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

  datatype *binop* = *Plus* | *Minus* | *Mult*

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

"$[\![Minus]\!] = (\lambda x\ y.\ x - y)$*"*

"$[\![Mult]\!] = (\lambda x\ y.\ x * y)$*"*

Expressions — Semantics

- Sematics for binary operations:

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

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

"$[\![$*Minus*$]\!]$ *= (*λ*x y. x - y)"*

"$[\![$*Mult*$]\!]$ *= (*λ*x 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

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*

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

Correctness

Executing compiled program yields value of expression

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

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.

Abstract Mathematics

- Concerns classes of objects specified by axioms, not concrete objects like the integers or reals.
- Objects are typically structures: $(G, \cdot, 1, {}^{-1})$

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.

Formalisation

- 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:

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.
- ı (`\<index>`) arguments in syntax declarations.

Support for Abstraction

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

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>`.

Index Arguments in Syntax Declarations

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

  $x \otimes_{\mathtt{G}} y \qquad \bigoplus_{\mathtt{R}} i \in \{0..n\}.\ f\ i$

Index Arguments in Syntax Declarations

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

  $x \otimes_{\mathtt{G}} y \qquad \bigoplus_{\mathtt{R}} i \in \{0..n\}.\ f\ i$

- Good for denoting record fields.

Index Arguments in Syntax Declarations

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

  $x \otimes_{\mathtt{G}} y \qquad \bigoplus_{\mathtt{R}} i \in \{0..n\}.\ f\ i$
- Good for denoting record fields.
- Can declare default by **(structure)**.

Index Arguments in Syntax Declarations

- One function argument may be `\<index>`.
- Works also for infix operators and binders:
  $x \otimes_{\mathtt{G}} y \qquad \bigoplus_{\mathtt{R}} i \in \{0..n\}.\ f\ i$
- Good for denoting record fields.
- Can declare default by **(structure)**.
- Yields a concise syntax for G while allowing references to other groups.

Index Arguments in Syntax Declarations

- One function argument may be `\<index>`.
- Works also for infix operators and binders:
  $x \otimes_{\mathtt{G}} y \qquad \bigoplus_{\mathtt{R}} i \in \{0..n\}.\ f\ i$
- 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.

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.

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]:
    "⟦ 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 Locale for Groups

A group is a monoid whose elements have inverses.

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

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. $y \otimes x = \mathbf{1} \land x \otimes y = \mathbf{1}$"

- 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:
    "$x \in$ carrier G $\Longrightarrow \exists y \in$ carrier G. $y \otimes x = 1 \wedge x \otimes y = 1$"

- 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 ("$\mathbf{0}$*ı*")
  add :: "['a, 'a] $\Rightarrow$ 'a" (**infixl** "$\oplus$*ı*" 65)

Hierarchy of Structures

record 'a ring = "'a monoid" +
  zero :: 'a ("$\mathbf{0}$*ı*")
  add :: "['a, 'a] $\Rightarrow$ 'a" (**infixl** "$\oplus$*ı*" 65)

record ('a, 'b) module = "'b ring" +
  smult :: "['a, 'b] $\Rightarrow$ 'b" (**infixl** "$\odot$*ı*" 70)

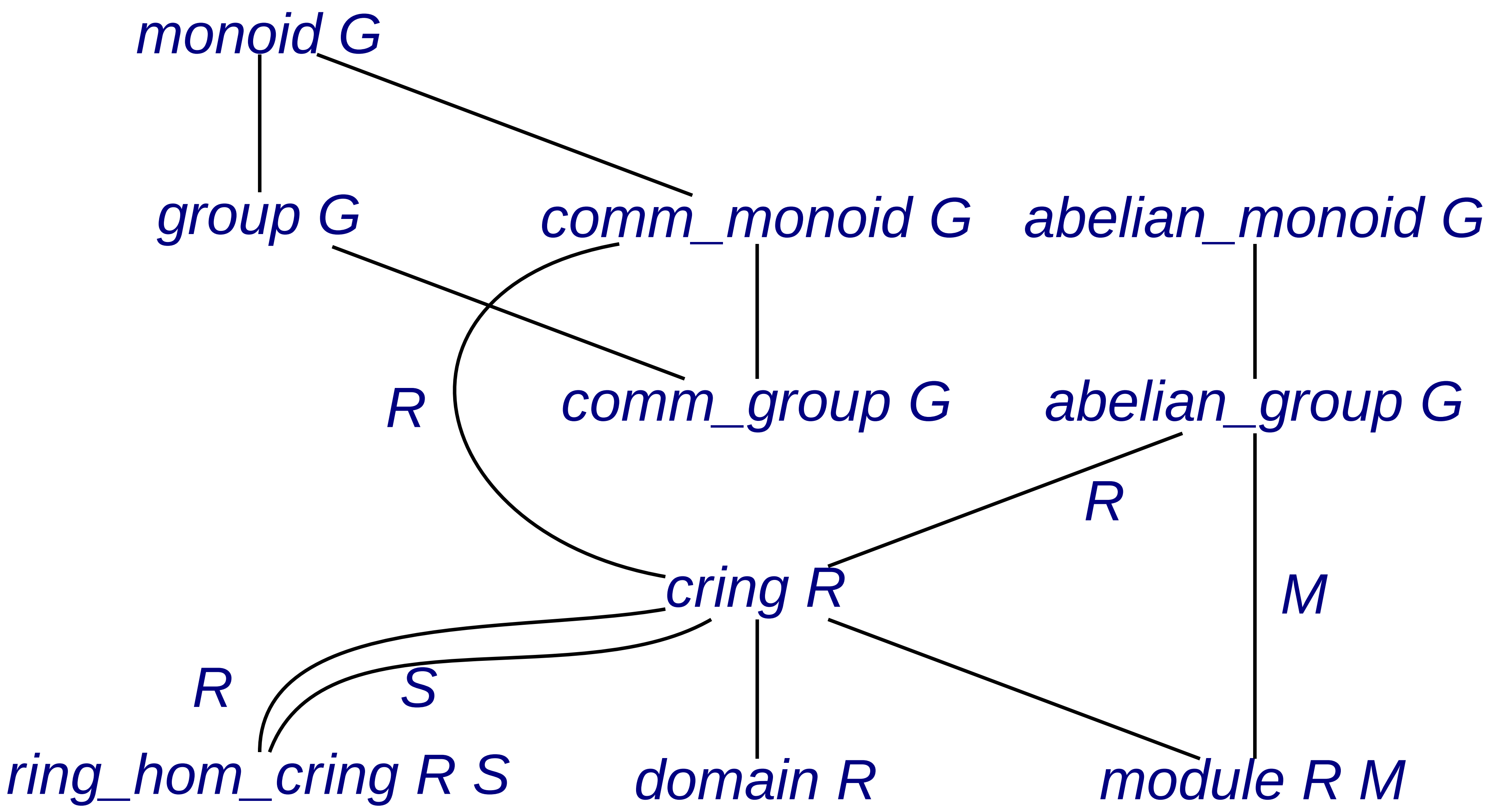
Hierarchy of Structures

```
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.

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

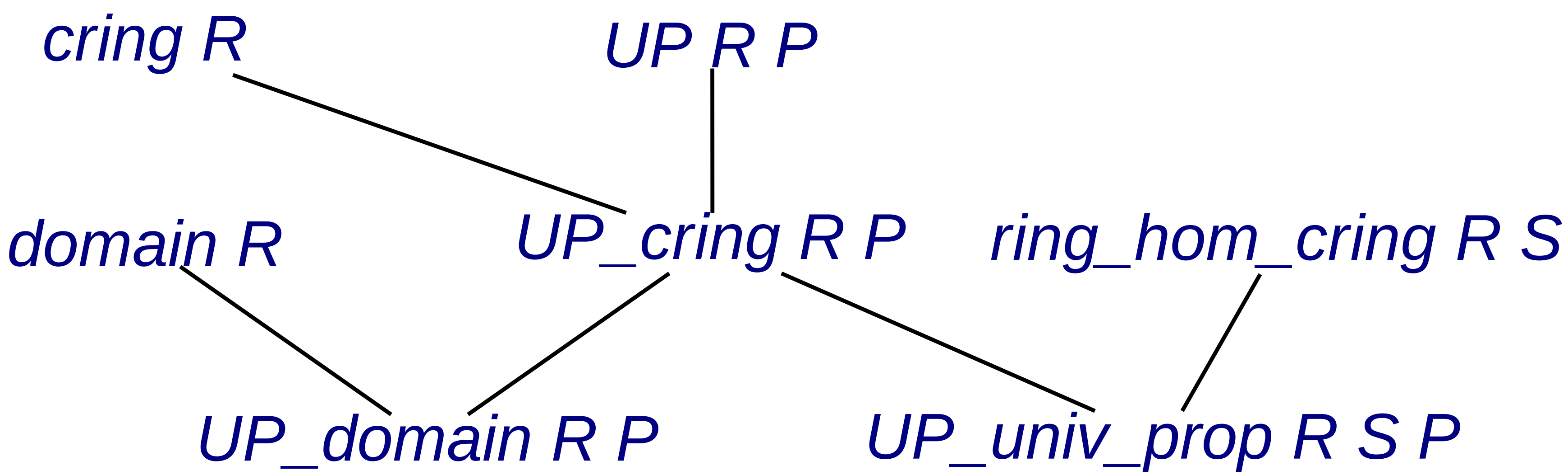
locale UP = struct R + struct P +
  defines P_def: "P $\equiv$ UP R"

Locales for Polynomials

- Make the polynomial ring a locale parameter

locale UP = struct R + struct P +
  defines P_def: "P $\equiv$ 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

$$\begin{tikzcd} R \arrow[r, "\varphi"] \arrow[d] & S \\ UP\,R \arrow[ur, "\Phi"'] & \end{tikzcd}$$

The Universal Property

$$
\begin{tikzcd}
R \arrow[r, "\varphi"] \arrow[d] & S \\
UP\,R \arrow[ur, "{\Phi\text{, unique for } \Phi\, X = s}"'] &
\end{tikzcd}
$$

The Universal Property

$$\begin{tikzcd}
R \arrow[r, "\varphi"] \arrow[d] & S \\
UP\ R \arrow[ur, "{\Phi\text{, unique for } \Phi\, X = s}"'] &
\end{tikzcd}$$

- 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

$$\begin{tikzcd} R \arrow[r, "\varphi"] \arrow[d] & S \\ \mathit{UP}\ R \arrow[ur, "\Phi\text{, unique for } \Phi\, X = s"'] & \end{tikzcd}$$

- Uniqueness of Φ:

  Show that two homomorphisms $\Phi, \Psi : \mathit{UP}\ R \to S$ with $\Phi\, X = \Psi\, X = s$ are identical.

The Universal Property

$$\begin{tikzcd} R \arrow[r, "\varphi"] \arrow[d] & S \\ \mathit{UP}\ R \arrow[ur, "{\Phi\text{, unique for } \Phi\, X = s}"'] & \end{tikzcd}$$

- Uniqueness of Φ:

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