

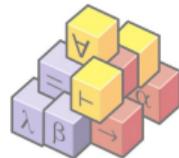
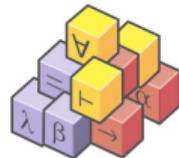
Functional Programming in Isabelle/HOL

Simon Foster **Jim Woodcock**
University of York

16th August 2022

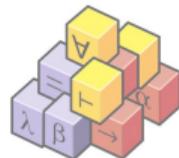
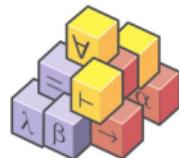
Overview

- 1 HOL as a Functional Programming Language
- 2 Type System
- 3 Algebraic Datatypes
- 4 Recursive Functions
- 5 Code Generator



Overview

- 1 HOL as a Functional Programming Language
- 2 Type System
- 3 Algebraic Datatypes
- 4 Recursive Functions
- 5 Code Generator



Functional Programming

- Paradigm: computations expressed as mathematical functions ($A \rightarrow B$)
- Origin: λ -calculus, formal system for computation ($\lambda x. \lambda y. x + y$)
- Function = abstraction
- Functions are side-effect free: everything captured by output type B
- No first-class notion of assignment ($x = e$) or iteration (`while` / `do` / `for`)
- Recursion instead of iteration
- Functions can also be higher order, taking functions as arguments
- Languages: Haskell, Scala, F#, Clojure, Erlang, OCaml, Elm
- Functional features: Python, Go, Java, C#, Rust, C++

Functional Programming

- **Paradigm:** computations expressed as **mathematical functions** ($A \Rightarrow B$).
- **Origin:** λ -calculus, formal system for computation $\lambda x : \mathbb{N}. x + 1$.
- Functional \neq imperative.
- Functions are **side-effect free**: everything captured by output type B .
- No first-class notion of assignment ($x := e$) or iteration (while b do S).
- Recursion instead of iteration.
- Functions can also be **higher order**: taking functions as arguments.
- **Languages:** Haskell, SML, F#, Clojure, Erlang, OCaml, Lisp.
- **Functional features:** Python, Scala, Java, Go, Rust, C#.

Functional Programming

- **Paradigm**: computations expressed as **mathematical functions** ($A \Rightarrow B$).
- **Origin**: λ -calculus, formal system for computation $\lambda x : \mathbb{N}. x + 1$.
- Functional \neq imperative.
- Functions are **side-effect free**: everything captured by output type B .
- No first-class notion of assignment ($x := e$) or iteration (while b do S).
- Recursion instead of iteration.
- Functions can also be **higher order**: taking functions as arguments.
- **Languages**: Haskell, SML, F#, Clojure, Erlang, OCaml, Lisp.
- **Functional features**: Python, Scala, Java, Go, Rust, C#.

Functional Programming

- **Paradigm**: computations expressed as **mathematical functions** ($A \Rightarrow B$).
- **Origin**: λ -calculus, formal system for computation $\lambda x : \mathbb{N}. x + 1$.
- **Functional** \neq **imperative**.
 - Functions are **side-effect free**: everything captured by output type B .
 - No first-class notion of assignment ($x := e$) or iteration (**while** b **do** S).
 - Recursion instead of iteration.
 - Functions can also be **higher order**: taking functions as arguments.
 - **Languages**: Haskell, SML, F#, Clojure, Erlang, OCaml, Lisp.
 - **Functional features**: Python, Scala, Java, Go, Rust, C#.

Functional Programming

- **Paradigm**: computations expressed as **mathematical functions** ($A \Rightarrow B$).
- **Origin**: λ -calculus, formal system for computation $\lambda x : \mathbb{N}. x + 1$.
- **Functional** \neq imperative.
- Functions are **side-effect free**: everything captured by output type B .
- No first-class notion of assignment ($x := e$) or iteration (**while** b do S).
- Recursion instead of iteration.
- Functions can also be **higher order**: taking functions as arguments.
- **Languages**: Haskell, SML, F#, Clojure, Erlang, OCaml, Lisp.
- **Functional features**: Python, Scala, Java, Go, Rust, C#.

Functional Programming

- **Paradigm**: computations expressed as **mathematical functions** ($A \Rightarrow B$).
- **Origin**: λ -calculus, formal system for computation $\lambda x : \mathbb{N}. x + 1$.
- **Functional** \neq imperative.
- Functions are **side-effect free**: everything captured by output type B .
- No first-class notion of assignment ($x := e$) or iteration (**while b do S**).
- Recursion instead of iteration.
- Functions can also be **higher order**: taking functions as arguments.
- **Languages**: Haskell, SML, F#, Clojure, Erlang, OCaml, Lisp.
- **Functional features**: Python, Scala, Java, Go, Rust, C#.

Functional Programming

- **Paradigm**: computations expressed as **mathematical functions** ($A \Rightarrow B$).
- **Origin**: λ -calculus, formal system for computation $\lambda x : \mathbb{N}. x + 1$.
- **Functional** \neq imperative.
- Functions are **side-effect free**: everything captured by output type B .
- No first-class notion of assignment ($x := e$) or iteration (**while** b **do** S).
- Recursion instead of iteration.
- Functions can also be **higher order**: taking functions as arguments.
- **Languages**: Haskell, SML, F#, Clojure, Erlang, OCaml, Lisp.
- **Functional features**: Python, Scala, Java, Go, Rust, C#.

Functional Programming

- **Paradigm**: computations expressed as **mathematical functions** ($A \Rightarrow B$).
- **Origin**: λ -calculus, formal system for computation $\lambda x : \mathbb{N}. x + 1$.
- **Functional** \neq imperative.
- Functions are **side-effect free**: everything captured by output type B .
- No first-class notion of assignment ($x := e$) or iteration (**while** b **do** S).
- Recursion instead of iteration.
- Functions can also be **higher order**: taking functions as arguments.
- **Languages**: Haskell, SML, F#, Clojure, Erlang, OCaml, Lisp.
- **Functional features**: Python, Scala, Java, Go, Rust, C#.

Functional Programming

- **Paradigm**: computations expressed as **mathematical functions** ($A \Rightarrow B$).
- **Origin**: λ -calculus, formal system for computation $\lambda x : \mathbb{N}. x + 1$.
- **Functional** \neq imperative.
- Functions are **side-effect free**: everything captured by output type B .
- No first-class notion of assignment ($x := e$) or iteration (**while** b **do** S).
- Recursion instead of iteration.
- Functions can also be **higher order**: taking functions as arguments.
- **Languages**: Haskell, SML, F#, Clojure, Erlang, OCaml, Lisp.
- **Functional features**: Python, Scala, Java, Go, Rust, C#.

Functional Programming

- **Paradigm**: computations expressed as **mathematical functions** ($A \Rightarrow B$).
- **Origin**: λ -calculus, formal system for computation $\lambda x : \mathbb{N}. x + 1$.
- **Functional** \neq imperative.
- Functions are **side-effect free**: everything captured by output type B .
- No first-class notion of assignment ($x := e$) or iteration (**while** b **do** S).
- Recursion instead of iteration.
- Functions can also be **higher order**: taking functions as arguments.
- **Languages**: Haskell, SML, F#, Clojure, Erlang, OCaml, Lisp.
- **Functional features**: Python, Scala, Java, Go, Rust, C#.

Higher Order Logic

- Ideally typed functional specification and programming language
- Like Haskell, but closer to its older cousin ML (meta-language).
- Functional programming ideally suited to verification, e.g., induction.
- Not everything need be executable; e.g., uncountably infinite sets.
- Type system features:
 - ✓ Algebraic datatypes and recursive functions.
 - ✓ Type variables (cf. generics).
 - ✓ Polymorphic type classes and overloading.
 - ✓ Type constructor parameters, and more exotic type system features.
- Haskell-style *axiomatic* proofs. Haskell (free) logic.
- Haskell has uncomputable objects: \mathbb{N} , \sqrt{x} , etc.

Higher Order Logic

- Statically typed functional specification and programming language.
- Like Haskell, but closer to its older cousin ML (meta-language).
- Functional programming ideally suited to verification, e.g., induction.
- Not everything need be executable; e.g., uncountably infinite sets.

