Seminar & Reading Club:

Automated Deductive Reasoning with ISABELLE

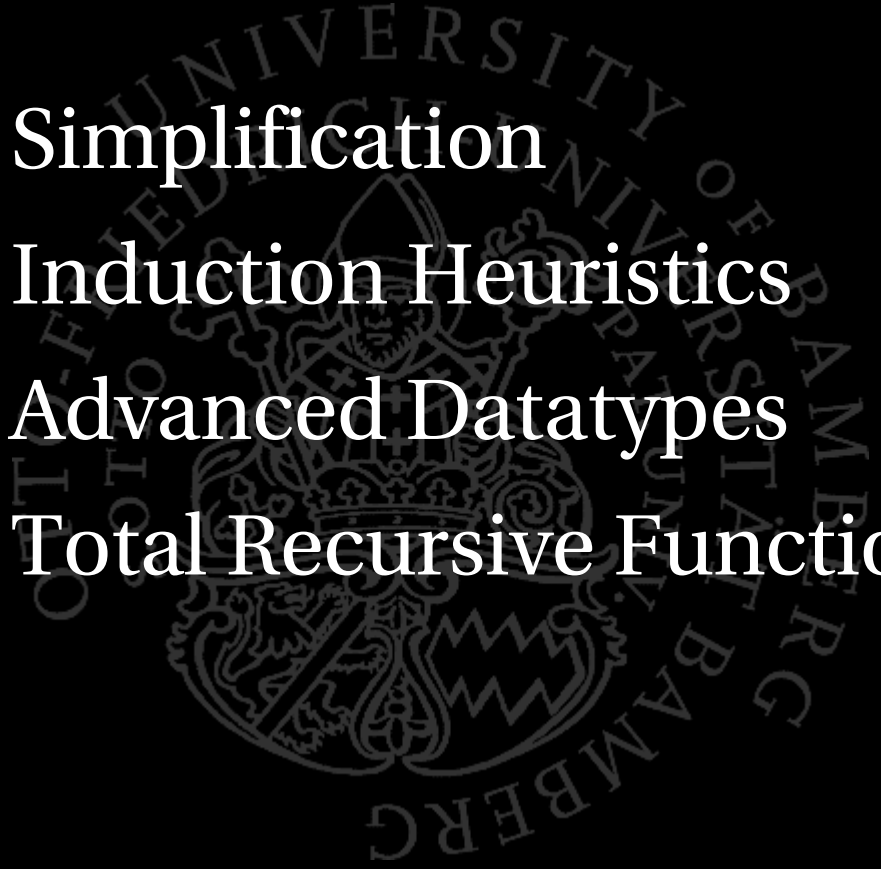
May 30th 2007

Chapter 3: More Functional Programming

Isabelle / HOL
A Proof Assistant for Higher-Order Logic
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel
LNCS 2283

Dominik.Mayer@Stud.Uni-Bamberg.de

3. More Functional Programming



3. More Functional Programming

Simplification

… what was that again?

3. More Functional Programming

3.1. Simplification

If we take the rules for “@” and apply them to the term

```
[0,1] @ []
```

the results of simplification would lead to the following

```
(0 # 1 # []) @ []
      ...
 0 # 1 # []
```

in Isabelle the common usage is just

```
apply(simp)
```

Simplification
=
Term Rewriting

... but it's not always going to get simpler!

In fact, by applying rewrite rules a small term
can expand to a bigger problem than it was.

Using the attribute `[simp]` to declare theorems to be simplification rules:

```
theorem theorem-name [simp]: "equation"
...
declare theorem-name [simp del]
...
declare theorem-name [simp]
```

Attention: Handle with care!

Simplification rules can lead to non-termination!

The `simp` Method

```
simp list of modifiers
simp_all list of modifiers
```

simp attacks only the first subgoal.
So make sure to use *simp_all* to get them all.

Modifiers

```
add:    list of theorem names
del:    list of theorem names
only:   list of theorem names
```

There is no need for a global simplification rule.

Apply the theorems needed for simplification “just in time”!

Assumptions

```
lemma example:
"[| xs @ zs = ys @ xs; [] @ xs = [] @ [] |]
  ==> ys = zs"
apply(simp)
done
```

By simplification these assumptions get reduced just the way we want them to. But sometimes using assumptions for simplification can lead to nontermination.

Modifying `simp` even more...

`(no_asm)`

Assumptions are completely ignored.

`(no_asm_simp)`

Assumptions are not simplified, but used for the simplification of the conclusion.

`(no_asm_use)`

Assumptions are simplified, but not used for the simplification of each other or the conclusion.

Definitions

```
constdefs
  xor :: "bool => bool => bool"
  "xor A B == (A & ~B) | (~A & B)"
```

Constant definitions can be used as simplification rules. But by default they are not expanded automatically to make proofs more robust. If a definition has to be changed only the proofs of the abstract properties will be affected.

`let`-Expressions

```
lemma "(let xs = [] in xs@ys@xs) = ys"
apply(simp add: Let_def)
done
```

Most of the times a lemma or theorem containing a `let`-expression has to be expanded by rewriting with `Let_def`.

`If`-Expressions

```
lemma "ALL xs. if xs = []
               then rev xs = []
               else rev xs ~= []"
apply(split split_if)
apply(auto)
done
```

Since splitting the `if`-expression is usually the right proof strategy, the simplifier does it automatically.

Case-Expressions

```
lemma
  "(case xs of []   => zs
             | y#ys => y#(ys@zs)) = xs@zs"
apply(simp split: list.split)
done
```

The simplifier does not split case-expressions, but as every datatype *t* comes with a theorem *t*.split, the modifier `split` can be applied to `simp` as shown.

Splitting Assumptions

```
lemma
  "if xs = []
   then ys ~= []
   else ys = []
  ==> xs @ ys ~= []"
apply(split split_if_asm)
apply(simp_all)
done
```

To split assumptions `split_if_asm` (or for case-expressions `t.split_asm`) has to be applied.

