

Examples of Inductive and Coinductive Definitions in HOL

Stefan Berghofer
Tobias Nipkow
Lawrence C Paulson
Markus Wenzel

December 17, 2025

Abstract

This is a collection of small examples to demonstrate Isabelle/HOL's (co)inductive definitions package. Large examples appear on many other sessions, such as Lambda, IMP, and Auth.

Contents

1	Common patterns of induction	4
1.1	Variations on statement structure	4
1.1.1	Local facts and parameters	4
1.1.2	Local definitions	4
1.1.3	Simple simultaneous goals	5
1.1.4	Compound simultaneous goals	6
1.2	Multiple rules	7
1.3	Inductive predicates	9
2	Nested datatypes	11
2.1	Terms and substitution	11
2.2	Alternative induction	12
3	Defining an Initial Algebra by Quotienting a Free Algebra	12
3.1	Defining the Free Algebra	12
3.2	Some Functions on the Free Algebra	13
3.2.1	The Set of Nonces	13
3.2.2	The Left Projection	14
3.2.3	The Right Projection	14
3.2.4	The Discriminator for Constructors	14
3.3	The Initial Algebra: A Quotiented Message Type	15
3.3.1	Characteristic Equations for the Abstract Constructors	15

3.4	The Abstract Function to Return the Set of Nonces	16
3.5	The Abstract Function to Return the Left Part	17
3.6	The Abstract Function to Return the Right Part	18
3.7	Injectivity Properties of Some Constructors	18
3.8	The Abstract Discriminator	20
4	Quotienting a Free Algebra Involving Nested Recursion	21
4.1	Defining the Free Algebra	21
4.2	Some Functions on the Free Algebra	22
4.2.1	The Set of Variables	22
4.2.2	Functions for Freeness	23
4.3	The Initial Algebra: A Quotiented Message Type	24
4.4	Every list of abstract expressions can be expressed in terms of a list of concrete expressions	25
4.4.1	Characteristic Equations for the Abstract Constructors	25
4.5	The Abstract Function to Return the Set of Variables	26
4.6	Injectivity Properties of Some Constructors	27
4.7	Injectivity of <i>FnCall</i>	27
4.8	The Abstract Discriminator	28
5	Terms over a given alphabet	30
6	Extended List Theory (old)	34
7	Arithmetic and boolean expressions	42
8	Infinitely branching trees	43
8.1	The Brouwer ordinals, as in ZF/Induct/Brouwer.thy.	44
8.2	A WF Ordering for The Brouwer ordinals (Michael Compton)	44
9	Ordinals	45
10	Sigma algebras	46
11	Combinatory Logic example: the Church-Rosser Theorem	47
11.1	Definitions	47
11.2	Reflexive/Transitive closure preserves Church-Rosser property	48
11.3	Non-contraction results	49
11.4	Results about Parallel Contraction	50
11.5	Basic properties of parallel contraction	50
11.6	Equivalence of $p \rightarrow q$ and $p \Rightarrow q$	50

12 Meta-theory of propositional logic	51
12.1 The datatype of propositions	51
12.2 The proof system	51
12.3 The semantics	52
12.3.1 Semantics of propositional logic.	52
12.3.2 Logical consequence	52
12.4 Proof theory of propositional logic	52
12.4.1 Weakening, left and right	52
12.4.2 The deduction theorem	53
12.4.3 The cut rule	53
12.4.4 Soundness of the rules wrt truth-table semantics . . .	53
12.5 Completeness	53
12.5.1 Towards the completeness proof	53
12.6 Completeness – lemmas for reducing the set of assumptions .	54
12.6.1 Completeness theorem	55
13 Mutual Induction via Iterated Inductive Definitions	56
13.1 Commands	56
13.2 Expressions	58
13.3 Equivalence of IF e THEN c;;(WHILE e DO c) ELSE SKIP and WHILE e DO c	60
13.4 Equivalence of (IF e THEN c1 ELSE c2);;c and IF e THEN (c1;;c) ELSE (c2;;c)	60
13.5 Equivalence of VALOF c1 RESULTIS (VALOF c2 RESULTIS e) and VALOF c1;;c2 RESULTIS e	61
13.6 Equivalence of VALOF SKIP RESULTIS e and e	61
13.7 Equivalence of VALOF x:=e RESULTIS x and e	62

1 Common patterns of induction

```
theory Common-Patterns
imports Main
begin
```

The subsequent Isar proof schemes illustrate common proof patterns supported by the generic *induct* method.

To demonstrate variations on statement (goal) structure we refer to the induction rule of Peano natural numbers: $\llbracket P\ 0; \bigwedge nat. P\ nat \implies P\ (Suc\ nat) \rrbracket \implies P\ nat$, which is the simplest case of datatype induction. We shall also see more complex (mutual) datatype inductions involving several rules. Working with inductive predicates is similar, but involves explicit facts about membership, instead of implicit syntactic typing.

1.1 Variations on statement structure

1.1.1 Local facts and parameters

Augmenting a problem by additional facts and locally fixed variables is a bread-and-butter method in many applications. This is where unwieldy object-level \forall and \longrightarrow used to occur in the past. The *induct* method works with primary means of the proof language instead.

```
lemma
  fixes  $n :: nat$ 
    and  $x :: 'a$ 
  assumes  $A\ n\ x$ 
  shows  $P\ n\ x$  using  $\langle A\ n\ x \rangle$ 
proof (induct  $n$  arbitrary:  $x$ )
  case 0
  note  $prem = \langle A\ 0\ x \rangle$ 
  show  $P\ 0\ x$   $\langle proof \rangle$ 
next
  case ( $Suc\ n$ )
  note  $hyp = \langle \bigwedge x. A\ n\ x \implies P\ n\ x \rangle$ 
    and  $prem = \langle A\ (Suc\ n)\ x \rangle$ 
  show  $P\ (Suc\ n)\ x$   $\langle proof \rangle$ 
qed
```

1.1.2 Local definitions

Here the idea is to turn sub-expressions of the problem into a defined induction variable. This is often accompanied with fixing of auxiliary parameters in the original expression, otherwise the induction step would refer invariably to particular entities. This combination essentially expresses a partially abstracted representation of inductive expressions.

```
lemma
```

```

fixes  $a :: 'a \Rightarrow nat$ 
assumes  $A (a\ x)$ 
shows  $P (a\ x)$  using  $\langle A (a\ x) \rangle$ 
proof ( $induct\ n \equiv a\ x\ arbitrary: x$ )
  case 0
    note  $prem = \langle A (a\ x) \rangle$ 
    and  $defn = \langle 0 = a\ x \rangle$ 
    show  $P (a\ x)$   $\langle proof \rangle$ 
  next
    case ( $Suc\ n$ )
    note  $hyp = \langle \bigwedge x. n = a\ x \implies A (a\ x) \implies P (a\ x) \rangle$ 
    and  $prem = \langle A (a\ x) \rangle$ 
    and  $defn = \langle Suc\ n = a\ x \rangle$ 
    show  $P (a\ x)$   $\langle proof \rangle$ 
qed

```

Observe how the local definition $n = a\ x$ recurs in the inductive cases as $0 = a\ x$ and $Suc\ n = a\ x$, according to underlying induction rule.

1.1.3 Simple simultaneous goals

The most basic simultaneous induction operates on several goals one-by-one, where each case refers to induction hypotheses that are duplicated according to the number of conclusions.

```

lemma
  fixes  $n :: nat$ 
  shows  $P\ n$  and  $Q\ n$ 
proof ( $induct\ n$ )
  case 0 case 1
    show  $P\ 0$   $\langle proof \rangle$ 
  next
    case 0 case 2
    show  $Q\ 0$   $\langle proof \rangle$ 
  next
    case ( $Suc\ n$ ) case 1
    note  $hyps = \langle P\ n \rangle \langle Q\ n \rangle$ 
    show  $P (Suc\ n)$   $\langle proof \rangle$ 
  next
    case ( $Suc\ n$ ) case 2
    note  $hyps = \langle P\ n \rangle \langle Q\ n \rangle$ 
    show  $Q (Suc\ n)$   $\langle proof \rangle$ 
qed

```

The split into subcases may be deferred as follows – this is particularly relevant for goal statements with local premises.

```

lemma
  fixes  $n :: nat$ 
  shows  $A\ n \implies P\ n$ 

```

```

    and  $B\ n \implies Q\ n$ 
  proof (induct n)
    case 0
    {
      case 1
      note  $\langle A\ 0 \rangle$ 
      show  $P\ 0$   $\langle proof \rangle$ 
    next
      case 2
      note  $\langle B\ 0 \rangle$ 
      show  $Q\ 0$   $\langle proof \rangle$ 
    }
  next
    case (Suc n)
    note  $\langle A\ n \implies P\ n \rangle$ 
    and  $\langle B\ n \implies Q\ n \rangle$ 
    {
      case 1
      note  $\langle A\ (Suc\ n) \rangle$ 
      show  $P\ (Suc\ n)$   $\langle proof \rangle$ 
    next
      case 2
      note  $\langle B\ (Suc\ n) \rangle$ 
      show  $Q\ (Suc\ n)$   $\langle proof \rangle$ 
    }
  qed

```

1.1.4 Compound simultaneous goals

The following pattern illustrates the slightly more complex situation of simultaneous goals with individual local assumptions. In compound simultaneous statements like this, local assumptions need to be included into each goal, using \implies of the Pure framework. In contrast, local parameters do not require separate \bigwedge prefixes here, but may be moved into the common context of the whole statement.

```

lemma
  fixes  $n :: nat$ 
  and  $x :: 'a$ 
  and  $y :: 'b$ 
  shows  $A\ n\ x \implies P\ n\ x$ 
  and  $B\ n\ y \implies Q\ n\ y$ 
proof (induct n arbitrary: x y)
  case 0
  {
    case 1
    note  $prem = \langle A\ 0\ x \rangle$ 
    show  $P\ 0\ x$   $\langle proof \rangle$ 
  }

```

```

{
  case 2
  note prem = ⟨B 0 y⟩
  show Q 0 y ⟨proof⟩
}
next
case (Suc n)
note hyps = ⟨ $\bigwedge x. A\ n\ x \implies P\ n\ x$ ⟩ ⟨ $\bigwedge y. B\ n\ y \implies Q\ n\ y$ ⟩
then have some-intermediate-result ⟨proof⟩
{
  case 1
  note prem = ⟨A (Suc n) x⟩
  show P (Suc n) x ⟨proof⟩
}
{
  case 2
  note prem = ⟨B (Suc n) y⟩
  show Q (Suc n) y ⟨proof⟩
}
}
qed

```

Here *induct* provides again nested cases with numbered sub-cases, which allows to share common parts of the body context. In typical applications, there could be a long intermediate proof of general consequences of the induction hypotheses, before finishing each conclusion separately.

1.2 Multiple rules

Multiple induction rules emerge from mutual definitions of datatypes, inductive predicates, functions etc. The *induct* method accepts replicated arguments (with *and* separator), corresponding to each projection of the induction principle.

The goal statement essentially follows the same arrangement, although it might be subdivided into simultaneous sub-problems as before!

```

datatype foo = Foo1 nat | Foo2 bar
and bar = Bar1 bool | Bar2 bazar
and bazar = Bazar foo

```

The pack of induction rules for this datatype is:

```

[[ $\bigwedge x. P1\ (Foo1\ x); \bigwedge x. P2\ x \implies P1\ (Foo2\ x); \bigwedge x. P2\ (Bar1\ x);$ 
 $\bigwedge x. P3\ x \implies P2\ (Bar2\ x); \bigwedge x. P1\ x \implies P3\ (Bazar\ x)$ ]]
 $\implies P1\ foo$ 
[[ $\bigwedge x. P1\ (Foo1\ x); \bigwedge x. P2\ x \implies P1\ (Foo2\ x); \bigwedge x. P2\ (Bar1\ x);$ 
 $\bigwedge x. P3\ x \implies P2\ (Bar2\ x); \bigwedge x. P1\ x \implies P3\ (Bazar\ x)$ ]]
 $\implies P2\ bar$ 
[[ $\bigwedge x. P1\ (Foo1\ x); \bigwedge x. P2\ x \implies P1\ (Foo2\ x); \bigwedge x. P2\ (Bar1\ x);$ 
 $\bigwedge x. P3\ x \implies P2\ (Bar2\ x); \bigwedge x. P1\ x \implies P3\ (Bazar\ x)$ ]]

```

$\implies P\mathcal{J} \text{ bazar}$

This corresponds to the following basic proof pattern:

```
lemma
  fixes foo :: foo
    and bar :: bar
    and bazar :: bazar
  shows P foo
    and Q bar
    and R bazar
proof (induct foo and bar and bazar)
  case (Foo1 n)
  show P (Foo1 n) <proof>
next
  case (Foo2 bar)
  note <Q bar>
  show P (Foo2 bar) <proof>
next
  case (Bar1 b)
  show Q (Bar1 b) <proof>
next
  case (Bar2 bazar)
  note <R bazar>
  show Q (Bar2 bazar) <proof>
next
  case (Bazar foo)
  note <P foo>
  show R (Bazar foo) <proof>
qed
```

This can be combined with the previous techniques for compound statements, e.g. like this.

```
lemma
  fixes x :: 'a and y :: 'b and z :: 'c
    and foo :: foo
    and bar :: bar
    and bazar :: bazar
  shows
    A x foo  $\implies$  P x foo
  and
    B1 y bar  $\implies$  Q1 y bar
    B2 y bar  $\implies$  Q2 y bar
  and
    C1 z bazar  $\implies$  R1 z bazar
    C2 z bazar  $\implies$  R2 z bazar
    C3 z bazar  $\implies$  R3 z bazar
proof (induct foo and bar and bazar arbitrary: x and y and z)
oops
```


1.3 Inductive predicates

The most basic form of induction involving predicates (or sets) essentially eliminates a given membership fact.

```
inductive Even :: nat ⇒ bool where
  zero: Even 0
| double: Even (2 * n) if Even n for n
```

```
lemma
  assumes Even n
  shows P n
  using assms
proof induct
  case zero
  show P 0 ⟨proof⟩
next
  case (double n)
  note ⟨Even n⟩ and ⟨P n⟩
  show P (2 * n) ⟨proof⟩
qed
```

Alternatively, an initial rule statement may be proven as follows, performing “in-situ” elimination with explicit rule specification.

```
lemma Even n ⇒ P n
proof (induct rule: Even.induct)
  oops
```

Simultaneous goals do not introduce anything new.

```
lemma
  assumes Even n
  shows P1 n and P2 n
  using assms
proof induct
  case zero
  {
    case 1
    show P1 0 ⟨proof⟩
  }
next
  case 2
  show P2 0 ⟨proof⟩
}
next
case (double n)
note ⟨Even n⟩ and ⟨P1 n⟩ and ⟨P2 n⟩
{
  case 1
  show P1 (2 * n) ⟨proof⟩
}
next
```

```

    case 2
    show P2 (2 * n) ⟨proof⟩
  }
qed

```

Working with mutual rules requires special care in composing the statement as a two-level conjunction, using lists of propositions separated by *and*. For example:

```

inductive Evn :: nat ⇒ bool and Odd :: nat ⇒ bool
where
  zero: Evn 0
| succ-Evn: Odd (Suc n) if Evn n for n
| succ-Odd: Evn (Suc n) if Odd n for n

```

```

lemma
  Evn n ⇒ P1 n
  Evn n ⇒ P2 n
  Evn n ⇒ P3 n
and
  Odd n ⇒ Q1 n
  Odd n ⇒ Q2 n
proof (induct rule: Evn-Odd.inducts)
  case zero
  { case 1 show P1 0 ⟨proof⟩ }
  { case 2 show P2 0 ⟨proof⟩ }
  { case 3 show P3 0 ⟨proof⟩ }
next
  case (succ-Evn n)
  note ⟨Evn n⟩ and ⟨P1 n⟩ ⟨P2 n⟩ ⟨P3 n⟩
  { case 1 show Q1 (Suc n) ⟨proof⟩ }
  { case 2 show Q2 (Suc n) ⟨proof⟩ }
next
  case (succ-Odd n)
  note ⟨Odd n⟩ and ⟨Q1 n⟩ ⟨Q2 n⟩
  { case 1 show P1 (Suc n) ⟨proof⟩ }
  { case 2 show P2 (Suc n) ⟨proof⟩ }
  { case 3 show P3 (Suc n) ⟨proof⟩ }
qed