Haskell Similarities

- ✓ Algebraic datatypes and recursive functions.
- ✓ Type variables (cf. generics).
- ✓ Polymorphic type classes and overloading.
- ✗ Type constructor parameters, and more exotic type system features.
- Isabelle supports proofs. Haskell doesn't.
- Isabelle has uncomputable objects: \mathbb{R} , \sqrt{x} , π , etc.

Higher Order Logic

- Statically typed functional specification and programming language.
- Like Haskell, but closer to its older cousin **ML** (meta-language).
- Functional programming ideally suited to verification, e.g., **induction**.
- Not everything need be **executable**; e.g., uncountably infinite sets.

Haskell Similarities

- ✓ Algebraic datatypes and recursive functions.
- ✓ Type variables (cf. generics).
- ✓ Polymorphic type classes and overloading.
- ✗ Type constructor parameters, and more exotic type system features.
- Isabelle supports **proofs**. Haskell doesn't.
- Isabelle has **uncomputable** objects: \mathbb{R} , \sqrt{x} , π , etc.

Higher Order Logic

- Statically typed functional specification and programming language.
- Like Haskell, but closer to its older cousin **ML** (meta-language).
- Functional programming ideally suited to verification, e.g., **induction**.
- Not everything need be **executable**; e.g., uncountably infinite sets.

Haskell Similarities

- ✓ Algebraic datatypes and recursive functions.
- ✓ Type variables (cf. generics).
- ✓ Polymorphic type classes and overloading.
- ✗ Type constructor parameters, and more exotic type system features.
- Isabelle supports **proofs**. Haskell doesn't.
- Isabelle has **uncomputable** objects: \mathbb{R} , \sqrt{x} , π , etc.

Higher Order Logic

- Statically typed functional specification and programming language.
- Like Haskell, but closer to its older cousin **ML** (meta-language).
- Functional programming ideally suited to verification, e.g., **induction**.
- Not everything need be **executable**; e.g., **uncountably infinite sets**.

Haskell Similarities

- ✓ Algebraic datatypes and recursive functions.
 - ✓ Type variables (cf. generics).
 - ✓ Polymorphic type classes and overloading.
 - ✗ Type constructor parameters, and more exotic type system features.
-
- Isabelle supports **proofs**. Haskell doesn't.
 - Isabelle has **uncomputable** objects: \mathbb{R} , \sqrt{x} , π , etc.

Higher Order Logic

- Statically typed functional specification and programming language.
- Like Haskell, but closer to its older cousin **ML** (meta-language).
- Functional programming ideally suited to verification, e.g., **induction**.
- Not everything need be **executable**; e.g., **uncountably infinite sets**.

Haskell Similarities

- ✓ Algebraic datatypes and recursive functions.
 - ✓ Type variables (cf. generics).
 - ✓ Polymorphic type classes and overloading.
 - ✗ Type constructor parameters, and more exotic type system features.
-
- Isabelle supports **proofs**. Haskell doesn't.
 - Isabelle has **uncomputable** objects: \mathbb{R} , \sqrt{x} , π , etc.

Higher Order Logic

- Statically typed functional specification and programming language.
- Like Haskell, but closer to its older cousin **ML** (meta-language).
- Functional programming ideally suited to verification, e.g., **induction**.
- Not everything need be **executable**; e.g., **uncountably infinite sets**.

Haskell Similarities

- ✓ Algebraic datatypes and recursive functions.
- ✓ Type variables (cf. generics).
- ✓ Polymorphic type classes and overloading.
- ✗ Type constructor parameters, and more exotic type system features.
- Isabelle supports **proofs**. Haskell doesn't.
- Isabelle has **uncomputable** objects: \mathbb{R} , \sqrt{x} , π , etc.

Higher Order Logic

- Statically typed functional specification and programming language.
- Like Haskell, but closer to its older cousin **ML** (meta-language).
- Functional programming ideally suited to verification, e.g., **induction**.
- Not everything need be **executable**; e.g., **uncountably infinite sets**.

Haskell Similarities

- ✓ Algebraic datatypes and recursive functions.
- ✓ Type variables (cf. generics).
- ✓ Polymorphic type classes and overloading.
- ✗ Type constructor parameters, and more exotic type system features.
- Isabelle supports **proofs**. Haskell doesn't.
- Isabelle has **uncomputable** objects: \mathbb{R} , \sqrt{x} , π , etc.

Higher Order Logic

- Statically typed functional specification and programming language.
- Like Haskell, but closer to its older cousin **ML** (meta-language).
- Functional programming ideally suited to verification, e.g., **induction**.
- Not everything need be **executable**; e.g., **uncountably infinite sets**.

Haskell Similarities

- ✓ Algebraic datatypes and recursive functions.
- ✓ Type variables (cf. generics).
- ✓ Polymorphic type classes and overloading.
- ✗ Type constructor parameters, and more exotic type system features.
- Isabelle supports **proofs**. Haskell doesn't.
- Isabelle has **uncomputable** objects: \mathbb{R} , \sqrt{x} , π , etc.

Higher Order Logic

- Statically typed functional specification and programming language.
- Like Haskell, but closer to its older cousin **ML** (meta-language).
- Functional programming ideally suited to verification, e.g., **induction**.
- Not everything need be **executable**; e.g., **uncountably infinite sets**.

Haskell Similarities

- ✓ Algebraic datatypes and recursive functions.
 - ✓ Type variables (cf. generics).
 - ✓ Polymorphic type classes and overloading.
 - ✗ Type constructor parameters, and more exotic type system features.
-
- Isabelle supports **proofs**. Haskell doesn't.
 - Isabelle has **uncomputable** objects: \mathbb{R} , \sqrt{x} , π , etc.

Higher Order Logic

- Statically typed functional specification and programming language.
- Like Haskell, but closer to its older cousin **ML** (meta-language).
- Functional programming ideally suited to verification, e.g., **induction**.
- Not everything need be **executable**; e.g., **uncountably infinite sets**.

Haskell Similarities

- ✓ Algebraic datatypes and recursive functions.
- ✓ Type variables (cf. generics).
- ✓ Polymorphic type classes and overloading.
- ✗ Type constructor parameters, and more exotic type system features.
- Isabelle supports **proofs**. Haskell doesn't.
- Isabelle has **uncomputable** objects: \mathbb{R} , \sqrt{x} , π , etc.

Higher Order Logic

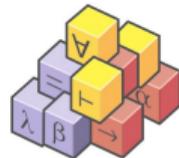
- Statically typed functional specification and programming language.
- Like Haskell, but closer to its older cousin **ML** (meta-language).
- Functional programming ideally suited to verification, e.g., **induction**.
- Not everything need be **executable**; e.g., **uncountably infinite sets**.

Haskell Similarities

- ✓ Algebraic datatypes and recursive functions.
 - ✓ Type variables (cf. generics).
 - ✓ Polymorphic type classes and overloading.
 - ✗ Type constructor parameters, and more exotic type system features.
-
- Isabelle supports **proofs**. Haskell doesn't.
 - Isabelle has **uncomputable** objects: \mathbb{R} , \sqrt{x} , π , etc.

Overview

- 1 HOL as a Functional Programming Language
- 2 Type System**
- 3 Algebraic Datatypes
- 4 Recursive Functions
- 5 Code Generator



Fundamental Types of HOL

- Booleans (true or false) True or False, \wedge , \vee , \neg , \rightarrow
- Natural numbers (nat or ℕ) 0, 1, 2, 3, 4, ... or Suc(0), Suc(Suc(0)) etc.
Arithmetic operators: $+$, $*$, $-$, $/$, div , mod
- Total functions ($A \rightarrow B$)
- Values $\lambda x. t(x) \rightarrow \lambda$ -abstraction (also anonymous functions, closures)
- For example, $\lambda x. \text{nat } x + 1$ has the type $\text{nat} \rightarrow \text{nat}$.
- Pairs $(A \times B). (x, y)$ for $x \in A$ and $y \in B$.
- Calculators: $\text{fst}(x, y) = x$ and $\text{snd}(x, y) = y$.
- Lists (lists of terms) $\text{list } A_1 \times A_2 \times A_3 \times \dots \times A_n$
- Explicitly design types to term type coercion, $x :: \tau$ \vdash E.

