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:

\[ \tau ::= (\tau) \]
\[ | \quad \text{'a} \ | \quad \text{'b} \ | \ldots \quad \text{type variables} \]
\[ | \quad \tau \Rightarrow \tau \quad \text{total functions} \]
\[ | \quad \text{bool} \ | \quad \text{nat} \ | \ldots \quad \text{HOL base types} \]
\[ | \quad \tau \times \tau \quad \text{HOL pairs (ascii: *)} \]
\[ | \quad \tau \ \text{list} \quad \text{HOL lists} \]
\[ | \quad \ldots \quad \text{user-defined types} \]

Parentheses: \[ T1 \Rightarrow T2 \Rightarrow T3 \quad \equiv \quad T1 \Rightarrow (T2 \Rightarrow T3) \]
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)

\[ term ::= (term) \]
\[ \mid a \quad \text{constant or variable (identifier)} \]
\[ \mid \text{term} \text{ term} \quad \text{function application} \]
\[ \mid \lambda x. \text{term} \quad \text{function “abstraction”} \]
\[ \mid \ldots \quad \text{lots of syntactic sugar} \]

Examples:
- \[ f (g x) y \]
- \[ h (\lambda x. f (g x)) \]

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** \(\implies\) (\(===>\))
For separating premises and conclusion of theorems.

**Equality** \(\equiv\) (\(==\))
For definitions.

**Universal quantifier** \(\forall\) (\(!!\))
For parameters in goals.

Do not use *inside* object-logic formulae.
Notation

\[
\begin{bmatrix}
A_1; \ldots; A_n
\end{bmatrix} \Rightarrow B
\]

abbreviates

\[
A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow B
\]

; ≈ “and”
Introducing New Theorems

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

! Axioms should mainly be used when specifying object-logics. !
Definition (non-recursive)