```

Cases and hypotheses in each case can be named explicitly.

```

inductive star :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool for r
where
  refl: star r x x for x
| step: star r x z if r x y and star r y z for x y z

```

Underscores are replaced by the default name hyps:

```

lemmas star-induct = star.induct [case-names base step[r - IH]]

```

```

lemma star r x y  $\implies$  star r y z  $\implies$  star r x z
proof (induct rule: star-induct) print-cases
  case base
  then show ?case .
next
  case (step a b c) print-facts
  from step.prems have star r b z by (rule step.IH)
  with step.r show ?case by (rule star.step)
qed

end

```

2 Nested datatypes

```

theory Nested-Datatype
imports Main
begin

```

2.1 Terms and substitution

```

datatype ('a, 'b) term =
  Var 'a
| App 'b ('a, 'b) term list

primrec subst-term :: ('a  $\Rightarrow$  ('a, 'b) term)  $\Rightarrow$  ('a, 'b) term  $\Rightarrow$  ('a, 'b) term
  and subst-term-list :: ('a  $\Rightarrow$  ('a, 'b) term)  $\Rightarrow$  ('a, 'b) term list  $\Rightarrow$  ('a, 'b) term
  list
where
  subst-term f (Var a) = f a
| subst-term f (App b ts) = App b (subst-term-list f ts)
| subst-term-list f [] = []
| subst-term-list f (t # ts) = subst-term f t # subst-term-list f ts

lemmas subst-simps = subst-term.simps subst-term-list.simps

```

A simple lemma about composition of substitutions.

```

lemma
  subst-term (subst-term f1  $\circ$  f2) t =
    subst-term f1 (subst-term f2 t)
and
  subst-term-list (subst-term f1  $\circ$  f2) ts =
    subst-term-list f1 (subst-term-list f2 ts)
by (induct t and ts rule: subst-term.induct subst-term-list.induct) simp-all

lemma subst-term (subst-term f1  $\circ$  f2) t = subst-term f1 (subst-term f2 t)
proof –
  let ?P t = ?thesis
  let ?Q =  $\lambda$ ts. subst-term-list (subst-term f1  $\circ$  f2) ts =

```

```

    subst-term-list f1 (subst-term-list f2 ts)
  show ?thesis
proof (induct t rule: subst-term.induct)
  show ?P (Var a) for a by simp
  show ?P (App b ts) if ?Q ts for b ts
    using that by (simp only: subst-simps)
  show ?Q [] by simp
  show ?Q (t # ts) if ?P t ?Q ts for t ts
    using that by (simp only: subst-simps)
qed
qed

```

2.2 Alternative induction

```

lemma subst-term (subst-term f1 ∘ f2) t = subst-term f1 (subst-term f2 t)
proof (induct t rule: term.induct)
  case (Var a)
  show ?case by (simp add: o-def)
next
  case (App b ts)
  then show ?case by (induct ts) simp-all
qed

end

```

3 Defining an Initial Algebra by Quotienting a Free Algebra

For Lawrence Paulson’s paper “Defining functions on equivalence classes” *ACM Transactions on Computational Logic* **7**:40 (2006), 658–675, illustrating bare-bones quotient constructions. Any comparison using lifting and transfer should be done in a separate theory.

```
theory QuoDataType imports Main begin
```

3.1 Defining the Free Algebra

Messages with encryption and decryption as free constructors.

```

datatype
  freemsg = NONCE nat
           | MPAIR freemsg freemsg
           | CRYPT nat freemsg
           | DECRYPT nat freemsg

```

The equivalence relation, which makes encryption and decryption inverses provided the keys are the same.

The first two rules are the desired equations. The next four rules make the equations applicable to subterms. The last two rules are symmetry and transitivity.

inductive-set

```

msgrel :: (freemsg * freemsg) set
and msg-rel :: [freemsg, freemsg] => bool (infixl <~> 50)
where
  X ~ Y == (X,Y) ∈ msgrel
| CD:   CRYPT K (DECRYPT K X) ~ X
| DC:   DECRYPT K (CRYPT K X) ~ X
| NONCE: NONCE N ~ NONCE N
| MPAIR: [X ~ X'; Y ~ Y'] ==> MPAIR X Y ~ MPAIR X' Y'
| CRYPT: X ~ X' ==> CRYPT K X ~ CRYPT K X'
| DECRYPT: X ~ X' ==> DECRYPT K X ~ DECRYPT K X'
| SYM:   X ~ Y ==> Y ~ X
| TRANS: [X ~ Y; Y ~ Z] ==> X ~ Z

```

Proving that it is an equivalence relation

```

lemma msgrel-refl: X ~ X
by (induct X) (blast intro: msgrel.intros)+

```

theorem equiv-msgrel: equiv UNIV msgrel

proof (rule equivI)

```

show msgrel ⊆ UNIV × UNIV by simp
show refl msgrel by (simp add: refl-on-def msgrel-refl)
show sym msgrel by (simp add: sym-def, blast intro: msgrel.SYM)
show trans msgrel by (simp add: trans-def, blast intro: msgrel.TRANS)
qed

```

3.2 Some Functions on the Free Algebra

3.2.1 The Set of Nonces

A function to return the set of nonces present in a message. It will be lifted to the initial algebra, to serve as an example of that process.

primrec freenonces :: freemsg ⇒ nat set **where**

```

freenonces (NONCE N) = {N}
| freenonces (MPAIR X Y) = freenonces X ∪ freenonces Y
| freenonces (CRYPT K X) = freenonces X
| freenonces (DECRYPT K X) = freenonces X

```

This theorem lets us prove that the nonces function respects the equivalence relation. It also helps us prove that Nonce (the abstract constructor) is injective

```

theorem msgrel-imp-eq-freenonces: U ~ V ==> freenonces U = freenonces V
by (induct set: msgrel) auto

```

3.2.2 The Left Projection

A function to return the left part of the top pair in a message. It will be lifted to the initial algebra, to serve as an example of that process.

```
primrec freeleft :: freemsg  $\Rightarrow$  freemsg where
  freeleft (NONCE N) = NONCE N
| freeleft (MPAIR X Y) = X
| freeleft (CRYPT K X) = freeleft X
| freeleft (DECRYPT K X) = freeleft X
```

This theorem lets us prove that the left function respects the equivalence relation. It also helps us prove that MPair (the abstract constructor) is injective

```
theorem msgrel-imp-eqv-freeleft:
   $U \sim V \implies \text{freeleft } U \sim \text{freeleft } V$ 
by (induct set: msgrel) (auto intro: msgrel.intros)
```

3.2.3 The Right Projection

A function to return the right part of the top pair in a message.

```
primrec freeright :: freemsg  $\Rightarrow$  freemsg where
  freeright (NONCE N) = NONCE N
| freeright (MPAIR X Y) = Y
| freeright (CRYPT K X) = freeright X
| freeright (DECRYPT K X) = freeright X
```

This theorem lets us prove that the right function respects the equivalence relation. It also helps us prove that MPair (the abstract constructor) is injective

```
theorem msgrel-imp-eqv-freeright:
   $U \sim V \implies \text{freeright } U \sim \text{freeright } V$ 
by (induct set: msgrel) (auto intro: msgrel.intros)
```

3.2.4 The Discriminator for Constructors

A function to distinguish nonces, mpairs and encryptions

```
primrec freediscrim :: freemsg  $\Rightarrow$  int where
  freediscrim (NONCE N) = 0
| freediscrim (MPAIR X Y) = 1
| freediscrim (CRYPT K X) = freediscrim X + 2
| freediscrim (DECRYPT K X) = freediscrim X - 2
```

This theorem helps us prove $\text{Nonce } N \neq \text{MPair } X Y$

```
theorem msgrel-imp-eq-freediscrim:
   $U \sim V \implies \text{freediscrim } U = \text{freediscrim } V$ 
by (induct set: msgrel) auto
```

3.3 The Initial Algebra: A Quotiented Message Type

definition $Msg = UNIV // msgrel$

typedef $msg = Msg$
morphisms $Rep-Msg Abs-Msg$
unfolding $Msg-def$ **by** $(auto simp add: quotient-def)$

The abstract message constructors

definition
 $Nonce :: nat \Rightarrow msg$ **where**
 $Nonce\ N = Abs-Msg(msgrel\ \{\{NONCE\ N\}\})$

definition
 $MPair :: [msg, msg] \Rightarrow msg$ **where**
 $MPair\ X\ Y =$
 $Abs-Msg(\bigcup U \in Rep-Msg\ X. \bigcup V \in Rep-Msg\ Y. msgrel\ \{\{MPAIR\ U\ V\}\})$

definition
 $Crypt :: [nat, msg] \Rightarrow msg$ **where**
 $Crypt\ K\ X =$
 $Abs-Msg(\bigcup U \in Rep-Msg\ X. msgrel\ \{\{CRYPT\ K\ U\}\})$

definition
 $Decrypt :: [nat, msg] \Rightarrow msg$ **where**
 $Decrypt\ K\ X =$
 $Abs-Msg(\bigcup U \in Rep-Msg\ X. msgrel\ \{\{DECRYPT\ K\ U\}\})$

Reduces equality of equivalence classes to the $msgrel$ relation: $(msgrel\ \{\{x\}\} = msgrel\ \{\{y\}\}) = (x \sim y)$

lemmas $equiv-msgrel-iff = eq-equiv-class-iff$ $[OF\ equiv-msgrel\ UNIV-I\ UNIV-I]$

declare $equiv-msgrel-iff$ $[simp]$

All equivalence classes belong to set of representatives

lemma $[simp]: msgrel\ \{\{U\}\} \in Msg$
by $(auto simp add: Msg-def quotient-def intro: msgrel-refl)$

lemma $inj-on-Abs-Msg: inj-on\ Abs-Msg\ Msg$
by $(meson\ Abs-Msg-inject\ inj-onI)$

Reduces equality on abstractions to equality on representatives

declare $inj-on-Abs-Msg$ $[THEN\ inj-on-eq-iff, simp]$

declare $Abs-Msg-inverse$ $[simp]$

3.3.1 Characteristic Equations for the Abstract Constructors

lemma $MPair: MPair\ (Abs-Msg(msgrel\ \{\{U\}\}))\ (Abs-Msg(msgrel\ \{\{V\}\})) =$

$Abs\text{-}Msg\ (msgrel\ \{\{MPAIR\ U\ V\}\})$

proof –
 have $(\lambda U\ V. msgrel\ \{\{MPAIR\ U\ V\}\})\ respects2\ msgrel$
 by $(auto\ simp\ add:\ congruent2\text{-}def\ msgrel.MPAIR)$
 thus $?thesis$
 by $(simp\ add:\ MPair\text{-}def\ UN\text{-}equiv\text{-}class2\ [OF\ equiv\text{-}msgrel\ equiv\text{-}msgrel])$
qed

lemma *Crypt*: $Crypt\ K\ (Abs\text{-}Msg(msgrel\ \{\{U\}\})) = Abs\text{-}Msg\ (msgrel\ \{\{CRYPT\ K\ U\}\})$
proof –
 have $(\lambda U. msgrel\ \{\{CRYPT\ K\ U\}\})\ respects\ msgrel$
 by $(auto\ simp\ add:\ congruent\text{-}def\ msgrel.CRYPT)$
 thus $?thesis$
 by $(simp\ add:\ Crypt\text{-}def\ UN\text{-}equiv\text{-}class\ [OF\ equiv\text{-}msgrel])$
qed

lemma *Decrypt*:
 $Decrypt\ K\ (Abs\text{-}Msg(msgrel\ \{\{U\}\})) = Abs\text{-}Msg\ (msgrel\ \{\{DECRYPT\ K\ U\}\})$
proof –
 have $(\lambda U. msgrel\ \{\{DECRYPT\ K\ U\}\})\ respects\ msgrel$
 by $(auto\ simp\ add:\ congruent\text{-}def\ msgrel.DECRYPT)$
 thus $?thesis$
 by $(simp\ add:\ Decrypt\text{-}def\ UN\text{-}equiv\text{-}class\ [OF\ equiv\text{-}msgrel])$
qed

Case analysis on the representation of a msg as an equivalence class.

lemma *eq-Abs-Msg* $[case\text{-}names\ Abs\text{-}Msg,\ cases\ type:\ msg]:$
 $(\bigwedge U. z = Abs\text{-}Msg\ (msgrel\ \{\{U\}\}) \implies P) \implies P$
 by $(metis\ Abs\text{-}Msg\text{-}cases\ Msg\text{-}def\ quotientE)$

Establishing these two equations is the point of the whole exercise

theorem *CD-eq* $[simp]: Crypt\ K\ (Decrypt\ K\ X) = X$
 by $(cases\ X,\ simp\ add:\ Crypt\ Decrypt\ CD)$

theorem *DC-eq* $[simp]: Decrypt\ K\ (Crypt\ K\ X) = X$
 by $(cases\ X,\ simp\ add:\ Crypt\ Decrypt\ DC)$

3.4 The Abstract Function to Return the Set of Nonces

definition

$nonces :: msg \Rightarrow nat\ set$ **where**
 $nonces\ X = (\bigcup U \in Rep\text{-}Msg\ X. freenonces\ U)$

lemma *nonces-congruent*: $freenonces\ respects\ msgrel$
 by $(auto\ simp\ add:\ congruent\text{-}def\ msgrel\text{-}imp\text{-}eq\text{-}freenonces)$

Now prove the four equations for *nonces*

lemma *nonces-Nonce* $[simp]: nonces\ (Nonce\ N) = \{N\}$

by (simp add: nonces-def Nonce-def
UN-equiv-class [OF equiv-msgrel nonces-congruent])

lemma nonces-MPair [simp]: nonces (MPair X Y) = nonces X \cup nonces Y

proof –

have $\bigwedge U V. \llbracket X = \text{Abs-Msg} (\text{msgrel} \text{ “ } \{U\}); Y = \text{Abs-Msg} (\text{msgrel} \text{ “ } \{V\}) \rrbracket$
 $\implies \text{nonces} (\text{MPair } X Y) = \text{nonces } X \cup \text{nonces } Y$

by (simp add: nonces-def MPair
UN-equiv-class [OF equiv-msgrel nonces-congruent])

then show ?thesis

by (meson eq-Abs-Msg)

qed

lemma nonces-Crypt [simp]: nonces (Crypt K X) = nonces X

proof –

have $\bigwedge U. X = \text{Abs-Msg} (\text{msgrel} \text{ “ } \{U\}) \implies \text{nonces} (\text{Crypt } K X) = \text{nonces } X$
by (simp add: nonces-def Crypt UN-equiv-class [OF equiv-msgrel nonces-congruent])

then show ?thesis

by (meson eq-Abs-Msg)

qed

lemma nonces-Decrypt [simp]: nonces (Decrypt K X) = nonces X

proof –

have $\bigwedge U. X = \text{Abs-Msg} (\text{msgrel} \text{ “ } \{U\}) \implies \text{nonces} (\text{Decrypt } K X) = \text{nonces } X$
by (simp add: nonces-def Decrypt UN-equiv-class [OF equiv-msgrel nonces-congruent])

then show ?thesis

by (meson eq-Abs-Msg)

qed

3.5 The Abstract Function to Return the Left Part

definition

left :: msg \Rightarrow msg

where left X = Abs-Msg ($\bigcup U \in \text{Rep-Msg } X. \text{msgrel} \text{ “ } \{\text{freeleft } U\}$)

lemma left-congruent: $(\lambda U. \text{msgrel} \text{ “ } \{\text{freeleft } U\})$ respects msgrel

by (auto simp add: congruent-def msgrel-imp-equiv-freeleft)

Now prove the four equations for left

lemma left-Nonce [simp]: left (Nonce N) = Nonce N

by (simp add: left-def Nonce-def

UN-equiv-class [OF equiv-msgrel left-congruent])

lemma left-MPair [simp]: left (MPair X Y) = X

by (cases X, cases Y) (simp add: left-def MPair UN-equiv-class [OF equiv-msgrel
left-congruent])

lemma *left-Crypt* [simp]: *left* (*Crypt* *K* *X*) = *left* *X*
by (*cases* *X*) (*simp* *add*: *left-def* *Crypt* *UN-equiv-class* [*OF* *equiv-msgrel* *left-congruent*])

lemma *left-Decrypt* [simp]: *left* (*Decrypt* *K* *X*) = *left* *X*
by (*metis* *CD-eq* *left-Crypt*)