Fundamental Types of HOL

- Booleans (**bool** or \mathbb{B}): **True** or **False**, \wedge , \vee , \neg , \longrightarrow .
- Natural numbers (**nat** or \mathbb{N}): 0, 1, 2, 3, 4 \dots or *Suc* 0, *Suc* (*Suc* 0) etc.
Arithmetic operators: +, -, *, n^2 , /.
- Total functions ($A \Rightarrow B$):
- Values: $\lambda x. f(x)$ – λ -abstraction (aka anonymous functions, closures).
- For example: $\lambda x :: \text{nat}. x + 1$ has the type $\text{nat} \Rightarrow \text{nat}$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $\text{fst}(x, y) = x$ and $\text{snd}(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \dots \times A_n$.
- Explicitly assign type to term: type coercion, $x :: T$.

Fundamental Types of HOL

- Booleans (**bool** or \mathbb{B}): **True** or **False**, \wedge , \vee , \neg , \longrightarrow .
- Natural numbers (**nat** or \mathbb{N}): **0, 1, 2, 3, 4...** or **Suc 0, Suc (Suc 0)** etc.
Arithmetic operators: $+$, $-$, $*$, n^2 , $/$.
- Total functions ($A \Rightarrow B$):
- Values: $\lambda x. f(x)$ – λ -abstraction (aka anonymous functions, closures).
- For example: $\lambda x :: \text{nat}. x + 1$ has the type $\text{nat} \Rightarrow \text{nat}$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $\text{fst}(x, y) = x$ and $\text{snd}(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \dots \times A_n$.
- Explicitly assign type to term: type coercion, $x :: T$.

Fundamental Types of HOL

- Booleans (**bool** or \mathbb{B}): **True** or **False**, \wedge , \vee , \neg , \longrightarrow .
- Natural numbers (**nat** or \mathbb{N}): **0, 1, 2, 3, 4...** or **Suc 0, Suc (Suc 0)** etc.
Arithmetic operators: $+$, $-$, $*$, n^2 , $/$.
- Total functions ($A \Rightarrow B$):
- Values: $\lambda x. f(x)$ – λ -abstraction (aka anonymous functions, closures).
- For example: $\lambda x :: \text{nat}. x + 1$ has the type $\text{nat} \Rightarrow \text{nat}$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $\text{fst}(x, y) = x$ and $\text{snd}(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \dots \times A_n$.
- Explicitly assign type to term: type coercion, $x :: T$.

Fundamental Types of HOL

- Booleans (**bool** or \mathbb{B}): **True** or **False**, \wedge , \vee , \neg , \longrightarrow .
- Natural numbers (**nat** or \mathbb{N}): **0, 1, 2, 3, 4...** or **Suc 0, Suc (Suc 0)** etc.
Arithmetic operators: $+$, $-$, $*$, n^2 , $/$.
- Total functions ($A \Rightarrow B$):
 - Values: $\lambda x. f(x)$ – λ -abstraction (aka anonymous functions, closures).
 - For example: $\lambda x :: \text{nat}. x + 1$ has the type $\text{nat} \Rightarrow \text{nat}$.
 - Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
 - Selectors: $\text{fst}(x, y) = x$ and $\text{snd}(x, y) = y$.
 - Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \dots \times A_n$.
 - Explicitly assign type to term: type coercion, $x :: T$.

Fundamental Types of HOL

- Booleans (**bool** or \mathbb{B}): **True** or **False**, \wedge , \vee , \neg , \longrightarrow .
- Natural numbers (**nat** or \mathbb{N}): **0, 1, 2, 3, 4...** or **Suc 0, Suc (Suc 0)** etc.
Arithmetic operators: $+$, $-$, $*$, n^2 , $/$.
- Total functions ($A \Rightarrow B$):
- Values: $\lambda x. f(x)$ – λ -abstraction (aka **anonymous functions, closures**).
 - For example: $\lambda x :: \text{nat}. x + 1$ has the type $\text{nat} \Rightarrow \text{nat}$.
 - Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
 - Selectors: $\text{fst}(x, y) = x$ and $\text{snd}(x, y) = y$.
 - Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \dots \times A_n$.
 - Explicitly assign type to term: **type coercion**, $x :: T$.

Fundamental Types of HOL

- Booleans (**bool** or \mathbb{B}): **True** or **False**, \wedge , \vee , \neg , \longrightarrow .
- Natural numbers (**nat** or \mathbb{N}): **0, 1, 2, 3, 4...** or **Suc 0, Suc (Suc 0)** etc.
Arithmetic operators: $+$, $-$, $*$, n^2 , $/$.
- Total functions ($A \Rightarrow B$):
- Values: $\lambda x. f(x)$ – λ -abstraction (aka **anonymous functions, closures**).
- For example: $\lambda x :: \text{nat}. x + 1$ has the type $\text{nat} \Rightarrow \text{nat}$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $\text{fst}(x, y) = x$ and $\text{snd}(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \dots \times A_n$.
- Explicitly assign type to term: **type coercion**, $x :: T$.

Fundamental Types of HOL

- Booleans (**bool** or \mathbb{B}): **True** or **False**, \wedge , \vee , \neg , \longrightarrow .
- Natural numbers (**nat** or \mathbb{N}): **0, 1, 2, 3, 4...** or **Suc 0, Suc (Suc 0)** etc.
Arithmetic operators: $+$, $-$, $*$, n^2 , $/$.
- Total functions ($A \Rightarrow B$):
- Values: $\lambda x. f(x)$ – λ -abstraction (aka **anonymous functions, closures**).
- For example: $\lambda x :: \text{nat}. x + 1$ has the type $\text{nat} \Rightarrow \text{nat}$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
 - Selectors: $\text{fst}(x, y) = x$ and $\text{snd}(x, y) = y$.
 - Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \dots \times A_n$.
 - Explicitly assign type to term: **type coercion**, $x :: T$.

Fundamental Types of HOL

- Booleans (**bool** or \mathbb{B}): **True** or **False**, \wedge , \vee , \neg , \longrightarrow .
- Natural numbers (**nat** or \mathbb{N}): **0, 1, 2, 3, 4...** or **Suc 0, Suc (Suc 0)** etc.
Arithmetic operators: $+$, $-$, $*$, n^2 , $/$.
- Total functions ($A \Rightarrow B$):
- Values: $\lambda x. f(x)$ – λ -abstraction (aka **anonymous functions, closures**).
- For example: $\lambda x :: \text{nat}. x + 1$ has the type $\text{nat} \Rightarrow \text{nat}$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $\text{fst}(x, y) = x$ and $\text{snd}(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \dots \times A_n$.
- Explicitly assign type to term: **type coercion**, $x :: T$.

Fundamental Types of HOL

- Booleans (**bool** or \mathbb{B}): **True** or **False**, \wedge , \vee , \neg , \longrightarrow .
- Natural numbers (**nat** or \mathbb{N}): **0, 1, 2, 3, 4...** or **Suc 0, Suc (Suc 0)** etc.
Arithmetic operators: $+$, $-$, $*$, n^2 , $/$.
- Total functions ($A \Rightarrow B$):
- Values: $\lambda x. f(x)$ – λ -abstraction (aka **anonymous functions, closures**).
- For example: $\lambda x :: \text{nat}. x + 1$ has the type $\text{nat} \Rightarrow \text{nat}$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $\text{fst}(x, y) = x$ and $\text{snd}(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \dots \times A_n$.
- Explicitly assign type to term: **type coercion**, $x :: T$.

Fundamental Types of HOL

- Booleans (**bool** or \mathbb{B}): **True** or **False**, \wedge , \vee , \neg , \longrightarrow .
- Natural numbers (**nat** or \mathbb{N}): **0, 1, 2, 3, 4...** or **Suc 0, Suc (Suc 0)** etc.
Arithmetic operators: $+$, $-$, $*$, n^2 , $/$.
- Total functions ($A \Rightarrow B$):
- Values: $\lambda x. f(x)$ – λ -abstraction (aka **anonymous functions, closures**).
- For example: $\lambda x :: \text{nat}. x + 1$ has the type $\text{nat} \Rightarrow \text{nat}$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $\text{fst}(x, y) = x$ and $\text{snd}(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \dots \times A_n$.
- Explicitly assign type to term: **type coercion**, $x :: T$.