Declaration:

```text
consts
  sq :: nat ⇒ nat
```

Definition:

```text
defs
  sq_def: sq n ≡ n*n
```

Declaration + definition:

```text
constdefs
  sq :: nat ⇒ nat
  sq n ≡ 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. $\land x_1 \ldots x_p. \left[ A_1, \ldots ; A_n \right] \implies B$
2. $\land y_1 \ldots y_q. \left[ C_1, \ldots ; C_n \right] \implies D$

$x_1 \ldots x_p$ Parameters
$A_1 \ldots A_n$ Local assumptions
$B$ Actual (sub)goal
Isabelle Theories
Theory = Source file

Syntax:

\[
\text{theory } MyTh = \text{ImpTh}_1 + \ldots + \text{ImpTh}_n:\n\]
(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:

\[
\text{theory } MyTh = \text{Main}:
\]
X-Symbols

Input of funny symbols in Proof General

- via menu (“X-Symbol”)
- via ascii encoding (similar to \texttt{LATEX}): \texttt{\&\&}, \texttt{\lor}, ...
- via abbreviation: /\, \setminus, \rightarrow, ...

<table>
<thead>
<tr>
<th>x-symbol</th>
<th>\forall</th>
<th>\exists</th>
<th>\lambda</th>
<th>\neg</th>
<th>\land</th>
<th>\lor</th>
<th>\rightarrow</th>
<th>\Rightarrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii (1)</td>
<td>\texttt{\forall}</td>
<td>\texttt{\exists}</td>
<td>\texttt{\lambda}</td>
<td>\texttt{\neg}</td>
<td>\texttt{\land}</td>
<td>\texttt{\lor}</td>
<td>\texttt{\rightarrow}</td>
<td>\texttt{\Rightarrow}</td>
</tr>
<tr>
<td>ascii (2)</td>
<td>\texttt{\textsc{all}}</td>
<td>\texttt{ex}</td>
<td>%</td>
<td>~{}</td>
<td>&amp;</td>
<td>|</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) is converted to x-symbol, (2) stays ascii.
Demo: Isabelle theories
Natural Deduction
Rules

\[
\frac{A \quad B}{A \land B} \quad \text{conjI}
\]

\[
\frac{A \quad B}{A \lor B} \quad \frac{B}{A \lor B} \quad \text{disjI1/2}
\]

\[
\frac{A \to B}{A \to B} \quad \text{impl}
\]

\[
A \land B \quad \frac{[A;B]}{C} \quad \text{conjE}
\]

\[
\frac{A \lor B}{A \quad C \quad B \quad C} \quad \frac{A}{C} \quad \frac{B}{C} \quad \text{disjE}
\]

\[
\frac{A \to B \quad A \quad B \to C}{C} \quad \text{impE}
\]
Proof by assumption

apply assumption

proves

1. \[[ B_1; \ldots ; B_m \] \implies 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 \( \rightarrow \).
  
  \[
  \text{apply}(\text{rule } <\text{intro-rule}>)
  \]

  Applying rule \([ A_1; \ldots ; A_n ] \rightarrow A\) 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 \( \rightarrow \).
  
  \[
  \text{apply}(\text{erule } <\text{elim-rule}>)
  \]

  Like \( \text{rule} \) 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 \( \forall, \exists, \ldots \): ends with ; or \( \supseteq \)

\[
\wedge x y. \left[ \forall y. P y \rightarrow Q z y; Q x y \right] \supseteq \exists x. Q x y
\]

means

\[
\wedge x y. \left[ \forall y_1. P y_1 \rightarrow Q z y_1; Q x y \right] \supseteq (\exists x_1. Q x_1 y)
\]
Natural deduction for quantifiers

\[ \forall x. P x \quad \forall x. P x \quad \text{allI} \]

\[ \forall x. P x \quad P ?x \implies R \quad \text{allE} \]

\[ P ?x \quad \exists x. P x \quad \text{exI} \]

\[ \exists x. P x \quad \forall x. P x \implies R \quad \text{exE} \]

- allI and exE introduce new parameters (\( \forall x \)).
- allE and exI introduce new unknowns (\( ?x \)).
Instantiating rules

\texttt{apply}(\texttt{rule_tac} x = "{term}" \textit{in} \texttt{rule})

Like \textit{rule}, but \(?x\) in \textit{rule} is instantiated by \textit{term} before application.

Similar: \texttt{erule_tac}

\(!x\) \textit{is in} \textit{rule}, \textit{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: $A_1 \implies A$
Subgoal: $1. \left[ B_1; \ldots ; B_n \right] \implies C$

Unifies: one $B_i$ with $A_1$
New subgoal: $1. \left[ B_1; \ldots ; B_n; A \right] \implies C$

apply(drule rulename)

Like frule but also deletes $B_i$
Demo: quantifier proofs
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 \( \sim \) rewrite rule
Example:

Equation: \( 0 + n = n \)

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

Result: \( a + (b + c) \)

Rewrite rules can be conditional: \([P_1 \ldots P_n] \implies l = r\) is used

- like \( l = r \), but
- \( P_1, \ldots, P_n \) must be proved by rewriting first.
Simplification in Isabelle

Goal: \[ \{ P_1; \ldots ; P_m \} \implies C \]

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: \( f(x) = g(x), g(x) = f(x) \)

\[
[P_1 \ldots P_n] \implies l = r
\]

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

\[
n < m \implies (n < \text{Suc } m) = \text{True } \quad \text{YES}
\]

\[
\text{Suc } n < m \implies (n < m) = \text{True } \quad \text{NO}
\]
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
Set trace mode on/off in Proof General:

Isabelle/Isar → Settings → Trace simplifier

Output in separate buffer:

Proof-General → Buffers → 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 $name = \tau$

Introduces an *abbreviation* $name$ for type $\tau$

Examples:

