Session IV

Case Studies
Case Study
Compiling Expressions
The Task

- develop a compiler
The Task

- develop a compiler
- from expressions
The Task

- develop a compiler
- from expressions
- to a stack machine
The Task

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

Syntax for
  ▶ binary operations
  ▶ expressions

Design decision:
  ▶ no syntax for variables and values

Instead:
  ▶ expressions generic in variable names,
  ▶ \textit{nat} for values.
Expressions — Data Type

- Binary operations

```plaintext
datatype binop = Plus | Minus | Mult
```
Expressions — Data Type

- Binary operations

```haskell
datatype binop = Plus | Minus | Mult
```

- Expressions

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

- 'v = variable names
Semantics for binary operations:

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

\textbf{primrec} "[\textit{Plus}] = (\lambda x\, y. \ x + \ y)"
"[\textit{Minus}] = (\lambda x\, y. \ x - \ y)"
"[\textit{Mult}] = (\lambda x\, y. \ x * \ y)"
Expressions — Semantics

Semantics for binary operations:

```ml
consts semop :: "binop ⇒ nat ⇒ nat ⇒ nat" ("\[\_\] ")
primrec "\[Plus\] = (\lambda x y. x + y)"
          "\[Minus\] = (\lambda x y. x - y)"
          "\[Mult\] = (\lambda x y. x * y)"
```

Semantics for expressions:

```ml
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
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** 

\[ 'v \, instr = \text{Push} \, nat \]
\[ | \, \text{Load} \, 'v \]
\[ | \, \text{Apply} \, \text{binop} \]
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*
The Compiler

Compilation easy:

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

**consts** \( \text{comp} :: "'v expr \Rightarrow 'v instr list" \)

**primrec**

"\( \text{comp} (\text{Const } v) = [\text{Push } v] \)"

"\( \text{comp} (\text{Var } a) = [\text{Load } a] \)"

"\( \text{comp} (\text{Binop } f \ e_1 \ e_2) = (\text{comp } e_2) @ (\text{comp } e_1) @ [\text{Apply } f] \)"
Correctness

Executing compiled program yields value of expression
Correctness

Executing compiled program yields value of expression

\textbf{theorem} \ "\texttt{exec (comp e) s [] = [value e s]}"
Correctness

Executing compiled program yields value of expression

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

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.
Abstract Mathematics

- Concerns \textit{classes} of objects specified by axioms, not concrete objects like the integers or reals.

- Objects are typically \textit{structures}: $(G, \cdot, 1, -1)$
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
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.
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 are not theories of proof tools.
Structures must be *first-class values*. 
Formalisation

- Structures are not theories of proof tools.
- Structures must be first-class values.
- Syntax should reflect context:
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 $G$. 
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.
Support for Abstraction

- **Locales**: portable contexts.
- \( I(\langle\text{index}\rangle) \) arguments in syntax declarations.
Support for Abstraction

- **Locales**: portable contexts.
- I (\index) arguments in syntax declarations.
- Extensible **records** (in HOL).
Support for Abstraction

- **Locales**: portable contexts.
- I ($\langle \text{index} \rangle$) arguments in syntax declarations.
- Extensible **records** (in HOL).
- Locale **instantiation**.
Index Arguments in Syntax Declarations

- One function argument may be $\langle \text{index} \rangle$. 
Index Arguments in Syntax Declarations

- One function argument may be $\langle\text{index}\rangle$.
- Works also for infix operators and binders:
  $x \otimes_G y \uplus_{R} i \in \{0..n\}. f i$
One function argument may be \(<\text{index}\>\).

Works also for infix operators and binders:

\[
x \otimes_G y \quad \bigoplus_R i \in \{0..n\}. \quad f_i
\]

Good for denoting record fields.
Index Arguments in Syntax Declarations

- One function argument may be \(<\text{index}>\).
- 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 (\text{structure}).
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 \texttt{(structure)}.
- Yields a concise syntax for $G$ while allowing references to other groups.
One function argument may be \(<\text{index}>\).

Works also for infix operators and binders:
\[
x \otimes_G y \bigoplus_R i \in \{0..n\}. f_i
\]

Good for denoting record fields.

Can declare default by \texttt{(structure)}.

Yields a concise syntax for \texttt{G} while allowing references to other groups.

Letter subscripts for \(<\text{index}>\) only available in current development version of Isabelle.
Records

- Are used to represent structures.
Records

- Are used to represent structures.
- Fields are functions and can have special syntax.
Records

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

```plaintext
locale group = monoid +
assumes inv_ex:
"x ∈ carrier G ⟹ ∃ y ∈ carrier G. y ⊗ x = 1 ∧ x ⊗ y = 1"
```
A **group** is a monoid whose elements have inverses.

**locale group = monoid +**

*assumes inv_ex:*

"\(x \in \text{carrier } G \implies \exists \, y \in \text{carrier } G. \, y \otimes x = 1 \land x \otimes y = 1\)"

► Reasoning in locale group makes implicit the assumption that **G** is a group.
A Locale for Groups

A **group** is a monoid whose elements have inverses.

**locale** group = monoid +
**assumes** inv_ex:
"\( x \in \text{carrier } G \implies \exists y \in \text{carrier } G. y \otimes x = 1 \land x \otimes y = 1 \)"

► 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)
Hierarchy of Structures

```haskell
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)
```
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
Polynomials

Functor $UP$ that maps ring structures to polynomial structures.
Polynomials

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

constdefs (structure $R$)
$\text{UP} :: \text{"('}a, 'm\text{) ring_scheme} \Rightarrow \text{('}a, \text{nat} \Rightarrow \text{'a)} \text{ up_ring}"

"UP R \equiv ( \mid \text{carrier} = \text{up R},$
\begin{align*}
\text{mult} &= (\lambda p \in \text{up R}. \lambda q \in \text{up R}. \lambda n. \bigoplus i \in \{..n\}. p i \otimes q (n-i)), \\
\text{one} &= (\lambda i. \text{if } i=0 \text{ then } 1 \text{ else } 0), \\
\text{zero} &= (\lambda i. 0), \\
\text{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

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

```
cring R  UP R P

domain R  UP_cring R P  ring_hom_cring R S

UP_domain R P  UP_univ_prop R S P
```
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

\[
\begin{array}{ccc}
R & \xrightarrow{\varphi} & S \\
\downarrow{\Phi} & & \downarrow{\Phi} \\
UP R & & \end{array}
\]
The Universal Property

\[
\begin{align*}
R & \xrightarrow{\varphi} S \\
\downarrow \Phi & \\
UP R &
\end{align*}
\]

\(\Phi\), unique for \(\Phi X = s\)
The Universal Property

$\varphi : R \rightarrow S$

$\Phi$, unique for $\Phi X = s$

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

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

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

$\varphi$

$R \xrightarrow{\Phi} S$

$\Phi$, unique for $\Phi X = s$

$UP R$

- Uniqueness of $\Phi$:

Show that two homomorphisms $\Phi, \Psi : UP R \to S$ with $\Phi X = \Psi X = s$ are identical.
The Universal Property

Uniqueness of $\Phi$:

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