Example: Evaluating Terms

```
value "True  $\wedge$  False" (* Returns False *)
```

```
value "True  $\vee$  False" (* Returns True *)
```

```
value "(3::nat) + 2" (* Returns 5 *)
```

```
value "(\lambda x::nat. x + 1) 6" (* Returns 7 *)
```

```
value "(\lambda x::nat. x2 + 1) 6" (* Returns 37 *)
```

```
value "(\lambda (x::nat,y::nat). x*y) (6,7)" (* Returns 42 *)
```

```
value "fst (2::nat, False) * 3" (* Returns 6 *)
```

Example: Evaluating Terms

```
value "True  $\wedge$  False" (* Returns False *)
```

```
value "True  $\vee$  False" (* Returns True *)
```

```
value "(3::nat) + 2" (* Returns 5 *)
```

```
value "(\lambda x::nat. x + 1) 6" (* Returns 7 *)
```

```
value "(\lambda x::nat. x2 + 1) 6" (* Returns 37 *)
```

```
value "(\lambda (x::nat,y::nat). x*y) (6,7)" (* Returns 42 *)
```

```
value "fst (2::nat, False) * 3" (* Returns 6 *)
```

Example: Evaluating Terms

```
value "True  $\wedge$  False" (* Returns False *)
```

```
value "True  $\vee$  False" (* Returns True *)
```

```
value "(3::nat) + 2" (* Returns 5 *)
```

```
value "(\lambda x::nat. x + 1) 6" (* Returns 7 *)
```

```
value "(\lambda x::nat. x2 + 1) 6" (* Returns 37 *)
```

```
value "(\lambda (x::nat,y::nat). x*y) (6,7)" (* Returns 42 *)
```

```
value "fst (2::nat, False) * 3" (* Returns 6 *)
```

Example: Evaluating Terms

```
value "True  $\wedge$  False" (* Returns False *)
```

```
value "True  $\vee$  False" (* Returns True *)
```

```
value "(3::nat) + 2" (* Returns 5 *)
```

```
value "(\lambda x::nat. x + 1) 6" (* Returns 7 *)
```

```
value "(\lambda x::nat. x2 + 1) 6" (* Returns 37 *)
```

```
value "(\lambda (x::nat,y::nat). x*y) (6,7)" (* Returns 42 *)
```

```
value "fst (2::nat, False) * 3" (* Returns 6 *)
```

Example: Evaluating Terms

value "True \wedge False" (* Returns False *)

value "True \vee False" (* Returns True *)

value "(3::nat) + 2" (* Returns 5 *)

value "(\lambda x::nat. x + 1) 6" (* Returns 7 *)

value "(\lambda x::nat. x² + 1) 6" (* Returns 37 *)

value "(\lambda (x::nat,y::nat). x*y) (6,7)" (* Returns 42 *)

value "fst (2::nat, False) * 3" (* Returns 6 *)

Example: Evaluating Terms

```
value "True  $\wedge$  False" (* Returns False *)
```

```
value "True  $\vee$  False" (* Returns True *)
```

```
value "(3::nat) + 2" (* Returns 5 *)
```

```
value "(\lambda x::nat. x + 1) 6" (* Returns 7 *)
```

```
value "(\lambda x::nat. x2 + 1) 6" (* Returns 37 *)
```

```
value "(\lambda (x::nat,y::nat). x*y) (6,7)" (* Returns 42 *)
```

```
value "fst (2::nat, False) * 3" (* Returns 6 *)
```

Example: Evaluating Terms

```
value "True  $\wedge$  False" (* Returns False *)
```

```
value "True  $\vee$  False" (* Returns True *)
```

```
value "(3::nat) + 2" (* Returns 5 *)
```

```
value "( $\lambda$  x::nat. x + 1) 6" (* Returns 7 *)
```

```
value "( $\lambda$  x::nat. x2 + 1) 6" (* Returns 37 *)
```

```
value "( $\lambda$  (x::nat, y::nat). x*y) (6,7)" (* Returns 42 *)
```

```
value "fst (2::nat, False) * 3" (* Returns 6 *)
```

Example: Evaluating Terms

```
value "True  $\wedge$  False" (* Returns False *)
```

```
value "True  $\vee$  False" (* Returns True *)
```

```
value "(3::nat) + 2" (* Returns 5 *)
```

```
value "(\lambda x::nat. x + 1) 6" (* Returns 7 *)
```

```
value "(\lambda x::nat. x2 + 1) 6" (* Returns 37 *)
```

```
value "(\lambda (x::nat, y::nat). x*y) (6,7)" (* Returns 42 *)
```

```
value "fst (2::nat, False) * 3" (* Returns 6 *)
```

Basic Commands

- Create new types as synonyms for existing types:

```
type nat_pair = "(nat × nat)"
```

- Create new simple functions:

```
let add_pair : nat_pair → nat → nat → nat
```

```
  add_pair = (λ(x, y). λz. y)
```

```
let square : nat → nat → nat
```

```
  square = λx. λy. x2
```

- Type-check functions using `check` and evaluate them using `eval`:

```
check (add_pair (3, 5) 7) -- type: nat
```

```
eval (add_pair (3, 5) 7) -- value: 5
```

Basic Commands

- Create new types as **synonyms** for existing types:

```
type_synonym nat_pair = "(nat × nat)"
```

- Create new simple functions:

```
definition add_pair :: "nat_pair ⇒ nat" where  
  "add_pair = (λ(x, y). x + y)"
```

```
definition square :: "nat ⇒ nat" where  
  "square x = x * x"
```

- Type check functions using **term** and evaluate them using **value**:

```
term "add_pair (3, 5)" (* Returns nat *)  
value "add_pair (3, 5)" (* Returns 8 *)
```

Basic Commands

- Create new types as **synonyms** for existing types:

```
type_synonym nat_pair = "(nat × nat)"
```

- Create new simple functions:

```
definition add_pair :: "nat_pair ⇒ nat" where  
  "add_pair = (λ(x, y). x + y)"
```

```
definition square :: "nat ⇒ nat" where  
  "square x = x * x"
```

- Type check functions using **term** and evaluate them using **value**:

```
term "add_pair (3, 5)" (* Returns nat *)  
value "add_pair (3, 5)" (* Returns 8 *)
```

Basic Commands

- Create new types as **synonyms** for existing types:

```
type_synonym nat_pair = "(nat × nat)"
```

- Create new simple functions:

```
definition add_pair :: "nat_pair ⇒ nat" where  
  "add_pair = (λ(x, y). x + y)"
```

```
definition square :: "nat ⇒ nat" where  
  "square x = x * x"
```

- Type check functions using **term** and evaluate them using **value**:

```
term "add_pair (3, 5)" (* Returns nat *)  
value "add_pair (3, 5)" (* Returns 8 *)
```

Polymorphism

- Types contain type parameters of the form α , β , γ , etc.
- Instantiated to ground types (i.e., types without variables).

```
inductive bag : nat -> nat
```

```
inductive school : nat | nat1 | nat2 | nat3 | nat4 | nat5 | nat6
```

Checked with the command `check eq (1 : nat) (bag)`

We define polymorphic functions over such types.

```
def empty_bag : nat -> bag
```

```
empty_bag = (λ x. 0)
```

```
def add_bag : (A : nat -> bag) -> (A -> bag) -> bag
```

```
add_bag A B = (λ x. A x + B x)
```

Polymorphism

- Types contain **type parameters** of the form **'a**, **'b**, **'c** etc.
- Instantiated to **ground types** (i.e., types without variables):

```
type_synonym 'a bag = "'a ⇒ nat"
```

- Instantiations: `bool ⇒ nat`; `nat ⇒ nat`; `nat × nat ⇒ nat` etc.
- Checked with the command **typ**, e.g. **typ "nat bag"**
- We define **polymorphic functions** over such types.

```
definition empty_bag :: "'a bag" where  
  "empty_bag = (λ x. 0)"
```

```
definition add_bag :: "'a bag ⇒ 'a bag ⇒ 'a bag"  
  where "add_bag A B = (λ x. A x + B x)"
```

Polymorphism

- Types contain **type parameters** of the form **'a**, **'b**, **'c** etc.
- Instantiated to **ground types** (i.e., types without variables):

```
type_synonym 'a bag = "'a  $\Rightarrow$  nat"
```

