = *x*
from *fF xF* **have** *mc*: *setmax* (*insert x F*) = *max x* (*setmax F*)
apply (*simp only*: *setmax-def*)
apply (*rule AC-max.fold-insert*)
apply *auto* **done**

with *xax* **show** $xa \le setmax\ (insert\ x\ F)$
apply *clarify* **by** *simp*
next
fix *x* **and** F::*nat set* **and** *xa*
assume *fF*: *finite F* **and** *xF*: $x \notin F$
and *axF*: $\forall x \in F.\ x \le setmax\ F$
and *xsxF*: $xa \in insert\ x\ F$
and *xax*: $xa \neq x$
from *xax xsxF* **have** *xaF*: $xa \in F$ **by** *auto*
with *axF* **have** *xasF*: $xa \le setmax\ F$ **by** *blast*
from *fF xF* **have** *mc*: $setmax\ (insert\ x\ F) = max\ x\ (setmax\ F)$
apply (*simp only*: *setmax-def*)
apply (*rule AC-max.fold-insert*)
apply *auto* **done**
with *xasF* **show** $xa \le setmax\ (insert\ x\ F)$ **by** *auto*
qed

lemma *max-is-fresh*[*simp*]:
assumes *F*: *finite L* **shows** $Suc\ (setmax\ L) \notin L$
proof
assume *ssl*: $Suc\ (setmax\ L) \in L$
with *F max-ge* **have** $Suc\ (setmax\ L) \le setmax\ L$ **by** *blast*
thus *False* **by** *simp*
qed

lemma *greaterthan-max-is-fresh*[*simp*]:
assumes *F*: *finite L* **and** *I*: $setmax\ L < i$
shows $i \notin L$
proof
assume *ssl*: $i \in L$
with *F max-ge* **have** $i \le setmax\ L$ **by** *blast*
with *I* **show** *False* **by** *simp*
qed

12.4 Well-typed expressions

types $env = nat \Rightarrow ty\ option$

constdefs *remove-bind* :: $env \Rightarrow nat \Rightarrow env \Rightarrow bool$ (- $-$ - $\subset$ - [*50*,*50*,*50*] *49*)
$\Gamma - z \subset \Gamma' \equiv \forall\ x\ \tau.\ x \neq z \wedge \Gamma\ x = Some\ \tau \longrightarrow \Gamma'\ x = Some\ \tau$

constdefs *finite-env* :: $env \Rightarrow bool$
finite-env $\Gamma \equiv$ *finite* (*dom* Γ)
declare *finite-env-def*[*simp*]

consts *TypeOf* :: *const* $\Rightarrow$ *ty*
primrec
 TypeOf (*IntC n*) = *IntT*
 TypeOf (*BoolC b*) = *BoolT*
 TypeOf Succ = *IntT* $\rightarrow$ *IntT*
 TypeOf IsZero = *IntT* $\rightarrow$ *BoolT*

consts *wte* :: (*env* $\times$ *expr* $\times$ *ty*) *set*
syntax *wte* :: *env* $\Rightarrow$ [*expr*,*ty*] $\Rightarrow$ *bool* (- $\vdash$ - : - [*52*,*52*,*52*] *51*)
translations $\Gamma \vdash e : \tau \rightleftharpoons (\Gamma, e, \tau) \in wte$
inductive *wte* **intros**
 wte-var: $\Gamma\ x = Some\ \tau \Longrightarrow \Gamma \vdash FVar\ x : \tau$
 wte-const: $\Gamma \vdash Const\ c : TypeOf\ c$
 wte-abs: $[\![$ *finite L*; *dom* $\Gamma \subseteq L$;
  $\forall\ x.\ x \notin L \longrightarrow \Gamma(x \mapsto \sigma) \vdash \{0 \rightarrow FVar\ x\}e : \tau\ ]\!]$
  $\Longrightarrow \Gamma \vdash (\lambda{:}\sigma.\ e) : \sigma \rightarrow \tau$
 wte-app: $[\![\ \Gamma \vdash e1 : \sigma \rightarrow \tau; \Gamma \vdash e2 : \sigma\ ]\!]$
  $\Longrightarrow \Gamma \vdash App\ e1\ e2 : \tau$

thm *wte.induct*

12.5 Properties of substitution

lemma *bsubst-cross*[*rule-format*]:
 $\forall\ i\ j\ u\ v.\ i \neq j \wedge\ \{i \rightarrow u\}(\{j \rightarrow v\}t) = \{j \rightarrow v\}t \longrightarrow \{i \rightarrow u\}t = t$
 apply (*induct t*)
 apply *force*
 apply *force*
 apply *force*
 apply *clarify*
  apply (*erule-tac x*=*Suc i* **in** *allE*)
  apply (*erule-tac x*=*Suc j* **in** *allE*)
  apply (*erule-tac x*=*u* **in** *allE*)
  apply (*erule-tac x*=*v* **in** *allE*)
  apply *simp*
 apply *clarify*
  apply (*erule-tac x*=*i* **in** *allE*)
  apply (*erule-tac x*=*i* **in** *allE*)
  apply (*erule-tac x*=*j* **in** *allE*)
  apply (*erule-tac x*=*j* **in** *allE*)
  apply *simp* **apply** *blast*
 done

lemma *bsubst-wt*:
 $[\![\ \Gamma \vdash e : \tau;$ *finite-env* $\Gamma\ ]\!] \Longrightarrow \forall\ k\ e'.\ \{k \rightarrow e'\}e = e$

apply (*induct rule*: *wte.induct*)
apply *force*
apply *force*
apply *clarify* **apply** *simp*
apply (*erule-tac x=Suc* (*setmax L*) **in** *allE*)
apply (*erule impE*)
apply (*rule max-is-fresh*) **apply** *simp*
apply (*erule conjE*)+
apply (*erule-tac x=Suc k* **in** *allE*)
apply (*erule-tac x*=e' **in** *allE*)
apply (*rule bsubst-cross*) **apply** *blast*
apply *force*
done

lemma *subst-permute-impl*[*rule-format*]:
$\forall\ j\ x\ z\ \Gamma\ \tau\ e'.\ x \neq z \land \Gamma \vdash e' : \tau \land \textit{finite-env}\ \Gamma$
$\longrightarrow [z{\rightarrow}e'](\{j{\rightarrow}\textit{FVar}\ x\}e) = \{j{\rightarrow}\ \textit{FVar}\ x\}([z{\rightarrow}e']e)$
apply (*induct e*)
apply *force*
apply *simp* **apply** *clarify*
apply (*frule bsubst-wt*)
apply *simp*
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=FVar x* **in** *allE*)
apply *simp*
apply *simp*
apply *simp* **apply** *clarify* **apply** *blast*
apply *simp* **apply** *clarify*
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=x* **in** *allE*)
apply (*erule-tac x=x* **in** *allE*)
apply (*erule-tac x=z* **in** *allE*)
apply (*erule-tac x=z* **in** *allE*)
apply (*erule-tac x*=Γ **in** *allE*)
apply (*erule-tac x*=Γ **in** *allE*)
apply *blast*
done

lemma *subst-permute*:
$[\![\ x \neq z;\ \Gamma \vdash e' : \tau;\ \textit{finite-env}\ \Gamma\ ]\!]$
$\Longrightarrow \{j{\rightarrow}\ \textit{FVar}\ x\}([z{\rightarrow}e']e) = [z{\rightarrow}e'](\{j{\rightarrow}\textit{FVar}\ x\}e)$
using *subst-permute-impl*[*of x z* Γ e' τ *j e*] **by** *simp*

lemma *decompose-subst*[*rule-format*]:

$\forall\ u\ x\ i.\ x \notin FV\ e \longrightarrow \{i{\to}u\}e = [x{\to}u](\{i{\to}FVar\ x\}e)$
apply (*induct e*)
apply *force*
apply *force*
apply *force*
apply *clarify*
  apply (*erule-tac x=u* **in** *allE*)
  apply (*erule-tac x=x* **in** *allE*)
  apply (*erule-tac x=Suc i* **in** *allE*)
  apply *simp*
apply *force*
done

12.6 Properties of environments and rule induction

constdefs *subseteq* :: *env* ⇒ *env* ⇒ *bool* (**infixl** ⊆ *80*)
  $\Gamma \subseteq \Gamma' \equiv \forall\ x\ \tau.\ \Gamma\ x = Some\ \tau \longrightarrow \Gamma'\ x = Some\ \tau$

lemma *env-weakening*:
  $\Gamma \vdash e : \tau \Longrightarrow \forall\ \Gamma'.\ \Gamma \subseteq \Gamma' \wedge \textit{finite-env}\ \Gamma' \longrightarrow \Gamma' \vdash e : \tau$
  apply (*induct rule*: *wte.induct*)
  using *subseteq-def wte-var* **apply** *blast*
  using *wte-const* **apply** *blast*
  prefer *2* **using** *wte-app* **apply** *blast*
  apply (*rule allI*) **apply** (*rule impI*)
proof −
  fix $L\ \Gamma\ \sigma\ \tau\ e\ \Gamma'$
  assume *fL*: *finite L* **and** *GL*: $dom\ \Gamma \subseteq L$
    and *IH*: $\forall x.\ x \notin L \longrightarrow$
    $(\Gamma(x{\mapsto}\sigma) \vdash \{0{\to}FVar\ x\}e : \tau\ \wedge$
    $(\forall \Gamma'.\ \ \Gamma(x{\mapsto}\sigma) \subseteq \Gamma' \wedge \textit{finite-env}\ \Gamma' \longrightarrow \Gamma' \vdash \{0{\to}FVar\ x\}e : \tau))$
    and *GGP*: $\Gamma \subseteq \Gamma' \wedge \textit{finite-env}\ \Gamma'$
  let $?L = L \cup dom\ \Gamma'$
  from *GGP* **have** *finite* $(dom\ \Gamma')$ **by** *auto*
  with *fL* **have** *fL2*: *finite ?L* **by** *auto*
  { fix *x* **assume** *xL*: $x \notin\ ?L$
    from *GGP* **have** *xGxGP*: $\Gamma(x{\mapsto}\sigma) \subseteq\ \ \Gamma'(x{\mapsto}\sigma)$ **using** *subseteq-def* **by** *auto*
    from *GGP* **have** *fGP*: *finite-env* $(\Gamma'(x{\mapsto}\sigma))$ **by** *auto*
    from *xL fGP IH xGxGP* **have** $\Gamma'(x{\mapsto}\sigma) \vdash \{0{\to}FVar\ x\}e : \tau$ **by** *blast*
  } hence *X*: $\forall\ x.\ x \notin\ ?L \longrightarrow \Gamma'(x{\mapsto}\sigma) \vdash \{0{\to}FVar\ x\}e : \tau$ **by** *blast*
  have *dGL*: $dom\ \Gamma' \subseteq\ ?L$ **by** *auto*
  from *fL2 dGL X* **show** $\Gamma' \vdash (\lambda{:}\sigma.\ e) : \sigma \to \tau$ **by** (*rule wte-abs*)
qed