```
types
  name = string
  (\texttt{'}a,\texttt{'}b)foo = "\texttt{'}a list \times \texttt{'}b list"
```

Type abbreviations are expanded after parsing
Not present in internal representation and Isabelle output
datatype 'a list = Nil | Cons 'a "'a list"

Properties:

- **Types:** $\text{Nil} :: \ 'a \ \text{list}$
  
  $\text{Cons} :: \ 'a \mapsto \ 'a \ \text{list} \rightarrow \ 'a \ \text{list}$

- **Distinctness:** $\text{Nil} \neq \text{Cons} \ x \ \text{xs}$

- **Injectivity:** $(\text{Cons} \ x \ \text{xs} = \text{Cons} \ y \ \text{ys}) = (x = y \land \ \text{xs} = \ \text{ys})$
case

Every datatype introduces a case construct, e.g.

\[(\text{case } xs \text{ of } \text{Nil} \Rightarrow \ldots \mid \text{Cons } y \text{ ys} \Rightarrow \ldots y \ldots \text{ys} \ldots)\]

- one case per constructor
- no nested patterns \((\text{Cons } x (\text{Cons } y \text{ zs}))\)
- but nested cases

apply \((\text{case\_tac } xs)\) ⇒ one subgoal for each constructor

\[xs = \text{Nil} \Rightarrow \ldots\]
\[xs = \text{Cons } a \text{ list} \Rightarrow \ldots\]
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
consts app :: "'a list ⇒ 'a list ⇒ '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 \( P \,(\text{Cons} \,x \,xs) \)

Induction theorem \texttt{list.induct}:

\[
\begin{align*}
& P \,\text{Nil}; \lor a \,\text{list.} \,P \,\text{list} \implies P \,(\text{Cons} \,a \,\text{list}) \\
\rightarrow & \rightarrow \ P \,\text{list}
\end{align*}
\]

- General proof method for induction: \texttt{(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 ⇒ ’a list ⇒ ’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

\textbf{lemma} \ \textit{itrev} \ \textit{xs} \ \textit{ys} = \textit{rev} \ \textit{xs} \ \textit{@} \ \textit{ys}

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

\textbf{lemma} \ \forall \ \textit{ys}. \ \textit{itrev} \ \textit{xs} \ \textit{ys} = \textit{rev} \ \textit{xs} \ \textit{@} \ \textit{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


\textbf{recdef — examples}

\textbf{consts} \texttt{sep :: "'a \times \text{ 'a list} \Rightarrow \text{ 'a list}"}

\textbf{recdef} \texttt{sep "measure (\lambda(a, xs). size xs)"}

\texttt{"sep (a, x \# y \# zs) = x \# a \# sep (a, y \# zs)"}
\texttt{"sep (a, xs) = xs"}

\textbf{consts} \texttt{ack :: "nat \times nat \Rightarrow nat"}

\textbf{recdef} \texttt{ack "measure (\lambda m. m) \text{ <\text{*lex*}> measure (\lambda n. n)}"}

\texttt{"ack (0, n) = Suc n"}
\texttt{"ack (Suc m, 0) = ack (m, 1)"}
\texttt{"ack (Suc m, Suc n) = ack (m, ack (Suc m, n))"}
recdef

- The definition:
  - one parameter
  - free pattern matching, order of rules important
  - termination relation
    \((\text{measure} \text{ sufficient for most cases})\)

- 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\}
- \text{insert} :: 'a \Rightarrow 'a set \Rightarrow 'a set
- f ' A \equiv \{y. \exists x \in A. y = f x\}
- \ldots
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:

\begin{verbatim}
consts Ev :: nat set
inductive Ev
intros
  0 ∈ Ev
  n ∈ Ev ⟹  n + 2 ∈ Ev
\end{verbatim}

— The set of all even numbers
Rule induction for Ev

To prove

\[ n \in Ev \implies P\ n \]

by rule induction on \( n \in Ev \) we must prove

- \( P\ 0 \)
- \( P\ n \implies P(n+2) \)

Rule Ev.induct:

\[ \left[ n \in Ev; P\ 0; \bigwedge n. P\ n \implies P(n+2) \right] \implies P\ n \]

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 \( \text{formula}_0 \)

have \( \text{formula}_1 \) by simp

: 

have \( \text{formula}_n \) by blast

show \( \text{formula}_{n+1} \) by \ldots

qed

proves \( \text{formula}_0 \Rightarrow \text{formula}_{n+1} \)
Isar core syntax

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

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

statement = fix variables (∧)
           | assume proposition (⇒)
           | [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:
  \[ \text{from } \neg a \text{ have formula proof} \]

- \[ \text{from } \neg a \text{ have formula proof (rule rule)} \]
  
  \( \neg a \) must prove the first \( n \) premises of \( \text{rule} \)
  in the right order
  the others are left as new subgoals

- \[ \text{proof alone abbreviates proof rule} \]

- \( \text{rule} \): tries elim rules first
  (if there are incoming facts \( \neg a! \)
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

\textit{this} = the previous proposition proved or assumed
\textit{then} = \textit{from} \textit{this}
\textit{with} \overset{\sim}{a} = \textit{from} \overset{\sim}{a} \textit{this}

\textit{thesis} = the last enclosing \textit{show} formula
Mixing proof styles

from . . .

have . . .

apply -
apply (. . .)

:::

apply (. . .)

done

make incoming facts assumptions
Demo: Abbreviations
Predicate Calculus
Syntax:

\texttt{fix variables}

Introduces new arbitrary but fixed variables

(\sim\ parameters)
obtain

Syntax:

\textbf{obtain} \textit{variables \textbf{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 $formula_1$ . . . $formula_n$ into the proof
proof . . .
Demo: moreover/ultimately
General case distinctions

show formula

proof -
  have $P_1 \lor P_2 \lor P_3 \ldots$
  moreover
  { assume $P_1 \ldots$ have $?thesis \ldots$ }
  moreover
  { assume $P_2 \ldots$ have $?thesis \ldots$ }
  moreover
  { assume $P_3 \ldots$ have $?thesis \ldots$ }
  ultimately show $?thesis$ by blast

qed
Chains of equations

- Keywords also and finally.
- \ldots \text{:} predefined schematic term variable, refers to the right hand side of the last expression.
- Uses transitivity rule.
also/finaly

have "t₀ = t₁" ...
also
have "... = t₂" ...
also
... :
also
have "... = tₙ" ...
finally show ...
— pipes fact t₀ = tₙ into the proof
proof :
...
More about also

- Works for all combinations of $=\,$, $\leq\,$ and $<\,$.
- Uses rules declared as \([\text{trans}]\).
- To view all combinations in Proof General: Isabelle/Isar → Show me → Transitivity rules
Demo: also/finally
Locales

Isabelle's Module System
Isar is based on contexts

\textbf{theorem } \bigwedge x. A \Rightarrow C \\
\textbf{proof -} \\
\hspace{1em} \text{fix } x \\
\hspace{2em} \text{assume } \texttt{Ass: } A \\
\hspace{3em} \text{from } \texttt{Ass} \text{ show } C \ldots \\
\textbf{qed} \\
x \text{ and } \texttt{Ass} \text{ are visible inside this context}
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
Locales consist of context elements.

- **fixes**: Parameter, with syntax
- **assumes**: Assumption
- **defines**: Definition
- **notes**: Record a theorem
Declaring locales

locale \textit{loc} = \textit{loc1} + \textit{fixes} \ldots \textit{assumes} \ldots

Declares named locale \textit{loc}.
Declaring locales

Theorems may be stated relative to a named locale.

\begin{verbatim}
lemma (in \textit{loc}) \textit{P} [simp]: \textit{proposition}
proof
\end{verbatim}

- Adds theorem \textit{P} to context \textit{loc}.
- Theorem \textit{P} is in the simpset in context \textit{loc}.
- Exported theorem \textit{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!
Instantiation

Move from abstract to concrete.

\texttt{instantiate \textit{label} : \textit{loc}}

- From chained fact $\textit{loc} \ t_1 \ldots \ t_n$ instantiate locale $\textit{loc}$.
- Imports all theorems of $\textit{loc}$ into current context.
  - Instantiates the parameters with $t_1 \ldots t_n$.
  - Interprets attributes of theorems.
  - Prefixes theorem names with \textit{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

Semantics for binary operations:

\textbf{consts} \ \texttt{semop} :: "binop \Rightarrow nat \Rightarrow nat \Rightarrow nat" ("[\_\_\_\_\_\_\_\_]")

\textbf{primrec} "[Plus] = (\lambda x \ y. x + y)"
"[Minus] = (\lambda x \ y. x - y)"
"[Mult] = (\lambda x \ y. x \ast y)"

Semantics for expressions:

\textbf{consts} \ \texttt{value} :: "'v expr \Rightarrow ('v \Rightarrow nat) \Rightarrow nat"

\textbf{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
consts exec :: "'v instr list ⇒ ('v ⇒ nat) ⇒ nat list ⇒ nat list"

primrec
"exec [] s vs = vs"
"exec (i#is) s vs = (case i of
  Push v ⇒ exec is s (v # vs)
  | Load a ⇒ exec is s (s a # vs)
  | Apply f ⇒ let v₁ = hd vs; v₂ = hd (tl vs); ts = tl (tl vs) in
    exec is s (\[f\] v₁ v₂ # ts))"

- hd and tl are head and tail of lists
The Compiler

Compilation easy:

- **Constants** ⇒ **Push**
- **Variables** ⇒ **Load**
- **Binop** ⇒ **Apply**

const  
\begin{align*}
\text{comp} ::& \quad 'v \text{ expr} \Rightarrow 'v \text{ instr \ list} \\
\text{primrec} \quad & \\
"\text{comp} (\text{Const } v) = [\text{Push } v]" \\
"\text{comp} (\text{Var } a) = [\text{Load } a]" \\
"\text{comp} (\text{Binop } f \ e_1 \ e_2) = (\text{comp } e_2) @ (\text{comp } e_1) @ [\text{Apply } f]"
\end{align*}
Correctness

Executing compiled program yields value of expression

\[ \text{theorem } "exec (\text{comp } e) \ s \ [] = [\text{value } e \ s]" \]

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.
- $I (\langle index\rangle)$ arguments in syntax declarations.
- Extensible **records** (in HOL).
- Locale **instantiation**.
Index Arguments in Syntax Declarations

- One function argument may be $\langle\text{index}\rangle$.
- Works also for infix operators and binders:
  \[
  x \otimes_G y \oplus_R i \in \{0..n\}. f i
  \]
- 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 $\langle\text{index}\rangle$ 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.

```plaintext
record 'a monoid =
  carrier :: "a set"
  mult :: "[a, a] => a" (infixl "\otimes/" 70)
  one :: 'a ("1/")
```
locale monoid = struct G +
  assumes m_closed [intro, simp]: 
    
  and m_assoc: 
    
  and one_closed [intro, simp]: "1 ∈ carrier G"
  and l_one [simp]: "x ∈ carrier G ⟹ 1 ⊗ x = x"
  and r_one [simp]: "x ∈ carrier G ⟹ x ⊗ 1 = x"
A group is a monoid whose elements have inverses.

locale group = monoid +
  assumes inv_ex:
  "x \in \text{carrier } G \Rightarrow \exists y \in \text{carrier } G. y \otimes x = 1 \land x \otimes y = 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 ("0/")
  add :: "['a, 'a] ⇒ 'a" (infixl "⊕/" 65)

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

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

- **monoid G**
  - **group G**
  - **comm_monoid G**: subcategory of monoid G
  - **abelian_monoid G**: subcategory of comm_monoid G

- **cring R**
  - **comm_group G**: subcategory of cring R
  - **abelian_group G**: subcategory of comm_group G

- **ring_hom_cring R S**
  - **domain R**
  - **module R**

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

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

- Add information about base ring
Properties of \textit{UP}

Polynomials over a ring form a ring.
\textbf{theorem} \textbf{(in UP\_cring)} UP\_cring: "cring P"

Polynomials over an integral domain form a domain.
\textbf{theorem} \textbf{(in UP\_domain)} UP\_domain: "domain P"
The Universal Property

\[ R \xrightarrow{\varphi} S \]
\[ \Phi, \text{ unique for } \Phi X = s \]

Existence of \( \Phi \):

\[ \text{eval } R S \phi s \equiv \lambda p \in \text{carrier } (UP R). \]
\[ \bigoplus i \in \{..\deg R p\}. \phi (\text{coeff } (UP R) p i) \otimes s (^) i \]

Show that \( \text{eval } R S \phi \) is a homomorphism.
The Universal Property

Uniqueness of $\Phi$:

Show that two homomorphisms $\Phi, \Psi : UP R \to S$ 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