- Instantiations: `bool \Rightarrow nat`; `nat \Rightarrow nat`; `nat \times nat \Rightarrow nat` etc.
- Checked with the command `typ`, e.g. `typ "nat bag"`
- We define **polymorphic functions** over such types.

```
definition empty_bag :: "'a bag" where  
  "empty_bag = ( $\lambda$  x. 0)"
```

```
definition add_bag :: "'a bag  $\Rightarrow$  'a bag  $\Rightarrow$  'a bag"  
  where "add_bag A B = ( $\lambda$  x. A x + B x)"
```

Polymorphism

- Types contain **type parameters** of the form **'a**, **'b**, **'c** etc.
- Instantiated to **ground types** (i.e., types without variables):

```
type_synonym 'a bag = "'a ⇒ nat"
```

- Instantiations: **bool** ⇒ **nat**; **nat** ⇒ **nat**; **nat**×**nat** ⇒ **nat** etc.
- Checked with the command **typ**, e.g. **typ** "nat bag"
- We define **polymorphic functions** over such types.

```
definition empty_bag :: "'a bag" where  
  "empty_bag = (λ x. 0)"
```

```
definition add_bag :: "'a bag ⇒ 'a bag ⇒ 'a bag"  
  where "add_bag A B = (λ x. A x + B x)"
```

Polymorphism

- Types contain **type parameters** of the form **'a**, **'b**, **'c** etc.
- Instantiated to **ground types** (i.e., types without variables):

```
type_synonym 'a bag = "'a ⇒ nat"
```

- Instantiations: **bool** ⇒ **nat**; **nat** ⇒ **nat**; **nat**×**nat** ⇒ **nat** etc.
- Checked with the command **typ**, e.g. **typ** "nat bag"
- We define **polymorphic functions** over such types.

```
definition empty_bag :: "'a bag" where  
  "empty_bag = (λ x. 0)"
```

```
definition add_bag :: "'a bag ⇒ 'a bag ⇒ 'a bag"  
  where "add_bag A B = (λ x. A x + B x)"
```

Polymorphism

- Types contain **type parameters** of the form **'a**, **'b**, **'c** etc.
- Instantiated to **ground types** (i.e., types without variables):

```
type_synonym 'a bag = "'a  $\Rightarrow$  nat"
```

- Instantiations: **bool** \Rightarrow **nat**; **nat** \Rightarrow **nat**; **nat** \times **nat** \Rightarrow **nat** etc.
- Checked with the command **typ**, e.g. **typ** "nat bag"
- We define **polymorphic functions** over such types.

```
definition empty_bag :: "'a bag" where  
  "empty_bag = ( $\lambda$  x. 0)"
```

```
definition add_bag :: "'a bag  $\Rightarrow$  'a bag  $\Rightarrow$  'a bag"  
  where "add_bag A B = ( $\lambda$  x. A x + B x)"
```

Overview

- 1 HOL as a Functional Programming Language
- 2 Type System
- 3 Algebraic Datatypes**
- 4 Recursive Functions
- 5 Code Generator



Algebraic Datatypes

Types are built using disjoint constructors.

```
data Nat = C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10
```

Constructors take parameters that can be self-referential.

Datatypes can be parametric, with type parameters defined as follows.

```
data Nat a = Zero | Suc a
```

In particular, `Zero :: Nat a` and `Suc :: a -> Nat a`.

Examples: `Zero`, `Suc Zero`, `Suc (Suc Zero)`

Algebraic Datatypes

- Types are built using disjoint **constructors**.

```
datatype T = C1 P1 | C2 P2 | C3 P3 | ...
```

- **Constructors** take parameters that can be **self-referential**.
- **Datatypes** can be **parametric**, with type parameters denoted as **'a** etc.

```
datatype nat = Zero | Suc nat
```

- Two constructors: `Zero :: nat` and `Suc :: nat ⇒ nat`.
- **Examples**: `Zero`, `Suc Zero`, `Suc (Suc Zero)`.

Algebraic Datatypes

- Types are built using disjoint **constructors**.

```
datatype T = C1 P1 | C2 P2 | C3 P3 | ...
```

- **Constructors** take parameters that can be **self-referential**.
- Datatypes can be **parametric**, with type parameters denoted as 'a etc.

```
datatype nat = Zero | Suc nat
```

- Two constructors: `Zero :: nat` and `Suc :: nat ⇒ nat`.
- Examples: `Zero`, `Suc Zero`, `Suc (Suc Zero)`.

Algebraic Datatypes

- Types are built using disjoint **constructors**.

```
datatype T = C1 P1 | C2 P2 | C3 P3 | ...
```

- **Constructors** take parameters that can be **self-referential**.
- **Datatypes** can be **parametric**, with type parameters denoted as '**a**' etc.

```
datatype nat = Zero | Suc nat
```

- Two constructors: `Zero :: nat` and `Suc :: nat ⇒ nat`.
- Examples: `Zero`, `Suc Zero`, `Suc (Suc Zero)`.

Algebraic Datatypes

- Types are built using disjoint **constructors**.

```
datatype T = C1 P1 | C2 P2 | C3 P3 | ...
```

- **Constructors** take parameters that can be **self-referential**.
- **Datatypes** can be **parametric**, with type parameters denoted as **'a** etc.

```
datatype nat = Zero | Suc nat
```

- Two constructors: **Zero :: nat** and **Suc :: nat ⇒ nat**.
- **Examples**: Zero, Suc Zero, Suc (Suc Zero).

Algebraic Datatypes

- Types are built using disjoint **constructors**.

```
datatype T = C1 P1 | C2 P2 | C3 P3 | ...
```

- **Constructors** take parameters that can be **self-referential**.
- **Datatypes** can be **parametric**, with type parameters denoted as 'a' etc.

```
datatype nat = Zero | Suc nat
```

- Two constructors: **Zero :: nat** and **Suc :: nat ⇒ nat**.
- **Examples**: **Zero**, **Suc Zero**, **Suc (Suc Zero)**.

Inductive Lists

An inductive list is either empty, or a head followed by a tail.

```
Inductive list =  
  Nil ("[]") | Cons "cons" (list) (A "A")
```

A list is either Nil or an element of type A followed by a list.

Here, we assign Nil the (optional) syntax [], and Cons the operator #.

Example

```
1 2 3 4 [] is nat list and True # False # [] is bool list
```

Syntactic sugar for

```
Cons 1 (Cons 2 Nil) and Cons True (Cons False Nil)
```

Inductive Lists

- An **inductive** list is either empty, or a head followed by a tail.

```
datatype 'a list =  
  Nil ("[]") | Cons 'a "'a list" (infixr "#" 65)
```

- A list is empty `Nil` or an element of type `'a` followed by a list `Cons`.
- Here, we assign `Nil` the (optional) syntax `[]` and `Cons` infix operator `#`.

- Example

```
1 # 2 # [] :: nat list and True # False # [] :: bool list
```

- Syntactic sugar for

```
Cons 1 (Cons 2 Nil) and Cons True (Cons False Nil)
```

Inductive Lists

- An **inductive** list is either empty, or a head followed by a tail.

```
datatype 'a list =  
  Nil ("[]") | Cons 'a "'a list" (infixr "#" 65)
```

- A list is empty **Nil** or an element of type **'a** followed by a list **Cons**.
 - Here, we assign **Nil** the (optional) syntax **[]** and **Cons** infix operator **#**.
 - Example
- ```
1 # 2 # [] :: nat list and True # False # [] :: bool list
```
- Syntactic sugar for

```
Cons 1 (Cons 2 Nil) and Cons True (Cons False Nil)
```

# Inductive Lists

- An **inductive** list is either empty, or a head followed by a tail.

```
datatype 'a list =
 Nil ("[]") | Cons 'a "'a list" (infixr "#" 65)
```

- A list is empty **Nil** or an element of type **'a** followed by a list **Cons**.
- Here, we assign **Nil** the (optional) syntax `[]` and **Cons** infix operator `#`.