12.7 The substition lemma

lemma *substitution*:
$[\![\ \Gamma \vdash e1 : \tau; \Gamma\ x = Some\ \sigma; finite\text{-}env\ \Gamma\ ]\!] \Longrightarrow$
$(\forall\ \Gamma'.\ finite\text{-}env\ \Gamma' \wedge \Gamma - x \subset \Gamma' \wedge \Gamma' \vdash e2 : \sigma \longrightarrow$
$\Gamma' \vdash [x{\rightarrow}e2]e1 : \tau)$
apply (*induct rule* : *wte.induct*)
apply (*case-tac* $x = xa$) **apply** *simp*
apply *clarify* **apply** (*simp only*: *remove-bind-def*)
apply (*erule-tac* $x{=}xa$ **in** *allE*) **apply** *simp* **apply** (*rule wte-var*) **apply** *assumption*
using *wte-const* **apply** *force*
prefer *2* **apply** *clarify* **apply** *simp* **apply** (*rule wte-app*) **apply** *blast* **apply** *blast*
proof *clarify*
fix L::*nat set* **and** Γ::*env* **and** σ'::*ty* **and** $\tau\ e\ \Gamma'$
assume *fL*: *finite* L **and** $dom\ \Gamma \subseteq L$
and *IH*: $\forall xa.\ xa \notin L \longrightarrow$
$(\Gamma(xa \mapsto \sigma') \vdash \{0{\rightarrow}FVar\ xa\}e : \tau\ \wedge$
$((\Gamma(xa \mapsto \sigma'))\ x = Some\ \sigma \longrightarrow$
$finite\text{-}env\ (\Gamma(xa \mapsto \sigma')) \longrightarrow$
$(\forall \Gamma'.\ finite\text{-}env\ \Gamma' \wedge \Gamma(xa \mapsto \sigma') - x \subset \Gamma' \wedge \Gamma' \vdash e2 : \sigma \longrightarrow$
$\Gamma' \vdash [x{\rightarrow}e2](\{0{\rightarrow}FVar\ xa\}e) : \tau)))$
and *xG*: $\Gamma\ x = Some\ \sigma$ **and** *fG*: *finite-env* Γ
and *fGP*: *finite-env* Γ'
and *GxG*: $\Gamma - x \subset \Gamma'$ **and** *wte2*: $\Gamma' \vdash e2 : \sigma$
let $?L = insert\ x\ (L \cup dom\ \Gamma \cup dom\ \Gamma')$
show $\Gamma' \vdash [x{\rightarrow}e2](\lambda{:}\sigma'.\ e) : \sigma' \rightarrow \tau$
proof *simp*
show $\Gamma' \vdash (\lambda{:}\sigma'.\ [x{\rightarrow}e2]e) : \sigma' \rightarrow \tau$
proof (*rule wte-abs*[*of ?L*])
from *fL fG fGP* **show** *finite ?L* **by** *auto*
next
show $dom\ \Gamma' \subseteq ?L$ **by** *auto*
next
show $\forall xa.\ xa \notin ?L \longrightarrow \Gamma'(xa \mapsto \sigma') \vdash \{0{\rightarrow}FVar\ xa\}([x{\rightarrow}e2]e) : \tau$
proof (*rule allI*, *rule impI*)
fix x' **assume** *xL*: $x' \notin ?L$
let $?GP = \Gamma'(x' {\mapsto} \sigma')$

from *xL fGP wte2*
have *wte2b*: $?GP \vdash e2 : \sigma$
using *subseteq-def env-weakening* **by** *force*

from *xG xL wte2b fG fGP GxG IH*
have *wte*: $?GP \vdash [x{\rightarrow}e2](\{0{\rightarrow}FVar\ x'\}e) : \tau$
using *remove-bind-def* **by** *auto*

from *xL wte2b fGP*
have $\{0{\to}FVar\ x'\}([x{\to}e2]e) = [x{\to}e2](\{0{\to}FVar\ x'\}e)$
using *subst-permute* **by** *auto*

with *wte xL* **show** $?GP \vdash \{0{\to}FVar\ x'\}([x{\to}e2]e) : \tau$ **by** *auto*
qed
qed
qed
qed

12.8 Inversion rules and canonical forms

We use Isabelle's **inductive-cases** form to generate inversion rules for expressions with certain types, such as integers and functions. These rules are called "inversion" rules because they let you use the inductive definitions in reverse, going from the conclusions to the premises.

inductive-cases *wte-int-inv*: $empty \vdash e : IntT$

From the above, Isabelle generates

$$\frac{empty \vdash e : IntT \qquad \bigwedge x.\ \dfrac{None = Some\ IntT \qquad e = FVar\ x}{P} \qquad \bigwedge c.\ \dfrac{e = Const\ c \qquad IntT = TypeOf\ c}{P} \qquad \bigwedge \sigma\ e1\ e2.\ \dfrac{empty \vdash e1 : \sigma \to IntT \qquad empty \vdash e2 : \sigma \qquad e = App\ e1\ e2}{P}}{P}$$

inductive-cases *wte-fun-inv*: $empty \vdash e : \sigma \to \tau$

and Isabelle generates

$$\frac{empty \vdash e : \sigma \to \tau \qquad \bigwedge x.\ \dfrac{None = Some\ (\sigma \to \tau) \qquad e = FVar\ x}{P} \qquad \bigwedge c.\ \dfrac{e = Const\ c \qquad \sigma \to \tau = TypeOf\ c}{P} \qquad \bigwedge L\ ea.\ \dfrac{finite\ L \qquad dom\ empty \subseteq L \qquad \forall x.\ x \notin L \longrightarrow [x \mapsto \sigma] \vdash \{0{\to}FVar\ x\}ea : \tau \qquad e = \lambda{:}\sigma.\ ea}{P} \qquad \bigwedge \sigma'\ e1\ e2.\ \dfrac{empty \vdash e1 : \sigma' \to \sigma \to \tau \qquad empty \vdash e2 : \sigma' \qquad e = App\ e1\ e2}{P}}{P}$$

The following canonical forms lemmas describe what kinds of *values* have certain types. For example, the only value that has type *IntT* is an integer constant. The canonical forms lemmas are needed to prove subject reduction.

lemma *canonical-form-int*:
assumes *eint*: $empty \vdash e : IntT$ **and** *ve*: *Values e*
shows $\exists\ n.\ e = Const\ (IntC\ n)$
using *eint* **apply** (*rule wte-int-inv*)
using *ve* **apply** *auto* **apply** (*case-tac c*) **by** *auto*

lemma *canonical-form-fun*:
assumes *wtf*: *empty* $\vdash v : \sigma \rightarrow \tau$ **and** *v*: *Values v*
shows $(\exists\ e.\ v = \lambda{:}\sigma.\ e) \vee (\exists\ c.\ v = Const\ c)$
using *wtf* **apply** (*rule wte-fun-inv*) **using** *v* **by** *auto*

12.9 Subject reduction

lemma *delta-typability*:
assumes *tc*: *TypeOf* $c = \tau' \rightarrow \tau$ **and** *vt*: *empty* $\vdash v : \tau'$ **and** *vv*: *Values v*
shows $\exists\ v'.\ \delta\ c\ v = Some\ v' \wedge empty \vdash v' : \tau$
using *tc vt vv* **apply** (*cases c*) **apply** *simp* **apply** *simp*
proof –
assume *tc*: *TypeOf* $c = \tau' \rightarrow \tau$ **and** *vt*: *empty* $\vdash v : \tau'$
and *vv*: *Values v* **and** *c*: $c = Succ$
from *c tc* **have** *st*: $\tau' = IntT \wedge \tau = IntT$ **by** *simp*
from *st vt vv* **obtain** *n* **where** *v*: $v = Const\ (IntC\ n)$
apply *simp* **using** *canonical-form-int* **by** *blast*
let $?VP = Const\ (IntC\ (n + 1))$
have *wtvp*: *empty* $\vdash\ ?VP : IntT$
using *wte-const*[*of empty IntC* $(n + 1)$] **by** *auto*
from *c v* **have** *d*: $\delta\ c\ v = Some\ ?VP$ **by** *simp*
from *d wtvp st* **show** *?thesis* **by** *simp*
next
assume *tc*: *TypeOf* $c = \tau' \rightarrow \tau$ **and** *vt*: *empty* $\vdash v : \tau'$
and *vv*: *Values v* **and** *c*: $c = IsZero$
from *c tc* **have** *st*: $\tau' = IntT \wedge \tau = BoolT$ **by** *simp*
from *st vt vv* **obtain** *n* **where** *v*: $v = Const\ (IntC\ n)$
apply *simp* **using** *canonical-form-int* **by** *blast*
let $?VP = Const\ (BoolC\ (n = 0))$
have *wtvp*: *empty* $\vdash\ ?VP : BoolT$
using *wte-const*[*of empty BoolC* $(n = 0)$] **by** *auto*
from *c v* **have** *d*: $\delta\ c\ v = Some\ ?VP$ **by** *simp*
from *d wtvp st* **show** *?thesis* **by** *simp*
qed

lemma *subject-reduction*:
assumes *wte*: $\Gamma \vdash e : \tau$ **and** *g*: $\Gamma = empty$ **and** *red*: $e \longrightarrow e'$
shows *empty* $\vdash e' : \tau$
using *wte g red*
apply (*cases rule*: *wte.cases*)
apply *simp-all*
apply *force*
apply (*cases rule*: *reduces.cases*) **apply** *simp+*
apply (*cases rule*: *reduces.cases*) **apply** *simp+*
apply *clarify*

proof −
— Beta
fix Γ::*env* **and** $\sigma\ \tau'\ e1\ e2$
assume *wte1*: $empty \vdash e1 : \sigma \to \tau$
and *wte2*: $empty \vdash e2 : \sigma$
and *red*: $App\ e1\ e2 \longrightarrow e'$
— Would be cleaner to use an inductive cases for the above 'red'
from *red* **show** $empty \vdash e' : \tau$
proof (*cases rule*: *reduces.cases*)
fix $\tau''\ b\ v$ **assume** *a*: $(App\ e1\ e2, e') = (App\ (\lambda{:}\tau''.\ b)\ v, \{0 \to v\}b)$
and *vv*: *Values v*
have *fe*: *finite* {} **by** *simp*
have *xL*: $(0{::}nat) \notin \{\}$ **by** *simp*
from *wte1 fe a xL* **obtain** *L*
where *fL*: *finite L*
and *wtb*: $\forall x.\ x \notin L \longrightarrow [x \mapsto \sigma] \vdash \{0 \to FVar\ x\}b : \tau$
apply (*cases rule*: *wte.cases*) **by** *auto*
let $?X = Suc\ (max\ (setmax\ L)\ (setmax\ (FV\ b)))$
have *xgel*: $setmax\ L < ?X$ **by** *auto*
have *xgeb*: $setmax\ (FV\ b) < ?X$ **by** *auto*

