IJCAR 2004 — Tutorial 4

Introduction to the Isabelle Proof Assistant

Clemens Ballarin Gerwin Klein

Tutorial Schedule

- Session I
  - Basics
- Session II
  - Specification Tools
  - Readable Proofs
- Session III
  - More on Readable Proofs
  - Modules
- Session IV
  - Applications
  - Q & A session with Larry Paulson

Session I

Basics

System Architecture

User can access all layers!

Proof General — User interface

HOL, ZF — Object-logics

Isabelle — Generic, interactive theorem prover

Standard ML — Logic implemented as ADT

Documentation

Available from http://isabelle.in.tum.de

- Learning Isabelle
  - Tutorial on Isabelle/HOL (LNCS 2283)
  - Tutorial on Isar
  - Tutorial on Locales
- Reference Manuals
  - Isabelle/Isar Reference Manual
  - Isabelle Reference Manual
  - Isabelle System Manual
- Reference Manuals for Object-Logics

Isabelle's Meta-Logic

- Intuitionistic fragment of Church's theory of simple types.
- With type variables.
- Can be used to formalise your own object-logic.
- Normally, use rich infrastructure of the object-logics HOL and ZF.
- This presentation assumes HOL.

Types

Syntax

Syntax:

τ	$::=$	(τ)	
	$\mid$	*'a* $\mid$ *'b* $\mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	total functions
	$\mid$	*bool* $\mid$ *nat* $\mid \ldots$	HOL base types
	$\mid$	$\tau \times \tau$	HOL pairs (ascii: $*$)
	$\mid$	τ *list*	HOL lists
	$\mid$	$\ldots$	user-defined types

Parentheses: *T1* $\Rightarrow$ *T2* $\Rightarrow$ *T3* $\equiv$ *T1* $\Rightarrow$ (*T2* $\Rightarrow$ *T3*)

Introducing new Types: typedecl

typedecl $name$

Introduces new “opaque” type $name$ without definition.

Example:

typedecl *addr* — An abstract type of addresses.

Terms

Syntax

Syntax: (curried version)

$term$	$::=$	$(term)$	
	$\mid$	a	constant or variable (identifier)
	$\mid$	$term\ term$	function application
	$\mid$	$\lambda x.\ term$	function “abstraction”
	$\mid$	$\ldots$	lots of syntactic sugar

Examples: $f\,(g\,x)\,y$ $\qquad h\,(\lambda x.\,f\,(g\,x))$

Parentheses: $f\,a_1\,a_2\,a_3 \equiv ((f\,a_1)\,a_2)\,a_3$

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ ("unknown")

- Logically: free = schematic
- Operationally:
  - free variables are fixed
  - schematic variables are instantiated by substitutions and unification

Theorems

Connectives of the Meta-Logic

Implication $\Longrightarrow$ (==>)

For separating premises and conclusion of theorems.

Equality $\equiv$ (==)

For definitions.

Universal quantifier $\bigwedge$ (!!)

For parameters in goals.

Do not use *inside* object-logic formulae.

Notation

$$[\![A_1; \ \dots \ ; A_n]\!] \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \dots \Longrightarrow A_n \Longrightarrow B$$

$;$ $\approx$ “and”

Introducing New Theorems

- As axioms.
- Through definitions.
- Through proofs.

! Axioms should mainly be used when specifying object-logics. !

Definition (non-recursive)

Declaration:

consts

sq :: nat $\Rightarrow$ *nat*

Definition:

defs

sq_def: sq n $\equiv$ *n*n*

Declaration + definition:

constdefs

sq :: nat $\Rightarrow$ *nat*
sq n $\equiv$ *n*n*

Proofs

General schema:

lemma $name$: <*goal*>
  apply <*method*>
  apply <*method*>
  ⋮
  done

- Sequential application of methods until all subgoals are solved.

The proof state

1. $\bigwedge x_1 \ldots x_p.\ [\![A_1; \ldots ; A_n]\!] \Longrightarrow B$
2. $\bigwedge y_1 \ldots y_q.\ [\![C_1; \ldots ; C_n]\!] \Longrightarrow D$

$x_1 \ldots x_p$	Parameters
$A_1 \ldots A_n$	Local assumptions
B	Actual (sub)goal

Isabelle Theories

Theory = Source file

Syntax:

theory $MyTh$ imports $ImpTh_1 \ldots ImpTh_n$ begin
(declarations, definitions, theorems, proofs, ...)*
end

- $MyTh$: name of theory. Must live in file $MyTh$.thy
- $ImpTh_i$: name of *imported* theories. Import transitive.

Unless you need something special:

theory $MyTh$ imports Main begin

X-Symbols

Input of funny symbols in Proof General

- via menu ("X-Symbol")
- via ascii encoding (similar to LaTeX): \<and>, \<or>, . . .
- via abbreviation: /\, \/, -->, . . .

x-symbol	$\forall$	$\exists$	λ	$\neg$	$\wedge$	$\vee$	$\longrightarrow$	$\Rightarrow$
ascii (1)	\<forall>	\<exists>	\<lambda>	\<not>	/\	\/	-->	=>
ascii (2)	ALL	EX	%	~	&	|		

(1) is converted to x-symbol, (2) stays ascii.

Demo: Isabelle theories

Natural Deduction

Rules

$$\frac{A \quad B}{A \wedge B} \text{ conjI} \qquad \frac{A \wedge B \quad [\![A;B]\!] \Longrightarrow C}{C} \text{ conjE}$$

$$\frac{A}{A \vee B} \quad \frac{B}{A \vee B} \text{ disjI1/2} \qquad \frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C} \text{ disjE}$$

$$\frac{A \Longrightarrow B}{A \longrightarrow B} \text{ impI} \qquad \frac{A \longrightarrow B \quad A \quad B \Longrightarrow C}{C} \text{ impE}$$

Proof by assumption

apply *assumption*

proves

$1.\ [\![\, B_1; \ldots ; B_m \,]\!] \Longrightarrow C$

by unifying C with one of the B_i (backtracking!)

How to prove it by natural deduction

- Intro rules decompose formulae to the right of $\Longrightarrow$.

  apply(*rule <intro-rule>*)

  Applying rule $[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$ to subgoal C:
  - Unify A and C
  - Replace C with n new subgoals $A_1 \ldots A_n$
- Elim rules decompose formulae on the left of $\Longrightarrow$.

  apply(*erule <elim-rule>*)

  Like *rule* but also
  - unifies first premise of rule with an assumption
  - eliminates that assumption

Demo: natural deduction

Safe and unsafe rules

Safe rules preserve provability

`conjI, impI, conjE, disjE,`
`notI, iffI, refl, ccontr, classical`

Unsafe rules can turn provable goal into unprovable goal

`disjI1, disjI2, impE,`
`iffD1, iffD2, notE`

Apply safe rules before unsafe ones

Predicate Logic: $\forall$ and $\exists$

Scope

- Scope of parameters: whole subgoal
- Scope of $\forall$, $\exists$, ...: ends with *;* or $\Longrightarrow$

$$\bigwedge x\, y.\ [\![\forall y.\ P\ y \longrightarrow Q\ z\ y;\ \ Q\ x\ y\ ]\!] \Longrightarrow \exists x.\ Q\ x\ y$$

means

$$\bigwedge x\, y.\ [\![(\forall y_1.\ P\ y_1 \longrightarrow Q\ z\ y_1);\ \ Q\ x\ y\ ]\!] \Longrightarrow (\exists x_1.\ Q\ x_1\ y)$$

Natural deduction for quantifiers

$$\frac{\bigwedge x.\ P\ x}{\forall x.\ P\ x}\ \texttt{allI} \qquad \frac{\forall x.\ P\ x \quad P\ ?x \Longrightarrow R}{R}\ \texttt{allE}$$

$$\frac{P\ ?x}{\exists x.\ P\ x}\ \texttt{exI} \qquad \frac{\exists x.\ P\ x \quad \bigwedge x.\ P\ x \Longrightarrow R}{R}\ \texttt{exE}$$

- `allI` and `exE` introduce new parameters ($\bigwedge x$).
- `allE` and `exI` introduce new unknowns ($?x$).

Instantiating rules

apply(*rule_tac x* = "$term$" *in* $rule$)

Like *rule*, but *?x* in *rule* is instantiated by *term* before application.

Similar: *erule_tac*

! *x* is in *rule*, not in the goal **!**

Safe and unsafe rules

Safe allI, exE

Unsafe allE, exI

Create parameters first, unknowns later

Forward proofs: frule and drule

apply(*frule* $rulename$)

Forward rule: $A_1 \Longrightarrow A$

Subgoal: *1.* $\llbracket B_1; \ldots ; B_n \rrbracket \Longrightarrow C$

Unifies: one B_i with A_1

New subgoal: *1.* $\llbracket B_1; \ldots ; B_n; A \rrbracket \Longrightarrow C$

apply(*drule* $rulename$)

Like *frule* but also deletes B_i

Demo: quantifier proofs

Practical Session I

In the cool morning
A man simplifies, a goal
A theorem is born.

— Don Syme

Session II

HOL = Functional programming + Logic

Proof by Term Rewriting

Term rewriting means . . .

Using equations $l = r$ from left to right

as long as possible

Terminology: equation $\rightsquigarrow$ rewrite rule

Example

Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$

Result: $a + (b + c)$

Rewrite rules can be conditional: $[\![P_1 \ldots P_n]\!] \Longrightarrow l = r$ is used

- like $l = r$, but
- P_1, . . . , P_n must be proved by rewriting first.

Simplification in Isabelle

Goal: *1.* $[\![$ *P*$_1$; $\ldots$; *P*$_m$ $]\!] \Longrightarrow$ *C*

apply(*simp add: eq*$_1$ $\ldots$ *eq*$_n$)

Simplify *P*$_1$ $\ldots$ *P*$_m$ and *C* using

- lemmas with attribute *simp*
- additional lemmas *eq*$_1$ $\ldots$ *eq*$_n$
- assumptions *P*$_1$ $\ldots$ *P*$_m$

Variations:

- *(simp* $\ldots$ *del:* $\ldots$*)* removes *simp*-lemmas
- *add* and *del* are optional

Termination

Simplification may not terminate.
Isabelle uses *simp*-rules (almost) blindly from left to right.

Example: $f(x) = g(x), g(x) = f(x)$

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

is suitable as a *simp*-rule only
if l is “bigger” than r and each P_i

n < m $\Longrightarrow$ *(n < Suc m) = True* YES
Suc n < m $\Longrightarrow$ *(n < m) = True* NO

How to ignore assumptions

Assumptions sometimes cause problems, e.g. nontermination. How to exclude them from *simp*:

apply*(simp (no_asm_simp) ...)*
Simplify only conclusion

apply*(simp (no_asm_use) ...)*
Simplify but do not use assumptions

apply*(simp (no_asm) ...)*
Ignore assumptions completely

Tracing

Set trace mode on/off in Proof General:

Isabelle/Isar $\longrightarrow$ Settings $\longrightarrow$ Trace simplifier

Output in separate buffer:

Proof-General $\longrightarrow$ Buffers $\longrightarrow$ Trace

auto

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1
- *auto* applies *simp* and more

Demo: simp

Type definitions in Isabelle/HOL

Keywords:

- **typedecl**: pure declaration (session 1)
- **types**: abbreviation
- **datatype**: recursive datatype

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

Examples:

types
name = string
('a,'b)foo = "'a list $\times$ *'b list"*

Type abbreviations are expanded after parsing
Not present in internal representation and Isabelle output

datatype

datatype *'a list = Nil | Cons 'a "'a list"*

Properties:

- Types: *Nil* :: *'a list*

  Cons :: *'a* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list*
- Distinctness: *Nil* $\neq$ *Cons x xs*
- Injectivity: *(Cons x xs = Cons y ys) = (x = y* $\wedge$ *xs = ys)*

case

Every datatype introduces a *case* construct, e.g.

$$(case\ xs\ of\ Nil \Rightarrow \ldots \mid Cons\ y\ ys \Rightarrow \ldots\ y \ldots\ ys \ldots)$$

- one case per constructor
- no nested patterns (*Cons x (Cons y zs)*)
- but nested cases

apply(*case_tac xs*) $\Rightarrow$ one subgoal for each constructor

$xs = Nil \Longrightarrow \ldots$
$xs = Cons\ a\ list \Longrightarrow \ldots$

Function definition schemas in Isabelle/HOL

- Non-recursive with **constdefs** (session 1)
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction
- Well-founded recursion with **recdef**
  User must (help to) prove termination

primrec

consts *app :: "'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list"*
primrec
"app Nil ys = ys"
"app (Cons x xs) ys = Cons x (app xs ys)"

- Each recursive call structurally smaller than lhs.
- Equations used automatically in simplifier

Structural induction

P xs holds for all lists *xs* if

- *P Nil*
- and for arbitrary *x* and *xs*, *P xs* implies *P (Cons x xs)*

  Induction theorem list.induct:
  $[\![P\ Nil;\ \bigwedge a\ list.\ P\ list \Longrightarrow P\ (Cons\ a\ list)]\!]$

  $\Longrightarrow P\ list$
- General proof method for induction: *(induct x)*
  - *x* must be a free variable in the first subgoal.
  - The type of *x* must be a datatype.

Induction heuristics

Theorems about recursive functions proved by induction

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

lemma *itrev xs [] = rev xs*

Demo: proof attempt

Generalisation

Replace constants by variables

lemma *itrev xs ys = rev xs @ ys*

Quantify free variables by $\forall$
(except the induction variable)

lemma $\forall$ *ys. itrev xs ys = rev xs @ ys*

Function definition schemas in Isabelle/HOL

- Non-recursive with **constdefs** (session 1)
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction
- Well-founded recursion with **recdef**
  User must (help to) prove termination

recdef — examples

consts *sep* :: "'a $\times$ 'a list $\Rightarrow$ 'a list"
recdef *sep* "measure (λ(a, xs). size xs)"

  "sep (a, x # y # zs) = x # a # sep (a, y # zs)"
  "sep (a, xs) = xs"

consts *ack* :: "nat $\times$ nat $\Rightarrow$ nat"
recdef *ack* "measure (λm. m) <*lex*> measure (λn. n)"
  "ack (0, n) = Suc n"
  "ack (Suc m, 0) = ack (m, 1)"
  "ack (Suc m, Suc n) = ack (m, ack (Suc m, n))"

recdef

- The definiton:
  - one parameter
  - free pattern matching, order of rules important
  - termination relation
    (*measure* sufficient for most cases)
- Termination relation:
  - must decrease for each recursive call
  - must be well founded
- Generates own induction principle.

Demo: recdef and induction

Sets

Notation

Type *'a set*: sets over type *'a*

- $\{\}$, $\{e_1,\ldots,e_n\}$, $\{x.\ P\ x\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, A - B, - A
- $\bigcup_{x\in A} B\ x$, $\bigcap_{x\in A} B\ x$
- $\{i..j\}$
- *insert* :: $'a \Rightarrow$ *'a set* $\Rightarrow$ *'a set*
- f ` $A \equiv \{y.\ \exists x{\in}A.\ y = f\ x\}$
- ...

Inductively defined sets: even numbers

Informally:

- 0 is even
- If n is even, so is $n + 2$
- These are the only even numbers

In Isabelle/HOL:

consts *Ev :: nat set* — The set of all even numbers

inductive *Ev*

intros

$0 \in Ev$

$n \in Ev \Longrightarrow n + 2 \in Ev$

Rule induction for Ev

To prove

$$n \in Ev \Longrightarrow P\ n$$

by *rule induction* on $n \in Ev$ we must prove

- $P\ 0$
- $P\ n \Longrightarrow P(n+2)$

Rule `Ev.induct`:

$$[\![\ n \in Ev;\ P\ 0;\ \bigwedge n.\ P\ n \Longrightarrow P(n+2)\ ]\!] \Longrightarrow P\ n$$

An elimination rule

Demo: inductively defined sets

Isar

A Language for Structured Proofs

Apply scripts

- unreadable
- hard to maintain
- do not scale

No structure!

A typical Isar proof

proof

  assume $formula_0$

  have $formula_1$ **by** *simp*

  $\vdots$

  have $formula_n$ **by** *blast*

  show $formula_{n+1}$ **by** $\ldots$

qed

proves $formula_0 \Longrightarrow formula_{n+1}$

Isar core syntax

proof = **proof** [method] statement* **qed**
| **by** method

method = *(simp ...)* | *(blast ...)* | *(rule ...)* | ...

statement = **fix** variables ($\bigwedge$)
| **assume** proposition ($\Longrightarrow$)
| [**from** name$^+$] (**have** | **show**) proposition proof
| **next** (separates subgoals)

proposition = [name:] formula

Demo: propositional logic

Elimination rules / forward reasoning

- Elim rules are triggered by facts fed into a proof:
  from $\vec{a}$ **have** $formula$ **proof**
- **from** $\vec{a}$ **have** $formula$ **proof** ***(****rule* $rule$***)***

  $\vec{a}$ must prove the first n premises of $rule$
  in the right order
  the others are left as new subgoals
- **proof** alone abbreviates **proof** *rule*
- *rule*: tries elim rules first
  (if there are incoming facts $\vec{a}$!)

Practical Session II

Theorem proving and sanity; Oh, my! What a delicate balance.

— Victor Carreno

Session III

More about Isar

Overview

- Abbreviations
- Predicate Logic
- Accumulating facts
- Reasoning with chains of equations
- Locales: the module system

Abbreviations

this	=	the previous proposition proved or assumed
then	=	**from** *this*
with $\vec{a}$	=	**from** $\vec{a}$ *this*
?thesis	=	the last enclosing **show** formula

Mixing proof styles

from . . .

have . . .

  apply - make incoming facts assumptions

  apply(. . .)

  ⋮

  apply(. . .)

  done

Demo: Abbreviations

Predicate Calculus

fix

Syntax:

fix variables

Introduces new arbitrary but fixed variables
($\sim$ parameters)

obtain

Syntax:

obtain variables **where** proposition proof

Introduces new variables together with property

Demo: predicate calculus

moreover/ultimately

have $formula_1$ **. . .**

moreover

have $formula_2$ **. . .**

moreover

⋮

moreover

have $formula_n$ **. . .**

ultimately

show . . .

— pipes facts $formula_1 \ldots formula_n$ into the proof

proof . . .

Demo: moreover/ultimately

General case distinctions

show $formula$

proof -

have $P_1 \vee P_2 \vee P_3$ ...

moreover

{ **assume** P_1 ... **have** *?thesis* ... }

moreover

{ **assume** P_2 ... **have** *?thesis* ... }

moreover

{ **assume** P_3 ... **have** *?thesis* ... }

ultimately **show** *?thesis* **by** *blast*

qed

Chains of equations

- Keywords **also** and **finally**.
- $\ldots$: predefined schematic term variable,
  refers to the right hand side of the last expression.
- Uses transitivity rule.

also/finally

have "t_0 = t_1" ...

also $t_0 = t_1$

have "$\ldots$ = t_2" ...

also $t_0 = t_2$

⋮ ⋮

also $t_0 = t_{\texttt{n-1}}$

have "$\ldots$ = t_n" ...

finally show ...

— pipes fact $t_0 = t_n$ into the proof

proof

⋮

More about also

- Works for all combinations of =, $\leq$ and <.
- Uses rules declared as `[trans]`.
- To view all combinations in Proof General:
  Isabelle/Isar $\rightarrow$ Show me $\rightarrow$ Transitivity rules

Demo: also/finally

Locales

Isabelle's Module System

Isar is based on contexts

theorem $\bigwedge x.\ A \Longrightarrow C$

proof -

fix x

assume Ass: A

$\vdots$ — x and Ass are visible

from Ass **show** C ... — inside this context

qed

Beyond Isar contexts

Locales are extended contexts

- Locales are named
- Fixed variables may have syntax
- It is possible to add and export theorems
- Locale expression: combine and modify locales

Context elements

Locales consist of context elements.

fixes	Parameter, with syntax
assumes	Assumption
defines	Definition
notes	Record a theorem

Declaring locales

locale *loc* =

    loc1 +   Import

    fixes . . .   Context elements

    assumes . . .

Declares named locale *loc*.

Declaring locales

Theorems may be stated relative to a named locale.

lemma (**in** *loc*) *P* [simp]: *proposition*
proof

- Adds theorem *P* to context *loc*.
- Theorem *P* is in the simpset in context *loc*.
- Exported theorem *loc.P* visible in the entire theory.

Demo: locales 1

Parameters must be consistent!

- Parameters in fixes are distinct.
- Free variables in assumes and defines occur in preceding fixes.
- Defined parameters must neither occur in preceding assumes nor defines.

Locale expressions

Locale name: n

Rename: $e\ q_1\ \ldots\ q_n$

Change names of parameters in e.

Merge: $e_1 + e_2$

Context elements of e_1, then e_2.

- Syntax is lost after rename (currently).

Demo: locales 2

Normal form of locale expressions

Locale expressions are converted to flattened lists of locale names.

- With full parameter lists
- Duplicates removed

Allows for multiple inheritance!

Interpretation

Move from abstract to concrete.

interpret *label* : *loc* $[t_1 \ldots t_n]$ *proof*

- Interpret *loc* with parameters $t_1 \ldots t_n$
- Generates proof obligation.
- Imports all theorems of *loc* into current context.
  - Instantiates the parameters with $t_1 \ldots t_n$.
  - Interprets attributes of theorems.
  - Prefixes theorem names with *label*
- Currently only works inside Isar contexts.

Demo: locales 3

Practical Session III

The sun spills darkness
A dog howls after midnight
Goals remain unsolved.

— Chris Owens

Session IV

Case Studies

Case Study

Compiling Expressions

The Task

- develop a compiler
- from expressions
- to a stack machine
- and show its correctness

- expressions built from
  - variables
  - constants
  - binary operations

Expressions — Syntax

Syntax for

- binary operations
- expressions

Design decision:

- no syntax for variables and values

Instead:

- expressions generic in variable names,
- *nat* for values.

Expressions — Data Type

- Binary operations

  datatype *binop = Plus | Minus | Mult*

- Expressions

  datatype *'v expr = Const nat*
  | Var 'v
  | Binop binop "'v expr" "'v expr"

- *'v* = variable names

Expressions — Semantics

- Sematics for binary operations:

  consts *semop :: "binop $\Rightarrow$ nat $\Rightarrow$ nat $\Rightarrow$ nat" ("$[\![\_]\!]$")*

  primrec *"$[\![Plus]\!]$ = (λx y. x + y)"*

  "$[\![Minus]\!]$ = (λx y. x - y)"

  *"$[\![Mult]\!]$ = (λx y. x * y)"*

- Sematics for expressions:

  consts *"value" :: "'v expr $\Rightarrow$ ('v $\Rightarrow$ nat) $\Rightarrow$ nat"*

  primrec

  "value (Const v) E = v"

  "value (Var a) E = E a"

  "value (Binop f e_1 e_2) E = $[\![f]\!]$ (value e_1 E) (value e_2 E)"

Stack Machine — Syntax

Machine with 3 instructions:

- push constant value onto stack
- load contents of register onto stack
- apply binary operator to top of stack

Simplification: register names = variable names

datatype *'v instr = Push nat*
| Load 'v
| Apply binop

Stack Machine — Execution

Modelled by a function taking

- list of instructions (program)
- store (register names to values)
- list of values (stack)

Returns

- new stack

exec

consts *exec :: "'v instr list* $\Rightarrow$ *('v* $\Rightarrow$ *nat)* $\Rightarrow$ *nat list* $\Rightarrow$ *nat list"*

primrec

"exec [] s vs = vs"

"exec (i#is) s vs = (case i of

Push v $\Rightarrow$ *exec is s (v # vs)*

| Load a $\Rightarrow$ *exec is s (s a # vs)*

| Apply f $\Rightarrow$ *let* v_1 *= hd vs;* v_2 *= hd (tl vs); ts = tl (tl vs) in*

exec is s ($[\![f]\!]$ v_1 v_2 *# ts))"*

- *hd* and *tl* are head and tail of lists

The Compiler

Compilation easy:

- *Constants* $\Rightarrow$ *Push*
- *Variables* $\Rightarrow$ *Load*
- *Binop* $\Rightarrow$ *Apply*

consts *comp :: "'v expr* $\Rightarrow$ *'v instr list"*

primrec

"comp (Const v) = [Push v]"

"comp (Var a) = [Load a]"

"comp (Binop f e_1 e_2*) = (comp* e_2*) @ (comp* e_1*) @ [Apply f]"*

Correctness

Executing compiled program yields value of expression

theorem *"exec (comp e) s [] = [value e s]"*

Proof?

Demo: correctness proof

Case Study
Commutative Algebra

Abstract Mathematics

- Concerns classes of objects specified by axioms, not concrete objects like the integers or reals.
- Objects are typically structures: $(G, \cdot, 1, {}^{-1})$
  - Groups, rings, lattices, topological spaces
- Concepts are frequently combined and extended.
- Instances may be concrete or abstract.

Formalisation

- Structures are not theories of proof tools.
- Structures must be first-class values.
- Syntax should reflect context:
  - If G is a group, then $(x \cdot y)^{-1} = y^{-1} \cdot x^{-1}$ refers implicitly to G.
- Inheritance of syntax and theorems should be automatic.

Support for Abstraction

- Locales: portable contexts.
- ı (`\<index>`) arguments in syntax declarations.
- Extensible records (in HOL).
- Locale instantiation.

Index Arguments in Syntax Declarations

- One function argument may be \<index>.
- Works also for infix operators and binders:

  $x \otimes_{G} y \qquad \bigoplus_{R} i \in \{0..n\}.\ f\ i$
- Good for denoting record fields.
- Can declare default by **(structure)**.
- Yields a concise syntax for G while allowing references to other groups.
- Letter subscripts for \<index> only available in current development version of Isabelle.

Records

- Are used to represent structures.
- Fields are functions and can have special syntax.
- Records can be extended with additional fields.

record 'a monoid =
  carrier :: "'a set"
  mult :: "['a, 'a] $\Rightarrow$ 'a" (**infixl** "$\otimes\imath$" 70)
  one :: 'a ("$1\imath$")

A Locale for Monoids

locale monoid = struct G +
assumes m_closed [intro, simp]:
"$\llbracket$ x $\in$ carrier G; y $\in$ carrier G $\rrbracket \Longrightarrow$ x $\otimes$ y $\in$ carrier G"
and m_assoc:
"$\llbracket$ x $\in$ carrier G; y $\in$ carrier G; z $\in$ carrier G $\rrbracket$
$\Longrightarrow$ (x $\otimes$ y) $\otimes$ z = x $\otimes$ (y $\otimes$ z)"
and one_closed [intro, simp]: "$\mathbf{1}$ $\in$ carrier G"
and l_one [simp]: "x $\in$ carrier G $\Longrightarrow$ $\mathbf{1}$ $\otimes$ x = x"
and r_one [simp]: "x $\in$ carrier G $\Longrightarrow$ x $\otimes$ $\mathbf{1}$ = x"

A Locale for Groups

A group is a monoid whose elements have inverses.

locale group = monoid +
 assumes inv_ex:
  "$x \in$ carrier G $\Longrightarrow \exists y \in$ carrier G. $y \otimes x = \mathbf{1} \wedge x \otimes y = \mathbf{1}$"

- Reasoning in locale group makes implicit the assumption that *G* is a group.
- Inverse operation is derived, not part of the record.

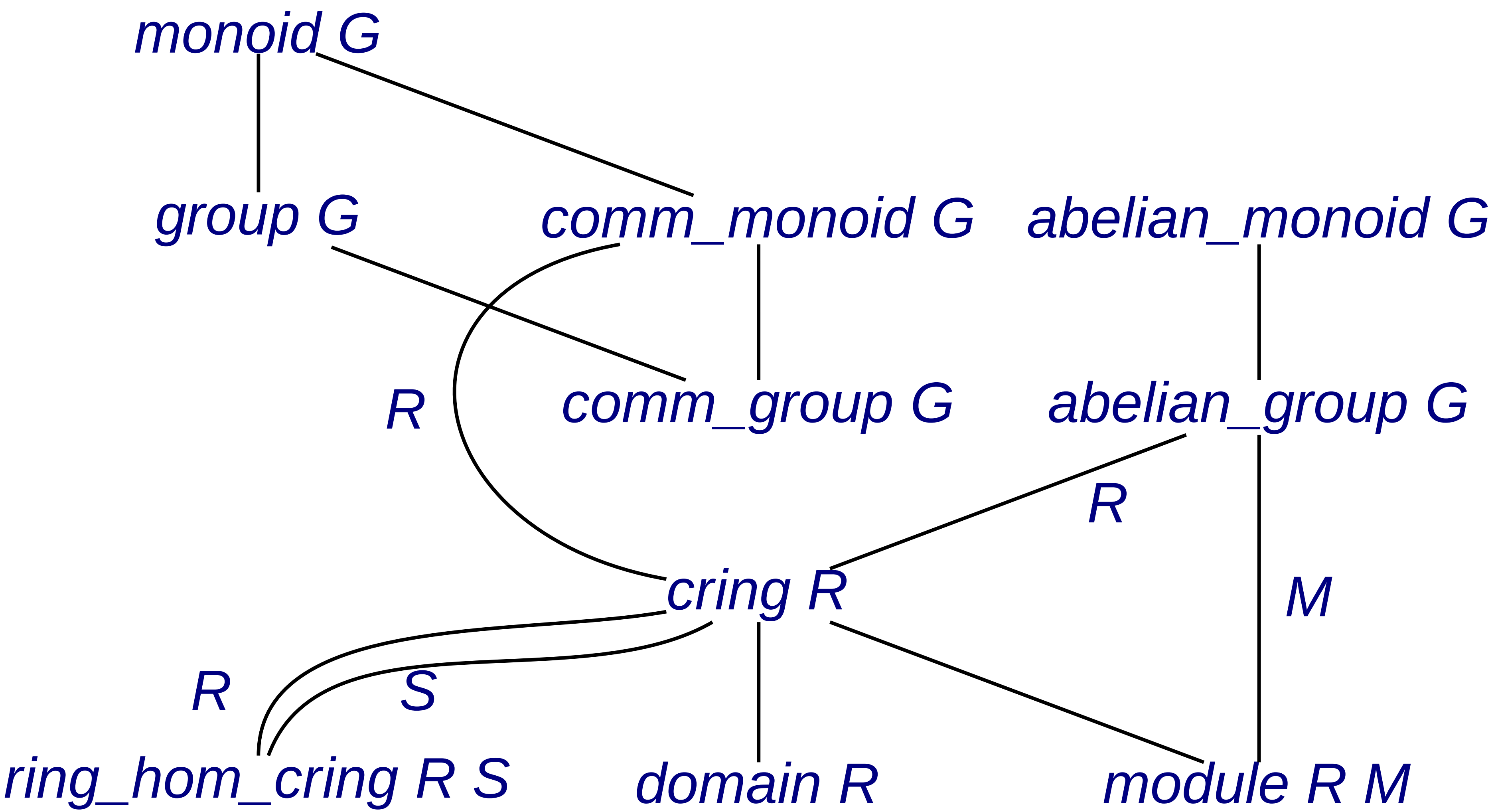
Hierarchy of Structures

```
record 'a ring = "'a monoid" +
  zero :: 'a ("0ı")
  add :: "['a, 'a] ⇒ 'a" (infixl "⊕ı" 65)

record ('a, 'b) module = "'b ring" +
  smult :: "['a, 'b] ⇒ 'b" (infixl "⊙ı" 70)

record ('a, 'p) up_ring = "('a, 'p) module" +
  monom :: "['a, nat] ⇒ 'p"
  coeff :: "['p, nat] ⇒ 'a"
```

Hierarchy of Specifications



Polynomials

Functor *UP* that maps ring structures to polynomial structures.

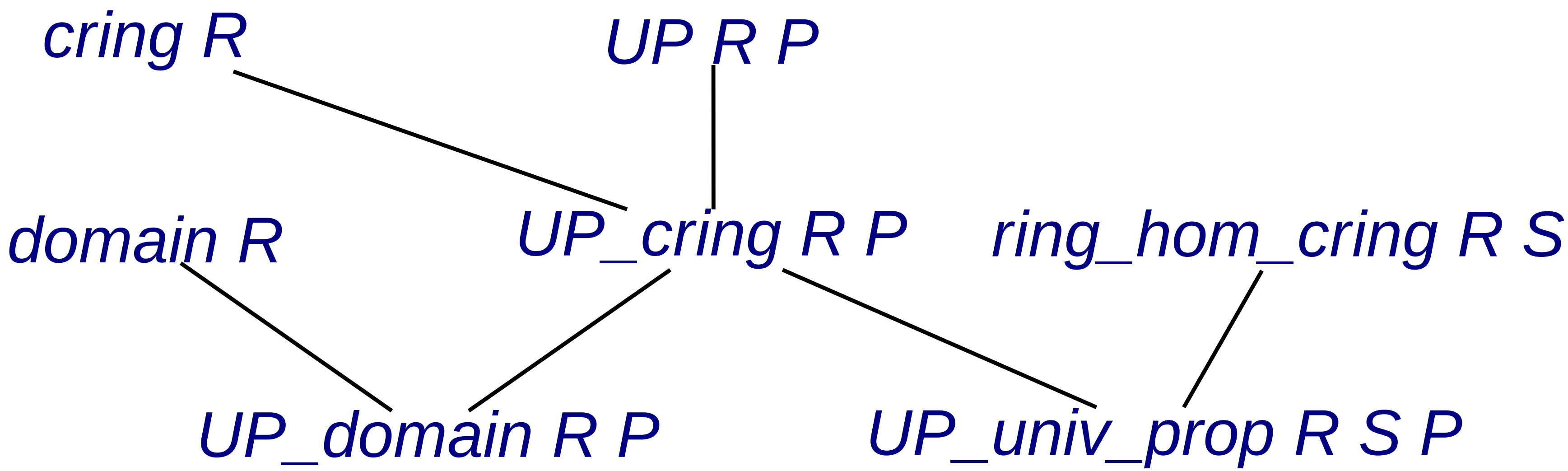
constdefs (structure R)
UP :: "('a, 'm) ring_scheme ⇒ ('a, nat ⇒ 'a) up_ring"
"UP R ≡ (| carrier = up R,
mult = (λp∈up R. λq∈up R. λn. ⨁i ∈ {..n}. p i ⊗ q (n-i)),
one = (λi. if i=0 then 𝟏 else 𝟎),
zero = (λi. 𝟎),
add = (λp∈up R. λq∈up R. λi. p i ⊕ q i),
smult = (λa∈carrier R. λp∈up R. λi. a ⊗ p i),
monom = (λa∈carrier R. λn i. if i=n then a else 𝟎),
coeff = (λp∈up R. λn. p n) |)"

Locales for Polynomials

- Make the polynomial ring a locale parameter

  locale UP = struct R + struct P +
  defines P_def: "P ≡ UP R"

- Add information about base ring



Properties of *UP*

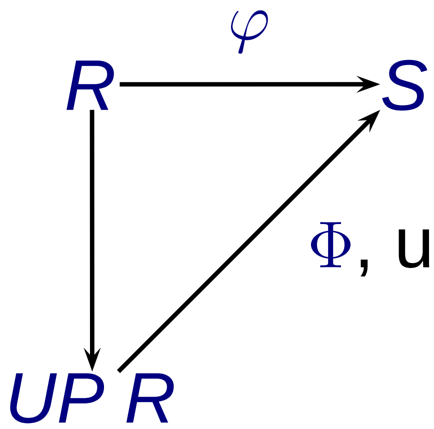
Polynomials over a ring form a ring.

theorem (in UP_cring) UP_cring: "cring P"

Polynomials over an integral domain form a domain.

theorem (in UP_domain) UP_domain: "domain P"

The Universal Property



Φ, unique for $\Phi\, X = s$

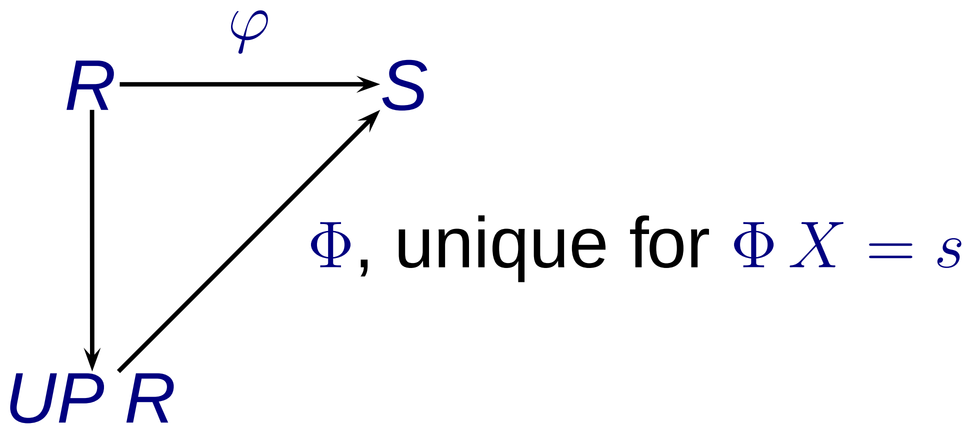
- Existence of Φ:

eval R S phi s $\equiv \lambda$*p* $\in$ *carrier (UP R).*

$\bigoplus$ *i* $\in$ *{..deg R p}. phi (coeff (UP R) p i)* $\otimes$ *s (^) i*

Show that *eval R S phi* is a homomorphism.

The Universal Property



- Uniqueness of Φ:

  Show that two homomorphisms $\Phi, \Psi :$ *UP R* $\rightarrow$ *S* with $\Phi\, X = \Psi\, X = s$ are identical.

Demo: uniqueness

Questions answered by Larry Paulson

Hah! A proof of False
Your axioms are bogus
Go back to square one.

— Larry Paulson