- Example

```
1 # 2 # [] :: nat list and True # False # [] :: bool list
```

- Syntactic sugar for

```
Cons 1 (Cons 2 Nil) and Cons True (Cons False Nil)
```

# Inductive Lists

- An **inductive** list is either empty, or a head followed by a tail.

```
datatype 'a list =
 Nil ("[]") | Cons 'a "'a list" (infixr "#" 65)
```

- A list is empty **Nil** or an element of type **'a** followed by a list **Cons**.
- Here, we assign **Nil** the (optional) syntax `[]` and **Cons** infix operator `#`.
- **Example**

```
1 # 2 # [] :: nat list and True # False # [] :: bool list
```

- Syntactic sugar for

```
Cons 1 (Cons 2 Nil) and Cons True (Cons False Nil)
```

# Inductive Lists

- An **inductive** list is either empty, or a head followed by a tail.

```
datatype 'a list =
 Nil ("[]") | Cons 'a "'a list" (infixr "#" 65)
```

- A list is empty **Nil** or an element of type **'a** followed by a list **Cons**.
- Here, we assign **Nil** the (optional) syntax **[]** and **Cons** infix operator **#**.
- **Example**

```
1 # 2 # [] :: nat list and True # False # [] :: bool list
```

- Syntactic sugar for

```
Cons 1 (Cons 2 Nil) and Cons True (Cons False Nil)
```

# Inductive Lists

- An **inductive** list is either empty, or a head followed by a tail.

```
datatype 'a list =
 Nil ("[]") | Cons 'a "'a list" (infixr "#" 65)
```

- A list is empty **Nil** or an element of type **'a** followed by a list **Cons**.
- Here, we assign **Nil** the (optional) syntax `[]` and **Cons** infix operator `#`.

- **Example**

```
1 # 2 # [] :: nat list and True # False # [] :: bool list
```

- **Syntactic sugar** for

```
Cons 1 (Cons 2 Nil) and Cons True (Cons False Nil)
```

# Inductive Lists

- An **inductive** list is either empty, or a head followed by a tail.

```
datatype 'a list =
 Nil ("[]") | Cons 'a "'a list" (infixr "#" 65)
```

- A list is empty **Nil** or an element of type **'a** followed by a list **Cons**.
- Here, we assign **Nil** the (optional) syntax `[]` and **Cons** infix operator `#`.

- **Example**

```
1 # 2 # [] :: nat list and True # False # [] :: bool list
```

- **Syntactic sugar** for

```
Cons 1 (Cons 2 Nil) and Cons True (Cons False Nil)
```

# More Datatype Examples

```
datatype ocean = Atlantic | Arctic | Indian | Pacific
```

```
datatype 'a option = None | Some 'a
```

```
datatype 'a tree = Empty
 | Leaf 'a
 | Node "'a tree" "'a tree"
```

# More Datatype Examples

```
datatype ocean = Atlantic | Arctic | Indian | Pacific
```

```
datatype 'a option = None | Some 'a
```

```
datatype 'a tree = Empty
 | Leaf 'a
 | Node "'a tree" "'a tree"
```

# More Datatype Examples

```
datatype ocean = Atlantic | Arctic | Indian | Pacific
```

```
datatype 'a option = None | Some 'a
```

```
datatype 'a tree = Empty
 | Leaf 'a
 | Node "'a tree" "'a tree"
```

## More Datatype Examples

```
datatype ocean = Atlantic | Arctic | Indian | Pacific
```

```
datatype 'a option = None | Some 'a
```

```
datatype 'a tree = Empty
 | Leaf 'a
 | Node "'a tree" "'a tree"
```

# Functions

- Functions built by pattern matching on algebraic datatype
  - Patterns have one or more constructors and variables for parameters
  - Definition of  $f$  consists of equations  $f(C\ x) = G(x)$
  - Overlapping patterns evaluated in the order they're given
- Is zero or not?

```
isZero :: Int -> Bool
isZero (Suc x) = False
isZero Zero = True
```
- Number the oceans

```
numOcean :: Ocean -> Int
numOcean Atlantic = 1
numOcean Pacific = 2
numOcean Indian = 3
numOcean Arctic = 4
```

# Functions

- Function  $f$  built by **pattern matching** on algebraic datatype.
- Patterns have one or more **constructors** and **variables** for parameters.
- Definition of  $f$  consists of equations  $f(C\ x) = G(x)$ .
- Overlapping patterns evaluated in the order they're given.
- Test for zero:

```
fun is_zero :: "nat ⇒ bool" where
 "is_zero (Suc x) = False" | "is_zero Zero = True"
```

- Number the oceans:

```
fun num_ocean :: "ocean ⇒ nat" where
 "num_ocean Atlantic = 1" | "num_ocean Pacific = 2" |
 "num_ocean Indian = 3" | "num_ocean Arctic = 4"
```

# Functions

- Function  $f$  built by **pattern matching** on algebraic datatype.
- Patterns have one or more **constructors** and **variables** for parameters.
- Definition of  $f$  consists of equations  $f(C\ x) = G(x)$ .
- Overlapping patterns evaluated in the order they're given.
- Test for zero:

```
fun is_zero :: "nat \Rightarrow bool" where
 "is_zero (Suc x) = False" | "is_zero Zero = True"
```

- Number the oceans:

```
fun num_ocean :: "ocean \Rightarrow nat" where
 "num_ocean Atlantic = 1" | "num_ocean Pacific = 2" |
 "num_ocean Indian = 3" | "num_ocean Arctic = 4"
```

# Functions

- Function  $f$  built by **pattern matching** on algebraic datatype.
- Patterns have one or more **constructors** and **variables** for parameters.
- Definition of  $f$  consists of equations  $f(C\ x) = G(x)$ .
- Overlapping patterns evaluated in the order they're given.
- Test for zero:

```
fun is_zero :: "nat ⇒ bool" where
 "is_zero (Suc x) = False" | "is_zero Zero = True"
```

- Number the oceans:

```
fun num_ocean :: "ocean ⇒ nat" where
 "num_ocean Atlantic = 1" | "num_ocean Pacific = 2" |
 "num_ocean Indian = 3" | "num_ocean Arctic = 4"
```

# Functions

- Function  $f$  built by **pattern matching** on algebraic datatype.
- Patterns have one or more **constructors** and **variables** for parameters.
- Definition of  $f$  consists of equations  $f(C\ x) = G(x)$ .
- Overlapping patterns evaluated in the order they're given.

- Test for zero:

```
fun is_zero :: "nat ⇒ bool" where
 "is_zero (Suc x) = False" | "is_zero Zero = True"
```

- Number the oceans:

```
fun num_ocean :: "ocean ⇒ nat" where
 "num_ocean Atlantic = 1" | "num_ocean Pacific = 2" |
 "num_ocean Indian = 3" | "num_ocean Arctic = 4"
```

# Functions

- Function  $f$  built by **pattern matching** on algebraic datatype.
- Patterns have one or more **constructors** and **variables** for parameters.
- Definition of  $f$  consists of equations  $f(C\ x) = G(x)$ .
- Overlapping patterns evaluated in the order they're given.
- Test for zero:

```
fun is_zero :: "nat \Rightarrow bool" where
 "is_zero (Suc x) = False" | "is_zero Zero = True"
```

- Number the oceans:

```
fun num_ocean :: "ocean \Rightarrow nat" where
 "num_ocean Atlantic = 1" | "num_ocean Pacific = 2" |
 "num_ocean Indian = 3" | "num_ocean Arctic = 4"
```

# Functions

- Function  $f$  built by **pattern matching** on algebraic datatype.
- Patterns have one or more **constructors** and **variables** for parameters.
- Definition of  $f$  consists of equations  $f(C\ x) = G(x)$ .
- Overlapping patterns evaluated in the order they're given.
- Test for zero:

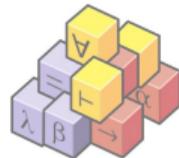
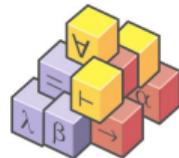
```
fun is_zero :: "nat \Rightarrow bool" where
 "is_zero (Suc x) = False" | "is_zero Zero = True"
```

- Number the oceans:

```
fun num_ocean :: "ocean \Rightarrow nat" where
 "num_ocean Atlantic = 1" | "num_ocean Pacific = 2" |
 "num_ocean Indian = 3" | "num_ocean Arctic = 4"
```

# Overview

- 1 HOL as a Functional Programming Language
- 2 Type System
- 3 Algebraic Datatypes
- 4 Recursive Functions**
- 5 Code Generator