— Set up for and apply the substitution lemma
from *fL xgel* **have** *xL*: $?X \notin L$ **by** (*rule greaterthan-max-is-fresh*)
with *wtb* **have** *wtb2*: $[?X \mapsto \sigma] \vdash \{0 \to FVar\ ?X\}b : \tau$ **by** *blast*
have *gxs*: $[?X \mapsto \sigma]\ ?X = Some\ \sigma$ **by** *simp*
have *fg*: *finite-env* $[?X \mapsto \sigma]$ **by** *simp*
have *fgp*: *finite-env empty* **by** *simp*
have *gxgp*: $[?X \mapsto \sigma] - ?X \subset empty$ **by** (*simp add*: *remove-bind-def*)
from *wtb2 gxs fg fgp gxgp wte2*
have *wtb*: $empty \vdash [?X \to e2](\{0 \to FVar\ ?X\}b) : \tau$
using *substitution* **by** *blast*

— Use the substitution decomposition lemma
have *finb*: *finite* (*FV b*) **by** (*rule finite-FV*)
from *finb xgeb* **have** *xb*: $?X \notin FV\ b$ **by** (*rule greaterthan-max-is-fresh*)
from *xb* **have** $\{0 \to e2\}b = [?X \to e2](\{0 \to FVar\ ?X\}b)$
by (*rule decompose-subst*)
with *wtb a* **show** $empty \vdash e' : \tau$ **by** *simp*

next — Delta
fix $c\ v\ v'$
assume *a*: $(App\ e1\ e2, e') = (App\ (Const\ c)\ v, v')$
and *d*: $\delta\ c\ v = Some\ v'$ **and** *vv*: *Values v*
from *wte1 a* **have** *tc*: $TypeOf\ c = \sigma \to \tau$
apply (*cases rule*: *wte.cases*) **by** *auto*

from *a tc wte2 vv* **obtain** v'' **where** *dd*: $\delta\ c\ v = Some\ v''$
and *wtvp*: $empty \vdash v'' : \tau$ **using** *delta-typability* **by** *blast*
from *wtvp a d dd* **show** $empty \vdash e' : \tau$ **by** *simp*
qed
qed

12.10 Decomposition

consts *welltyped-ctx* :: $(env \times ctx \times ty \times ty)$ *set*
syntax *welltyped-ctx* :: $env \Rightarrow ctx \Rightarrow ty \Rightarrow ty \Rightarrow bool$ (- $\vdash$ - : -$\Rightarrow$- [52,52,52,52] 51)
translations $\Gamma \vdash E : \sigma \Rightarrow \tau == (\Gamma, E, \sigma, \tau) \in$ *welltyped-ctx*
inductive *welltyped-ctx* **intros**
WTHole: $\Gamma \vdash Hole : \tau \Rightarrow \tau$
WTAppL: $[\![\Gamma \vdash E : \sigma \Rightarrow (\varrho \to \tau); \Gamma \vdash e : \varrho]\!]$
$\Longrightarrow \Gamma \vdash AppL\ E\ e : \sigma \Rightarrow \tau$
WTAppR: $[\![\Gamma \vdash e : \varrho \to \tau; \Gamma \vdash E : \sigma \Rightarrow \varrho]\!]$
$\Longrightarrow \Gamma \vdash AppR\ e\ E : \sigma \Rightarrow \tau$

lemma *welltyped-decomposition*:
$\Gamma \vdash e : \tau \Longrightarrow$
$\Gamma = empty \longrightarrow Values\ e \lor (\exists\ \sigma\ E\ r.\ e = E[r] \land \Gamma \vdash E : \sigma \Rightarrow \tau \land E \in wf\text{-}ctx$
$\land \Gamma \vdash r : \sigma \land redex\ r)$
(**is** $\Gamma \vdash e : \tau \Longrightarrow ?P\ \Gamma\ e\ \tau$)
apply (*induct rule*: *wte.induct*)
apply *simp* **apply** *simp* **apply** *simp* **apply** (*rule impI*)
proof –
fix $\Gamma\ \sigma\ \tau\ e1\ e2$
assume *wte1*: $\Gamma \vdash e1 : \sigma \to \tau$ **and** *IH1*: $?P\ \Gamma\ e1\ (\sigma{\to}\tau)$
and *wte2*: $\Gamma \vdash e2 : \sigma$ **and** *IH2*: $?P\ \Gamma\ e2\ \sigma$ **and** *g*: $\Gamma = empty$
show *Values* (*App e1 e2*) $\lor$
$(\exists \sigma\ E\ r.\ App\ e1\ e2 = E[r] \land \Gamma \vdash E : \sigma{\Rightarrow}\tau \land E \in wf\text{-}ctx \land \Gamma \vdash r : \sigma \land redex\ r)$
proof (*cases Values e1*)
assume *ve1*: *Values e1*
show *?thesis*
proof (*cases Values e2*)
assume *ve2*: *Values e2*
have *h*: *App e1 e2* = *Hole*[*App e1 e2*] **by** *simp*
have *wth*: $empty \vdash Hole : \tau{\Rightarrow}\tau$ **by** (*rule WTHole*)
from *wte1 wte2 g* **have** *wta*: $empty \vdash App\ e1\ e2 : \tau$
apply *simp* **by** (*rule wte-app*)

from *wte1 ve1 g* **have** $(\exists\ e.\ e1 = \lambda{:}\sigma.\ e) \lor (\exists\ c.\ e1 = Const\ c)$
apply *simp* **apply** (*rule canonical-form-fun*) **by** *auto*
moreover { **assume** *x*: $\exists\ e.\ e1 = \lambda{:}\sigma.\ e$
— Beta

```
      from x obtain b where e1: e1 = λ:σ. b by blast
      from e1 ve2 have App e1 e2 −→ {0→e2}b apply simp by (rule Beta)
      hence r: redex (App e1 e2) using redex-def by blast
      have wfh: Hole ∈ wf-ctx by (rule WFHole)
      from h wth wfh wta r g have ?thesis by blast
    } moreover { assume x: ∃ c. e1 = Const c
        — Delta
      from x obtain c where e1: e1 = Const c by blast
      from wte1 e1 have tc: TypeOf c = σ → τ
        apply (cases rule: wte.cases) by auto
      from tc wte2 ve2 g obtain v'' where dd: δ c e2 = Some v''
        using delta-typability by blast
      from dd ve2 e1 have App e1 e2 −→ v'' apply simp by (rule Delta)
      hence r: redex (App e1 e2) using redex-def by blast
      have wfh: Hole ∈ wf-ctx by (rule WFHole)
      with h wth wfh wta r g have ?thesis by blast
    } ultimately show ?thesis by blast
  next
    assume ve2: ¬ Values e2
    from ve2 IH2 g obtain σ' E r where e2: e2 = E[r]
      and wtE: Γ ⊢ E : σ'⇒σ and wfE: E ∈ wf-ctx
      and wtr: Γ ⊢ r : σ' and rr: redex r
      by blast
    from e2 have App e1 e2 = (AppR e1 E)[r] by simp
    moreover from wte1 wtE g have empty ⊢ AppR e1 E : σ' ⇒ τ
      apply simp apply (rule WTAppR) apply auto done
    moreover from ve1 wfE have AppR e1 E ∈ wf-ctx by (rule WFAppR)
    moreover note wtr rr g
    ultimately show ?thesis by blast
  qed
 next
  assume ve1: ¬ Values e1
  from ve1 IH1 g obtain σ' E r where e1: e1 = E[r]
    and wtE: Γ ⊢ E : σ'⇒σ→τ and wfE: E ∈ wf-ctx and wtr: Γ ⊢ r : σ' and rr: redex r
    by blast
  from e1 have App e1 e2 = (AppL E e2)[r] by simp
  moreover from wtE wte2 g have empty ⊢ AppL E e2 : σ' ⇒ τ
    apply simp apply (rule WTAppL) apply auto done
  moreover from wfE have AppL E e2 ∈ wf-ctx by (rule WFAppL)
  moreover note wtr rr g
  ultimately show ?thesis by blast
 qed
qed
```

lemma *welltyped-expr-ctx-impl*:

$\Gamma \vdash e : \tau \Longrightarrow \forall\ E\, r.\ e = E[r]$
$\longrightarrow (\exists\ \sigma.\ \Gamma \vdash E : \sigma \Rightarrow \tau \wedge \Gamma \vdash r : \sigma)$
apply (*induct rule*: *wte.induct*)
apply *clarify*
apply (*rule-tac x*=τ **in** *exI*)
apply (*case-tac E*)
using *wte-var WTHole* **apply** *force*
apply *simp* **apply** *simp*
apply *clarify*
apply (*rule-tac x*=*TypeOf c* **in** *exI*)
apply (*case-tac E*) **using** *wte-const WTHole* **apply** *force*
apply *simp* **apply** *simp*
apply *clarify*
apply (*case-tac E*)
apply (*rule-tac x*=$\sigma \rightarrow \tau$ **in** *exI*)
apply *simp* **using** *wte-abs WTHole* **apply** *force*
apply *simp* **apply** *simp*
apply *clarify*
apply (*case-tac E*)
apply (*rule-tac x*=τ **in** *exI*) **using** *wte-app WTHole* **apply** *force*
apply (*erule-tac x*=*ctx* **in** *allE*)
apply (*erule-tac x*=*ctx* **in** *allE*)
apply (*erule-tac x*=*r* **in** *allE*)
apply (*erule-tac x*=*r* **in** *allE*)
apply *simp* **using** *WTAppL* **apply** *blast*
apply (*erule-tac x*=*ctx* **in** *allE*)
apply (*erule-tac x*=*ctx* **in** *allE*)
apply (*erule-tac x*=*r* **in** *allE*)
apply (*erule-tac x*=*r* **in** *allE*)
apply *simp* **using** *WTAppR* **apply** *blast*
done

lemma *welltyped-expr-ctx*:
$\Gamma \vdash E[r] : \tau \Longrightarrow \exists\ \sigma.\ \Gamma \vdash E : \sigma \Rightarrow \tau \wedge \Gamma \vdash r : \sigma$
using *welltyped-expr-ctx-impl* **by** *simp*

lemma *fill-ctx-welltyped*[*rule-format*]:
$\Gamma \vdash E : \sigma \Rightarrow \tau \Longrightarrow \forall\ r.\ \Gamma \vdash r : \sigma \longrightarrow \Gamma \vdash \mathit{fill}\ E\ r : \tau$
apply (*induct rule*: *welltyped-ctx.induct*)
apply *simp*
using *wte-app* **apply** *force*
using *wte-app* **apply** *force*
done