3.6 The Abstract Function to Return the Right Part

definition

right :: *msg* \Rightarrow *msg*
where *right* *X* = *Abs-Msg* ($\bigcup U \in \text{Rep-Msg } X. \text{msgrel} \{ \text{freeright } U \}$)

lemma *right-congruent*: $(\lambda U. \text{msgrel} \{ \text{freeright } U \})$ respects *msgrel*
by (*auto* *simp* *add*: *congruent-def* *msgrel-imp-equiv-freeright*)

Now prove the four equations for *right*

lemma *right-Nonce* [simp]: *right* (*Nonce* *N*) = *Nonce* *N*
by (*simp* *add*: *right-def* *Nonce-def* *UN-equiv-class* [*OF* *equiv-msgrel* *right-congruent*])

lemma *right-MPair* [simp]: *right* (*MPair* *X* *Y*) = *Y*
by (*cases* *X*, *cases* *Y*) (*simp* *add*: *right-def* *MPair* *UN-equiv-class* [*OF* *equiv-msgrel* *right-congruent*])

lemma *right-Crypt* [simp]: *right* (*Crypt* *K* *X*) = *right* *X*
by (*cases* *X*) (*simp* *add*: *right-def* *Crypt* *UN-equiv-class* [*OF* *equiv-msgrel* *right-congruent*])

lemma *right-Decrypt* [simp]: *right* (*Decrypt* *K* *X*) = *right* *X*
by (*metis* *CD-eq* *right-Crypt*)

3.7 Injectivity Properties of Some Constructors

lemma *NONCE-imp-eq*: *NONCE* *m* \sim *NONCE* *n* \Longrightarrow *m* = *n*
by (*drule* *msgrel-imp-eq-freenonces*, *simp*)

Can also be proved using the function *nonces*

lemma *Nonce-Nonce-eq* [iff]: (*Nonce* *m* = *Nonce* *n*) = (*m* = *n*)
by (*auto* *simp* *add*: *Nonce-def* *msgrel-refl* *dest*: *NONCE-imp-eq*)

lemma *MPAIR-imp-equiv-left*: *MPAIR* *X* *Y* \sim *MPAIR* *X'* *Y'* \Longrightarrow *X* \sim *X'*
by (*drule* *msgrel-imp-equiv-freeleft*, *simp*)

lemma *MPair-imp-equiv-left*:
assumes *eq*: *MPair* *X* *Y* = *MPair* *X'* *Y'* **shows** *X* = *X'*
proof –
from *eq*
have *left* (*MPair* *X* *Y*) = *left* (*MPair* *X'* *Y'*) **by** *simp*

thus *?thesis* **by** *simp*
qed

lemma *MPAIR-imp-eqv-right*: $MPAIR\ X\ Y \sim MPAIR\ X'\ Y' \implies Y \sim Y'$
by (*drule msgrel-imp-eqv-freeright, simp*)

lemma *MPair-imp-eq-right*: $MPair\ X\ Y = MPair\ X'\ Y' \implies Y = Y'$
by (*metis right-MPair*)

theorem *MPair-MPair-eq [iff]*: $(MPair\ X\ Y = MPair\ X'\ Y') = (X=X' \ \&\ Y=Y')$
by (*blast dest: MPair-imp-eq-left MPair-imp-eq-right*)

lemma *NONCE-neqv-MPAIR*: $NONCE\ m \sim MPAIR\ X\ Y \implies False$
by (*drule msgrel-imp-eq-freediscrim, simp*)

theorem *Nonce-neq-MPair [iff]*: $Nonce\ N \neq MPair\ X\ Y$
by (*cases X, cases Y*) (*use MPair NONCE-neqv-MPAIR Nonce-def in fastforce*)

Example suggested by a referee

theorem *Crypt-Nonce-neq-Nonce*: $Crypt\ K\ (Nonce\ M) \neq Nonce\ N$
by (*auto simp add: Nonce-def Crypt dest: msgrel-imp-eq-freediscrim*)

...and many similar results

theorem *Crypt2-Nonce-neq-Nonce*: $Crypt\ K\ (Crypt\ K'\ (Nonce\ M)) \neq Nonce\ N$
by (*auto simp add: Nonce-def Crypt dest: msgrel-imp-eq-freediscrim*)

theorem *Crypt-Crypt-eq [iff]*: $(Crypt\ K\ X = Crypt\ K\ X') = (X=X')$

proof

assume $Crypt\ K\ X = Crypt\ K\ X'$

hence $Decrypt\ K\ (Crypt\ K\ X) = Decrypt\ K\ (Crypt\ K\ X')$ **by** *simp*

thus $X = X'$ **by** *simp*

next

assume $X = X'$

thus $Crypt\ K\ X = Crypt\ K\ X'$ **by** *simp*

qed

theorem *Decrypt-Decrypt-eq [iff]*: $(Decrypt\ K\ X = Decrypt\ K\ X') = (X=X')$

proof

assume $Decrypt\ K\ X = Decrypt\ K\ X'$

hence $Crypt\ K\ (Decrypt\ K\ X) = Crypt\ K\ (Decrypt\ K\ X')$ **by** *simp*

thus $X = X'$ **by** *simp*

next

assume $X = X'$

thus $Decrypt\ K\ X = Decrypt\ K\ X'$ **by** *simp*

qed

lemma *msg-induct [case-names Nonce MPair Crypt Decrypt, cases type: msg]*:
assumes $N: \bigwedge N. P\ (Nonce\ N)$

```

    and M:  $\bigwedge X Y. \llbracket P X; P Y \rrbracket \implies P (MPair X Y)$ 
    and C:  $\bigwedge K X. P X \implies P (Crypt K X)$ 
    and D:  $\bigwedge K X. P X \implies P (Decrypt K X)$ 
  shows  $P msg$ 
proof (cases msg)
  case (Abs-Msg U)
  have  $P (Abs-Msg (msgrel `` \{U\}))$ 
  proof (induct U)
    case (NONCE N)
    with N show ?case by (simp add: Nonce-def)
  next
    case (MPAIR X Y)
    with M [of Abs-Msg (msgrel `` \{X\}) Abs-Msg (msgrel `` \{Y\})]
    show ?case by (simp add: MPair)
  next
    case (CRYPT K X)
    with C [of Abs-Msg (msgrel `` \{X\})]
    show ?case by (simp add: Crypt)
  next
    case (DECRYPT K X)
    with D [of Abs-Msg (msgrel `` \{X\})]
    show ?case by (simp add: Decrypt)
  qed
  with Abs-Msg show ?thesis by (simp only:)
qed

```

3.8 The Abstract Discriminator

However, as *Crypt-Nonce-neq-Nonce* above illustrates, we don't need this function in order to prove discrimination theorems.

definition

```

discrim :: msg  $\Rightarrow$  int where
  discrim X = the-elem ( $\bigcup U \in Rep-Msg X. \{freediscrim U\}$ )

```

lemma *discrim-congruent*: $(\lambda U. \{freediscrim U\})$ respects msgrel
 by (auto simp add: congruent-def msgrel-imp-eq-freediscrim)

Now prove the four equations for *discrim*

lemma *discrim-Nonce* [simp]: $discrim (Nonce N) = 0$
 by (simp add: discrim-def Nonce-def
 UN-equiv-class [OF equiv-msgrel discrim-congruent])

lemma *discrim-MPair* [simp]: $discrim (MPair X Y) = 1$
proof –
 have $\bigwedge U V. discrim (MPair (Abs-Msg (msgrel `` \{U\})) (Abs-Msg (msgrel `` \{V\}))) = 1$
 by (simp add: discrim-def MPair UN-equiv-class [OF equiv-msgrel discrim-congruent])

```

then show ?thesis
  by (metis eq-Abs-Msg)
qed

lemma discrim-Crypt [simp]:  $\text{discrim } (\text{Crypt } K \ X) = \text{discrim } X + 2$ 
  by (cases X) (use Crypt UN-equiv-class discrim-congruent discrim-def equiv-msgrel
in fastforce)

lemma discrim-Decrypt [simp]:  $\text{discrim } (\text{Decrypt } K \ X) = \text{discrim } X - 2$ 
  by (cases X) (use Decrypt UN-equiv-class discrim-congruent discrim-def equiv-msgrel
in fastforce)

end

```

4 Quotienting a Free Algebra Involving Nested Recursion

This is the development promised in Lawrence Paulson’s paper “Defining functions on equivalence classes” *ACM Transactions on Computational Logic* **7:40** (2006), 658–675, illustrating bare-bones quotient constructions. Any comparison using lifting and transfer should be done in a separate theory.

```
theory QuoNestedDataType imports Main begin
```

4.1 Defining the Free Algebra

Messages with encryption and decryption as free constructors.

```
datatype
  freeExp = VAR nat
          | PLUS freeExp freeExp
          | FNCALL nat freeExp list
```

```
datatype-compat freeExp
```

The equivalence relation, which makes PLUS associative.

The first rule is the desired equation. The next three rules make the equations applicable to subterms. The last two rules are symmetry and transitivity.

```
inductive-set
  exprel :: (freeExp * freeExp) set
  and exp-rel :: [freeExp, freeExp] => bool (infixl <~> 50)
  where
    X ~ Y ≡ (X, Y) ∈ exprel
    | ASSOC: PLUS X (PLUS Y Z) ~ PLUS (PLUS X Y) Z
    | VAR: VAR N ~ VAR N
    | PLUS:  $\llbracket X \sim X'; Y \sim Y' \rrbracket \implies \text{PLUS } X \ Y \sim \text{PLUS } X' \ Y'$ 
```

```

| FNCALL:  $(Xs, Xs') \in \text{listrel } \text{exprel} \implies \text{FNCALL } F \ Xs \sim \text{FNCALL } F \ Xs'$ 
| SYM:  $X \sim Y \implies Y \sim X$ 
| TRANS:  $\llbracket X \sim Y; Y \sim Z \rrbracket \implies X \sim Z$ 
monos listrel-mono

```

Proving that it is an equivalence relation

```

lemma exprel-refl:  $X \sim X$ 
  and list-exprel-refl:  $(Xs, Xs) \in \text{listrel}(\text{exprel})$ 
  by (induct  $X$  and  $Xs$  rule: compat-freeExp.induct compat-freeExp-list.induct)
    (blast intro: exprel.intros listrel.intros)+

```

theorem *equiv-exprel*: *equiv UNIV exprel*

proof (*rule equivI*)

```

  show  $\text{exprel} \subseteq \text{UNIV} \times \text{UNIV}$  by simp
  show refl exprel by (simp add: refl-on-def exprel-refl)
  show sym exprel by (simp add: sym-def, blast intro: exprel.SYM)
  show trans exprel by (simp add: trans-def, blast intro: exprel.TRANS)

```

qed

theorem *equiv-list-exprel*: *equiv UNIV (listrel exprel)*

using *equiv-listrel [OF equiv-exprel]* **by** *simp*

lemma *FNCALL-Cons*:

```

 $\llbracket X \sim X'; (Xs, Xs') \in \text{listrel}(\text{exprel}) \rrbracket \implies \text{FNCALL } F \ (X \# Xs) \sim \text{FNCALL } F \ (X' \# Xs')$ 
by (blast intro: exprel.intros listrel.intros)

```

4.2 Some Functions on the Free Algebra

4.2.1 The Set of Variables

A function to return the set of variables present in a message. It will be lifted to the initial algebra, to serve as an example of that process. Note that the "free" refers to the free datatype rather than to the concept of a free variable.

primrec *freevars* :: *freeExp* \Rightarrow *nat set* **and** *freevars-list* :: *freeExp list* \Rightarrow *nat set*

where

```

  freevars (VAR  $N$ ) =  $\{N\}$ 
| freevars (PLUS  $X \ Y$ ) = freevars  $X \cup \text{freevars } Y$ 
| freevars (FNCALL  $F \ Xs$ ) = freevars-list  $Xs$ 

| freevars-list [] =  $\{\}$ 
| freevars-list ( $X \ \# \ Xs$ ) = freevars  $X \cup \text{freevars-list } Xs$ 

```

This theorem lets us prove that the vars function respects the equivalence relation. It also helps us prove that Variable (the abstract constructor) is injective

theorem *exprel-imp-eq-freevars*: $U \sim V \implies \text{freevars } U = \text{freevars } V$

```

proof (induct set: exprel)
  case (FNCALL  $Xs$   $Xs'$   $F$ )
  then show ?case
    by (induct rule: listrel.induct) auto
qed (simp-all add: Un-assoc)

```

4.2.2 Functions for Freeness

A discriminator function to distinguish vars, sums and function calls

```

primrec freediscrim :: freeExp  $\Rightarrow$  int where
  freediscrim (VAR  $N$ ) = 0
| freediscrim (PLUS  $X$   $Y$ ) = 1
| freediscrim (FNCALL  $F$   $Xs$ ) = 2

```

```

theorem exprel-imp-eq-freediscrim:
   $U \sim V \Longrightarrow \text{freediscrim } U = \text{freediscrim } V$ 
by (induct set: exprel) auto

```

This function, which returns the function name, is used to prove part of the injectivity property for FnCall.

```

primrec freefun :: freeExp  $\Rightarrow$  nat where
  freefun (VAR  $N$ ) = 0
| freefun (PLUS  $X$   $Y$ ) = 0
| freefun (FNCALL  $F$   $Xs$ ) =  $F$ 

```

```

theorem exprel-imp-eq-freefun:
   $U \sim V \Longrightarrow \text{freefun } U = \text{freefun } V$ 
by (induct set: exprel) (simp-all add: listrel.intros)

```

This function, which returns the list of function arguments, is used to prove part of the injectivity property for FnCall.

```

primrec freeargs :: freeExp  $\Rightarrow$  freeExp list where
  freeargs (VAR  $N$ ) = []
| freeargs (PLUS  $X$   $Y$ ) = []
| freeargs (FNCALL  $F$   $Xs$ ) =  $Xs$ 

```

```

theorem exprel-imp-eqv-freeargs:
  assumes  $U \sim V$ 
  shows (freeargs  $U$ , freeargs  $V$ )  $\in$  listrel exprel
  using assms
proof induction
  case (FNCALL  $Xs$   $Xs'$   $F$ )
  then show ?case
    by (simp add: listrel-iff-nth)
next
  case (SYM  $X$   $Y$ )
  then show ?case

```

```

    by (meson equivE equiv-list-exprel symD)
next
  case (TRANS X Y Z)
  then show ?case
    by (meson equivE equiv-list-exprel transD)
qed (use listrel.simps in auto)

```

4.3 The Initial Algebra: A Quotiented Message Type

definition $Exp = UNIV // exprel$

```

typedef exp = Exp
morphisms Rep-Exp Abs-Exp
unfolding Exp-def by (auto simp add: quotient-def)

```

The abstract message constructors

definition

```

Var :: nat  $\Rightarrow$  exp where
Var N = Abs-Exp (exprel "{VAR N}")

```

definition

```

Plus :: [exp, exp]  $\Rightarrow$  exp where
Plus X Y =
  Abs-Exp ( $\bigcup U \in Rep\text{-}Exp\ X. \bigcup V \in Rep\text{-}Exp\ Y. exprel "{PLUS\ U\ V}"$ )

```

definition

```

FnCall :: [nat, exp list]  $\Rightarrow$  exp where
FnCall F Xs =
  Abs-Exp ( $\bigcup Us \in listset\ (map\ Rep\text{-}Exp\ Xs). exprel "{FNCALL\ F\ Us}"$ )

```

Reduces equality of equivalence classes to the *exprel* relation: ($exprel\ \{\{x\}\} = exprel\ \{\{y\}\} = (x \sim y)$)

lemmas *equiv-exprel-iff* = *eq-equiv-class-iff* [*OF equiv-exprel UNIV-I UNIV-I*]

declare *equiv-exprel-iff* [*simp*]

All equivalence classes belong to set of representatives

lemma *exprel-in-Exp* [*simp*]: $exprel\ \{\{U\}\} \in Exp$
by (*simp add: Exp-def quotientI*)

lemma *inj-on-Abs-Exp*: *inj-on Abs-Exp Exp*
by (*meson Abs-Exp-inject inj-onI*)

Reduces equality on abstractions to equality on representatives

declare *inj-on-Abs-Exp* [*THEN inj-on-eq-iff, simp*]

declare *Abs-Exp-inverse* [*simp*]