# Recursive Functions

- Define a function by recursion over the algebraic datatype
- Example: Fibonacci numbers
  - $f(0) = 0$ ,  $f(1) = 1$ ,  $f(2) = 1$ ,  $f(3) = 2$ ,  $f(4) = 3$ ,  $f(5) = 5$ ,  $f(6) = 8$ ,  $f(7) = 13$ ,  $f(8) = 21$ ,  $f(9) = 34$ ,  $f(10) = 55$ ,  $f(11) = 89$ ,  $f(12) = 144$ ,  $f(13) = 233$ ,  $f(14) = 377$ ,  $f(15) = 610$ ,  $f(16) = 987$ ,  $f(17) = 1597$ ,  $f(18) = 2584$ ,  $f(19) = 4181$ ,  $f(20) = 6765$ ,  $f(21) = 10946$ ,  $f(22) = 17711$ ,  $f(23) = 28657$ ,  $f(24) = 46368$ ,  $f(25) = 75025$ ,  $f(26) = 121393$ ,  $f(27) = 196418$ ,  $f(28) = 317811$ ,  $f(29) = 514130$ ,  $f(30) = 832040$ ,  $f(31) = 1346209$ ,  $f(32) = 2178309$ ,  $f(33) = 3542248$ ,  $f(34) = 5720557$ ,  $f(35) = 9272793$ ,  $f(36) = 14930352$ ,  $f(37) = 24214963$ ,  $f(38) = 39186375$ ,  $f(39) = 63496038$ ,  $f(40) = 102334155$ ,  $f(41) = 165580141$ ,  $f(42) = 267914296$ ,  $f(43) = 433494437$ ,  $f(44) = 701408733$ ,  $f(45) = 1134903170$ ,  $f(46) = 1836311903$ ,  $f(47) = 2971215073$ ,  $f(48) = 4807526976$ ,  $f(49) = 7778752049$ ,  $f(50) = 12586269025$
  - $f(C1 + 1) = f(C1) + f(C1 - 1)$
- Match the input against the first equation satisfying the pattern
- Recurse checks that all possibilities for  $T1$  are covered by an equation

```
plus :: Nat -> Nat -> Nat
plus Zero n = n
plus (Suc m) n = Suc (plus m n)
```

- Length of a list

```
length :: List a -> Nat
length [] = 0
length (x : xs) = Suc (length xs)
```

# Recursive Functions

- Define a function by recursion over the algebraic datatype **T1**

```
fun rf :: "T1 \Rightarrow T2" where
 "rf (C1 x) = V1" | "rf (C2 x) = V2" | ...
```

- Match the input against the first equation satisfying the pattern.
- Isabelle checks that all possibilities for T1 are covered by an equation.

```
fun plus :: "nat \Rightarrow nat \Rightarrow nat" where
 "plus Zero n = n" |
 "plus (Suc m) n = Suc (plus m n)"
```

- Length of a list:

```
fun len :: "'a list \Rightarrow nat" where
 "len [] = 0" | "len (x # xs) = len xs + 1"
```

# Recursive Functions

- Define a function by recursion over the algebraic datatype **T1**

```
fun rf :: "T1 \Rightarrow T2" where
 "rf (C1 x) = V1" | "rf (C2 x) = V2" | ...
```

- Match the input against the first equation satisfying the pattern.
- Isabelle checks that all possibilities for T1 are covered by an equation.

```
fun plus :: "nat \Rightarrow nat \Rightarrow nat" where
 "plus Zero n = n" |
 "plus (Suc m) n = Suc (plus m n)"
```

- Length of a list:

```
fun len :: "'a list \Rightarrow nat" where
 "len [] = 0" | "len (x # xs) = len xs + 1"
```

# Recursive Functions

- Define a function by recursion over the algebraic datatype **T1**

```
fun rf :: "T1 \Rightarrow T2" where
 "rf (C1 x) = V1" | "rf (C2 x) = V2" | ...
```

- Match the input against the first equation satisfying the pattern.
- Isabelle checks that all possibilities for **T1** are covered by an equation.

```
fun plus :: "nat \Rightarrow nat \Rightarrow nat" where
 "plus Zero n = n" |
 "plus (Suc m) n = Suc (plus m n)"
```

- Length of a list:

```
fun len :: "'a list \Rightarrow nat" where
 "len [] = 0" | "len (x # xs) = len xs + 1"
```

# Recursive Functions

- Define a function by recursion over the algebraic datatype **T1**

```
fun rf :: "T1 \Rightarrow T2" where
 "rf (C1 x) = V1" | "rf (C2 x) = V2" | ...
```

- Match the input against the first equation satisfying the pattern.
- Isabelle checks that all possibilities for **T1** are covered by an equation.

```
fun plus :: "nat \Rightarrow nat \Rightarrow nat" where
 "plus Zero n = n" |
 "plus (Suc m) n = Suc (plus m n)"
```

- Length of a list:

```
fun len :: "'a list \Rightarrow nat" where
 "len [] = 0" | "len (x # xs) = len xs + 1"
```

# Example: Non-Termination

This function does not terminate. It's an infinite loop.

```
def rec(x: Int) = rec(x)
def rec(x: Int) = rec(x + 1)
```

It's rejected by Labelle/IDL with an error message.

Labelle requires that each function *terminates*.

• This is checked automatically.

• The check is produced automatically using flow heuristics.

## Example: Non-Termination

- This function does not **terminate**. It's an infinite loop.

```
fun bad :: "nat \Rightarrow nat" where
 "bad x = bad x + 1"
```

- It's **rejected** by Isabelle/HOL with an error message.
- Isabelle requires that each function **terminates**.
- This is checked automatically.
- The proof is produced automatically using clever **heuristics**.

# Example: Non-Termination

- This function does not **terminate**. It's an infinite loop.

```
fun bad :: "nat \Rightarrow nat" where
 "bad x = bad x + 1"
```

- It's **rejected** by Isabelle/HOL with an error message.
- Isabelle requires that each function **terminates**.
- This is checked **automatically**.
- The proof is produced **automatically** using **clever heuristics**.

## Example: Non-Termination

- This function does not **terminate**. It's an infinite loop.

```
fun bad :: "nat \Rightarrow nat" where
 "bad x = bad x + 1"
```

- It's **rejected** by Isabelle/HOL with an error message.
- Isabelle requires that each function **terminates**.
- This is checked **automatically**.
- The proof is produced **automatically** using **clever heuristics**.

# Example: Non-Termination

- This function does not **terminate**. It's an infinite loop.

```
fun bad :: "nat \Rightarrow nat" where
 "bad x = bad x + 1"
```

- It's **rejected** by Isabelle/HOL with an error message.
- Isabelle requires that each function **terminates**.
- This is checked **automatically**.
- The proof is produced automatically using clever heuristics.

## Example: Non-Termination

- This function does not **terminate**. It's an infinite loop.

```
fun bad :: "nat \Rightarrow nat" where
 "bad x = bad x + 1"
```

- It's **rejected** by Isabelle/HOL with an error message.
- Isabelle requires that each function **terminates**.
- This is checked **automatically**.
- The proof is produced automatically using clever **heuristics**.

# Examples: Recursive Functions

```
fun append :: "'a list ⇒ 'a list ⇒ 'a list"
 (infixr "@" 65)
 where "[] @ xs = xs" | "(x # xs) @ ys = x # (xs @ ys)"
```

```
value "append ((0::nat) # 1 # []) (2 # 3 # [])"
 (* 0#1#2#3#[] *)
```

```
fun rev :: "'a list ⇒ 'a list" where
 "rev [] = []" | "rev (x # xs) = rev xs @ [x]"
```

```
value "rev ((0::nat) # 1 # 2 # 3 # [])"
 (* 3#2#1#0#[] *)
```

## Examples: Recursive Functions

```
fun append :: "'a list ⇒ 'a list ⇒ 'a list"
 (infixr "@" 65)
 where "[] @ xs = xs" | "(x # xs) @ ys = x # (xs @ ys)"
```

```
value "append ((0::nat) # 1 # []) (2 # 3 # [])"
 (* 0#1#2#3#[] *)
```

```
fun rev :: "'a list ⇒ 'a list" where
 "rev [] = []" | "rev (x # xs) = rev xs @ [x]"
```

```
value "rev ((0::nat) # 1 # 2 # 3 # [])"
 (* 3#2#1#0#[] *)
```

