Introduction to the Isabelle Proof Assistant
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:

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

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: (curried version)

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

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

Parentheses:
\[ f \; a_1 \; a_2 \; a_3 \equiv ((f \; a_1) \; a_2) \; a_3 \]
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 $\rightarrow (\implies)$
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

\[
\left[ A_1 ; \ldots ; A_n \right] \implies B
\]

abbreviates

\[ A_1 \implies \ldots \implies A_n \implies B \]

; \; \approx \; \text{“and”}
Introducing New Theorems

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

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

Declaration:

\textbf{consts}

\texttt{sq :: nat \Rightarrow nat}

Definition:

\textbf{defs}

\texttt{sq\_def: sq n \equiv n^*n}

Declaration + definition:

\textbf{constdefs}

\texttt{sq :: nat \Rightarrow nat}
\texttt{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. \( \land x_1 \ldots x_p. [ A_1; \ldots ; A_n ] \implies B \)
2. \( \land y_1 \ldots y_q. [ C_1; \ldots ; C_n ] \implies D \)

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

Syntax:

\[
\text{theory } \text{MyTh imports } \text{ImpTh}_1 \ldots \text{ImpTh}_n \text{begin (declarations, definitions, theorems, proofs, ...)*}
\]
\[
\text{end}
\]

- \text{MyTh}: name of theory. Must live in file \text{MyTh.thy}
- \text{ImpTh}_i: name of imported theories. Import transitive.

Unless you need something special:

\[
\text{theory } \text{MyTh imports } \text{Main begin}
\]
X-Symbols

Input of funny symbols in Proof General

- via menu (“X-Symbol”)
- via ascii encoding (similar to \( \text{\LaTeX} \)): \(<\text{and}>, \ <\text{or}>, \ldots\)
- via abbreviation: \(\text{//, \ \text{/}, \ ightarrow, \ldots}\)

<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>(&lt;\text{forall}&gt;)</td>
<td>(&lt;\text{exists}&gt;)</td>
<td>(&lt;\text{lambda}&gt;)</td>
<td>(&lt;\text{not}&gt;)</td>
<td>(\text{//})</td>
<td>(\text{/})</td>
<td>(\rightarrow)</td>
<td>(\Rightarrow)</td>
</tr>
<tr>
<td>ascii (2)</td>
<td>ALL</td>
<td>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

\[
\begin{align*}
\frac{A}{A \land B} & \quad \text{conjI} \\
\frac{A \land B}{A \land B} & \quad \text{conjE} \\
\frac{A}{A \lor B} & \quad \text{disjI1/2} \\
\frac{B}{A \lor B} & \quad \text{disjE} \\
\frac{A \rightarrow B}{A \rightarrow B} & \quad \text{impl} \\
\frac{A \rightarrow B}{A \rightarrow B} & \quad \text{implE}
\end{align*}
\]
Proof by assumption

apply assumption

proves

1. $\left[ B_1; \ldots ; B_m \right] \implies C$

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 $\Rightarrow$.

  apply\(\text{(rule <intro-rule>)}\)

  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 $\Rightarrow$.

  apply\(\text{(erule <elim-rule>)}\)

  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: ∀ and ∃
Scope

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

\[
\land x \; y. \quad (\forall y. \; P y \rightarrow Q z \; y; \; Q x \; y) = \Rightarrow \; \exists x. \; Q x \; y
\]

means

\[
\land x \; y. \quad (\forall y_1. \; P y_1 \rightarrow Q z \; y_1; \; Q x \; y) = \Rightarrow \; (\exists x_1. \; Q x_1 \; y)
\]
Natural deduction for quantifiers

\[ \forall x. P(x) \]

1. **allI**: \[ \frac{\forall x. P(x)}{\forall x. P(x)} \]

2. **allE**: \[ \frac{\forall x. P(x) \quad P(?x) \implies R}{R} \]

\[ \exists x. P(x) \]

3. **exI**: \[ \frac{P(?x)}{\exists x. P(x)} \]

4. **exE**: \[ \frac{\exists x. P(x) \quad \forall x. P(x) \implies R}{R} \]

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

\text{apply}(\text{rule\_tac \ x = \"term\" in rule})

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

Similar: \text{erule\_tac}

\text{\textbf{!}} \quad x \text{ is in } \text{rule}, \text{ not in the goal} \quad \text{\textbf{!}}
Safe and unsafe rules

Safe  allI, exE
Unsafe allE, exl

Create parameters first, unknowns later
Forward proofs: frule and drule

\textbf{apply(frule rulename)}

Forward rule: \( A_1 \implies A \)
Subgoal: \( 1. [ B_1; \ldots ; B_n ] \implies C \)

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

\textbf{apply(drule rulename)}

Like \textit{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: $[P_1 \ldots P_n] \Rightarrow l = r$

is used

► like $l = r$, but

► $P_1, \ldots, P_n$ must be proved by rewriting first.
Simplification in Isabelle

Goal: 1. \[ [ P_1; \ldots ; P_m ] \implies C \]

\textbf{apply}(simp add: eq_1 \ldots eq_n)

Simplify \( P_1 \ldots P_m \) and \( C \) using

- lemmas with attribute \textit{simp}
- additional lemmas \( eq_1 \ldots eq_n \)
- assumptions \( P_1 \ldots P_m \)

Variations:

- \textit{(simp} \ldots \textit{del}: \ldots \textit{)} removes \textit{simp}-lemmas
- \textit{add} and \textit{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\((\text{simp (no\_asm\_simp)} \ldots)\)
Simplify only conclusion

apply\((\text{simp (no\_asm\_use)} \ldots)\)
Simplify but do not use assumptions

apply\((\text{simp (no\_asm)} \ldots)\)
Ignore assumptions completely
Tracing

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:

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

```isar
types name = τ
```

Introduces an *abbreviation* `name` for type `τ`

Examples:

```isar
types
  name = string
  ('a,'b)foo = "'a list × '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: Nil :: 'a list

▸ Distinctness: Nil ≠ Cons x xs

▸ Injectivity: (Cons x xs = Cons y ys) = (x = y ∧ xs = ys)
Every datatype introduces a case construct, e.g.

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

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

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

\[xs = Nil \quad \Rightarrow \quad \ldots\]
\[xs = Cons \ a \ list \quad \Rightarrow \quad \ldots\]
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 list.induct:

\[
\left[ P \, \text{Nil}; \land a \text{ list.} \, P \, \text{list} \implies P \, (\text{Cons} \, a \, \text{list}) \right] \implies P \, \text{list}
\]

- General proof method for induction: \((\text{induct} \, x)\)
  - \( x \) 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 \textit{itrev} :: 'a list ⇒ 'a list ⇒ 'a list
primrec
  \textit{itrev} [] ys = ys
  \textit{itrev} (x#xs) ys = \textit{itrev} xs (x#ys)

lemma \textit{itrev} xs [] = \text{rev} xs
Demo: proof attempt
Generalisation

Replace constants by variables

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

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

\textbf{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
consts sep :: "'a × 'a list ⇒ '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 × nat ⇒ 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 definition:
- 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\}
- \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:

```isabelle
consts Ev :: nat set  
inductive Ev  
intros  
  0 ∈ Ev  
  \( n ∈ Ev \implies n + 2 ∈ Ev \)
```

---

IJCAR 2004, Tutorial T4 -- p.64
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** \( Ev\text{.induct} \):

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

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

proof = \text{proof} \ [\text{method}] \ \text{statement}^* \ \text{qed} \\
\quad \mid \ \text{by} \ \text{method}

method = (\text{simp} \ldots) \mid (\text{blast} \ldots) \mid (\text{rule} \ldots) \mid \ldots

statement = \text{fix} \ \text{variables} \ (\land) \\
\quad \mid \ \text{assume} \ \text{proposition} \ (\implies) \\
\quad \mid \ [\text{from} \ \text{name}^+] \ (\text{have} \mid \text{show}) \ \text{proposition} \ \text{proof} \\
\quad \mid \ \text{next} \ (\text{separates subgoals})

proposition = \ [\text{name:}] \ \text{formula}
Demo: propositional logic
Elim rules / forward reasoning

- Elim rules are triggered by facts fed into a proof:
  \[ \text{from } \vec{a} \text{ have } \text{formula } \text{proof} \]

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

- \(\text{proof}\) alone abbreviates \(\text{proof } \text{rule}\)

- \(\text{rule}\): 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

\[\begin{align*}
\textit{this} &= \text{the previous proposition proved or assumed} \\
\text{then} &= \text{from this} \\
\text{with } \vec{a} &= \text{from } \vec{a} \text{ this} \\
\textit{thesis} &= \text{the last enclosing show formula}
\end{align*}\]
Mixing proof styles

from . . .

have . . .

apply - make incoming facts assumptions

apply( . . . )

: 

apply( . . . )

done
Demo: Abbreviations
Predicate Calculus
Syntax:

\texttt{fix \ variables}

Introduces new arbitrary but fixed variables \((\sim \ \text{parameters})\)
Syntax:

\texttt{obtain variables where proposition proof}

Introduces new variables together with property
Demo: predicate calculus
Moreover/ultimately

have $f_{\text{ormula}_1} \ldots $  
moreover

have $f_{\text{ormula}_2} \ldots $  
moreover

:  
moreover

have $f_{\text{ormula}_n} \ldots $  
ultimately

show $\ldots $  
— pipes facts $f_{\text{ormula}_1} \ldots f_{\text{ormula}_n}$ into the proof  
proof $\ldots $
Demo: moreover/ultimately
General case distinctions

show \textit{formula}

proof -

have $P_1 \lor P_2 \lor P_3 \ldots$

moreover

\{ assume $P_1 \ldots$ have \textit{thesis} \ldots \}\n
moreover

\{ assume $P_2 \ldots$ have \textit{thesis} \ldots \}\n
moreover

\{ assume $P_3 \ldots$ have \textit{thesis} \ldots \}\n
ultimately show \textit{thesis} by \textit{blast}

qed
Chains of equations

- Keywords **also** and **finally**.
- …: 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 
have "... = \( t_2 \)" ... 
also 
: 
also 
have "... = \( t_n \)" ... 
finally show ... 
— pipes fact \( t_0 = t_n \) into the proof 
proof 
: 

\[ t_0 = t_1 \]
\[ t_0 = t_2 \]
\[ \vdots \]
\[ t_0 = t_{n-1} \]

IJCAR 2004, Tutorial T4 -- p.87
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

\[ \forall x. A \implies C \]

proof -

fix \( x \)

assume \( \text{Ass} : A \)

\( : \) \( x \) and \( \text{Ass} \) are visible

from \( \text{Ass} \) show \( C \) . . . inside this context

qed
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} + \text{Import}
    \text{fixes} . . . \text{Context elements}
    \text{assumes} . . .

Declares named locale \textit{loc}.
Declaring locales

Theorems may be stated relative to a named locale.

\textbf{lemma} (in \textit{loc}) \textit{P} [simp]: \textit{proposition}

\textit{proof}

- 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!
Interpretation

Move from abstract to concrete.

\texttt{interpret label : loc [t_1 \ldots t_n] proof}

- Interpret \textit{loc} with parameters \( t_1 \ldots t_n \)
- Generates proof obligation.
- 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
  ```plaintext
datatype binop = Plus | Minus | Mult
  ```

- Expressions
  ```plaintext
datatype 'v expr = Const nat
        | Var 'v
        | Binop binop "'v expr" "'v expr"
  ```

- 'v = variable names
Expressions — Semantics

Semantics for binary operations:

```plaintext
consts semop :: "binop ⇒ nat ⇒ nat ⇒ nat" ("[ ]")
primrec "Plus] = (λx y. x + y)"
"Minus] = (λx y. x - y)"
"Mult] = (λx y. x * y)"
```

Semantics for expressions:

```plaintext
consts "value" :: "'v expr ⇒ ('v ⇒ nat) ⇒ nat"
primrec
"value (Const v) E = v"
"value (Var a) E = E a"
"value (Binop f e₁ e₂) E = [f] (value e₁ E) (value e₂ 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

```plaintext
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 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
Compilation easy:

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

```
consts comp :: "'v expr ⇒ 'v instr list"
primrec
"comp (Const v) = [Push v]"
"comp (Var a) = [Load a]"
"comp (Binop f e₁ e₂) = (comp e₂) @ (comp e₁) @ [Apply f]"
```
Correctness

Executing compiled program yields value of expression

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

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.
- \( l (\langle index\rangle) \) 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:
  \( x \otimes_G y \uplus_R i \in \{0..n\}. f\ i \)
- 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.

```plaintext
record 'a monoid =
carrier :: "'a set"
mult :: "['a, 'a] ⇒ 'a" (infixl "⊗" 70)
one :: 'a ("1")
```
locale monoid = struct G +

assumes m_closed [intro, simp]:
"[ x ∈ carrier G; y ∈ carrier G ] ⇒ x ⊗ y ∈ carrier G"

and m_assoc:
"[ x ∈ carrier G; y ∈ carrier G; z ∈ carrier G ]
⇒ (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"

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.

```agda
definition group = monoid +
  assumes inv_ex:
  "x ∈ carrier G =⇒ ∃ y ∈ carrier G. y ⊗ x = 1 ∧ x ⊗ y = 1"
```

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

record 'a ring = "'a monoid" +
  zero :: 'a ("0/"")
  add :: "[ 'a, 'a ] \rightarrow 'a" (infixl "\oplus /" 65)

record ('a, 'b) module = "'b ring" +
  smult :: "[ 'a, 'b ] \rightarrow 'b" (infixl "\odot /" 70)

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

monoid $G$ 

comm_monoid $G$  

abelian_monoid $G$

comm_group $G$  

abelian_group $G$

cring $R$ 

ring_hom_cring $R$ $S$  

domain $R$  

module $R$ $M$
Polynomials

Functor \( \text{UP} \) that maps ring structures to polynomial structures.

\[
\text{constdefs} \ (\text{structure } R) \\
\text{UP} :: "('a, 'm) ring_scheme \Rightarrow ('a, nat \Rightarrow 'a) up_ring" \\
"\text{UP } R \equiv \{ | \text{carrier } = \text{up } R, \\
mult = (\lambda p \in \text{up } R. \lambda q \in \text{up } R. \lambda n. \bigoplus i \in \{..n\}. p \ i \times q \ (n-i)), \\
one = (\lambda i. \text{if } i=0 \text{ then } 1 \text{ else } 0), \\
zero = (\lambda i. 0), \\
add = (\lambda p \in \text{up } R. \lambda q \in \text{up } R. \lambda i. p \ i \oplus q \ i), \\
\text{smult} = (\lambda a \in \text{carrier } R. \lambda p \in \text{up } R. \lambda i. a \otimes p \ i), \\
\text{monom} = (\lambda a \in \text{carrier } R. \lambda n i. \text{if } i=n \text{ then } a \text{ else } 0), \\
\text{coeff} = (\lambda p \in \text{up } R. \lambda n. p \ n) | \}
\]
Locales for Polynomials

- Make the polynomial ring a locale parameter

\[
\text{locale } \text{UP} = \text{struct } \text{R} + \text{struct } \text{P} + \\
\text{defines } \text{P_def}: "\text{P} \equiv \text{UP R}"
\]

- Add information about base ring

\[
\text{cring } \text{R} \quad \text{UP } \text{R} \quad \text{P} \\
\text{domain } \text{R} \quad \text{UP_cring } \text{R} \quad \text{P} \quad \text{ring_hom_cring } \text{R} \quad \text{S} \\
\text{UP_domain } \text{R} \quad \text{P} \quad \text{UP_univ_prop } \text{R} \quad \text{S} \quad \text{P}
\]
Properties of $UP$

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

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

![Diagram]

Existence of $\Phi$:

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

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

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

Uniqueness of $\Phi$:

Show that two homomorphisms $\Phi, \Psi : \text{UP } R \rightarrow S$ 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