12.11 Progress and preservation

lemma *progress*:
 assumes *wte*: $empty \vdash e : \tau$
 shows $Values\ e \lor (\exists\ e'.\ e \longmapsto e')$
proof −
 show *?thesis*
 proof (*cases Values e*)
  assume *Values e* **thus** *?thesis* **by** *simp*
 next assume $\neg$ *Values e*
  with *wte* **have** *x*: $\exists\ \sigma\ E\ r.\ e = E[r] \land empty \vdash E : \sigma \Rightarrow \tau \land E \in wf\text{-}ctx$
    $\land\ empty \vdash r : \sigma\ \ \land redex\ r$
   using *welltyped-decomposition*[*of empty e* τ] **by** *simp*
  from *x* **obtain** σ *E r* **where** *eE*: $e = E[r]$ **and** *wtc*: $empty \vdash E : \sigma \Rightarrow \tau$
   and *wfE*: $E \in wf\text{-}ctx$ **and** *wtr*: $empty \vdash r : \sigma$ **and** *rr*: *redex r*
   by *blast*
  from *rr* **obtain** r' **where** *red*: $r \longrightarrow r'$ **using** *redex-def* **by** *blast*
  from *wfE red* **have** $E[r] \longmapsto E[r']$ **by** (*rule Step*)
  with *eE* **show** *?thesis* **by** *blast*
 qed
qed

lemma *preservation*:
 assumes *s*: $e \longmapsto e'$
 and *wte*: $empty \vdash e : \tau$
 shows $empty \vdash e' : \tau$
using *s*
proof (*cases rule*: *eval-step.cases*)
 fix $E\ r\ r'$
 assume *a*: $(e, e') = (E[r], E[r'])$
  and *wfE*: $E \in wf\text{-}ctx$
  and *rr*: $r \longrightarrow r'$
 from *a wte* **obtain** σ **where** *wtc*: $empty \vdash E : \sigma \Rightarrow \tau$
  and *wtr*: $empty \vdash r : \sigma$ **using** *welltyped-expr-ctx* **by** *blast*
 from *wtr rr*
 have *wtrp*: $empty \vdash r' : \sigma$ **using** *subject-reduction* **by** *blast*
 from *wtc wtrp* **have** $empty \vdash fill\ E\ r' : \tau$ **by** (*rule fill-ctx-welltyped*)
 with *a* **show** *?thesis* **by** *simp*
qed

12.12 Type safety

constdefs *finished* :: $expr \Rightarrow bool$
 $finished\ e \equiv \neg(\exists\ e'.\ e \longmapsto e')$

syntax *eval-step-rtrancl* :: $expr \Rightarrow expr \Rightarrow bool$ (**infixl** $\longmapsto^*$ *51*)

translations $e \longmapsto^* e' == (e,e') \in$ *eval-step**

theorem *type-safety*:
 assumes *et*: $empty \vdash e : \tau$
 and *ee*: $e \longmapsto^* e'$
 shows $empty \vdash e' : \tau \wedge (Values\ e' \vee \neg\ (finished\ e'))$
 using *ee et*
proof (*induct rule*: *rtrancl.induct*)
 fix *a* **assume** *wta*: $empty \vdash a : \tau$
 from *wta* **have** $Values\ a \vee (\exists\ e'.\ a \longmapsto e')$ **by** (*rule progress*)
 with *wta* **show** $empty \vdash a : \tau \wedge (Values\ a \vee \neg\ (finished\ a))$
  using *finished-def* **by** *auto*
next
 fix *a b c*
 assume *IH*: $empty \vdash a : \tau \Longrightarrow empty \vdash b : \tau \wedge (Values\ b \vee \neg(finished\ b))$
  and *bc*: $b \longmapsto c$ **and** *wta*: $empty \vdash a : \tau$
 from *wta IH* **have** *wtb*: $empty \vdash b : \tau$ **by** *simp*
 from *bc wtb* **have** *wtc*: $empty \vdash c : \tau$ **by** (*rule preservation*)
 from *wtc* **have** $Values\ c \vee (\exists\ e'.\ c \longmapsto e')$ **by** (*rule progress*)
 with *wtc* **show** $empty \vdash c : \tau \wedge (Values\ c \vee \neg\ (finished\ c))$
  using *finished-def* **by** *auto*
qed

13 Total recursive functions

Isabelle's **recdef** facility let you write functions without syntax restrictions on the recursion pattern (as with **primrec**). However, you must provide the termination measure. That is, you must provide a function that maps the input of your recursive function to an element of a well-founded set, such as the natural numbers, and show that these elements decrease for each recursive call.

13.1 The Fibonacci function

The following is a simple example of a recursive function, the Fibonacci function.

consts *fib* :: $nat \Rightarrow nat$
recdef *fib* $measure(\lambda\ n.\ n)$
 $fib\ 0 = 0$
 $fib\ (Suc\ 0) = 1$
 $fib\ (Suc\ (Suc\ x)) = fib\ x + fib\ (Suc\ x)$

thm *fib.induct*

lemma *fib* (*Suc* (*Suc* (*Suc* (*Suc 0*)))) = *3* **by** *simp*

13.2 Case study: Euclid's Algorithm

consts *compute-gcd* :: *nat* × *nat* ⇒ *nat*
recdef *compute-gcd measure*(λ(*m*,*n*). *n*)
compute-gcd(*m*, *n*) = (*if n* = *0 then m else compute-gcd*(*n*, *m mod n*))

thm *compute-gcd.induct*

constdefs *divisible-by* :: *nat* ⇒ *nat* ⇒ *bool* (- | - [*80*,*80*] *79*)
divisible-by m n ≡ (∃ *k*. *m* = *n* ∗ *k*)
declare *divisible-by-def*[*simp*]

constdefs *isGCD* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool*
isGCD k m n ≡ *m*|*k* ∧ *m*|*k* ∧ (∀ *k*′. *m*|*k*′ ∧ *n*|*k*′ ⟶ *k*|*k*′)
declare *isGCD-def*[*simp*]

theorem *isGCD* (*compute-gcd*(*m*,*n*)) *m n*
proof (*induct rule*: *compute-gcd.induct*)
  fix *m n*
  assume *IH*: *n* ≠ *0* ⟶ (*isGCD* (*compute-gcd* (*n*, *m mod n*)) *n* (*m mod n*))
  show *isGCD* (*compute-gcd* (*m*, *n*)) *m n*
  proof (*case-tac n* = *0*)
    assume *n* = *0* **thus** *?thesis* **by** *simp*
  next
    assume *N*: *n* ≠ *0*
    from *N IH* **have** *isGCD* (*compute-gcd* (*n*, *m mod n*)) *n* (*m mod n*)
      by *simp*
    oops

13.3 Merge sort

The goal of merge sort, of course, is to produce a sorted list.

consts *sorted* :: *nat list* ⇒ *bool*
primrec
sorted [] = *True*
sorted (*x*#*xs*) = ((∀ *y* ∈ *set xs*. *x* ≤ *y*) ∧ *sorted xs*)

The merge sort function will use the following auxiliary function to merge already sorted sub-lists. When using the **recdef** facility, the recursive function must have a single parameter but that parameter may be a tuple.

consts *merge* :: *nat list* $*$ *nat list* $\Rightarrow$ *nat list*
recdef *merge measure*$(\lambda(xs,ys).\ size\ xs + size\ ys)$
  $merge(x\#xs,\ y\#ys) =$
    $(if\ x \le y\ then\ x\ \#\ merge(xs,\ y\#ys)\ else\ y\ \#\ merge(x\#xs,\ ys))$
  $merge(xs,[]) = xs$
  $merge([],ys) = ys$

Isabelle generates a special purpose induction rule for each recursive function. Compare the following rule to the definition of merge.

$$\frac{\bigwedge x\ xs\ y\ ys.\ \dfrac{\neg\ x \le y \longrightarrow P\ (x{\cdot}xs)\ ys \qquad x \le y \longrightarrow P\ xs\ (y{\cdot}ys)}{P\ (x{\cdot}xs)\ (y{\cdot}ys)} \\ P\ []\ [] \qquad \bigwedge w\ z.\ P\ (w{\cdot}z)\ [] \qquad \bigwedge ac\ ad.\ P\ []\ (ac{\cdot}ad)}{P\ u\ v}$$

lemma *set-merge*[*simp*]: $set(merge(xs,ys)) = set\ xs \cup set\ ys$
  apply(*induct xs ys rule*: *merge.induct*)
  apply *auto*
  done

If the inputs to merge are sorted, then so is the output (and vice-versa).

lemma *sorted-merge*[*simp*]:
    $sorted\ (merge(xs,ys)) = (sorted\ xs \wedge sorted\ ys)$
  apply(*induct xs ys rule*: *merge.induct*)
  apply(*simp-all add*: *ball-Un linorder-not-le order-less-le*)
  apply(*blast intro*: *order-trans*)
  done

Here's the definition of merge sort.

consts *msort* :: *nat list* $\Rightarrow$ *nat list*
recdef *msort measure size*
    $msort\ [] = []$
    $msort\ [x] = [x]$
    $msort\ xs = merge(msort(take\ (size\ xs\ div\ 2)\ xs),$
                $msort(drop\ (size\ xs\ div\ 2)\ xs))$

The induction rule for msort is

$$\frac{P\ [] \qquad \bigwedge x.\ P\ [x] \\ \bigwedge u\ z\ aa.\ \dfrac{P\ (drop\ (|u{\cdot}z{\cdot}aa|\ div\ 2)\ (u{\cdot}z{\cdot}aa)) \qquad P\ (take\ (|u{\cdot}z{\cdot}aa|\ div\ 2)\ (u{\cdot}z{\cdot}aa))}{P\ (u{\cdot}z{\cdot}aa)}}{P\ x}$$

theorem *sorted-msort*: $sorted\ (msort\ xs)$
  by (*induct xs rule*: *msort.induct*) *simp-all*

13.4 Substitution and strong induction

We define the explicitly α-renaming version of substitution á la Curry [1, 5] using the **recdef** facility. The proof of termination relies on a proof by strong induction, an extremely general and powerful induction principle.

datatype *expr*
  $= \mathit{Var}\ \mathit{nat}$
  $\mid \mathit{Lam}\ \mathit{nat}\ \mathit{expr}$ (λ -. - [53,53] 52)
  $\mid \mathit{App}\ \mathit{expr}\ \mathit{expr}$

To be completely concrete (and computable), we choose fresh variables by computing the largest variable in the relevant terms and add 1, thereby guaranteeing that the new variable does not occur in these expressions.

consts $\mathit{maxv} :: \mathit{expr} \Rightarrow \mathit{nat}$
primrec
  $\mathit{maxv}\ (\mathit{Var}\ x) = x$
  $\mathit{maxv}\ (\lambda\, x.\ e) = \mathit{max}\ (\mathit{maxv}\ e)\ x$
  $\mathit{maxv}\ (\mathit{App}\ e_1\ e_2) = \mathit{max}\ (\mathit{maxv}\ e_1)\ (\mathit{maxv}\ e_2)$

constdefs $\mathit{fresh} :: \mathit{nat} \Rightarrow \mathit{expr} \Rightarrow \mathit{expr} \Rightarrow \mathit{nat}$
  $\mathit{fresh}\ x\ e\ e' \equiv (\mathit{max}\ (\mathit{max}\ (\mathit{maxv}\ e')\ x)\ (\mathit{maxv}\ e)) + 1$

Here's the definition of substitution. We label each clause so that we can used them as simplification rules.

consts $\mathit{subst} :: (\mathit{expr} \times \mathit{nat} \times \mathit{expr}) \Rightarrow \mathit{expr}$
syntax $\mathit{subst} :: \mathit{nat} \Rightarrow \mathit{expr} \Rightarrow \mathit{expr} \Rightarrow \mathit{expr}$ ([-:=-]- [100,100,100] 101)
translations $[x{:=}e']e == \mathit{subst}(e,x,e')$
recdef (**permissive**) $\mathit{subst}\ \mathit{measure}\ (\lambda\, p.\ \mathit{size}\ (\mathit{fst}\ p))$
  svar: $[x{:=}e](\mathit{Var}\ y) = (\mathit{if}\ y = x\ \mathit{then}\ e\ \mathit{else}\ \mathit{Var}\ y)$
  slam: $[x{:=}e](\lambda\, y.\ e') = (\mathit{let}\ z = \mathit{fresh}\ x\ e\ e'\ \mathit{in}\ \lambda\, z.\ [x{:=}e]([y{:=}\mathit{Var}\ z]e'))$
  sapp: $[x{:=}e](\mathit{App}\ e_1\ e_2) = \mathit{App}\ ([x{:=}e]e_1)\ ([x{:=}e]e_2)$

The use of **permissive** tells Isabelle not to immediately abort, but instead accept the *subst* function conditionally. Isabelle accepts a modified form of the *subst* function that includes extra 'if' statements to make sure that it terminates.

$$[x{:=}e]\mathit{Var}\ y = \mathsf{if}\ y = x\ \mathsf{then}\ e\ \mathsf{else}\ \mathit{Var}\ y$$
$$[x{:=}e](\lambda\, y.\ e') = \mathsf{let}\ z = \mathit{fresh}\ x\ e\ e'\ \mathsf{in}\ \lambda\, z.\ (\mathsf{if}\ \mathit{size}\ ([y{:=}\mathit{Var}\ z]e') < \mathit{Suc}\ (\mathit{size}\ e')\ \mathsf{then}\ [x{:=}e][y{:=}\mathit{Var}\ z]e'\ \mathsf{else}\ \mathit{arbitrary})$$
$$[x{:=}e]\mathit{App}\ e_1\ e_2 = \mathit{App}\ ([x{:=}e]e_1)\ ([x{:=}e]e_2)$$

The *response* window tells us that Isabelle could not prove termination and where it got stuck. We then create a lemma, slightly generalizing from the stuck proof state. The following lemma says that substituting a variable for a variable does not change the size of an expression. The proof cannot be done by structural induction on the expression because the nested substitution changes the expression, so the induction hypothesis is not applicable. Instead we use strong induction (aka course of values induction) on the size of expressions. With this style of induction, the induction hypothesis is applicable to any expression smaller than the current one. The following is the rule for strong induction.

$$\frac{\bigwedge n.\ \dfrac{\forall\, m<n.\ P\ m}{P\ n}}{P\ n}$$

lemma *alpha-subst-size*[*simp*]: $\forall\ x\ w\ e.\ size\ e = n \longrightarrow size\ ([x{:=}Var\ w]e) = n$
proof (*induct rule*: *nat-less-induct*)
  fix n
  assume *IH*: $\forall\, m<n.\ \forall\, x\ w\ e.\ size\ e = m \longrightarrow size\ ([x{:=}Var\ w]e) = m$
  show $\forall\, x\ w\ e.\ size\ e = n \longrightarrow size\ ([x{:=}Var\ w]e) = n$
  proof ((*rule allI*)+, *rule impI*)
    fix x **and** w **and** $e{::}expr$ **assume** *se*: $size\ e = n$
    show $size\ ([x{:=}Var\ w]e) = n$
    proof (*cases e*)
      fix y **assume** $e = Var\ y$ **thus** $size\ ([x{:=}Var\ w]e) = n$ **using** *se* **by** (*simp add*: *svar*)
    next
      fix $x'\ \tau\ e'$
      assume *E*: $e = \lambda\, x'.\ e'$
      let $?W = (max\ (max\ (maxv\ e')\ x)\ w) + 1$
      from *E se* **have** $Suc\ (size\ e') = n$ **by** *simp*
      with *IH* **have** *EP*: $Suc\ (size\ ([x'{:=}Var\ ?W]e')) = n$ **by** *auto*
      from *se EP E* **have** *EP2*: $size\ ([x'{:=}Var\ ?W]e') < Suc\ (size\ e')$ **by** *auto*
      from *EP IH* **have** $Suc\ (size\ ([x{:=}Var\ w]([x'{:=}Var\ ?W]e'))) = n$ **by** *auto*
      with *E EP2* **show** $size\ ([x{:=}Var\ w]e) = n$ **by** (*simp add*: *slam fresh-def*)
    next
      fix $e1\ e2$ **assume** *AP*: $e = App\ e1\ e2$
      from *AP se* **have** $size\ e1 < n$ **by** *auto*
      with *IH* **have** *E1*: $size\ ([x{:=}Var\ w]e1) = size\ e1$ **by** *auto*
      from *AP se* **have** $size\ e2 < n$ **by** *auto*
      with *IH* **have** *E2*: $size\ ([x{:=}Var\ w]e2) = size\ e2$ **by** *auto*
      from *AP E1 E2* **have** $size\ ([x{:=}Var\ w]e) = size\ e$ **by** (*simp add*: *sapp*)
      with *se* **show** $size\ ([x{:=}Var\ w]e) = n$ **by** *simp*
    qed
  qed
qed

With the above lemma established, we can resolve the termination conditions and update the simplification rules for the *subst* function.

recdef-tc *subst* (*1*) **by** *simp*
lemmas *subst-simps*[*simp*] = *subst.simps*[*simplified*]

lemma *subst-lam*: $z = fresh\ x\ e\ e' \Longrightarrow [x{:=}e](\lambda\, y.\ e') = (\lambda\, z.\ [x{:=}e][y{:=}Var\ z]e')$
  by (*simp add*: *fresh-def*)

13.5 Depth-First Search

typedecl *node*
types *graph* = (*node* $*$ *node*) *list*

consts
adj :: [*graph*, *node*] => *node list*
primrec
adj [] *n* = []
adj (*e*#*es*) *n* = (*if fst e* = *n then snd e* # *adj es n else adj es n*)

constdefs
adjs :: [*graph*, *node list*] => *node set*
adjs g xs $\equiv$ *set g* “ *set xs*

lemma *adj-set*: *y* $\in$ *set* (*adj g x*) = ((*x*,*y*) $\in$ *set g*)
by (*induct g*, *auto*)

lemma *adjs-Cons*: *adjs g* (*x*#*xs*) = *set* (*adj g x*) $\cup$ *adjs g xs*
by(*unfold adjs-def* ,*auto simp add*:*Image-def adj-set*)

constdefs
reachable :: [*graph*, *node list*] $\Rightarrow$ *node set*
reachable g xs $\equiv$ (*set g*)* “ *set xs*

constdefs
nodes-of :: *graph* $\Rightarrow$ *node set*
nodes-of g $\equiv$ *set* (*map fst g* @ *map snd g*)

lemma [*rule-format*, *simp*]: *x* $\notin$ *nodes-of g* $\longrightarrow$ *adj g x* = []
by (*induct g*, *auto simp add*: *nodes-of-def*)

constdefs
dfs-rel :: ((*graph* $*$ *node list* $*$ *node list*) $*$ (*graph* $*$ *node list* $*$ *node list*)) *set*
dfs-rel $\equiv$ *inv-image* (*finite-psubset* <$*$*lex*$*$> *less-than*)
(λ(*g*,*xs*,*ys*). (*nodes-of g* $-$ *set ys*, *size xs*))

lemma *dfs-rel-wf*: *wf dfs-rel*
by (*auto simp add*: *dfs-rel-def wf-finite-psubset*)

lemma [*simp*]: *finite* (*nodes-of g* $-$ *set ys*)
proof(*rule finite-subset*)
show *finite* (*nodes-of g*)
by (*auto simp add*: *nodes-of-def*)
qed (*auto*)

consts
dfs :: [*graph* $*$ *node list* $*$ *node list*] $\Rightarrow$ *node list*
recdef (**permissive**) *dfs dfs-rel*
dfs-base[*simp*]: *dfs* (*g*, [], *ys*) = *ys*
dfs-inductive: *dfs* (*g*, *x*#*xs*, *ys*) = (*if x mem ys then dfs* (*g*, *xs*, *ys*)
else dfs (*g*, *adj g x*@*xs*, *x*#*ys*))
(**hints** *recdef-simp add*: *dfs-rel-def finite-psubset-def recdef-wf add*: *dfs-rel-wf*)

- The second argument of *dfs* is a stack of nodes that will be visited.
- The third argument of *dfs* is a list of nodes that have been visited already.

recdef-tc *dfs-tc*: *dfs*
proof (*intro allI*)
fix *g x ys*
show $\neg$ *x mem ys* $\longrightarrow$
nodes-of g $-$ *insert x* (*set ys*) $\subset$ *nodes-of g* $-$ *set ys* $\vee$
nodes-of g $-$ *insert x* (*set ys*) = *nodes-of g* $-$ *set ys* $\wedge$ *adj g x* = []
by (*cases* $x \in$ *nodes-of g*, *auto simp add*: *mem-iff*)
qed

lemmas *dfs-induct* = *dfs.induct*[*OF dfs-tc*]
lemmas *dfs-inductive*[*simp*] = *dfs-inductive*[*OF dfs-tc*]

To do: proof of correctness.

13.6 Notes

The material on merge sort is from the `HOL/ex/MergeSort.thy` example from the Isabelle distribution.
The material on Depth-First Search is from [9].

14 METATHEORY OF PROPOSITIONAL LOGIC

We formalize the meaning of propositional formulas and define a proof system. We prove completeness of the proof system via Kalmar's variable elimination method. The material in this section is based on several texts on Mathematical Logic [2, 8] and Paulson's completeness proof in Isabelle/ZF [10].

14.1 Formulas and their meaning

datatype *formula*
  = *Atom nat*
  | *Neg formula*
  | *Implies formula formula*   (**infixl** $\rightarrow$ *101*)

consts *eval* :: (*nat* $\Rightarrow$ *bool*) $\Rightarrow$ *formula* $\Rightarrow$ *bool*
primrec
  eval v (*Atom a*) = *v a*
  eval v (*Neg* φ) = ($\neg$ (*eval v* φ))
  eval v ($\varphi \rightarrow \psi$) = ((*eval v* φ) $\longrightarrow$ *eval v* ψ)

constdefs *tautology* :: *formula* $\Rightarrow$ *bool*
  tautology $\varphi \equiv (\forall$ *v*. *eval v* φ)
  satisfies :: (*nat* $\Rightarrow$ *bool*) $\Rightarrow$ *formula set* $\Rightarrow$ *bool* (- *sats* - [*80*,*80*] *80*)
  v sats $\Sigma \equiv (\forall\ \varphi \in \Sigma$. *eval v* φ)
  satisfiable :: *formula set* $\Rightarrow$ *bool*
  satisfiable $\Sigma \equiv (\exists$ *v*. *v sats* Σ)
  implies :: *formula set* $\Rightarrow$ *formula* $\Rightarrow$ *bool* (- $\models$ - [*80*,*80*] *80*)
  $\Sigma \models \varphi \equiv (\forall$ *v*. *v sats* $\Sigma \longrightarrow$ *eval v* φ = *True*)

14.2 Axioms and proofs

constdefs
  A1 :: *formula*
  A1 $\equiv$ *Atom 0* $\rightarrow$ (*Atom 1* $\rightarrow$ *Atom 0*)
  A2 :: *formula*
  A2 $\equiv$ (((*Atom 0*) $\rightarrow$ (*Atom 1* $\rightarrow$ *Atom 2*)) $\rightarrow$
    (((*Atom 0*) $\rightarrow$ (*Atom 1*)) $\rightarrow$ ((*Atom 0*) $\rightarrow$ (*Atom 2*))))
  A3 :: *formula*
  A3 $\equiv$ (((*Neg* (*Atom 1*)) $\rightarrow$ (*Neg* (*Atom 0*)))
      $\rightarrow$ (((*Neg* (*Atom 1*)) $\rightarrow$ *Atom 0*) $\rightarrow$ *Atom 1*))
  Axioms :: *formula set*
  Axioms $\equiv$ { *A1*, *A2*, *A3* }
declare *A1-def*[*simp*] *A2-def*[*simp*] *A3-def*[*simp*] *Axioms-def*[*simp*]

lemma *tautology A1* **by** (*simp add*: *tautology-def*)
lemma *tautology A2* **by** (*simp add*: *tautology-def*)
lemma *tautology A3* **by** (*simp add*: *tautology-def*)

consts *subst* :: (*nat* $\Rightarrow$ *formula*) $\Rightarrow$ *formula* $\Rightarrow$ *formula*
primrec
  subst S (*Atom x*) = (*S x*)
  subst S (*Neg f*) = *Neg* (*subst S f*)
  subst S (*f1* $\rightarrow$ *f2*) = (*subst S f1*) $\rightarrow$ (*subst S f2*)

consts *deduction* :: (*formula set* $\times$ *formula*) *set*
syntax *deduction* :: *formula set* $\Rightarrow$ *formula* $\Rightarrow$ *bool* (- $\vdash$ - [*100*,*100*] *100*)
translations $\Sigma \vdash \varphi$ == $(\Sigma,\varphi) \in$ *deduction*
inductive *deduction* **intros**
hyp: $[\![\varphi \in \Sigma]\!] \Longrightarrow \Sigma \vdash \varphi$
ax: $\varphi \in$ *Axioms* $\Longrightarrow \Sigma \vdash$ *subst S* φ
mp: $[\![\Sigma \vdash (\varphi \to \psi); \Sigma \vdash \varphi]\!] \Longrightarrow \Sigma \vdash \psi$

constdefs *emp* :: *nat* $\Rightarrow$ *formula*
emp $\equiv$ (λ *x*. (*Atom x*))

14.3 Basic properties of the proof system

lemma *aa*: $\Gamma \vdash (\varphi \to \varphi)$
proof –
let *?S0* = ((*emp*(*0*:=φ))(*1*:=($\varphi\to\varphi$)))(*2*:=φ)
let *?S1* = (*emp*(*0*:=φ))(*1*:=(φ))
have *p1*: $\Gamma \vdash$ *subst ?S0 A2* **apply** (*rule ax*) **by** *simp*
have *p2*: $\Gamma \vdash$ *subst ?S0 A1* **apply** (*rule ax*) **by** *simp*
have *p3*: $\Gamma \vdash ((\varphi \to (\varphi \to \varphi)) \to (\varphi \to \varphi))$
using *p1 p2* **apply** *simp* **apply** (*rule mp*) **apply** *blast* **apply** *blast* **done**
have *p4*: $\Gamma \vdash$ *subst ?S1 A1* **apply** (*rule ax*) **by** *simp*
show $\Gamma \vdash (\varphi \to \varphi)$
using *p3 p4* **apply** *simp* **apply** (*rule mp*) **apply** *blast* **apply** *blast* **done**
qed

lemma *weakening*[*rule-format*]: $\Gamma \vdash \varphi \Longrightarrow (\forall\ \Delta.\ \Gamma \subseteq \Delta \longrightarrow \Delta \vdash \varphi)$
apply (*induct rule*: *deduction.induct*)
apply *clarify* **using** *hyp* **apply** *blast*
using *ax* **apply** *blast*
apply *clarify* **apply** (*erule-tac x*=Δ **in** *allE*)
apply (*erule-tac x*=Δ **in** *allE*) **apply** *clarify*
using *mp* **apply** *blast* **done**

theorem *soundness*:
assumes *d*: $\Delta \vdash \varphi$ **shows** $\Delta \models \varphi$
using *d*
apply (*induct rule*: *deduction.induct*)
apply (*auto simp add*: *implies-def satisfies-def*)
done

lemma *ppp*: **assumes** *A*: $\varphi \in \Gamma$ **shows** $\Gamma \vdash (\varphi' \to \varphi)$
proof –
let *?S* = (*emp*(*0*:=φ))(*1*:=φ')

have *p1*: $\Gamma \vdash$ *subst ?S A1* **apply** (*rule ax*) **by** *simp*
from *A* **have** *p2*: $\Gamma \vdash \varphi$ **by** (*rule hyp*)
from *p1 p2* **show** *?thesis*
apply *simp* **apply** (*rule mp*) **apply** *blast* **apply** *simp* **done**
qed

lemma *deduction-impl*:
$\Gamma' \vdash \psi \Longrightarrow (\forall\ \varphi\, \Gamma.\ \Gamma' = insert\ \varphi\ \Gamma \longrightarrow \Gamma \vdash (\varphi \to \psi))$
apply (*induct rule*: *deduction.induct*)
apply *clarify* **apply** (*case-tac* φ=φ') **apply** *simp* **apply** (*rule aa*)
apply *simp* **apply** (*rule ppp*) **apply** *simp*
apply *clarify*
defer
apply *clarify*
apply (*erule-tac x*=φ' **in** *allE*)
apply (*erule-tac x*=Γ **in** *allE*)
apply (*erule-tac x*=φ' **in** *allE*)
apply (*erule-tac x*=Γ **in** *allE*)
apply *simp*
proof –
fix $\varphi\ \psi\ \varphi'\ \Gamma$
assume *IH1*: $\Gamma \vdash (\varphi' \to (\varphi \to \psi))$ **and** *IH2*: $\Gamma \vdash (\varphi' \to \varphi)$
let *?S* = ((*emp*(*0*:=φ'))(*1*:=φ))(*2*:=ψ)
have *p1*: $\Gamma \vdash$ *subst ?S A2* **apply** (*rule ax*) **by** *simp*
from *IH1 p1* **have** *p2*: $\Gamma \vdash ((\varphi' \to \varphi) \to (\varphi' \to \psi))$
apply *simp* **apply** (*rule mp*) **apply** *blast* **apply** *blast* **done**
from *p2 IH2* **show** $\Gamma \vdash (\varphi' \to \psi)$ **by** (*rule mp*)
next
fix $S\ \Sigma\ \varphi\ \varphi'\ \Gamma$
assume *pa*: $\varphi \in$ *Axioms*
from *pa* **have** *A*: $\Gamma \vdash$ *subst S* φ **by** (*rule ax*)
let *?S* = (*emp*(*0*:=*subst S* φ))(*1*:=φ')
have *p1*: $\Gamma \vdash$ *subst ?S A1* **apply** (*rule ax*) **by** *simp*
from *A p1* **show** $\Gamma \vdash (\varphi' \to$ *subst S* $\varphi)$
apply *simp* **apply** (*rule mp*) **apply** *blast* **apply** *simp* **done**
qed

theorem *deduction*:
insert $\varphi\ \Gamma \vdash \psi \Longrightarrow \Gamma \vdash (\varphi \to \psi)$ **using** *deduction-impl* **by** *simp*

lemma *cut-rule*:
assumes *A*: $\Gamma \vdash \varphi$ **and** *B*: *insert* $\varphi\ \Gamma \vdash \psi$ **shows** $\Gamma \vdash \psi$
proof –
from *B* **have** *C*: $\Gamma \vdash (\varphi \to \psi)$ **by** (*rule deduction*)
from *C A* **show** *?thesis* **by** (*rule mp*)

qed

lemma *mphyp*:
  assumes *ppG*: $\varphi \to \psi \in \Gamma$ **and** *pG*: $\varphi \in \Gamma$ **shows** $\Gamma \vdash \psi$
proof –
  from *pG* **have** *Gp*: $\Gamma \vdash \varphi$ **by** (*rule hyp*)
  from *ppG* **have** *pp*: $\Gamma \vdash (\varphi \to \psi)$ **by** (*rule hyp*)
  from *pp Gp* **show** $\Gamma \vdash \psi$ **by** (*rule mp*)
qed

lemma *cor-1-10a*: $\{b{\to}c, c{\to}d\} \vdash (b{\to}d)$
proof –
  let *?E* = $\{b{\to}c, c{\to}d\}$
  have *C*: *insert b ?E* $\vdash$ *c* **apply** (*rule mphyp*) **apply** *blast* **apply** *blast* **done**
  have *CD*: *insert b ?E* $\vdash$ *c*$\to$*d* **apply** (*rule hyp*) **by** *blast*
  from *CD C* **have** *insert b ?E* $\vdash$ *d* **by** (*rule mp*)
  thus *?E* $\vdash$ ($b{\to}d$) **by** (*rule deduction*)
qed

lemma *cor-1-10b*: $\{b{\to}(c{\to}d), c\} \vdash (b{\to}d)$
proof –
  let *?E* = $\{b{\to}(c{\to}d), c\}$
  have *CD*: *insert b ?E* $\vdash$ ($c{\to}d$) **apply** (*rule mphyp*) **apply** *blast* **by** *blast*
  have *C*: *insert b ?E* $\vdash$ *c* **apply** (*rule hyp*) **by** *simp*
  from *CD C* **have** *insert b ?E* $\vdash$ *d* **by** (*rule mp*)
  thus *?thesis* **by** (*rule deduction*)
qed

lemma *lem-1-11-a*: $\Gamma \vdash$ ((*Neg* (*Neg* φ)) $\to \varphi$)
proof –
  let *?S* = (*emp*(*0*:=*Neg* φ))(*1*:=φ)
  have *p1*: $\Gamma \vdash$ *subst ?S A3* **apply** (*rule ax*) **by** *simp*
  have *p2*: $\Gamma \vdash$ (*Neg* $\varphi \to$ *Neg* φ) **by** (*rule aa*)

  have {*subst ?S A3*, (*Neg* $\varphi \to$ *Neg* φ)} $\vdash$ ((*Neg* $\varphi \to$ *Neg* (*Neg* φ)) $\to \varphi$)
    apply *simp* **by** (*rule cor-1-10b*)
  hence *A*: (*insert* (*subst ?S A3*) (*insert* (*Neg* $\varphi \to$ *Neg* φ) Γ)) $\vdash$ ((*Neg* $\varphi \to$ *Neg* (*Neg* φ)) $\to \varphi$)
    using *weakening* **apply** *blast* **done**
  obtain *x* **where** *X*: *x* = (*insert* (*Neg* $\varphi \to$ *Neg* φ) Γ) **by** *simp*
  from *X A* **have** *B*: (*insert* (*subst ?S A3*) *x*) $\vdash$ ((*Neg* $\varphi \to$ *Neg* (*Neg* φ)) $\to \varphi$) **by** *simp*
  from *p1 X* **have** *p3*: *x* $\vdash$ *subst ?S A3* **using** *weakening* **apply** *blast* **done**
  from *p3 B* **have** *x* $\vdash$ ((*Neg* $\varphi \to$ *Neg* (*Neg* φ)) $\to \varphi$)
    by (*rule cut-rule*)
  with *X* **have** *C*: *insert* (*Neg* $\varphi \to$ *Neg* φ) $\Gamma \vdash$ ((*Neg* $\varphi \to$ *Neg* (*Neg* φ)) $\to \varphi$) **by** *simp*
  let *?Y* = ((*Neg* $\varphi \to$ *Neg* (*Neg* φ)) $\to \varphi$)

from *p2 C* **have** *D*: $\Gamma \vdash$ *?Y* **by** (*rule cut-rule*)
let *?S1* = (*emp*(*0*:=*Neg* (*Neg* φ)))(*1*:=*Neg* φ)
have *E*: $\Gamma \vdash$ *subst ?S1 A1* **apply** (*rule ax*) **by** *simp*
let *?Z* = ((*Neg* (*Neg* φ)) $\rightarrow$ φ)
have *F*: {*subst ?S1 A1*, *?Y*} $\vdash$ *?Z* **apply** *simp* **by** (*rule cor-1-10a*)
obtain *y* **where** *Y*: *y* = (*insert ?Y* Γ) **by** *simp*
from *F Y* **have** *G*: *insert* (*subst ?S1 A1*) *y* $\vdash$ *?Z* **using** *weakening* **apply** *blast* **done**
from *E Y* **have** *H*: *y* $\vdash$ *subst ?S1 A1* **using** *weakening* **apply** *blast* **done**
from *H G* **have** *y* $\vdash$ *?Z* **by** (*rule cut-rule*)
with *Y* **have** *I*: *insert ?Y* $\Gamma \vdash$ *?Z* **by** *simp*
from *D I* **show** $\Gamma \vdash$ *?Z* **by** (*rule cut-rule*)
qed

lemma *lem-1-11-b*: $\Gamma \vdash$ ($\varphi \rightarrow$ (*Neg* (*Neg* φ)))
proof –
let *?S* = (*emp*(*0*:=φ))(*1*:=(*Neg* (*Neg* φ)))
have *p1*: $\Gamma \vdash$ *subst ?S A3* **apply** (*rule ax*) **by** *simp*
have *p2*: $\Gamma \vdash$ ((*Neg* (*Neg* (*Neg* φ))) $\rightarrow$ (*Neg* φ)) **by** (*rule lem-1-11-a*)
let *?P3* = ((*Neg* (*Neg* (*Neg* φ))) $\rightarrow \varphi$) $\rightarrow$ *Neg* (*Neg* φ)
from *p1 p2* **have** *p3*: $\Gamma \vdash$ *?P3* **apply** *simp* **apply** (*rule mp*) **apply** *blast* **apply** *blast* **done**
let *?S1* = (*emp*(*0*:=φ))(*1*:=(*Neg* (*Neg* (*Neg* φ))))
let *?P4* = $\varphi \rightarrow$(*Neg*(*Neg*(*Neg* φ))$\rightarrow\varphi$)
have $\Gamma \vdash$ *subst ?S1 A1* **apply** (*rule ax*) **by** *simp*
hence *p4*: *insert ?P3* $\Gamma \vdash$ *?P4* **apply** *simp* **using** *weakening* **by** *blast*
have {*?P4*, *?P3*} $\vdash \varphi \rightarrow$ (*Neg* (*Neg* φ)) **by** (*rule cor-1-10a*)
hence (*insert ?P4* (*insert ?P3* Γ)) $\vdash \varphi \rightarrow$ (*Neg* (*Neg* φ)) **using** *weakening* **by** *blast*
with *p4* **have** (*insert ?P3* Γ) $\vdash \varphi \rightarrow$ (*Neg* (*Neg* φ)) **using** *cut-rule* **by** *blast*
with *p3* **show** $\Gamma \vdash \varphi \rightarrow$ (*Neg* (*Neg* φ)) **using** *cut-rule* **by** *blast*
qed
lemma *lem-1-11-c*: $\Gamma \vdash$ (*Neg* $\varphi \rightarrow (\varphi \rightarrow \psi)$) **sorry**
lemma *lem-1-11-d*: $\Gamma \vdash$ ((*Neg* $\psi \rightarrow$ *Neg* φ) $\rightarrow (\varphi \rightarrow \psi)$) **sorry**
lemma *lem-1-11-e*: $\Gamma \vdash$ (($\varphi \rightarrow \psi$) $\rightarrow$ (*Neg* $\psi \rightarrow$ *Neg* φ)) **sorry**
lemma *lem-1-11-f*: $\Gamma \vdash$ ($\varphi \rightarrow$ (*Neg* $\psi \rightarrow$ *Neg* ($\varphi \rightarrow \psi$))) **sorry**
lemma *lem-1-11-g*: $\Gamma \vdash$ (($\varphi \rightarrow \psi$) $\rightarrow$ ((*Neg* $\varphi \rightarrow \psi$) $\rightarrow \psi$)) **sorry**

14.4 Completeness

consts *hyps* :: *nat set* $\Rightarrow$ *formula* $\Rightarrow$ *formula set*
primrec
hyps T (*Atom n*) = (*if n* $\in$ *T then* {*Atom n*} *else* {*Neg* (*Atom n*)})
hyps T (*Neg* φ) = *hyps T* φ
hyps T ($\varphi \rightarrow \psi$) = *hyps T* φ $\cup$ *hyps T* ψ

lemma *hyps-finite*: *finite* (*hyps T* φ)
apply (*induct* φ) **apply** *auto* **done**

```
lemma hyps-member: ∀ T x. x ∈ hyps T φ ⟶ (∃ α. (x = Atom α ∧ α ∈ T)
    ∨ (x = Neg (Atom α) ∧ α ∉ T))
  apply (induct φ) by auto

lemma hyps-diff: hyps (T−{α}) φ ⊆ insert (Neg (Atom α)) ((hyps T φ) − {Atom α})
  apply (induct φ) by auto

lemma hyps-cons:
    hyps (insert α T) φ ⊆ insert (Atom α) ((hyps T φ)−{Neg (Atom α)})
  by (induct-tac φ) auto

constdefs flip :: (nat set) ⇒ formula ⇒ formula
 flip T φ ≡ (if eval (λ x. x ∈ T) φ then φ else Neg φ)
lemma eval (λ x. x ∈ T) (flip T φ) by (simp add: flip-def)

lemma kalmar[rule-format]:
 ∀ φ. size φ = n ⟶ hyps v φ ⊢ flip v φ
  apply (induct rule: nat-less-induct)
  apply clarify
proof −
  fix n and φ::formula
  assume IH: ∀ m<size φ. ∀ φ. size φ = m ⟶ hyps v φ ⊢ flip v φ
  show hyps v φ ⊢ flip v φ
  proof (cases φ)
    fix α assume p: φ = Atom α
    thus ?thesis apply (simp add: flip-def) using hyp by blast
  next
    fix ψ assume p: φ = Neg ψ
    show ?thesis
    proof (cases eval (λ x. x ∈ v) ψ)
      assume ev: eval (λ x. x ∈ v) ψ
      from ev have evnp: ¬ (eval (λ x. x ∈ v) (Neg ψ)) by simp
      from ev have fp: flip v ψ = ψ by (simp add: flip-def)
      from evnp p have fnp: flip v φ = Neg φ by (simp add: flip-def)
      from p have size ψ < size φ by simp
      with IH have hyps v ψ ⊢ flip v ψ by blast
      with fp have A: hyps v ψ ⊢ ψ by simp
      have B: hyps v ψ ⊢ ψ → (Neg (Neg ψ)) by (rule lem-1-11-b)
      from B A have hyps v ψ ⊢ Neg (Neg ψ) by (rule mp)
      with fnp p show ?thesis by simp
    next
      assume ev: ¬ eval (λ x. x ∈ v) ψ
      from ev p have evp: eval (λ x. x ∈ v) φ by simp
      hence fp: flip v φ = φ by (simp add: flip-def)
```

```
    from ev have fps: flip v ψ = Neg ψ by (simp add: flip-def)
    from p have size ψ < size φ by simp
    with IH have hyps v ψ ⊢ flip v ψ by blast
    with fps p fp  show ?thesis by simp
  qed
next
  fix ψ1 ψ2 assume p: φ = ψ1 → ψ2
  from p have s1: size ψ1 < size φ by simp
  from s1 IH have IH1: hyps v ψ1 ⊢ flip v ψ1 by blast
  from p have s2: size ψ2 < size φ by simp
  from s2 IH have IH2: hyps v ψ2 ⊢ flip v ψ2 by blast
  show ?thesis
  proof (cases eval (λ x. x ∈ v) ψ1)
    assume ev1: eval (λ x. x ∈ v) ψ1
    from ev1 have f1: flip v ψ1 = ψ1 by (simp add: flip-def)
    show ?thesis
    proof (cases eval (λ x. x ∈ v) ψ2)
      assume ev2: eval (λ x. x ∈ v) ψ2
      from ev2 have f2: flip v ψ2 = ψ2 by (simp add: flip-def)
      from p ev2 have fp: flip v φ = φ by (simp add: flip-def)
      from f2 IH2 have ps2: hyps v ψ2 ⊢ ψ2 by simp
      let ?S = (emp(0:=ψ2))(1:=ψ1)
      have hyps v ψ2 ⊢ subst ?S A1 apply (rule ax) by simp
      with ps2 p have X: hyps v ψ2 ⊢ φ
        apply simp apply (rule mp)
        apply blast apply blast done
      from p have hyps v ψ2 ⊆ hyps v φ
        apply simp by blast
      with X have hyps v φ ⊢ φ by (rule weakening)
      with fp show ?thesis by simp
    next
      assume ev2: ¬ eval (λ x. x ∈ v) ψ2
      hence fp2: flip v ψ2 = Neg ψ2 by (simp add: flip-def)
      from p ev2 have ¬ eval (λ x. x ∈ v) φ by simp
      hence fp: flip v φ = Neg φ by (simp add: flip-def)

      from p have p1p: hyps v ψ1 ⊆ hyps v φ
        apply simp by blast
      from IH1 f1 have p1p1: hyps v ψ1 ⊢ ψ1 by simp
      from p1p1 p1p have p1: hyps v φ ⊢ ψ1 by (rule weakening)

      from p have p2p: hyps v ψ2 ⊆ hyps v φ
        apply simp by blast
      from IH2 fp2 have p2p2: hyps v ψ2 ⊢ Neg ψ2 by simp
      from p2p2 p2p have p2: hyps v φ ⊢ Neg ψ2 by (rule weakening)
```

```
    have hyps v φ ⊢ (ψ1 → (Neg ψ2 → Neg (ψ1 → ψ2)))
     by (rule lem-1-11-f)
    with p1 have hyps v φ ⊢ Neg ψ2 → Neg (ψ1 → ψ2)
     using mp by blast
    with p2 have hyps v φ ⊢ Neg (ψ1 → ψ2)
     using mp by blast
    with fp p show ?thesis by simp
   qed
  next
   assume ev1: ¬ eval (λ x. x ∈ v) ψ1
   with p have ep: eval (λ x. x ∈ v) φ by simp
   from ev1 have f1: flip v ψ1 = Neg ψ1 by (simp add: flip-def)
   from ep have fp: flip v φ = φ by (simp add: flip-def)
   from f1 IH1 have ps1: hyps v ψ1 ⊢ Neg ψ1 by simp
   have p12: hyps v ψ1 ⊢ (Neg ψ1 → (ψ1 → ψ2)) by (rule lem-1-11-c)
   from p12 ps1 have hyps v ψ1 ⊢ ψ1 → ψ2 by (rule mp)
   with p have X: hyps v ψ1 ⊢ φ by simp
   from p have Y: hyps v ψ1 ⊆ hyps v φ
    apply simp by blast
   from Y X fp show ?thesis apply simp apply (rule weakening) apply blast by blast
  qed
 qed
qed

lemma excluded-middle:
 assumes pp: insert φ Γ ⊢ ψ and npp: insert (Neg φ) Γ ⊢ ψ
 shows Γ ⊢ ψ
proof –
 from pp have a: Γ ⊢ φ → ψ  by (rule deduction)
 from npp have b: Γ ⊢ Neg φ → ψ  by (rule deduction)
 have c: Γ ⊢ (φ→ψ) → ((Neg φ → ψ) → ψ) by (rule lem-1-11-g)
 from c a have d: Γ ⊢ ((Neg φ → ψ) → ψ) by (rule mp)
 from d b show Γ ⊢ ψ by (rule mp)
qed

lemma variable-elimination:
 finite H ⟹ (∀ φ. tautology φ ∧ H ⊆ hyps T0 φ ⟶
   (∀ T.  (hyps T φ − H) ⊢ φ))
 apply (induct rule: finite-induct) apply clarify defer apply clarify defer
proof –
 fix φ T assume taut: tautology φ
 have hyps T φ ⊢ flip T φ apply (rule kalmar) by simp
 with taut show (hyps T φ − {}) ⊢ φ by (simp add: flip-def tautology-def)
next
```

```
  fix x H φ T
  assume IH: ∀ φ. tautology φ ∧ H ⊆ hyps T0 φ ⟶ (∀ T. (hyps T φ − H) ⊢ φ)
    and taut: tautology φ and xfh: insert x H ⊆ hyps T0 φ
  from xfh obtain α where X: (x = Atom α ∧ α ∈ T0)
    ∨ (x = Neg (Atom α) ∧ α ∉ T0) using hyps-member by blast
  moreover { assume X: x = Atom α ∧ α ∈ T0
    have (hyps T φ − insert (Atom α) H) ⊢ φ
    proof (rule excluded-middle[of Atom α])
      from taut xfh IH have a: (hyps T φ − H) ⊢ φ by blast
      have b: hyps T φ − H ⊆ insert (Atom α) (hyps T φ − insert (Atom α) H) by blast
      from a b show insert (Atom α) (hyps T φ − insert (Atom α) H) ⊢ φ using weakening by blast
    next
      from taut xfh IH have a: (hyps (T−{α}) φ − H) ⊢ φ by blast
      have hyps (T−{α}) φ ⊆ insert (Neg (Atom α)) ((hyps T φ) − {Atom α})
        by (rule hyps-diff)
      with X have b: (hyps (T−{α}) φ) − H ⊆ insert (Neg (Atom α)) (hyps T φ − insert (Atom α) H)
        by blast
      from a b show insert (Neg (Atom α)) (hyps T φ − insert (Atom α) H) ⊢ φ
        using weakening by blast
    qed
  } moreover { assume X: x = Neg (Atom α) ∧ α ∉ T0
    have (hyps T φ − insert (Neg (Atom α)) H) ⊢ φ
    proof (rule excluded-middle[of Atom α])
      from taut xfh IH have a: (hyps (insert α T) φ − H) ⊢ φ by blast
      have b: hyps (insert α T) φ ⊆ insert (Atom α) (hyps T φ − {Neg (Atom α)}) by (rule hyps-cons)
      from b have c: hyps (insert α T) φ − H
                ⊆ insert (Atom α) (hyps T φ − insert (Neg (Atom α)) H) by blast
      from a c show insert (Atom α) (hyps T φ − insert (Neg (Atom α)) H) ⊢ φ
        using weakening by blast
    next
      from taut xfh IH have a: (hyps T φ − H) ⊢ φ by blast
      have b: hyps T φ − H ⊆ insert (Neg (Atom α)) (hyps T φ − insert (Neg (Atom α)) H) by blast
      from a b show insert (Neg (Atom α)) (hyps T φ − (insert (Neg (Atom α)) H)) ⊢ φ
        using weakening by blast
    qed
  } ultimately show (hyps T φ − insert x H) ⊢ φ by blast
qed

theorem completeness:
  assumes taut: tautology φ shows {} ⊢ φ
proof −
  have finite (hyps T φ) by (rule hyps-finite)
  with taut have (hyps T φ − hyps T φ) ⊢ φ using variable-elimination by blast
  thus ?thesis by simp
qed
```

REFERENCES

[1] H. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic*. Elsevier, 1984.

[2] S. Bilaniuk. A problem course in mathematical logic.

[3] A. Charguéraud, B. C. Pierce, and S. Weirich. Proof engineering: Practical techniques for mechanized metatheory, Sept. 2006. Submitted for publication.

[4] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[5] H. B. Curry and R. Feys. *Combinatory Logic,*. North-Holland Publishing Co., Amsterdam, 1958.

[6] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[7] A. D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *HUG '93: Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 413–425, London, UK, 1994. Springer-Verlag.

[8] E. Mendelson. *Introduction to Mathematical Logic*. Chapman and Hall, 1997.

[9] T. Nishihara and Y. Minamide. Depth first search. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. http://afp.sourceforge.net/entries/Depth-First-Search.shtml, June 2004. Formal proof development.

[10] L. C. Paulson. Set theory for verification: II. Induction and recursion. *Journal of Automated Reasoning*, 15(2):167–215, 1995.

[11] R. Pollack. Closure under alpha-conversion. In *TYPES '93: Proceedings of the international workshop on Types for proofs and programs*, pages 313–332, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[12] D. J. Velleman. *How to Prove It*. Cambridge University Press, 1994.

[13] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.