## Examples: Recursive Functions

```
fun append :: "'a list ⇒ 'a list ⇒ 'a list"
 (infixr "@" 65)
 where "[] @ xs = xs" | "(x # xs) @ ys = x # (xs @ ys)"
```

```
value "append ((0::nat) # 1 # []) (2 # 3 # [])"
 (* 0#1#2#3#[] *)
```

```
fun rev :: "'a list ⇒ 'a list" where
 "rev [] = []" | "rev (x # xs) = rev xs @ [x]"
```

```
value "rev ((0::nat) # 1 # 2 # 3 # [])"
 (* 3#2#1#0#[] *)
```

## Examples: Recursive Functions

```
fun append :: "'a list ⇒ 'a list ⇒ 'a list"
 (infixr "@" 65)
 where "[] @ xs = xs" | "(x # xs) @ ys = x # (xs @ ys)"
```

```
value "append ((0::nat) # 1 # []) (2 # 3 # [])"
 (* 0#1#2#3#[] *)
```

```
fun rev :: "'a list ⇒ 'a list" where
 "rev [] = []" | "rev (x # xs) = rev xs @ [x]"
```

```
value "rev ((0::nat) # 1 # 2 # 3 # [])"
 (* 3#2#1#0#[] *)
```

## Examples: Recursive Functions

```
fun append :: "'a list ⇒ 'a list ⇒ 'a list"
 (infixr "@" 65)
 where "[] @ xs = xs" | "(x # xs) @ ys = x # (xs @ ys)"
```

```
value "append ((0::nat) # 1 # []) (2 # 3 # [])"
 (* 0#1#2#3#[] *)
```

```
fun rev :: "'a list ⇒ 'a list" where
 "rev [] = []" | "rev (x # xs) = rev xs @ [x]"
```

```
value "rev ((0::nat) # 1 # 2 # 3 # [])"
 (* 3#2#1#0#[] *)
```

# Mixfix Syntax Annotation

- Emacs has a flexible syntax for operators, called *mixfix*.
- Any operator, definition, or function can be given mixfix syntax.
- The operator is followed by a precedence, or *fixity*.
- `single` containing disambiguation symbols (`|`) and *precedence*.
- We use unicode symbols in mixfix annotation.
- Mixfix can be given in the definition, or using the `defun-autoload` command.
- `append'` → `list` → `list` → `list` (`fixity` = `0` or `95`)
- `append` (`fixity` = `95`)
- `single` (`fixity` = `95`) → `single` `x = x` (`fixity` = `95`)
- `fixity` is a syntactic constant. No logical meaning (not use `of`).

# Mixfix Syntax Annotation

- Isabelle has a flexible syntax for operators, called **mixfix**.
- Any constructor, definition, or function can be given mixfix syntax.
- **infix**, **infixl**, **infixr** followed by a precedence; or
- String containing placeholder symbols (`_`) and precedence.
- We use **unicode** symbols in mixfix annotation.
- Mixfix can be given in the definition, or using the **notation** command:

```
fun append::"'a list⇒'a list⇒'a list" (infixr "@" 65)
```

```
notation append (infixr "@" 65)
```

```
abbreviation single ("[_]") where "single x ≡ x#[_]"
```

- Abbreviation: **syntactic constant**. No logical meaning (note use of  $\equiv$ ).

# Mixfix Syntax Annotation

- Isabelle has a flexible syntax for operators, called **mixfix**.
- Any constructor, definition, or function can be given mixfix syntax.
- `infix`, `infixl`, `infixr` followed by a precedence; or
- String containing placeholder symbols (`_`) and precedence.
- We use **unicode** symbols in mixfix annotation.
- Mixfix can be given in the definition, or using the **notation** command:

```
fun append::"'a list ⇒ 'a list ⇒ 'a list" (infixr "@" 65)
```

```
notation append (infixr "@" 65)
```

```
abbreviation single ("[_]") where "single x ≡ x#[_]"
```

- Abbreviation: **syntactic constant**. No logical meaning (note use of  $\equiv$ ).

# Mixfix Syntax Annotation

- Isabelle has a flexible syntax for operators, called **mixfix**.
- Any constructor, definition, or function can be given mixfix syntax.
- **infix**, **infixl**, **infixr** followed by a precedence; or
- String containing placeholder symbols (`_`) and precedence.
- We use **unicode** symbols in mixfix annotation.
- Mixfix can be given in the definition, or using the **notation** command:  

```
fun append::"'a list⇒'a list⇒'a list" (infixr "@" 65)

notation append (infixr "@" 65)

abbreviation single ("[_]") where "single x ≡ x#[_]"
```
- Abbreviation: **syntactic constant**. No logical meaning (note use of  $\equiv$ ).

# Mixfix Syntax Annotation

- Isabelle has a flexible syntax for operators, called **mixfix**.
- Any constructor, definition, or function can be given mixfix syntax.
- **infix**, **infixl**, **infixr** followed by a precedence; or
- String containing placeholder symbols (`_`) and precedence.

● We use **unicode** symbols in mixfix annotation.

● Mixfix can be given in the definition, or using the **notation** command:

```
fun append:: "'a list ⇒ 'a list ⇒ 'a list" (infixr "@" 65)
```

```
notation append (infixr "@" 65)
```

```
abbreviation single ("[_]") where "single x ≡ x#[_]"
```

● Abbreviation: **syntactic constant**. No logical meaning (note use of  $\equiv$ ).

# Mixfix Syntax Annotation

- Isabelle has a flexible syntax for operators, called **mixfix**.
- Any constructor, definition, or function can be given mixfix syntax.
- **infix**, **infixl**, **infixr** followed by a precedence; or
- String containing placeholder symbols (`_`) and precedence.
- We use **unicode** symbols in mixfix annotation.
- Mixfix can be given in the definition, or using the **notation** command:

```
fun append::"'a list⇒'a list⇒'a list" (infixr "@" 65)

notation append (infixr "@" 65)

abbreviation single ("[_]") where "single x ≡ x#[_]"
```
- Abbreviation: **syntactic constant**. No logical meaning (note use of  $\equiv$ ).

# Mixfix Syntax Annotation

- Isabelle has a flexible syntax for operators, called **mixfix**.
- Any constructor, definition, or function can be given mixfix syntax.
- **infix**, **infixl**, **infixr** followed by a precedence; or
- String containing placeholder symbols (`_`) and precedence.
- We use **unicode** symbols in mixfix annotation.
- Mixfix can be given in the definition, or using the **notation** command:

```
fun append::"'a list⇒'a list⇒'a list" (infixr "@" 65)
```

```
notation append (infixr "@" 65)
```

```
abbreviation single ("[_]") where "single x ≡ x#[_]"
```

- **Abbreviation: syntactic constant.** No logical meaning (note use of  $\equiv$ ).

# Mixfix Syntax Annotation

- Isabelle has a flexible syntax for operators, called **mixfix**.
- Any constructor, definition, or function can be given mixfix syntax.
- **infix**, **infixl**, **infixr** followed by a precedence; or
- String containing placeholder symbols (`_`) and precedence.
- We use **unicode** symbols in mixfix annotation.
- Mixfix can be given in the definition, or using the **notation** command:

```
fun append::"'a list⇒'a list⇒'a list" (infixr "@" 65)
```

```
notation append (infixr "@" 65)
```

```
abbreviation single ("[_]") where "single x ≡ x#[_]"
```

- Abbreviation: **syntactic constant**. No logical meaning (note use of  $\equiv$ ).

# Higher Order Functions and Type Inference

• Higher order function takes function argument (e.g. `map`)

```
map :: (a -> b) -> [a] -> [b]
```

```
map f [1..10] == [f 1..f 10] == [2..11]
```

```
map f [1..10] == [f 1..f 10] == [2..11]
```

```
map f [] == []
```

• The effect is to apply the right hand function to every element of the list

• We can also partially apply `map` to a given function:

```
map f [1..10] == [f 1..f 10]
```

```
map f [] == []
```

• Type inference deduces the type of `f` from using the argument

• E.g. `f` instances `Int -> Int` to both be `map`

# Higher Order Functions and Type Inference

- **map**: higher order function takes function argument (cf.,  $\lambda x. x + 1$ ).

```
fun map :: "('a ⇒ 'b) ⇒ 'a list ