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,
- $nat$ for values.
Expressions — Data Type

- Binary operations

\[
\text{datatype } \textit{binop} = \textit{Plus} \mid \textit{Minus} \mid \textit{Mult}
\]
Expressions — Data Type

- Binary operations
  
  \[\text{datatype} \ \text{binop} = \text{Plus} \mid \text{Minus} \mid \text{Mult}\]

- Expressions
  
  \[\text{datatype} \ \text{'v expr} = \text{Const} \ \text{nat} \mid \text{Var} \ \text{'v} \mid \text{Binop} \ \text{binop} "'v expr" "'v expr"\]

- \text{'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)"
```

IJCAR 2004, Tutorial T4 – p.6
Expressions — Semantics

Semantics for binary operations:

```ml
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:

```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 = 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**
Compilation easy:

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

```plaintext
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
Correctness

Executing compiled program yields value of expression

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

Executing compiled program yields value of expression

\textbf{theorem} \quad \textquote{exec (comp e) s [] = [value e s]}

\textbf{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.
Concerns **classes** of objects specified by axioms, not concrete objects like the integers or reals.

Objects are typically **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**.
Structures are not theories of proof tools.

Syntax should reflect context: If $G$ is a group, then $(x^{-1}y^{-1}) = y^{-1}x^{-1}$ refers implicitly to $G$.

Inheritance of syntax and theorems should be automatic.
Formalisation

- Structures are not theories of proof tools.
- Structures must be *first-class values*. 

Syntax should reflect context:

If $G$ is a group, then $(x^{-1}y)^{-1} = y^{-1}x^{-1}$ refers implicitly to $G$. 

Inheritance of syntax and theorems should be automatic.
Formalisation

- Structures are not theories of proof tools.
- Structures must be first-class values.
- Syntax should reflect context:
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 $G$. 
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.
- \( l (\langle \text{index} \rangle) \) arguments in syntax declarations.
Support for Abstraction

- **Locales**: portable contexts.
- \( <\text{index}> \) arguments in syntax declarations.
- Extensible **records** (in HOL).
Support for Abstraction

- **Locales**: portable contexts.
- $1 (\langle{\text{index}}\rangle)$ arguments in syntax declarations.
- Extensible **records** (in HOL).
- **Locale instantiation**.
Index Arguments in Syntax Declarations

- One function argument may be \langle index\rangle.

Letter subscripts for \text{n}<index> only available in current development version of Isabelle.
One function argument may be $\langle\text{index}\rangle$.

Works also for infix operators and binders:

$$x \otimes_G y \quad \bigoplus_R i \in \{0..n\}. f i$$
Index Arguments in Syntax Declarations

- One function argument may be \(<\text{index}>\).
- Works also for infix operators and binders:
  \[ x \otimes_G y \quad \bigoplus_R i \in \{0..n\}. f i \]
- Good for denoting record fields.
Index Arguments in Syntax Declarations

- One function argument may be $\langle\text{index}\rangle$.
- Works also for infix operators and binders:
  \[ x \otimes_G y \quad \bigoplus_R i \in \{0..n\}. f i \]
- Good for denoting record fields.
- Can declare default by (structure).
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 \((\text{structure})\).
- Yields a concise syntax for \(G\) while allowing references to other groups.
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 (structure).
- Yields a concise syntax for \(G\) 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/")
```
A Locale for Monoids

locale monoid = struct G +
  assumes m_closed [intro, simp]:
    "[ x \in carrier G; y \in carrier G ] \Rightarrow x \otimes y \in carrier G"
  and m_assoc:
    "[ x \in carrier G; y \in carrier G; z \in carrier G ]
    \Rightarrow (x \otimes y) \otimes z = x \otimes (y \otimes z)"
  and one_closed [intro, simp]: "1 \in carrier G"
  and l_one [simp]: "x \in carrier G \Rightarrow 1 \otimes x = x"
  and r_one [simp]: "x \in carrier G \Rightarrow x \otimes 1 = x"
A group is a monoid whose elements have inverses.

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

A Locale for Groups

A group is a monoid whose elements have inverses.

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

- Reasoning in locale group makes implicit the assumption that \( G \) is a group.
A group is a monoid whose elements have inverses.

locale 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.
Hierarchy of Structures

record 'a ring = "'a monoid" +
    zero :: 'a ("0/")
    add :: "[a, a] ⇒ a" (infixl "⊕/" 65)
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)
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 $G$
- $comm\_monoid\ G$
- $abelian\_monoid\ G$

- $cring\ R$
- $comm\_group\ G$
- $abelian\_group\ G$

- $ring\_hom\_cring\ R\ S$
- domain $R$
- module $R\ M$

$IJCAR$ 2004, Tutorial T4 – p.22
Functor $UP$ that maps ring structures to polynomial structures.
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

```lean
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 \textit{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

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

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

\[ \begin{aligned} R & \xrightarrow{\varphi} S \\ UP R & \xrightarrow{\Phi} R \end{aligned} \]

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

 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 $\Phi, \Psi : UP R \rightarrow S$ with $\Phi X = \Psi X = s$ are identical.
The Universal Property

Uniqueness of $\Phi$:

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