Case analysis on the representation of a exp as an equivalence class.

lemma *eq-Abs-Exp* [case-names *Abs-Exp*, cases type: *exp*]:
 $(\bigwedge U. z = \text{Abs-Exp}(\text{exprel}\{\{U\}\}) \implies P) \implies P$
by (*metis Abs-Exp-cases Exp-def quotientE*)

4.4 Every list of abstract expressions can be expressed in terms of a list of concrete expressions

definition

Abs-ExpList :: *freeExp list => exp list* **where**
Abs-ExpList *Xs* $\equiv \text{map } (\lambda U. \text{Abs-Exp}(\text{exprel}\{\{U\}\})) \text{ } Xs$

lemma *Abs-ExpList-Nil* [*simp*]: *Abs-ExpList* [] = []
by (*simp add: Abs-ExpList-def*)

lemma *Abs-ExpList-Cons* [*simp*]:
Abs-ExpList (*X* # *Xs*) = *Abs-Exp* (*exprel*{*X*}) # *Abs-ExpList* *Xs*
by (*simp add: Abs-ExpList-def*)

lemma *ExpList-rep*: $\exists Us. z = \text{Abs-ExpList } Us$
by (*smt (verit, del-ists) Abs-ExpList-def eq-Abs-Exp ex-map-conv*)

4.4.1 Characteristic Equations for the Abstract Constructors

lemma *Plus*: *Plus* (*Abs-Exp*(*exprel*{*U*})) (*Abs-Exp*(*exprel*{*V*})) =
Abs-Exp (*exprel*{*PLUS U V*})

proof –

have ($\lambda U V. \text{exprel}\{\{PLUS\ U\ V\}\}$ *respects2* *exprel*)
by (*auto simp add: congruent2-def exprel.PLUS*)
thus *?thesis*
by (*simp add: Plus-def UN-equiv-class2 [OF equiv-exprel equiv-exprel]*)

qed

It is not clear what to do with *FnCall*: it's argument is an abstraction of an *exp list*. Is it just *Nil* or *Cons*? What seems to work best is to regard an *exp list* as a *listrel exprel* equivalence class

This theorem is easily proved but never used. There's no obvious way even to state the analogous result, *FnCall-Cons*.

lemma *FnCall-Nil*: *FnCall* *F* [] = *Abs-Exp* (*exprel*{*FNCALL F* []})
by (*simp add: FnCall-def*)

lemma *FnCall-respects*:

($\lambda Us. \text{exprel}\{\{FNCALL\ F\ Us\}\}$ *respects* (*listrel exprel*)
by (*auto simp add: congruent-def exprel.FNCALL*)

lemma *FnCall-sing*:

FnCall *F* [*Abs-Exp*(*exprel*{*U*})] = *Abs-Exp* (*exprel*{*FNCALL F* [*U*]})

proof –

have ($\lambda U. \text{exprel}\{\{FNCALL\ F\ [U]\}\}$ *respects* *exprel*)

```

    by (auto simp add: congruent-def FNCALL-Cons listrel.intros)
  thus ?thesis
    by (simp add: FnCall-def UN-equiv-class [OF equiv-exprel])
qed

```

lemma *listset-Rep-Exp-Abs-Exp*:

```

  listset (map Rep-Exp (Abs-ExpList Us)) = listrel exprel "{Us}"
  by (induct Us) (simp-all add: listrel-Cons Abs-ExpList-def)

```

lemma *FnCall*:

```

  FnCall F (Abs-ExpList Us) = Abs-Exp (exprel "{FNCALL F Us}")
proof -
  have (λUs. exprel "{FNCALL F Us}") respects (listrel exprel)
    by (auto simp add: congruent-def exprel.FNCALL)
  thus ?thesis
    by (simp add: FnCall-def UN-equiv-class [OF equiv-list-exprel]
        listset-Rep-Exp-Abs-Exp)
qed

```

Establishing this equation is the point of the whole exercise

theorem *Plus-assoc*: $Plus\ X\ (Plus\ Y\ Z) = Plus\ (Plus\ X\ Y)\ Z$

```

  by (cases X, cases Y, cases Z, simp add: Plus exprel.ASSOC)

```

4.5 The Abstract Function to Return the Set of Variables

definition

$vars :: exp \Rightarrow nat\ set$ **where** $vars\ X \equiv (\bigcup U \in Rep-Exp\ X. freevars\ U)$

lemma *vars-respects*: *freevars respects exprel*

```

  by (auto simp add: congruent-def exprel-imp-eq-freevars)

```

The extension of the function *vars* to lists

primrec *vars-list* :: $exp\ list \Rightarrow nat\ set$ **where**

```

  vars-list [] = {}
  | vars-list (E#Es) = vars E ∪ vars-list Es

```

Now prove the three equations for *vars*

lemma *vars-Variable* [simp]: $vars\ (Var\ N) = \{N\}$

```

  by (simp add: vars-def Var-def
      UN-equiv-class [OF equiv-exprel vars-respects])

```

lemma *vars-Plus* [simp]: $vars\ (Plus\ X\ Y) = vars\ X \cup vars\ Y$

proof -

```

  have ∧U V. [X = Abs-Exp (exprel "{U}"); Y = Abs-Exp (exprel "{V}")]
    ⇒ vars (Plus X Y) = vars X ∪ vars Y
    by (simp add: vars-def Plus UN-equiv-class [OF equiv-exprel vars-respects])
  then show ?thesis
    by (meson eq-Abs-Exp)
qed

```

lemma *vars-FnCall* [simp]: $\text{vars } (\text{FnCall } F \ Xs) = \text{vars-list } Xs$
proof –
 have $\text{vars } (\text{Abs-Exp } (\text{exprel}''\{\text{FNCALL } F \ Us\})) = \text{vars-list } (\text{Abs-ExpList } Us)$ **for** Us
 by (induct Us) (auto simp: vars-def UN-equiv-class [OF equiv-exprel vars-respects])
 then show ?thesis
 by (metis ExpList-rep FnCall)
qed

lemma *vars-FnCall-Nil*: $\text{vars } (\text{FnCall } F \ \text{Nil}) = \{\}$
 by simp

lemma *vars-FnCall-Cons*: $\text{vars } (\text{FnCall } F \ (X\#Xs)) = \text{vars } X \cup \text{vars-list } Xs$
 by simp

4.6 Injectivity Properties of Some Constructors

lemma *VAR-imp-eq*: $\text{VAR } m \sim \text{VAR } n \implies m = n$
 by (drule exprel-imp-eq-freevars, simp)

Can also be proved using the function *vars*

lemma *Var-Var-eq* [iff]: $(\text{Var } m = \text{Var } n) = (m = n)$
 by (auto simp add: Var-def exprel-refl dest: VAR-imp-eq)

lemma *VAR-neqv-PLUS*: $\text{VAR } m \sim \text{PLUS } X \ Y \implies \text{False}$
 using exprel-imp-eq-freediscrim **by** force

theorem *Var-neqv-Plus* [iff]: $\text{Var } N \neq \text{Plus } X \ Y$
proof –
 have $\bigwedge U \ V. \llbracket X = \text{Abs-Exp } (\text{exprel}''\{U\}); Y = \text{Abs-Exp } (\text{exprel}''\{V\}) \rrbracket \implies \text{Var } N \neq \text{Plus } X \ Y$
 using Plus VAR-neqv-PLUS Var-def **by** force
 then show ?thesis
 by (meson eq-Abs-Exp)
qed

theorem *Var-neqv-FnCall* [iff]: $\text{Var } N \neq \text{FnCall } F \ Xs$
proof –
 have $\bigwedge Us. \text{Var } N \neq \text{FnCall } F \ (\text{Abs-ExpList } Us)$
 using FnCall Var-def exprel-imp-eq-freediscrim **by** fastforce
 then show ?thesis
 by (metis ExpList-rep)
qed

4.7 Injectivity of *Fncall*

definition
 $\text{fun} :: \text{exp} \Rightarrow \text{nat}$

where $\text{fun } X \equiv \text{the-elem } (\bigcup U \in \text{Rep-Exp } X. \{\text{freefun } U\})$

lemma *fun-respects*: $(\lambda U. \{\text{freefun } U\})$ respects *exprel*
by (*auto simp add: congruent-def exprel-imp-eq-freefun*)

lemma *fun-FnCall [simp]*: $\text{fun } (\text{FnCall } F \text{ } Xs) = F$

proof –

have $\bigwedge Us. \text{fun } (\text{FnCall } F (\text{Abs-ExpList } Us)) = F$

using *FnCall UN-equiv-class [OF equiv-exprel] fun-def fun-respects* **by** *fastforce*

then show *?thesis*

by (*metis ExpList-rep*)

qed

definition

$\text{args} :: \text{exp} \Rightarrow \text{exp list}$ **where**

$\text{args } X = \text{the-elem } (\bigcup U \in \text{Rep-Exp } X. \{\text{Abs-ExpList } (\text{freeargs } U)\})$

This result can probably be generalized to arbitrary equivalence relations, but with little benefit here.

lemma *Abs-ExpList-eq*:

$(y, z) \in \text{listrel exprel} \implies \text{Abs-ExpList } (y) = \text{Abs-ExpList } (z)$

by (*induct set: listrel*) *simp-all*

lemma *args-respects*: $(\lambda U. \{\text{Abs-ExpList } (\text{freeargs } U)\})$ respects *exprel*

by (*auto simp add: congruent-def Abs-ExpList-eq exprel-imp-eqv-freeargs*)

lemma *args-FnCall [simp]*: $\text{args } (\text{FnCall } F \text{ } Xs) = Xs$

proof –

have $\bigwedge Us. Xs = \text{Abs-ExpList } Us \implies \text{args } (\text{FnCall } F \text{ } Xs) = Xs$

by (*simp add: FnCall args-def UN-equiv-class [OF equiv-exprel args-respects]*)

then show *?thesis*

by (*metis ExpList-rep*)

qed

lemma *FnCall-FnCall-eq [iff]*: $(\text{FnCall } F \text{ } Xs = \text{FnCall } F' \text{ } Xs') \longleftrightarrow (F=F' \wedge Xs=Xs')$

by (*metis args-FnCall fun-FnCall*)

4.8 The Abstract Discriminator

However, as *FnCall-Var-neq-Var* illustrates, we don't need this function in order to prove discrimination theorems.

definition

$\text{discrim} :: \text{exp} \Rightarrow \text{int}$ **where**

$\text{discrim } X = \text{the-elem } (\bigcup U \in \text{Rep-Exp } X. \{\text{freediscrim } U\})$

lemma *discrim-respects*: $(\lambda U. \{\text{freediscrim } U\})$ respects *exprel*

by (*auto simp add: congruent-def exprel-imp-eq-freediscrim*)

Now prove the four equations for *discrim*

lemma *discrim-Var* [simp]: *discrim* (Var *N*) = 0
 by (simp add: *discrim-def Var-def UN-equiv-class [OF equiv-exprel discrim-respects]*)

lemma *discrim-Plus* [simp]: *discrim* (Plus *X Y*) = 1
proof –
 have $\bigwedge U V. \llbracket X = \text{Abs-Exp } (\text{exprel}\{\!U\}) ; Y = \text{Abs-Exp } (\text{exprel}\{\!V\}) \rrbracket \implies$
discrim (Plus *X Y*) = 1
 by (simp add: *discrim-def Plus UN-equiv-class [OF equiv-exprel discrim-respects]*)
 then show ?thesis
 by (meson eq-Abs-Exp)
qed

lemma *discrim-FnCall* [simp]: *discrim* (FnCall *F Xs*) = 2
proof –
 have *discrim* (FnCall *F (Abs-ExpList Us)*) = 2 **for** *Us*
 by (simp add: *discrim-def FnCall UN-equiv-class [OF equiv-exprel discrim-respects]*)
 then show ?thesis
 by (metis ExpList-rep)
qed

The structural induction rule for the abstract type

theorem *exp-inducts*:
 assumes *V*: $\bigwedge \text{nat}. P1 \text{ (Var nat)}$
 and *P*: $\bigwedge \text{exp1 exp2}. \llbracket P1 \text{ exp1} ; P1 \text{ exp2} \rrbracket \implies P1 \text{ (Plus exp1 exp2)}$
 and *F*: $\bigwedge \text{nat list}. P2 \text{ list} \implies P1 \text{ (FnCall nat list)}$
 and *Nil*: *P2* []
 and *Cons*: $\bigwedge \text{exp list}. \llbracket P1 \text{ exp} ; P2 \text{ list} \rrbracket \implies P2 \text{ (exp \# list)}$
 shows *P1 exp and P2 list*
proof –
 obtain *U* **where** *exp*: *exp* = (Abs-Exp (exprel{*U*})) **by** (cases *exp*)
 obtain *Us* **where** *list*: *list* = Abs-ExpList *Us* **by** (metis ExpList-rep)
 have *P1* (Abs-Exp (exprel{*U*})) **and** *P2* (Abs-ExpList *Us*)
proof (induct *U* **and** *Us* rule: compat-freeExp.induct compat-freeExp-list.induct)
 case (VAR *nat*)
 with *V* show ?case **by** (simp add: Var-def)
 next
 case (PLUS *X Y*)
 with *P* [of Abs-Exp (exprel{*X*}) Abs-Exp (exprel{*Y*})]
 show ?case **by** (simp add: Plus)
 next
 case (FNCALL *nat list*)
 with *F* [of Abs-ExpList *list*]
 show ?case **by** (simp add: FnCall)
 next
 case Nil-freeExp

```

    with Nil show ?case by simp
  next
    case Cons-freeExp
    with Cons show ?case by simp
  qed
  with exp and list show P1 exp and P2 list by (simp-all only:)
qed
end

```

5 Terms over a given alphabet

```

theory Term
imports Main
begin

```

```

datatype ('a, 'b) term =
  Var 'a
| App 'b ('a, 'b) term list

```

Substitution function on terms

```

primrec subst-term :: ('a  $\Rightarrow$  ('a, 'b) term)  $\Rightarrow$  ('a, 'b) term  $\Rightarrow$  ('a, 'b) term
and subst-term-list :: ('a  $\Rightarrow$  ('a, 'b) term)  $\Rightarrow$  ('a, 'b) term list  $\Rightarrow$  ('a, 'b) term
list
where
  subst-term f (Var a) = f a
| subst-term f (App b ts) = App b (subst-term-list f ts)
| subst-term-list f [] = []
| subst-term-list f (t # ts) = subst-term f t # subst-term-list f ts

```

A simple theorem about composition of substitutions

```

lemma subst-comp:
  subst-term (subst-term f1  $\circ$  f2) t =
    subst-term f1 (subst-term f2 t)
and subst-term-list (subst-term f1  $\circ$  f2) ts =
  subst-term-list f1 (subst-term-list f2 ts)
by (induct t and ts rule: subst-term.induct subst-term-list.induct) simp-all

```

Alternative induction rule

```

lemma
  assumes var:  $\bigwedge v. P \text{ (Var } v)$ 
  and app:  $\bigwedge f \text{ ts. } (\forall t \in \text{set } ts. P \text{ } t) \implies P \text{ (App } f \text{ ts)}$ 
  shows term-induct2:  $P \text{ } t$ 
  and  $\forall t \in \text{set } ts. P \text{ } t$ 
  apply (induct t and ts rule: subst-term.induct subst-term-list.induct)
  apply (rule var)
  apply (rule app)

```

```

    apply assumption
    apply simp-all
done

```

```

end

```

```

theory Sexp
imports HOL-Library.Old-Datatype
begin

```

```

type-synonym 'a item = 'a Old-Datatype.item
abbreviation Leaf == Old-Datatype.Leaf
abbreviation Numb == Old-Datatype.Numb

```

```

inductive-set
  sexp      :: 'a item set
  where
    LeafI: Leaf(a) ∈ sexp
  | NumbI: Numb(i) ∈ sexp
  | SconsI: [| M ∈ sexp; N ∈ sexp |] ==> Scons M N ∈ sexp

```

```

definition
  sexp-case :: ['a=>'b, nat=>'b, ['a item, 'a item]=>'b,
               'a item]=>'b] where
  sexp-case c d e M = (THE z. (∃ x. M=Leaf(x) & z=c(x))
    | (∃ k. M=Numb(k) & z=d(k))
    | (∃ N1 N2. M = Scons N1 N2 & z=e N1 N2))

```

```

definition
  pred-sexp :: ('a item * 'a item)set where
  pred-sexp = (⋃ M ∈ sexp. ⋃ N ∈ sexp. {(M, Scons M N), (N, Scons M N)})

```

```

definition
  sexp-rec :: ['a item, 'a=>'b, nat=>'b,
               ['a item, 'a item, 'b, 'b]=>'b] => 'b where
  sexp-rec M c d e = wfrec pred-sexp
    (%g. sexp-case c d (%N1 N2. e N1 N2 (g N1) (g N2))) M

```

```

lemma sexp-case-Leaf [simp]: sexp-case c d e (Leaf a) = c(a)
by (simp add: sexp-case-def, blast)

```

```

lemma sexp-case-Numb [simp]: sexp-case c d e (Numb k) = d(k)
by (simp add: sexp-case-def, blast)

```

```

lemma sexp-case-Scons [simp]: sexp-case c d e (Scons M N) = e M N

```

by (*simp add: sexp-case-def*)

lemma *sexp-In0I*: $M \in \text{sexp} \implies \text{In0}(M) \in \text{sexp}$
apply (*simp add: In0-def*)
apply (*erule sexp.NumbI [THEN sexp.SconsI]*)
done

lemma *sexp-In1I*: $M \in \text{sexp} \implies \text{In1}(M) \in \text{sexp}$
apply (*simp add: In1-def*)
apply (*erule sexp.NumbI [THEN sexp.SconsI]*)
done

declare *sexp.intros* [*intro, simp*]

lemma *range-Leaf-subset-sexp*: $\text{range}(\text{Leaf}) \leq \text{sexp}$
by *blast*

lemma *Scons-D*: $\text{Scons } M \ N \in \text{sexp} \implies M \in \text{sexp} \ \& \ N \in \text{sexp}$
by (*induct S == Scons M N set: sexp*) *auto*

lemma *pred-sexp-subset-Sigma*: $\text{pred-sexp} \leq \text{sexp} \times \text{sexp}$
by (*simp add: pred-sexp-def*) *blast*

lemmas *trancI-pred-sexpD1* =
 pred-sexp-subset-Sigma
 [*THEN trancI-subset-Sigma, THEN subsetD, THEN SigmaD1*]
and *trancI-pred-sexpD2* =
 pred-sexp-subset-Sigma
 [*THEN trancI-subset-Sigma, THEN subsetD, THEN SigmaD2*]

lemma *pred-sexpI1*:
 [$M \in \text{sexp}; N \in \text{sexp}$] $\implies (M, \text{Scons } M \ N) \in \text{pred-sexp}$
by (*simp add: pred-sexp-def, blast*)

lemma *pred-sexpI2*:
 [$M \in \text{sexp}; N \in \text{sexp}$] $\implies (N, \text{Scons } M \ N) \in \text{pred-sexp}$
by (*simp add: pred-sexp-def, blast*)

lemmas *pred-sexp-t1* [*simp*] = *pred-sexpI1* [*THEN r-into-trancI*]
and *pred-sexp-t2* [*simp*] = *pred-sexpI2* [*THEN r-into-trancI*]

lemmas *pred-sexp-trans1* [*simp*] = *trans-trancl* [*THEN transD*, *OF - pred-sexp-t1*]
and *pred-sexp-trans2* [*simp*] = *trans-trancl* [*THEN transD*, *OF - pred-sexp-t2*]

declare *cut-apply* [*simp*]

lemma *pred-sexpE*:

[[*p* ∈ *pred-sexp*;
 !!*M N*. [[*p* = (*M*, *Scons M N*); *M* ∈ *sexp*; *N* ∈ *sexp*]] ==> *R*;
 !!*M N*. [[*p* = (*N*, *Scons M N*); *M* ∈ *sexp*; *N* ∈ *sexp*]] ==> *R*
]] ==> *R*
by (*simp add: pred-sexp-def, blast*)

lemma *wf-pred-sexp*: *wf*(*pred-sexp*)

apply (*rule pred-sexp-subset-Sigma* [*THEN wfI*])

apply (*erule sexp.induct*)

apply (*blast elim!: pred-sexpE*)+

done

lemma *sexp-rec-unfold-lemma*:

(%*M*. *sexp-rec M c d e*) ==
wfrec pred-sexp (%*g*. *sexp-case c d* (%*N1 N2*. *e N1 N2* (*g N1*) (*g N2*)))
by (*simp add: sexp-rec-def*)

lemmas *sexp-rec-unfold* = *def-wfrec* [*OF sexp-rec-unfold-lemma wf-pred-sexp*]

lemma *sexp-rec-Leaf*: *sexp-rec* (*Leaf a*) *c d h* = *c(a)*

apply (*subst sexp-rec-unfold*)

apply (*rule sexp-case-Leaf*)

done

lemma *sexp-rec-Numb*: *sexp-rec* (*Numb k*) *c d h* = *d(k)*

apply (*subst sexp-rec-unfold*)

apply (*rule sexp-case-Numb*)

done

lemma *sexp-rec-Scons*: [[*M* ∈ *sexp*; *N* ∈ *sexp*]] ==>

sexp-rec (*Scons M N*) *c d h* = *h M N* (*sexp-rec M c d h*) (*sexp-rec N c d h*)

apply (*rule sexp-rec-unfold* [*THEN trans*])

apply (*simp add: pred-sexpI1 pred-sexpI2*)

done

end

6 Extended List Theory (old)

theory *SList*
imports *Sexp*
begin

definition

NIL :: '*a item* **where**
NIL = *In0*(*Numb*(0))

definition

CONS :: ['*a item*, '*a item*] => '*a item* **where**
CONS *M N* = *In1*(*Scons* *M N*)

inductive-set

list :: '*a item set* => '*a item set*
for *A* :: '*a item set*
where
 NIL-I: *NIL* ∈ *list A*
 | *CONS-I*: [| *a* ∈ *A*; *M* ∈ *list A* |] ==> *CONS a M* ∈ *list A*

definition *List* = *list* (*range Leaf*)

typedef '*a list* = *List* :: '*a item set*
 morphisms *Rep-List Abs-List*
 unfolding *List-def* **by** (*blast intro: list.NIL-I*)

abbreviation *Case* == *Old-Datatype.Case*

abbreviation *Split* == *Old-Datatype.Split*

definition

List-case :: ['*b*, ['*a item*, '*a item*] => '*b*, '*a item*] => '*b* **where**
List-case c d = *Case*(%*x. c*)(*Split*(*d*))

definition

List-rec :: ['*a item*, '*b*, ['*a item*, '*a item*, '*b*] => '*b*] => '*b* **where**

$List-rec\ M\ c\ d = wfrec\ (pred-sexp^+)$
 $(\%g. List-case\ c\ (\%x\ y. d\ x\ y\ (g\ y)))\ M$

no-translations

$[x, xs] == x\#[xs]$
 $[x] == x\#[]$

unbundle *no list-syntax*

definition

$Nil :: 'a\ list$ $(\langle [] \rangle)$ **where**
 $Nil = Abs-List(NIL)$

definition

$Cons :: ['a, 'a\ list] => 'a\ list$ **(infixr** $\langle \# \rangle$ 65) **where**
 $x\#xs = Abs-List(CONS\ (Leaf\ x)(Rep-List\ xs))$

definition

$list-rec :: ['a\ list, 'b, ['a, 'a\ list, 'b] => 'b] => 'b$ **where**
 $list-rec\ l\ c\ d =$
 $List-rec(Rep-List\ l)\ c\ (\%x\ y\ r. d(inv\ Leaf\ x)(Abs-List\ y)\ r)$

definition

$list-case :: ['b, ['a, 'a\ list] => 'b, 'a\ list] => 'b$ **where**
 $list-case\ a\ f\ xs = list-rec\ xs\ a\ (\%x\ xs\ r. f\ x\ xs)$

translations

$[x, xs] == x\#[xs]$
 $[x] == x\#[]$

$case\ xs\ of\ [] => a \mid y\#ys => b == CONST\ list-case(a, \%y\ ys. b, xs)$

definition

Rep-map :: ('b => 'a item) => ('b list => 'a item) **where**
Rep-map f xs = list-rec xs NIL(%x l r. CONS(f x) r)

definition

Abs-map :: ('a item => 'b) => 'a item => 'b list **where**
Abs-map g M = List-rec M Nil (%N L r. g(N)#r)

definition

map :: ('a=>'b) => ('a list => 'b list) **where**
map f xs = list-rec xs [] (%x l r. f(x)#r)

primrec *take* :: ['a list,nat] => 'a list **where**

take-0: *take* xs 0 = []
| *take-Suc*: *take* xs (Suc n) = list-case [] (%x l. x # *take* l n) xs

lemma *ListI*: $x \in \text{list } (\text{range Leaf}) \implies x \in \text{List}$

by (*simp add: List-def*)

lemma *ListD*: $x \in \text{List} \implies x \in \text{list } (\text{range Leaf})$

by (*simp add: List-def*)

lemma *list-unfold*: $\text{list}(A) = \text{usum } \{\text{Numb}(0)\} (\text{uprod } A (\text{list}(A)))$

by (*fast intro!*: list.intros [unfolded NIL-def CONS-def]
elim: list.cases [unfolded NIL-def CONS-def])

lemma *list-mono*: $A \leq B \implies \text{list}(A) \leq \text{list}(B)$

apply (*rule subsetI*)

apply (*erule list.induct*)

apply (*auto intro!*: list.intros)

done

lemma *list-serp*: $\text{list}(\text{serp}) \leq \text{serp}$

apply (*rule subsetI*)

apply (*erule list.induct*)

apply (*unfold NIL-def CONS-def*)

apply (*auto intro*: serp.intros serp-In0I serp-In1I)

done

lemmas *list-subset-serp* = subset-trans [OF *list-mono list-serp*]

```

lemma list-induct:
  [|  $P(\text{Nil})$ ;
     $\forall x\ xs. P(xs) \implies P(x \# xs)$  |]  $\implies P(l)$ 
apply (unfold Nil-def Cons-def)
apply (rule Rep-List-inverse [THEN subst])

apply (rule Rep-List [unfolded List-def, THEN list.induct], simp)
apply (erule Abs-List-inverse [unfolded List-def, THEN subst], blast)
done

```

```

lemma inj-on-Abs-list: inj-on Abs-List (list(range Leaf))
apply (rule inj-on-inverseI)
apply (erule Abs-List-inverse [unfolded List-def])
done

```

```

lemma CONS-not-NIL [iff]:  $\text{CONS } M\ N \sim = \text{NIL}$ 
by (simp add: NIL-def CONS-def)

```

```

lemmas NIL-not-CONS [iff] = CONS-not-NIL [THEN not-sym]
lemmas CONS-neq-NIL = CONS-not-NIL [THEN notE]
lemmas NIL-neq-CONS = sym [THEN CONS-neq-NIL]

```

```

lemma Cons-not-Nil [iff]:  $x \# xs \sim = \text{Nil}$ 
apply (unfold Nil-def Cons-def)
apply (rule CONS-not-NIL [THEN inj-on-Abs-list [THEN inj-on-contrad]])
apply (simp-all add: list.intros rangeI Rep-List [unfolded List-def])
done

```

```

lemmas Nil-not-Cons = Cons-not-Nil [THEN not-sym]
declare Nil-not-Cons [iff]
lemmas Cons-neq-Nil = Cons-not-Nil [THEN notE]
lemmas Nil-neq-Cons = sym [THEN Cons-neq-Nil]

```

```

lemma CONS-CONS-eq [iff]:  $(\text{CONS } K\ M) = (\text{CONS } L\ N) = (K=L \ \& \ M=N)$ 
by (simp add: CONS-def)

```

```

declare Rep-List [THEN ListD, intro] ListI [intro]
declare list.intros [intro,simp]
declare Leaf-inject [dest!]

```

lemma *Cons-Cons-eq* [iff]: $(x\#xs=y\#ys) = (x=y \ \& \ xs=ys)$
apply (*simp add: Cons-def*)
apply (*subst Abs-List-inject*)
apply (*auto simp add: Rep-List-inject*)
done

lemmas *Cons-inject2* = *Cons-Cons-eq* [THEN *iffD1*, THEN *conjE*]

lemma *CONS-D*: $CONS \ M \ N \in list(A) \implies M \in A \ \& \ N \in list(A)$
by (*induct L == CONS M N rule: list.induct*) *auto*

lemma *sexp-CONS-D*: $CONS \ M \ N \in sexp \implies M \in sexp \wedge N \in sexp$
apply (*simp add: CONS-def In1-def*)
apply (*fast dest!: Scons-D*)
done

lemma *not-CONS-self*: $N \in list(A) \implies \forall M. N \neq CONS \ M \ N$
apply (*erule list.induct*) **apply** *simp-all* **done**

lemma *not-Cons-self2*: $\forall x. l \neq x\#l$
by (*induct l rule: list-induct*) *simp-all*

lemma *neq-Nil-conv2*: $(xs \neq []) = (\exists y \ ys. xs = y\#ys)$
by (*induct xs rule: list-induct*) *auto*

lemma *List-case-NIL* [*simp*]: $List-case \ c \ h \ NIL = c$
by (*simp add: List-case-def NIL-def*)

lemma *List-case-CONS* [*simp*]: $List-case \ c \ h \ (CONS \ M \ N) = h \ M \ N$
by (*simp add: List-case-def CONS-def*)

lemma *List-rec-unfold-lemma*:
 $(\lambda M. List-rec \ M \ c \ d) \equiv$
 $wfrec \ (pred-sexp^+) \ (\lambda g. List-case \ c \ (\lambda x \ y. d \ x \ y \ (g \ y)))$
by (*simp add: List-rec-def*)

lemmas *List-rec-unfold* =

def-wfrec [OF List-rec-unfold-lemma wf-pred-sexp [THEN wf-trancl]]

lemma *pred-sexp-CONS-I1*:

$[| M \in \text{sexp}; N \in \text{sexp} |] \implies (M, \text{CONS } M \ N) \in \text{pred-sexp}^+$
by (*simp add: CONS-def In1-def*)

lemma *pred-sexp-CONS-I2*:

$[| M \in \text{sexp}; N \in \text{sexp} |] \implies (N, \text{CONS } M \ N) \in \text{pred-sexp}^+$
by (*simp add: CONS-def In1-def*)

lemma *pred-sexp-CONS-D*:

$(\text{CONS } M1 \ M2, N) \in \text{pred-sexp}^+ \implies$
 $(M1, N) \in \text{pred-sexp}^+ \wedge (M2, N) \in \text{pred-sexp}^+$
apply (*frule pred-sexp-subset-Sigma [THEN trancl-subset-Sigma, THEN subsetD]*)
apply (*blast dest!: sexp-CONS-D intro: pred-sexp-CONS-I1 pred-sexp-CONS-I2*
trans-trancl [THEN transD])
done

lemma *List-rec-NIL [simp]*: *List-rec NIL c h = c*

apply (*rule List-rec-unfold [THEN trans]*)
apply (*simp add: List-case-NIL*)
done

lemma *List-rec-CONS [simp]*:

$[| M \in \text{sexp}; N \in \text{sexp} |]$
 $\implies \text{List-rec } (\text{CONS } M \ N) \ c \ h = h \ M \ N \ (\text{List-rec } N \ c \ h)$
apply (*rule List-rec-unfold [THEN trans]*)
apply (*simp add: pred-sexp-CONS-I2*)
done

lemmas *Rep-List-in-sexp =*

subsetD [OF range-Leaf-subset-sexp [THEN list-subset-sexp]
Rep-List [THEN ListD]]

lemma *list-rec-Nil [simp]*: *list-rec Nil c h = c*

by (*simp add: list-rec-def ListI [THEN Abs-List-inverse] Nil-def*)

lemma *list-rec-Cons [simp]*: *list-rec (a#l) c h = h a l (list-rec l c h)*

by (*simp add: list-rec-def ListI [THEN Abs-List-inverse] Cons-def*
Rep-List-inverse Rep-List [THEN ListD] inj-Leaf Rep-List-in-sexp)

lemma *List-rec-type*:

$[[M \in \text{list}(A);$
 $A \leq \text{sexp};$
 $c \in C(\text{NIL});$
 $\bigwedge x y r. [[x \in A; y \in \text{list}(A); r \in C(y)]] \implies h x y r \in C(\text{CONS } x y)$
 $]] \implies \text{List-rec } M c h \in C(M :: 'a \text{ item})$
apply (*erule list.induct, simp*)
apply (*insert list-subset-sexp*)
apply (*subst List-rec-CONS, blast+*)
done

lemma *Rep-map-Nil [simp]*: $\text{Rep-map } f \text{ Nil} = \text{NIL}$
by (*simp add: Rep-map-def*)

lemma *Rep-map-Cons [simp]*:
 $\text{Rep-map } f (x \# xs) = \text{CONS}(f x)(\text{Rep-map } f xs)$
by (*simp add: Rep-map-def*)

lemma *Rep-map-type*: $(\bigwedge x. f(x) \in A) \implies \text{Rep-map } f xs \in \text{list}(A)$
apply (*simp add: Rep-map-def*)
apply (*rule list-induct, auto*)
done

lemma *Abs-map-NIL [simp]*: $\text{Abs-map } g \text{ NIL} = \text{Nil}$
by (*simp add: Abs-map-def*)

lemma *Abs-map-CONS [simp]*:
 $[[M \in \text{sexp}; N \in \text{sexp}]] \implies \text{Abs-map } g (\text{CONS } M N) = g(M) \# \text{Abs-map } g N$
by (*simp add: Abs-map-def*)

lemma *def-list-rec-NilCons*:

$[[\bigwedge xs. f(xs) = \text{list-rec } xs c h]]$
 $\implies f [] = c \wedge f(x \# xs) = h x xs (f xs)$
by *simp*

lemma *Abs-map-inverse*:

$[[M \in \text{list}(A); A \leq \text{sexp}; \bigwedge z. z \in A \implies f(g(z)) = z]]$
 $\implies \text{Rep-map } f (\text{Abs-map } g M) = M$


```

apply (erule list.induct, simp-all)
apply (insert list-subset-sexp)
apply (subst Abs-map-CONS, blast)
apply blast
apply simp
done

```

Better to have a single theorem with a conjunctive conclusion.

```

declare def-list-rec-NilCons [OF list-case-def, simp]

```

```

lemma expand-list-case:
   $P(\text{list-case } a \ f \ xs) = ((xs = [] \longrightarrow P \ a) \wedge (\forall y \ ys. \ xs = y \# ys \longrightarrow P(f \ y \ ys)))$ 
by (induct xs rule: list-induct) simp-all

```

```

declare def-list-rec-NilCons [OF map-def, simp]

```

```

lemma Abs-Rep-map:
   $(\bigwedge x. f(x) \in \text{sexp}) \implies$ 
   $\text{Abs-map } g \ (\text{Rep-map } f \ xs) = \text{map } (\lambda t. g(f(t))) \ xs$ 
apply (induct xs rule: list-induct)
apply (simp-all add: Rep-map-type list-sexp [THEN subsetD])
done

```

```

lemma map-ident [simp]:  $\text{map}(\lambda x. x)(xs) = xs$ 
by (induct xs rule: list-induct) simp-all

```

```

lemma map-compose:  $\text{map}(f \ o \ g)(xs) = \text{map } f \ (\text{map } g \ xs)$ 
apply (simp add: o-def)
apply (induct xs rule: list-induct)
apply simp-all
done

```

```

lemma take-Suc1 [simp]:  $\text{take } [] \ (\text{Suc } x) = []$ 
by simp

```

```

lemma take-Suc2 [simp]:  $\text{take}(a \# xs)(\text{Suc } x) = a \# \text{take } xs \ x$ 

```

by *simp*

lemma *take-Nil* [*simp*]: *take* [] *n* = []
by (*induct n*) *simp-all*

lemma *take-take-eq* [*simp*]: $\forall n. \text{take } (\text{take } xs \ n) \ n = \text{take } xs \ n$
apply (*induct xs rule: list-induct*)
apply *simp-all*
apply (*rule allI*)
apply (*induct-tac n*)
apply *auto*
done

end

7 Arithmetic and boolean expressions

theory *ABexp*
imports *Main*
begin

datatype 'a *aexp* =
 IF 'a *bexp* 'a *aexp* 'a *aexp*
 | *Sum* 'a *aexp* 'a *aexp*
 | *Diff* 'a *aexp* 'a *aexp*
 | *Var* 'a
 | *Num* *nat*
and 'a *bexp* =
 Less 'a *aexp* 'a *aexp*
 | *And* 'a *bexp* 'a *bexp*
 | *Neg* 'a *bexp*

Evaluation of arithmetic and boolean expressions

primrec *evala* :: ('a \Rightarrow *nat*) \Rightarrow 'a *aexp* \Rightarrow *nat*
 and *evalb* :: ('a \Rightarrow *nat*) \Rightarrow 'a *bexp* \Rightarrow *bool*
where
 evala env (IF b a1 a2) = (*if evalb env b then evala env a1 else evala env a2*)
 | *evala env (Sum a1 a2)* = *evala env a1* + *evala env a2*
 | *evala env (Diff a1 a2)* = *evala env a1* - *evala env a2*
 | *evala env (Var v)* = *env v*
 | *evala env (Num n)* = *n*

 | *evalb env (Less a1 a2)* = (*evala env a1* < *evala env a2*)
 | *evalb env (And b1 b2)* = (*evalb env b1* \wedge *evalb env b2*)
 | *evalb env (Neg b)* = (\neg *evalb env b*)

Substitution on arithmetic and boolean expressions

primrec *subst* :: ('a \Rightarrow 'b *aexp*) \Rightarrow 'a *aexp* \Rightarrow 'b *aexp*

```

and substb :: ('a ⇒ 'b aexp) ⇒ 'a bexp ⇒ 'b bexp
where
  subst f (IF b a1 a2) = IF (subst f b) (subst f a1) (subst f a2)
| subst f (Sum a1 a2) = Sum (subst f a1) (subst f a2)
| subst f (Diff a1 a2) = Diff (subst f a1) (subst f a2)
| subst f (Var v) = f v
| subst f (Num n) = Num n

| substb f (Less a1 a2) = Less (subst f a1) (subst f a2)
| substb f (And b1 b2) = And (substb f b1) (substb f b2)
| substb f (Neg b) = Neg (substb f b)

lemma subst1-aexp:
  evala env (subst (Var (v := a')) a) = evala (env (v := evala env a')) a
and subst1-bexp:
  evalb env (substb (Var (v := a')) b) = evalb (env (v := evala env a')) b
  — one variable
by (induct a and b) simp-all

lemma subst-all-aexp:
  evala env (subst s a) = evala (λx. evala env (s x)) a
and subst-all-bexp:
  evalb env (substb s b) = evalb (λx. evala env (s x)) b
by (induct a and b) auto

end

```

8 Infinitely branching trees

```

theory Infinitely-Branching-Tree
imports Main
begin

```

```

datatype 'a tree =
  Atom 'a
| Branch nat ⇒ 'a tree

```

```

primrec map-tree :: ('a ⇒ 'b) ⇒ 'a tree ⇒ 'b tree
where
  map-tree f (Atom a) = Atom (f a)
| map-tree f (Branch ts) = Branch (λx. map-tree f (ts x))

```

```

lemma tree-map-compose: map-tree g (map-tree f t) = map-tree (g ∘ f) t
by (induct t) simp-all

```

```

primrec exists-tree :: ('a ⇒ bool) ⇒ 'a tree ⇒ bool
where
  exists-tree P (Atom a) = P a
| exists-tree P (Branch ts) = (∃ x. exists-tree P (ts x))

```

lemma *exists-map*:
 $(\bigwedge x. P\ x \implies Q\ (f\ x)) \implies$
 $\text{exists-tree } P\ ts \implies \text{exists-tree } Q\ (\text{map-tree } f\ ts)$
by (*induct ts*) *auto*

8.1 The Brouwer ordinals, as in ZF/Induct/Brouwer.thy.

datatype *brouwer* = *Zero* | *Succ brouwer* | *Lim nat \Rightarrow brouwer*

Addition of ordinals

primrec *add* :: *brouwer \Rightarrow brouwer \Rightarrow brouwer*
where
 $\text{add } i\ \text{Zero} = i$
 $\mid \text{add } i\ (\text{Succ } j) = \text{Succ } (\text{add } i\ j)$
 $\mid \text{add } i\ (\text{Lim } f) = \text{Lim } (\lambda n. \text{add } i\ (f\ n))$

lemma *add-assoc*: $\text{add } (\text{add } i\ j)\ k = \text{add } i\ (\text{add } j\ k)$
by (*induct k*) *auto*

Multiplication of ordinals

primrec *mult* :: *brouwer \Rightarrow brouwer \Rightarrow brouwer*
where
 $\text{mult } i\ \text{Zero} = \text{Zero}$
 $\mid \text{mult } i\ (\text{Succ } j) = \text{add } (\text{mult } i\ j)\ i$
 $\mid \text{mult } i\ (\text{Lim } f) = \text{Lim } (\lambda n. \text{mult } i\ (f\ n))$

lemma *add-mult-distrib*: $\text{mult } i\ (\text{add } j\ k) = \text{add } (\text{mult } i\ j)\ (\text{mult } i\ k)$
by (*induct k*) (*auto simp add: add-assoc*)

lemma *mult-assoc*: $\text{mult } (\text{mult } i\ j)\ k = \text{mult } i\ (\text{mult } j\ k)$
by (*induct k*) (*auto simp add: add-mult-distrib*)

We could probably instantiate some axiomatic type classes and use the standard infix operators.

8.2 A WF Ordering for The Brouwer ordinals (Michael Comp-ton)

To use the function package we need an ordering on the Brouwer ordinals. Start with a predecessor relation and form its transitive closure.

definition *brouwer-pred* :: (*brouwer \times brouwer*) *set*
where $\text{brouwer-pred} = (\bigcup i. \{(m, n). n = \text{Succ } m \vee (\exists f. n = \text{Lim } f \wedge m = f\ i)\})$

definition *brouwer-order* :: (*brouwer \times brouwer*) *set*
where $\text{brouwer-order} = \text{brouwer-pred}^+$

```

lemma wf-brouwer-pred: wf brouwer-pred
  unfolding wf-def brouwer-pred-def
  apply clarify
  apply (induct-tac x)
  apply blast+
done

lemma wf-brouwer-order[simp]: wf brouwer-order
  unfolding brouwer-order-def
  by (rule wf-trancl[OF wf-brouwer-pred])

lemma [simp]: (j, Succ j) ∈ brouwer-order
  by (auto simp add: brouwer-order-def brouwer-pred-def)

lemma [simp]: (f n, Lim f) ∈ brouwer-order
  by (auto simp add: brouwer-order-def brouwer-pred-def)

Example of a general function

function add2 :: brouwer ⇒ brouwer ⇒ brouwer
  where
    add2 i Zero = i
  | add2 i (Succ j) = Succ (add2 i j)
  | add2 i (Lim f) = Lim (λn. add2 i (f n))
  by pat-completeness auto
termination
  by (relation inv-image brouwer-order snd) auto

lemma add2-assoc: add2 (add2 i j) k = add2 i (add2 j k)
  by (induct k) auto

end

```

9 Ordinals

```

theory Ordinals
imports Main
begin

```

Some basic definitions of ordinal numbers. Draws an Agda development (in Martin-Löf type theory) by Peter Hancock (see <http://www.dcs.ed.ac.uk/home/pgh/chat.html>).

```

datatype ordinal =
  Zero
  | Succ ordinal
  | Limit nat ⇒ ordinal

primrec pred :: ordinal ⇒ nat ⇒ ordinal option
where

```

```

  pred Zero n = None
| pred (Succ a) n = Some a
| pred (Limit f) n = Some (f n)

```

abbreviation *(input) iter* :: ('a \Rightarrow 'a) \Rightarrow nat \Rightarrow ('a \Rightarrow 'a)
where *iter f n* \equiv $f \hat{\sim} n$

definition *OpLim* :: (nat \Rightarrow (ordinal \Rightarrow ordinal)) \Rightarrow (ordinal \Rightarrow ordinal)
where *OpLim F a* = *Limit* ($\lambda n. F n a$)

definition *OpItw* :: (ordinal \Rightarrow ordinal) \Rightarrow (ordinal \Rightarrow ordinal) ($\langle \sqcup \rangle$)
where $\sqcup f$ = *OpLim* (*iter f*)

primrec *cantor* :: ordinal \Rightarrow ordinal \Rightarrow ordinal
where

```

  cantor a Zero = Succ a
| cantor a (Succ b) =  $\sqcup (\lambda x. cantor x b)$  a
| cantor a (Limit f) = Limit ( $\lambda n. cantor a (f n)$ )

```

primrec *Nabla* :: (ordinal \Rightarrow ordinal) \Rightarrow (ordinal \Rightarrow ordinal) ($\langle \nabla \rangle$)
where

```

   $\nabla f$  Zero = f Zero
|  $\nabla f$  (Succ a) = f (Succ ( $\nabla f a$ ))
|  $\nabla f$  (Limit h) = Limit ( $\lambda n. \nabla f (h n)$ )

```

definition *deriv* :: (ordinal \Rightarrow ordinal) \Rightarrow (ordinal \Rightarrow ordinal)
where *deriv f* = $\nabla(\sqcup f)$

primrec *veblen* :: ordinal \Rightarrow ordinal \Rightarrow ordinal
where

```

  veblen Zero =  $\nabla(OpLim (iter (cantor Zero)))$ 
| veblen (Succ a) =  $\nabla(OpLim (iter (veblen a)))$ 
| veblen (Limit f) =  $\nabla(OpLim (\lambda n. veblen (f n)))$ 

```

definition *veb a* = *veblen a Zero*

definition ε_0 = *veb Zero*

definition Γ_0 = *Limit* ($\lambda n. iter veb n Zero$)

end

10 Sigma algebras

theory *Sigma-Algebra*

imports *Main*

begin

This is just a tiny example demonstrating the use of inductive definitions in classical mathematics. We define the least σ -algebra over a given set of sets.

inductive-set *σ -algebra* :: 'a set set \Rightarrow 'a set set **for** *A* :: 'a set set

where

$basic: a \in \sigma\text{-algebra } A \text{ if } a \in A \text{ for } a$
 $| UNIV: UNIV \in \sigma\text{-algebra } A$
 $| complement: - a \in \sigma\text{-algebra } A \text{ if } a \in \sigma\text{-algebra } A \text{ for } a$
 $| Union: (\bigcup i. a \ i) \in \sigma\text{-algebra } A \text{ if } \bigwedge i::nat. a \ i \in \sigma\text{-algebra } A \text{ for } a$

The following basic facts are consequences of the closure properties of any σ -algebra, merely using the introduction rules, but no induction nor cases.

theorem *sigma-algebra-empty*: $\{\} \in \sigma\text{-algebra } A$

proof –

have $UNIV \in \sigma\text{-algebra } A$ **by** (rule $\sigma\text{-algebra.UNIV}$)
 then have $-UNIV \in \sigma\text{-algebra } A$ **by** (rule $\sigma\text{-algebra.complement}$)
 also have $-UNIV = \{\}$ **by** *simp*
 finally show *?thesis* .

qed

theorem *sigma-algebra-Inter*:

$(\bigwedge i::nat. a \ i \in \sigma\text{-algebra } A) \implies (\bigcap i. a \ i) \in \sigma\text{-algebra } A$

proof –

assume $\bigwedge i::nat. a \ i \in \sigma\text{-algebra } A$
 then have $\bigwedge i::nat. -(a \ i) \in \sigma\text{-algebra } A$ **by** (rule $\sigma\text{-algebra.complement}$)
 then have $(\bigcup i. -(a \ i)) \in \sigma\text{-algebra } A$ **by** (rule $\sigma\text{-algebra.Union}$)
 then have $-(\bigcup i. -(a \ i)) \in \sigma\text{-algebra } A$ **by** (rule $\sigma\text{-algebra.complement}$)
 also have $-(\bigcup i. -(a \ i)) = (\bigcap i. a \ i)$ **by** *simp*
 finally show *?thesis* .

qed

end

11 Combinatory Logic example: the Church-Rosser Theorem

theory *Comb*

imports *Main*

begin

Combinator terms do not have free variables. Example taken from [1].

11.1 Definitions

Datatype definition of combinators S and K .

datatype *comb* = K

$| S$
 $| Ap \ comb \ comb \ (\text{infixl } \langle \cdot \rangle \ 90)$

Inductive definition of contractions, \rightarrow^1 and (multi-step) reductions, \rightarrow .

inductive *contract1* :: $[comb, comb] \Rightarrow bool$ (**infixl** $\langle \rightarrow^1 \rangle \ 50$)

where

$K: K \cdot x \cdot y \rightarrow^1 x$
 $| S: S \cdot x \cdot y \cdot z \rightarrow^1 (x \cdot z) \cdot (y \cdot z)$
 $| Ap1: x \rightarrow^1 y \implies x \cdot z \rightarrow^1 y \cdot z$
 $| Ap2: x \rightarrow^1 y \implies z \cdot x \rightarrow^1 z \cdot y$

abbreviation

$contract :: [comb, comb] \Rightarrow bool \quad (\mathbf{infixl} \langle \rightarrow \rangle 50) \text{ where}$
 $contract \equiv contract1^{**}$

Inductive definition of parallel contractions, \Rightarrow^1 and (multi-step) parallel reductions, \Rightarrow .

inductive $parcontract1 :: [comb, comb] \Rightarrow bool \quad (\mathbf{infixl} \langle \Rightarrow^1 \rangle 50)$

where

$refl: x \Rightarrow^1 x$
 $| K: K \cdot x \cdot y \Rightarrow^1 x$
 $| S: S \cdot x \cdot y \cdot z \Rightarrow^1 (x \cdot z) \cdot (y \cdot z)$
 $| Ap: \llbracket x \Rightarrow^1 y; z \Rightarrow^1 w \rrbracket \implies x \cdot z \Rightarrow^1 y \cdot w$

abbreviation

$parcontract :: [comb, comb] \Rightarrow bool \quad (\mathbf{infixl} \langle \Rightarrow \rangle 50) \text{ where}$
 $parcontract \equiv parcontract1^{**}$

Misc definitions.

definition

$I :: comb \text{ where}$
 $I \equiv S \cdot K \cdot K$

definition

$diamond :: ([comb, comb] \Rightarrow bool) \Rightarrow bool \text{ where}$
 — confluence; Lambda/Commutation treats this more abstractly
 $diamond\ r \equiv \forall x\ y.\ r\ x\ y \longrightarrow$
 $\quad (\forall y'.\ r\ x\ y' \longrightarrow$
 $\quad \quad (\exists z.\ r\ y\ z \wedge r\ y'\ z))$

11.2 Reflexive/Transitive closure preserves Church-Rosser property

Remark: So does the Transitive closure, with a similar proof

Strip lemma. The induction hypothesis covers all but the last diamond of the strip.

lemma *strip-lemma* [rule-format]:

assumes $diamond\ r$ **and** $r: r^{**}\ x\ y\ r\ x\ y'$

shows $\exists z.\ r^{**}\ y'\ z \wedge r\ y\ z$

using r

proof (induction rule: *rtranclp-induct*)

case *base*


```

    then show ?case
      by blast
next
  case (step y z)
  then show ?case
    using <diamond r> unfolding diamond-def
    by (metis rtranclp.rtrancl-into-rtrancl)
qed

proposition diamond-rtrancl:
  assumes diamond r
  shows diamond(r**)
  unfolding diamond-def
proof (intro strip)
  fix x y y'
  assume r** x y r** x y'
  then show  $\exists z. r** y z \wedge r** y' z$ 
proof (induction rule: rtranclp-induct)
    case base
    then show ?case
      by blast
  next
    case (step y z)
    then show ?case
      by (meson assms strip-lemma rtranclp.rtrancl-into-rtrancl)
  qed
qed

```

11.3 Non-contraction results

Derive a case for each combinator constructor.

inductive-cases

```

K-contractE [elim!]:  $K \rightarrow^1 r$ 
and S-contractE [elim!]:  $S \rightarrow^1 r$ 
and Ap-contractE [elim!]:  $p \cdot q \rightarrow^1 r$ 

```

```

declare contract1.K [intro!] contract1.S [intro!]
declare contract1.Ap1 [intro] contract1.Ap2 [intro]

```

```

lemma I-contract-E [iff]:  $\neg I \rightarrow^1 z$ 
  unfolding I-def by blast

```

```

lemma K1-contractD [elim!]:  $K \cdot x \rightarrow^1 z \implies (\exists x'. z = K \cdot x' \wedge x \rightarrow^1 x')$ 
  by blast

```

```

lemma Ap-reduce1 [intro]:  $x \rightarrow y \implies x \cdot z \rightarrow y \cdot z$ 
  by (induction rule: rtranclp-induct; blast intro: rtranclp-trans)

```

```

lemma Ap-reduce2 [intro]:  $x \rightarrow y \implies z \cdot x \rightarrow z \cdot y$ 

```

by (*induction rule: rtrancp-induct; blast intro: rtrancp-trans*)

Counterexample to the diamond property for $x \rightarrow^1 y$

lemma *not-diamond-contract*: $\neg \text{diamond}(\text{contract1})$
unfolding *diamond-def* **by** (*metis S-contractE contract1.K*)

11.4 Results about Parallel Contraction

Derive a case for each combinator constructor.

inductive-cases

K-parcontractE [*elim!*]: $K \Rightarrow^1 r$
and *S-parcontractE* [*elim!*]: $S \Rightarrow^1 r$
and *Ap-parcontractE* [*elim!*]: $p \cdot q \Rightarrow^1 r$

declare *parcontract1.intros* [*intro*]

11.5 Basic properties of parallel contraction

The rules below are not essential but make proofs much faster

lemma *K1-parcontractD* [*dest!*]: $K \cdot x \Rightarrow^1 z \implies (\exists x'. z = K \cdot x' \wedge x \Rightarrow^1 x')$
by *blast*

lemma *S1-parcontractD* [*dest!*]: $S \cdot x \Rightarrow^1 z \implies (\exists x'. z = S \cdot x' \wedge x \Rightarrow^1 x')$
by *blast*

lemma *S2-parcontractD* [*dest!*]: $S \cdot x \cdot y \Rightarrow^1 z \implies (\exists x' y'. z = S \cdot x' \cdot y' \wedge x \Rightarrow^1 x' \wedge y \Rightarrow^1 y')$
by *blast*

Church-Rosser property for parallel contraction

proposition *diamond-parcontract*: *diamond parcontract1*

proof –

have $(\exists z. w \Rightarrow^1 z \wedge y' \Rightarrow^1 z)$ **if** $y \Rightarrow^1 w$ $y \Rightarrow^1 y'$ **for** $w y y'$
using that **by** (*induction arbitrary: y' rule: parcontract1.induct*) *fast+*
then show *?thesis*
by (*auto simp: diamond-def*)

qed

11.6 Equivalence of $p \rightarrow q$ and $p \Rightarrow q$.

lemma *contract-imp-parcontract*: $x \rightarrow^1 y \implies x \Rightarrow^1 y$
by (*induction rule: contract1.induct; blast*)

Reductions: simply throw together reflexivity, transitivity and the one-step reductions

proposition *reduce-I*: $I \cdot x \rightarrow x$
unfolding *I-def*

```

by (meson contract1.K contract1.S r-into-rtrancpl rtrancpl.rtrancpl-into-rtrancpl)

lemma parcontract-imp-reduce:  $x \Rightarrow^1 y \implies x \rightarrow y$ 
proof (induction rule: parcontract1.induct)
  case (Ap x y z w)
  then show ?case
    by (meson Ap-reduce1 Ap-reduce2 rtrancpl-trans)
qed auto

lemma reduce-eq-parreduce:  $x \rightarrow y \longleftrightarrow x \Rightarrow y$ 
by (metis contract-imp-parcontract parcontract-imp-reduce predicate2I rtrancpl-subset)

theorem diamond-reduce: diamond(contract)
  using diamond-parcontract diamond-rtrancpl reduce-eq-parreduce by presburger

end

```

12 Meta-theory of propositional logic

theory *PropLog* **imports** *Main* **begin**

Datatype definition of propositional logic formulae and inductive definition of the propositional tautologies.

Inductive definition of propositional logic. Soundness and completeness w.r.t. truth-tables.

Prove: If $H \models p$ then $G \models p$ where $G \in \text{Fin}(H)$

12.1 The datatype of propositions

```

datatype 'a pl =
  false
| var 'a (‹#› [1000])
| imp 'a pl 'a pl (infixr ‹→› 90)

```

12.2 The proof system

```

inductive thms :: ['a pl set, 'a pl] ⇒ bool (infixl ‹⊢› 50)
  for H :: 'a pl set
  where
    H:  $p \in H \implies H \vdash p$ 
  | K:  $H \vdash p \rightarrow q \rightarrow p$ 
  | S:  $H \vdash (p \rightarrow q \rightarrow r) \rightarrow (p \rightarrow q) \rightarrow p \rightarrow r$ 
  | DN:  $H \vdash ((p \rightarrow \text{false}) \rightarrow \text{false}) \rightarrow p$ 
  | MP:  $\llbracket H \vdash p \rightarrow q; H \vdash p \rrbracket \implies H \vdash q$ 

```

12.3 The semantics

12.3.1 Semantics of propositional logic.

```

primrec eval :: ['a set, 'a pl] => bool  (‹[-]› [100,0] 100)
  where
    tt[false] = False
  | tt[#v] = (v ∈ tt)
  | eval-imp: tt[p→q] = (tt[p] → tt[q])

```

A finite set of hypotheses from t and the *Vars* in p .

```

primrec hys :: ['a pl, 'a set] => 'a pl set
  where
    hys false tt = {}
  | hys (#v) tt = {if v ∈ tt then #v else #v→false}
  | hys (p→q) tt = hys p tt Un hys q tt

```

12.3.2 Logical consequence

For every valuation, if all elements of H are true then so is p .

```

definition sat :: ['a pl set, 'a pl] => bool  (infixl ‹|=› 50)
  where H |= p = (∀ tt. (∀ q∈H. tt[q]) → tt[p])

```

12.4 Proof theory of propositional logic

lemma thms-mono:

assumes $G \subseteq H$ shows $\text{thms}(G) \leq \text{thms}(H)$

proof –

have $G \vdash p \implies H \vdash p$ for p

by (induction rule: thms.induct) (use assms in ‹auto intro: thms.intros›)

then show ?thesis

by blast

qed

lemma thms-I: $H \vdash p \rightarrow p$

— Called I for Identity Combinator, not for Introduction.

by (best intro: thms.K thms.S thms.MP)

12.4.1 Weakening, left and right

lemma weaken-left: $\llbracket G \subseteq H; G \vdash p \rrbracket \implies H \vdash p$

— Order of premises is convenient with *THEN*

by (meson predicate1D thms-mono)

lemma weaken-left-insert: $G \vdash p \implies \text{insert } a \ G \vdash p$

by (meson subset-insertI weaken-left)

lemma weaken-left-Un1: $G \vdash p \implies G \cup B \vdash p$

by (rule weaken-left) (rule Un-upper1)

lemma *weaken-left-Un2*: $G \vdash p \implies A \cup G \vdash p$
by (*metis Un-commute weaken-left-Un1*)

lemma *weaken-right*: $H \vdash q \implies H \vdash p \multimap q$
using *K MP* **by** *blast*

12.4.2 The deduction theorem

theorem *deduction*: $\text{insert } p \ H \vdash q \implies H \vdash p \multimap q$
proof (*induct set: thms*)
case ($H \ p$)
then show *?case*
using *thms.H thms-I weaken-right* **by** *fastforce*
qed (*metis thms.simps*)**+**

12.4.3 The cut rule

lemma *cut*: $\text{insert } p \ H \vdash q \implies H \vdash p \implies H \vdash q$
using *MP deduction* **by** *blast*

lemma *thms-falseE*: $H \vdash \text{false} \implies H \vdash q$
by (*metis thms.simps*)

lemma *thms-notE*: $H \vdash p \multimap \text{false} \implies H \vdash p \implies H \vdash q$
using *MP thms-falseE* **by** *blast*

12.4.4 Soundness of the rules wrt truth-table semantics

theorem *soundness*: $H \vdash p \implies H \models p$
by (*induct set: thms*) (*auto simp: sat-def*)

12.5 Completeness

12.5.1 Towards the completeness proof

lemma *false-imp*: $H \vdash p \multimap \text{false} \implies H \vdash p \multimap q$
by (*metis thms.simps*)

lemma *imp-false*:
 $\llbracket H \vdash p; \ H \vdash q \multimap \text{false} \rrbracket \implies H \vdash (p \multimap q) \multimap \text{false}$
by (*meson MP S weaken-right*)

lemma *hyps-thms-if*: $\text{hyps } p \ tt \vdash (\text{if } tt[[p]] \text{ then } p \text{ else } p \multimap \text{false})$
— Typical example of strengthening the induction statement.

proof (*induction p*)
case (*imp p1 p2*)
then show *?case*
by (*metis (full-types) eval-imp false-imp hyps.simps(3) imp-false weaken-left-Un1 weaken-left-Un2 weaken-right*)

qed (*simp-all add: thms-I thms.H*)

lemma *sat-thms-p*: $\{\} \models p \implies \text{hyps } p \text{ } tt \vdash p$

— Key lemma for completeness; yields a set of assumptions satisfying p

by (*metis (full-types) empty-iff hyps-thms-if sat-def*)

For proving certain theorems in our new propositional logic.

declare *deduction* [*intro!*]

declare *thms.H* [*THEN thms.MP, intro*]

The excluded middle in the form of an elimination rule.

lemma *thms-excluded-middle*: $H \vdash (p \rightarrow q) \rightarrow ((p \rightarrow \text{false}) \rightarrow q) \rightarrow q$

proof —

have *insert* $((p \rightarrow \text{false}) \rightarrow q)$ (*insert* $(p \rightarrow q)$ *H*) $\vdash (q \rightarrow \text{false}) \rightarrow \text{false}$

by (*best intro: H*)

then show *?thesis*

by (*metis deduction thms.simps*)

qed

lemma *thms-excluded-middle-rule*:

$\llbracket \text{insert } p \text{ } H \vdash q; \text{ insert } (p \rightarrow \text{false}) \text{ } H \vdash q \rrbracket \implies H \vdash q$

— Hard to prove directly because it requires cuts

by (*rule thms-excluded-middle [THEN thms.MP, THEN thms.MP], auto*)

12.6 Completeness – lemmas for reducing the set of assumptions

For the case $\text{hyps } p \text{ } t - \text{insert } \#v \text{ } Y \vdash p$ we also have $\text{hyps } p \text{ } t - \{\#v\} \subseteq \text{hyps } p \text{ } (t - \{v\})$.

lemma *hyps-Diff*: $\text{hyps } p \text{ } (t - \{v\}) \subseteq \text{insert } (\#v \rightarrow \text{false}) ((\text{hyps } p \text{ } t) - \{\#v\})$

by (*induct p*) *auto*

For the case $\text{hyps } p \text{ } t - \text{insert } (\#v \rightarrow \text{Fls}) \text{ } Y \vdash p$ we also have $\text{hyps } p \text{ } t - \{\#v \rightarrow \text{Fls}\} \subseteq \text{hyps } p \text{ } (\text{insert } v \text{ } t)$.

lemma *hyps-insert*: $\text{hyps } p \text{ } (\text{insert } v \text{ } t) \subseteq \text{insert } (\#v) (\text{hyps } p \text{ } t - \{\#v \rightarrow \text{false}\})$

by (*induct p*) *auto*

Two lemmas for use with *weaken-left*

lemma *insert-Diff-same*: $B - C \subseteq \text{insert } a (B - \text{insert } a \text{ } C)$

by *fast*

lemma *insert-Diff-subset2*: $\text{insert } a (B - \{c\}) - D \subseteq \text{insert } a (B - \text{insert } c \text{ } D)$

by *fast*

The set $\text{hyps } p \text{ } t$ is finite, and elements have the form $\#v$ or $\#v \rightarrow \text{Fls}$.

lemma *hyps-finite*: *finite*($\text{hyps } p \text{ } t$)

by (*induct p*) *auto*

lemma *hyps-subset*: $\text{hyps } p \ t \subseteq (\text{UN } v. \{\#v, \#v \rightarrow \text{false}\})$
by (*induct p*) *auto*

lemma *Diff-weaken-left*: $A \subseteq C \implies A - B \vdash p \implies C - B \vdash p$
by (*rule Diff-mono [OF - subset-refl, THEN weaken-left]*)

12.6.1 Completeness theorem

Induction on the finite set of assumptions *hyps p t0*. We may repeatedly subtract assumptions until none are left!

lemma *completeness-0*:
assumes $\{\} \models p$
shows $\{\} \vdash p$
proof –
 { **fix** *t t0*
 have $\text{hyps } p \ t - \text{hyps } p \ t0 \vdash p$
 using *hyps-finite hyps-subset*
 proof (*induction arbitrary: t rule: finite-subset-induct*)
 case *empty*
 then show *?case*
 by (*simp add: assms sat-thms-p*)
 next
 case (*insert q H*)
 then consider *v where q = #v | v where q = #v → false*
 by *blast*
 then show *?case*
 proof *cases*
 case *1*
 then show *?thesis*
 by (*metis (no-types, lifting) insert.IH thms-excluded-middle-rule insert-Diff-same*
 insert-Diff-subset2 weaken-left Diff-weaken-left hyps-Diff)
 next
 case *2*
 then show *?thesis*
 by (*metis (no-types, lifting) insert.IH thms-excluded-middle-rule insert-Diff-same*
 insert-Diff-subset2 weaken-left Diff-weaken-left hyps-insert)
 qed
 qed
 }
 then show *?thesis*
 by (*metis Diff-cancel*)
qed

A semantic analogue of the Deduction Theorem

lemma *sat-imp*: $\text{insert } p \ H \models q \implies H \models p \rightarrow q$

```

    by (auto simp: sat-def)

theorem completeness: finite  $H \implies H \models p \implies H \vdash p$ 
proof (induction arbitrary:  $p$  rule: finite-induct)
  case empty
  then show ?case
    by (simp add: completeness-0)
next
  case insert
  then show ?case
    by (meson  $H$  MP insertI1 sat-imp weaken-left-insert)
qed

theorem syntax-iff-semantics: finite  $H \implies (H \vdash p) = (H \models p)$ 
  by (blast intro: soundness completeness)

end

```

13 Mutual Induction via Iterated Inductive Definitions

```

theory Com imports Main begin

typedecl loc
type-synonym state = loc => nat

datatype
  exp = N nat
    | X loc
    | Op nat => nat => nat exp exp
    | valOf com exp      (‹VALOF - RESULTIS -› 60)
and
  com = SKIP
    | Assign loc exp      (infixl ‹:=› 60)
    | Semi com com        (‹-;› [60, 60] 60)
    | Cond exp com com    (‹IF - THEN - ELSE -› 60)
    | While exp com       (‹WHILE - DO -› 60)

```

13.1 Commands

Execution of commands

```

abbreviation (input)
  generic-rel (‹-/ -|[-]> -› [50,0,50] 50) where
    esig -|[[eval]]> ns == (esig,ns) ∈ eval

```

Command execution. Natural numbers represent Booleans: 0=True, 1=False

inductive-set


```

exec :: ((exp*state) * (nat*state)) set => ((com*state)*state)set
and exec-rel :: com * state => ((exp*state) * (nat*state)) set => state => bool
  (⟨-/ -[-]→ -⟩ [50,0,50] 50)
for eval :: ((exp*state) * (nat*state)) set
where
  csig -[eval]→ s == (csig,s) ∈ exec eval

| Skip:   (SKIP,s) -[eval]→ s

| Assign: (e,s) -[eval]→ (v,s') ==> (x := e, s) -[eval]→ s'(x:=v)

| Semi:   [| (c0,s) -[eval]→ s2; (c1,s2) -[eval]→ s1 |]
          ==> (c0 ;; c1, s) -[eval]→ s1

| IfTrue: [| (e,s) -[eval]→ (0,s'); (c0,s') -[eval]→ s1 |]
          ==> (IF e THEN c0 ELSE c1, s) -[eval]→ s1

| IfFalse: [| (e,s) -[eval]→ (Suc 0, s'); (c1,s') -[eval]→ s1 |]
           ==> (IF e THEN c0 ELSE c1, s) -[eval]→ s1

| WhileFalse: (e,s) -[eval]→ (Suc 0, s1)
              ==> (WHILE e DO c, s) -[eval]→ s1

| WhileTrue: [| (e,s) -[eval]→ (0,s1);
                (c,s1) -[eval]→ s2; (WHILE e DO c, s2) -[eval]→ s3 |]
              ==> (WHILE e DO c, s) -[eval]→ s3

```

declare *exec.intros* [intro]

inductive-cases

```

[elim!]: (SKIP,s) -[eval]→ t
and [elim!]: (x:=a,s) -[eval]→ t
and [elim!]: (c1;;c2, s) -[eval]→ t
and [elim!]: (IF e THEN c1 ELSE c2, s) -[eval]→ t
and exec-WHILE-case: (WHILE b DO c,s) -[eval]→ t

```

Justifies using "exec" in the inductive definition of "eval"

lemma *exec-mono*: $A \leq B \implies \text{exec}(A) \leq \text{exec}(B)$

apply (*rule subsetI*)

apply (*simp add: split-paired-all*)

apply (*erule exec.induct*)

apply *blast+*

done

lemma [*pred-set-conv*]:

$((\lambda x x' y y'. ((x, x'), (y, y')) \in R) \leq (\lambda x x' y y'. ((x, x'), (y, y')) \in S)) = (R \leq S)$

unfolding *subset-eq*

by (*auto simp add: le-fun-def*)

lemma [*pred-set-conv*]:

$((\lambda x x'. y. ((x, x'), y) \in R) \leq (\lambda x x'. y. ((x, x'), y) \in S)) = (R \leq S)$

unfolding *subset-eq*

by (*auto simp add: le-fun-def*)

Command execution is functional (deterministic) provided evaluation is

theorem *single-valued-exec: single-valued ev ==> single-valued(exec ev)*

apply (*simp add: single-valued-def*)

apply (*intro allI*)

apply (*rule impI*)

apply (*erule exec.induct*)

apply (*blast elim: exec-WHILE-case*)

done

13.2 Expressions

Evaluation of arithmetic expressions

inductive-set

eval :: $((exp*state) * (nat*state)) \text{ set}$

and *eval-rel* :: $[exp*state, nat*state] \Rightarrow \text{bool}$ (**infixl** $\langle -|-> \rangle$ 50)

where

$esig -|-> ns == (esig, ns) \in eval$

| *N* [*intro!*]: $(N(n), s) -|-> (n, s)$

| *X* [*intro!*]: $(X(x), s) -|-> (s(x), s)$

| *Op* [*intro*]: $[(e0, s) -|-> (n0, s0); (e1, s0) -|-> (n1, s1)] \Rightarrow (Op\ f\ e0\ e1, s) -|-> (f\ n0\ n1, s1)$

| *valOf* [*intro*]: $[(c, s) -[eval]-> s0; (e, s0) -|-> (n, s1)] \Rightarrow (VALOF\ c\ RESULTIS\ e, s) -|-> (n, s1)$

monos *exec-mono*

inductive-cases

[*elim!*]: $(N(n), sigma) -|-> (n', s')$

and [*elim!*]: $(X(x), sigma) -|-> (n, s')$

and [*elim!*]: $(Op\ f\ a1\ a2, sigma) -|-> (n, s')$

and [*elim!*]: $(VALOF\ c\ RESULTIS\ e, s) -|-> (n, s1)$

lemma *var-assign-eval* [*intro!*]: $(X\ x, s(x:=n)) -|-> (n, s(x:=n))$

by (*rule fun-upd-same [THEN subst]*) *fast*

Make the induction rule look nicer – though *eta-contract* makes the new

version look worse than it is...

lemma *split-lemma*: $\{((e,s),(n,s')). P\ e\ s\ n\ s'\} = \text{Collect} (\text{case-prod } (\%v. \text{case-prod } (\text{case-prod } P\ v)))$
by *auto*

New induction rule. Note the form of the VALOF induction hypothesis

lemma *eval-induct*
 $[\text{case-names } N\ X\ Op\ valOf, \text{ consumes } 1, \text{ induct set: eval}]:$
 $[[(e,s) \dashv\!\!\rightarrow (n,s');$
 $!!n\ s. P\ (N\ n)\ s\ n\ s;$
 $!!s\ x. P\ (X\ x)\ s\ (s\ x)\ s;$
 $!!e0\ e1\ f\ n0\ n1\ s\ s0\ s1.$
 $[[(e0,s) \dashv\!\!\rightarrow (n0,s0); P\ e0\ s\ n0\ s0;$
 $(e1,s0) \dashv\!\!\rightarrow (n1,s1); P\ e1\ s0\ n1\ s1$
 $] ==> P\ (Op\ f\ e0\ e1)\ s\ (f\ n0\ n1)\ s1;$
 $!!c\ e\ n\ s\ s0\ s1.$
 $[[(c,s) \dashv\!\!\rightarrow [eval\ Int\ \{((e,s),(n,s')). P\ e\ s\ n\ s'\}] \dashv\!\!\rightarrow s0;$
 $(c,s) \dashv\!\!\rightarrow [eval] \dashv\!\!\rightarrow s0;$
 $(e,s0) \dashv\!\!\rightarrow (n,s1); P\ e\ s0\ n\ s1\]]$
 $==> P\ (VALOF\ c\ RESULTIS\ e)\ s\ n\ s1$
 $] ==> P\ e\ s\ n\ s'$
apply (*induct set: eval*)
apply *blast*
apply *blast*
apply *blast*
apply (*frule Int-lower1 [THEN exec-mono, THEN subsetD]*)
apply (*auto simp add: split-lemma*)
done

Lemma for *Function-eval*. The major premise is that (c,s) executes to $s1$ using eval restricted to its functional part. Note that the execution $(c,s) \dashv\!\!\rightarrow [eval] \dashv\!\!\rightarrow s2$ can use unrestricted *eval*! The reason is that the execution $(c,s) \dashv\!\!\rightarrow [eval\ Int\ \{...\}] \dashv\!\!\rightarrow s1$ assures us that execution is functional on the argument (c,s) .

lemma *com-Unique*:
 $(c,s) \dashv\!\!\rightarrow [eval\ Int\ \{((e,s),(n,t)). \forall nt'. (e,s) \dashv\!\!\rightarrow nt' \dashv\!\!\rightarrow (n,t)=nt'\}] \dashv\!\!\rightarrow s1$
 $==> \forall s2. (c,s) \dashv\!\!\rightarrow [eval] \dashv\!\!\rightarrow s2 \dashv\!\!\rightarrow s2=s1$
apply (*induct set: exec*)
apply *simp-all*
apply *blast*
apply *force*
apply *blast*
apply *blast*
apply *blast*
apply (*blast elim: exec-WHILE-case*)
apply (*erule-tac V = (c,s2) \dashv\!\!\rightarrow [ev] \dashv\!\!\rightarrow s3 for c ev in thin-rl*)
apply *clarify*
apply (*erule exec-WHILE-case, blast+*)

done

Expression evaluation is functional, or deterministic

theorem *single-valued-eval: single-valued eval*
apply (*unfold single-valued-def*)
apply (*intro allI, rule impI*)
apply (*simp (no-asm-simp) only: split-tupled-all*)
apply (*erule eval-induct*)
apply (*drule-tac [4] com-Unique*)
apply (*simp-all (no-asm-use)*)
apply *blast+*
done

lemma *eval-N-E [dest!]: (N n, s) -|-> (v, s') ==> (v = n & s' = s)*
by (*induct e == N n s v s' set: eval*) *simp-all*

This theorem says that "WHILE TRUE DO c" cannot terminate

lemma *while-true-E:*
 $(c', s) -[eval]-> t ==> c' = \text{WHILE } (N \ 0) \text{ DO } c ==> \text{False}$
by (*induct set: exec*) *auto*

13.3 Equivalence of IF e THEN c;;(WHILE e DO c) ELSE SKIP and WHILE e DO c

lemma *while-if1:*
 $(c', s) -[eval]-> t$
 $==> c' = \text{WHILE } e \text{ DO } c ==>$
 $(\text{IF } e \text{ THEN } c;;c' \text{ ELSE SKIP}, s) -[eval]-> t$
by (*induct set: exec*) *auto*

lemma *while-if2:*
 $(c', s) -[eval]-> t$
 $==> c' = \text{IF } e \text{ THEN } c;;(\text{WHILE } e \text{ DO } c) \text{ ELSE SKIP} ==>$
 $(\text{WHILE } e \text{ DO } c, s) -[eval]-> t$
by (*induct set: exec*) *auto*

theorem *while-if:*
 $((\text{IF } e \text{ THEN } c;;(\text{WHILE } e \text{ DO } c) \text{ ELSE SKIP}, s) -[eval]-> t) =$
 $((\text{WHILE } e \text{ DO } c, s) -[eval]-> t)$
by (*blast intro: while-if1 while-if2*)

13.4 Equivalence of (IF e THEN c1 ELSE c2);;c and IF e THEN (c1;;c) ELSE (c2;;c)

lemma *if-semi1:*
 $(c', s) -[eval]-> t$
 $==> c' = (\text{IF } e \text{ THEN } c1 \text{ ELSE } c2);;c ==>$
 $(\text{IF } e \text{ THEN } (c1;;c) \text{ ELSE } (c2;;c), s) -[eval]-> t$

by (*induct set: exec*) *auto*

lemma *if-semi2*:

$(c', s) -[eval] \rightarrow t$
 $\implies c' = IF\ e\ THEN\ (c1;;c)\ ELSE\ (c2;;c) \implies$
 $((IF\ e\ THEN\ c1\ ELSE\ c2) ;; c, s) -[eval] \rightarrow t$
by (*induct set: exec*) *auto*

theorem *if-semi*: $((IF\ e\ THEN\ c1\ ELSE\ c2) ;; c, s) -[eval] \rightarrow t =$
 $((IF\ e\ THEN\ (c1;;c)\ ELSE\ (c2;;c), s) -[eval] \rightarrow t)$
by (*blast intro: if-semi1 if-semi2*)

13.5 Equivalence of VALOF c1 RESULTIS (VALOF c2 RESULTIS e) and VALOF c1;;c2 RESULTIS e

lemma *valof-valof1*:

$(e', s) -|-> (v, s')$
 $\implies e' = VALOF\ c1\ RESULTIS\ (VALOF\ c2\ RESULTIS\ e) \implies$
 $(VALOF\ c1;;c2\ RESULTIS\ e, s) -|-> (v, s')$
by (*induct set: eval*) *auto*

lemma *valof-valof2*:

$(e', s) -|-> (v, s')$
 $\implies e' = VALOF\ c1;;c2\ RESULTIS\ e \implies$
 $(VALOF\ c1\ RESULTIS\ (VALOF\ c2\ RESULTIS\ e), s) -|-> (v, s')$
by (*induct set: eval*) *auto*

theorem *valof-valof*:

$((VALOF\ c1\ RESULTIS\ (VALOF\ c2\ RESULTIS\ e), s) -|-> (v, s')) =$
 $((VALOF\ c1;;c2\ RESULTIS\ e, s) -|-> (v, s'))$
by (*blast intro: valof-valof1 valof-valof2*)

13.6 Equivalence of VALOF SKIP RESULTIS e and e

lemma *valof-skip1*:

$(e', s) -|-> (v, s')$
 $\implies e' = VALOF\ SKIP\ RESULTIS\ e \implies$
 $(e, s) -|-> (v, s')$
by (*induct set: eval*) *auto*

lemma *valof-skip2*:

$(e, s) -|-> (v, s') \implies (VALOF\ SKIP\ RESULTIS\ e, s) -|-> (v, s')$
by *blast*

theorem *valof-skip*:

$((VALOF\ SKIP\ RESULTIS\ e, s) -|-> (v, s')) = ((e, s) -|-> (v, s'))$
by (*blast intro: valof-skip1 valof-skip2*)

13.7 Equivalence of VALOF $x:=e$ RESULTIS x and e

lemma *valof-assign1*:

$(e', s) \dashv\vdash (v, s'')$

$\implies e' = \text{VALOF } x:=e \text{ RESULTIS } X x \implies$

$(\exists s'. (e, s) \dashv\vdash (v, s') \ \& \ (s'' = s'(x:=v)))$

by (*induct set: eval*) (*simp-all del: fun-upd-apply, clarify, auto*)

lemma *valof-assign2*:

$(e, s) \dashv\vdash (v, s') \implies (\text{VALOF } x:=e \text{ RESULTIS } X x, s) \dashv\vdash (v, s'(x:=v))$

by *blast*

end

References

- [1] J. Camilleri and T. F. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, Computer Laboratory, University of Cambridge, Aug. 1992.