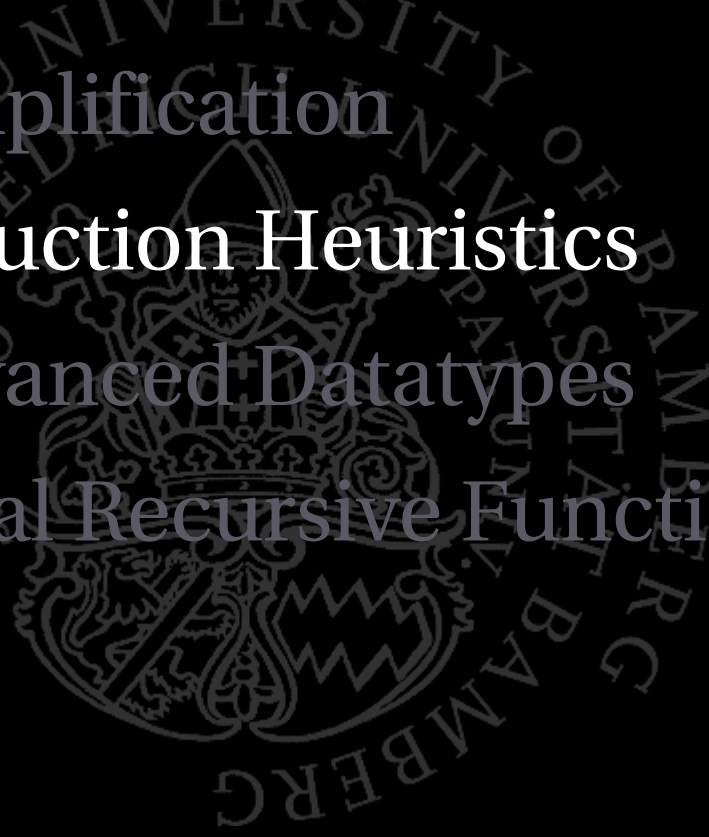
Tracing

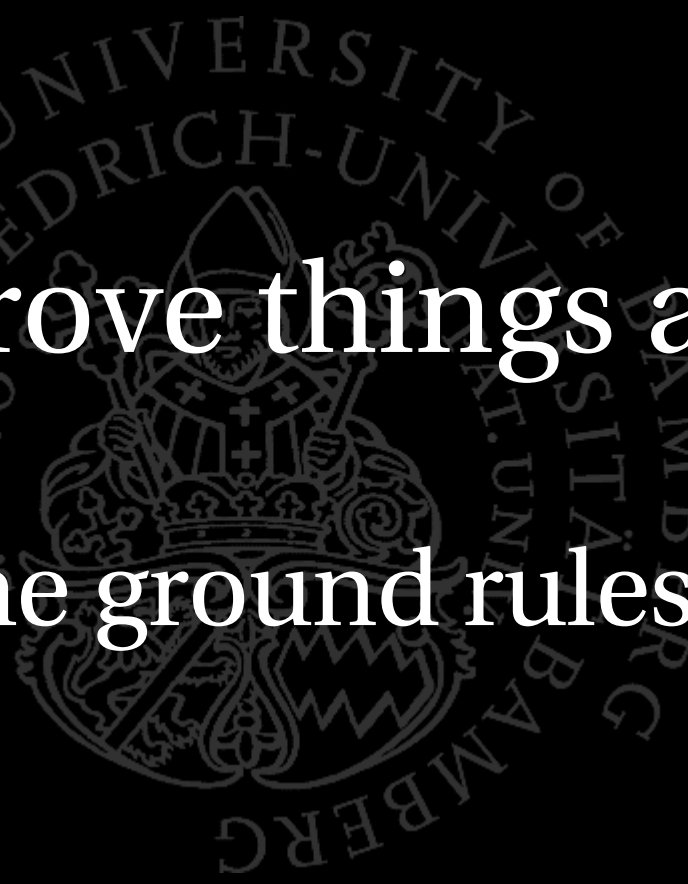
```
ML   "set trace_simp"
ML "reset trace_simp"
```

To better understand whats going on,
the simplification can be traced.

The trace lists every rule applied!

3. More Functional Programming



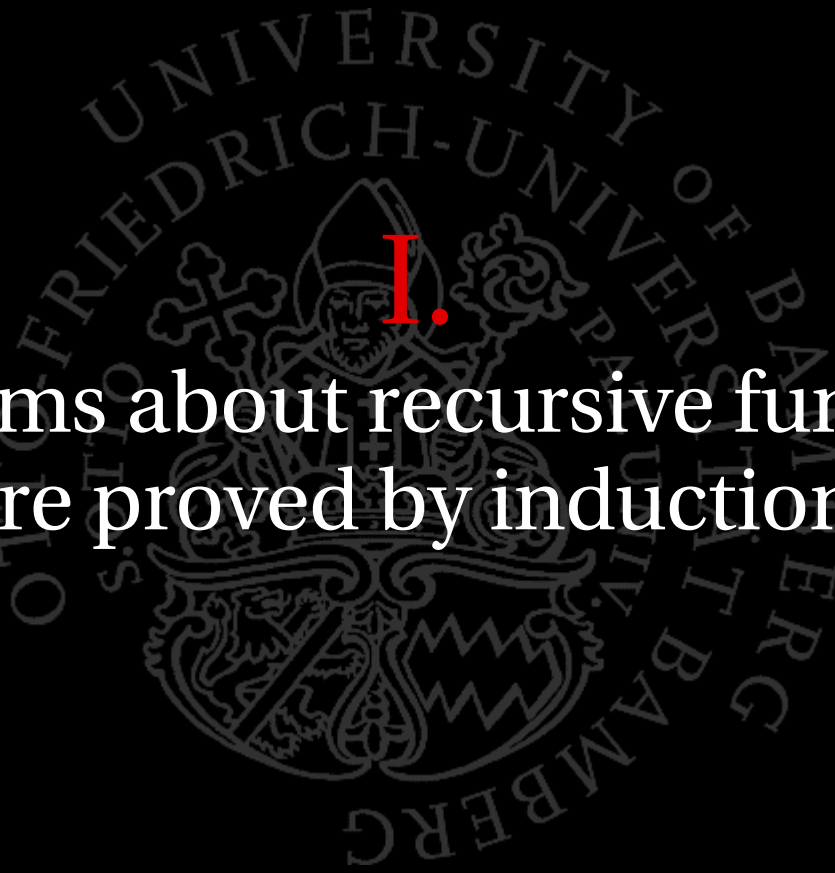


How to prove things anyway?

Some ground rules...

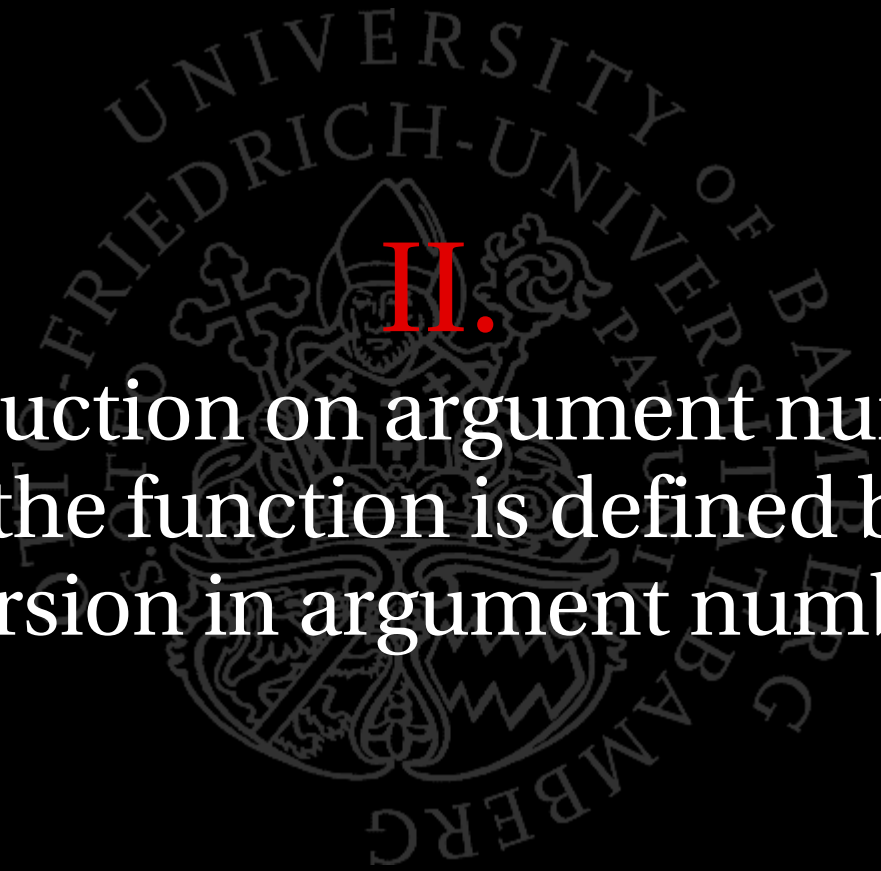
3. More Functional Programming

3.2. Induction Heuristics



I.

Theorems about recursive functions
are proved by induction.

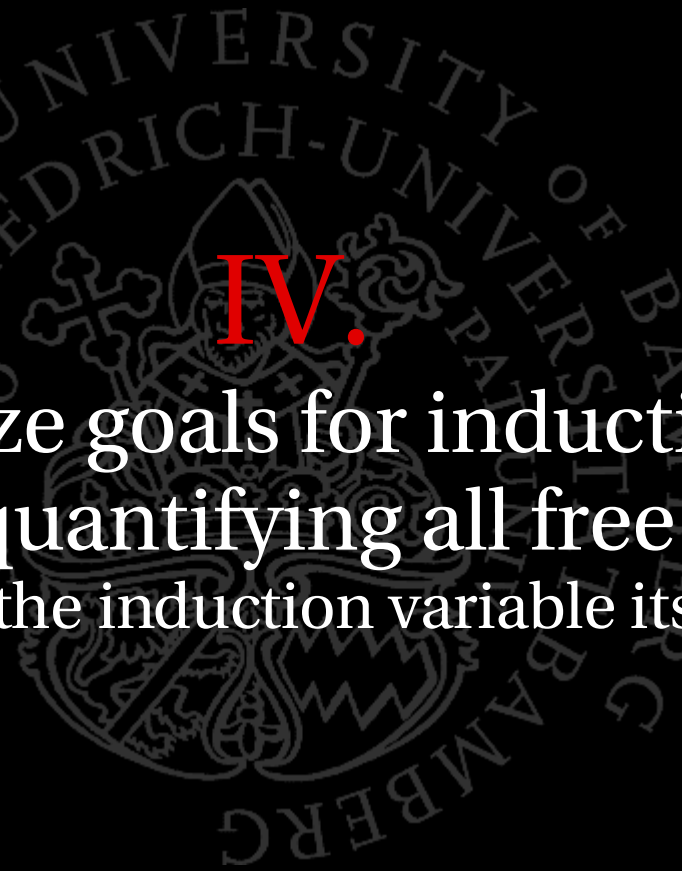


II.

Do induction on argument number i
if the function is defined by
recursion in argument number i.

III.

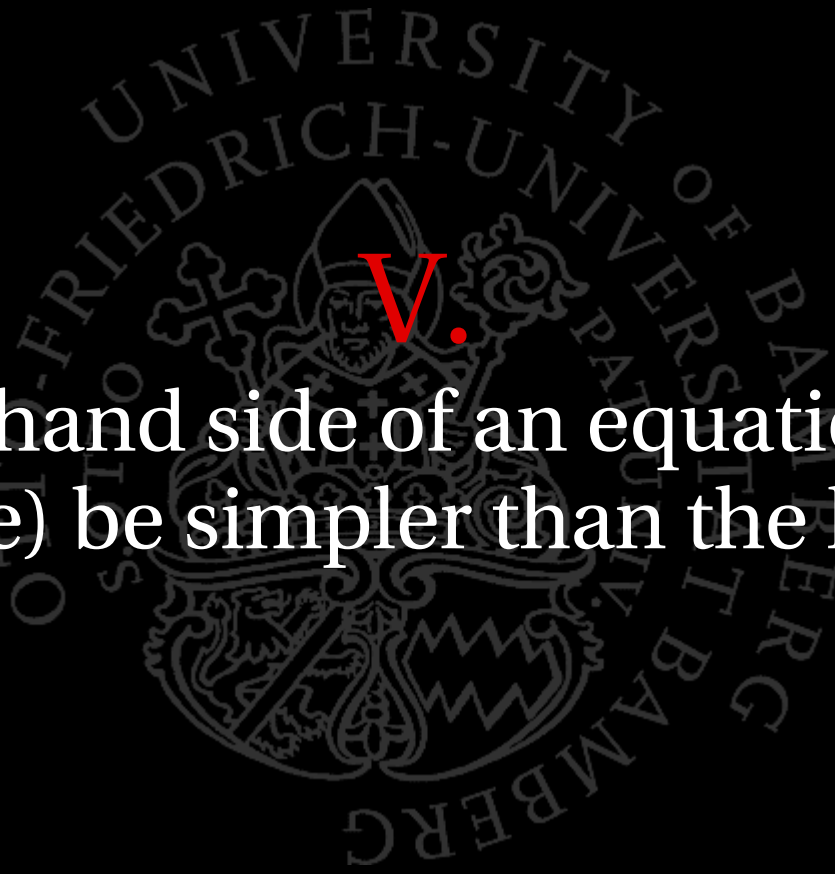
Generalize goals for induction
by replacing constants by variables.



IV.

Generalize goals for induction by universally quantifying all free variables.

(except the induction variable itself !)



V.

The right-hand side of an equation should
(in some sense) be simpler than the left-hand side.

3. More Functional Programming

3.2. Induction Heuristics

I. Theorems about recursive functions are proved by induction.

II. Do induction on argument number i if the function is defined by recursion in argument number i.

III. Generalize goals for induction by replacing constants by variables.

IV. Generalize goals for induction by universally quantifying all free variables.

V. The right-hand side of an equation should (in some sense) be simpler than the left-hand side.

Let's try that!

Example: `itrev`

```
consts
  itrev :: "'a list => 'a list => 'a list"

primrec
  "itrev [] ys = ys"
  "itrev (x#xs) ys = itrev xs (x#ys)"
```

`itrev` reverses its first argument by
stacking its elements onto the second argument,
which is returned when the first one is empty.

3. More Functional Programming

3.2. Induction Heuristics

Now, this should be working like the old one, shouldn't it?

```
lemma "itrev xs [] = rev xs"
```

The obvious way to do it:

```
apply(induct_tac xs, simp_all)
```

Unfortunately it doesn't seem to work.

```
goal (lemma, 1 subgoal):
1. !!a list.
     itrev list []  = rev list ==>
     itrev list [a] = rev list @ [a]
```

Let's add some number III. and see what's happening.

```
lemma "itrev xs ys = rev xs @ ys"
```

Still not very promising...

```
goal (lemma, 1 subgoal):
1. !!a list.
     itrev list       ys  = rev list @ ys ==>
     itrev list (a # ys) = rev list @ a # ys
```

This time `ys` is fixed throughout the subgoal,
but the induction hypothesis
needs to be applied with `a#ys` instead of `ys`.

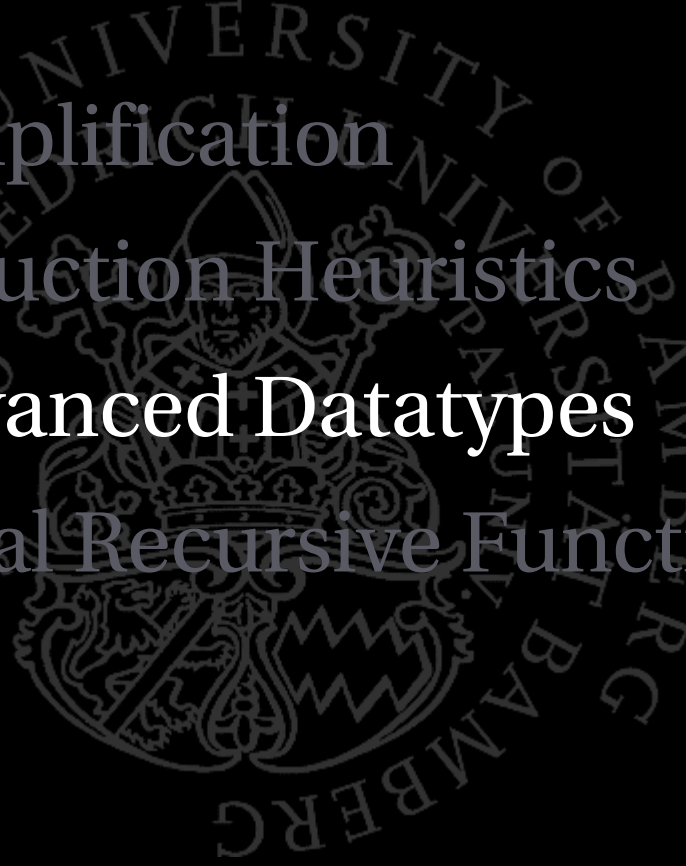
Universally quantifying the free variables could help:

```
lemma "ALL ys. itrev xs ys = rev xs @ ys"
```

And we're done!
All subgoals have been proved.

```
goal (lemma): No subgoals!
```

3. More Functional Programming



Of mutual and nested recursion...

Mutual Recursion

```
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"
```

Defining two datatypes that depend on each other:
Arithmetic and Boolean Expressions

Mutual Recursion

```
consts
  evala :: "'a aexp => ('a => nat) => nat"
  evalb :: "'a bexp => ('a => nat) => bool"
```

Semantics for evaluation functions.
Both functions take an expression and
an environment (a mapping from variables to `nat`)
and return its value.

Mutual Recursion

Since the datatypes are mutually recursive,
so are functions that operate on them.
Hence they need to be defined
in a single primrec section.

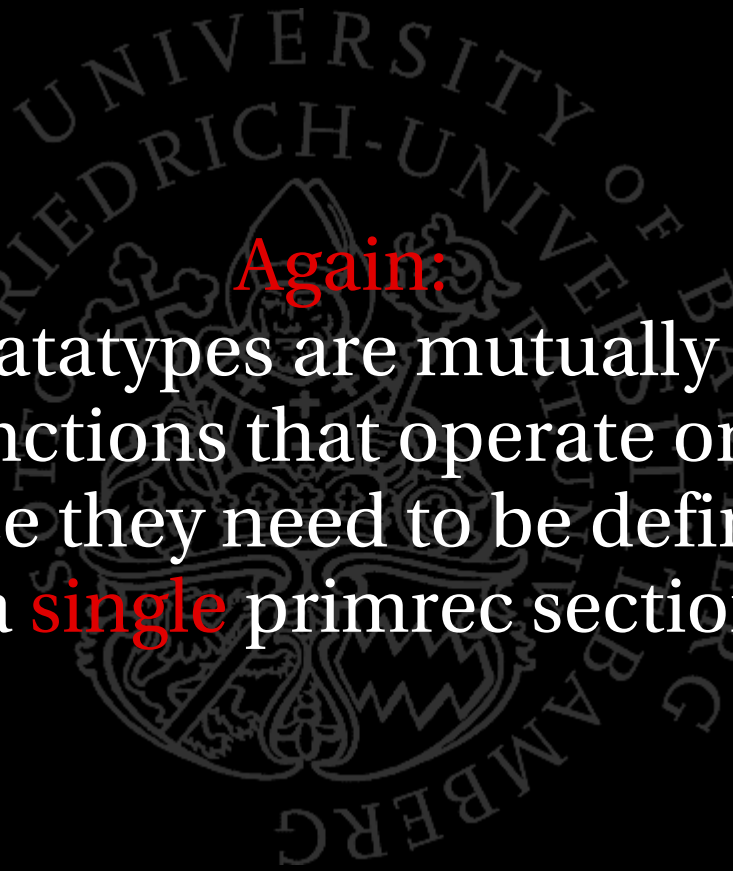
```
primrec
  "evala (IF b a1 a2) env =
     (if evalb b env
      then evala a1 env
      else evala a2 env)"
  "evala (Sum a1 a2) env =
      evala a1 env + evala a2 env"
  "evala (Diff a1 a2) env =
      evala a1 env - evala a2 env"
  "evala (Var v) env = env v"
  "evala (Num n) env = n"
  "evalb (Less a1 a2) env =
     (evala a1 env < evala a2 env)"
  "evalb (And b1 b2) env =
     (evalb b1 env & evalb b2 env)"
  "evalb (Neg b) env = (~ evalb b env)"
```

Mutual Recursion

```
consts
  substa ::
   "('a => 'b aexp) => 'a aexp => 'b aexp"
  substb ::
   "('a => 'b aexp) => 'a bexp => 'b bexp"
```

Semantics for substition functions.
The first argument is a function mapping variables to expressions, the substitution.
As it is applied to all variables in the second argument, the type of variables in the expression changes from `'a` to `'b`.

Mutual Recursion



Again:

Since the datatypes are mutually recursive,
so are functions that operate on them.
Hence they need to be defined
in a single primrec section.

```
primrec
  "substa s (IF b a1 a2) =
    IF (substb s b)
       (substa s a1)
       (substa s a2)"
  "substa s (Sum  a1 a2) =
           Sum  (substa s a1) (substa s a2)"
  "substa s (Diff a1 a2) =
           Diff (substa s a1) (substa s a2)"
  "substa s (Var v) = s v"
  "substa s (Num n) = Num n"
  "substb s (Less a1 a2) =
           Less (substa s a1) (substa s a2)"
  "substb s (And b1 b2) =
           And  (substb s b1) (substb s b2)"
  "substb s (Neg b) = Neg (substb s b)"
```

Mutual Recursion

```
lemma
  "evala (substa s a) env =
        evala a (%x. evala (s x) env) &

   evalb (substb s b) env =
        evalb b (%x. evala (s x) env)"

apply(induct_tac a and b)
apply(simp_all)
done
```

We need to state and prove both theorems simultaneously.

Mutual Recursion

Given n mutually recursive datatypes

$$\tau_1, \ldots, \tau_n,$$

an inductive proof expects a goal of the form

$$P_1(x_1) \wedge \ldots \wedge P_n(x_n)$$

where each variable x_i is of type τ_i.

```
apply(induct_tac x1 and ... and xn)
```

Nested Recursion

```
datatype ('v,'f)"term" =
  Var 'v | App 'f "('v,'f)term list"
```

A datatype (term) is nested inside another datatype (list).

Here function symbols can be applied to a list of arguments. Parameter 'v is the type of variables and 'f the type of function symbols.

A mathematical term f(x, g(y)) becomes
`f [Var x, App g [Var y]]`.

Nested Recursion

```
datatype
  ('v,'f)"term" =
    Var 'v | App 'f "('v,'f)term_list"
and
  ('v,'f)term_list = Nil
    | Cons "('v,'f)term" "('v,'f)term_list"
```

Elimination of nested recursion in favor of mutual recursion.

Here term and term_list are mutual recursive.

Nested Recursion

```
consts
  subst  :: "('v => ('v,'f)term) =>
       ('v,'f)term        => ('v,'f)term"
  substs :: "('v => ('v,'f)term) =>
       ('v,'f)term list => ('v,'f)term list"
```

Definition for substitution functions.
Because terms involve term lists, we need to define both functions simultaneously...

Nested Recursion

```
primrec
  "subst s (Var x) = s x"
  "subst s (App f ts) =
                App f (substs s ts)"
  "substs s [] = []"
  "substs s (t # ts) =
                subst s t # substs s ts"
```

... and again put them in one single `primrec`-block.

Nested Recursion

```
lemma
  "subst  Var t  =  t  ::('v,'f)term) &
   substs Var ts = (ts ::('v,'f)term list)"

apply(induct_tac t and ts, simp_all)
done
```

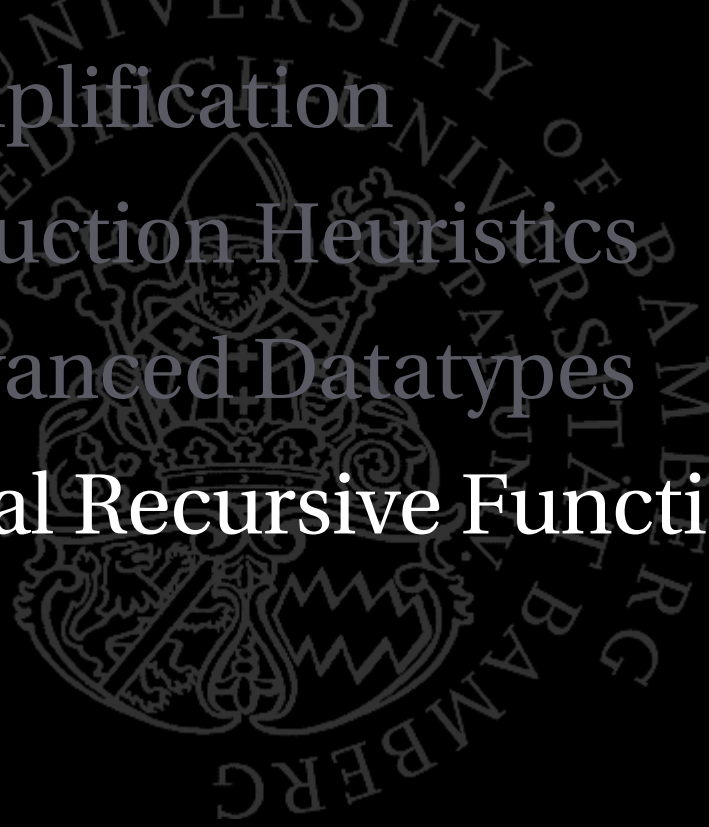
When proving a statement about terms, we need to prove a related statement about term lists simultaneously.

The identity substitution

`subst Var (Var x) = Var x`

does not change a term.

3. More Functional Programming



Finally, the good stuff!

Full pattern-matching,
real recursion,
proof of termination.

Total Recursive Functions

```
consts
  fib :: "nat => nat"

recdef fib "measure(%n. n)"
  "fib 0 = 0"
  "fib (Suc 0) = 1"
  "fib (Suc(Suc x)) = fib x + fib (Suc x)"
```

The `measure` functions maps the argument of `fib` to a natural number.

In each equation the measure of the argument on the left-hand side must be greater than the measure of the argument of each recursive call.

Total Recursive Functions

```
consts
  sep :: "'a * 'a list => 'a list"

recdef sep "measure (%(a,xs). length xs)"
  "sep(a, []) = []"
  "sep(a, [x]) = [x]"
  "sep(a, x#y#zs) = x # a # sep(a, y#zs)"
```

Inserting a separator to a list.

Here `measure` is the length of the list,
which is decreasing in each recursive call.

Total Recursive Functions

```
consts
  last :: "'a list => 'a"

recdef last "measure (%xs. length xs)"
  "last [x] = x"
  "last (x#y#zs) = last (y#zs)"
```

Pattern-matching need not be exhaustive.

Total Recursive Functions

```
consts
  sep1 :: "'a * 'a list => 'a list"

recdef sep1 "measure (%(a,xs). length xs)"
  "sep1(a, x#y#zs) = x # a # sep1(a, y#zs)"
  "sep1(a, xs) = xs"
```

Overlapping patterns are disambiguated by taking the order of equations into account.

By internal replacing in ISABELLE
the two functions `sep` and `sep1` are identical.

Total Recursive Functions

```
consts
  sep2 :: "'a list => 'a => 'a list"

recdef sep2 "measure length"
 "sep2(x#y#zs) = (%a. x # a # sep2(y#zs) a)"
 "sep2 xs = (%a. xs)"
```

As only the first argument can be used
for pattern matching and termination measure,
in general we will have to combine several arguments in to a tuple. If only one argument is relevant for termination we can rearrange the order of arguments.

Total Recursive Functions

```
consts
  qs :: "nat list => nat list"

recdef qs "measure length"
  "qs [] = []"
  "qs(x#xs) = qs(filter (%y. y<=x) xs)
      @ [x] @ qs(filter (%y. x< y) xs)"
  (hints recdef_simp: less_Suc_eq_le)
```

Helping ISABELLE prove termination by giving `hints`.

If auto-proving for termination fails
we get an error message what it failed to prove
and (of course) can't use the function.

Total Recursive Functions

Once we have proved all the termination conditions, the `recdef` recursion equations become simplification rules.

Because of this we must pay attention to `if`-expressions, as they are split automatically.

Try to give a definition by pattern-matching on the left-hand side instead of `if` on the right-hand side.

Or replace `if` by `case`, which is not automatically split.

Total Recursive Functions

```
lemma
  "map f (sep(x,xs)) = sep(f x, map f xs)"
apply(induct_tac x xs rule: sep.induct)
apply(simp_all)
done
```

Proving theorems about `recdef`-functions (like map here) works by induction on the variables and the automatically generated rule *`function_name`*`.induct`.

3. More Functional Programming

... any questions left ?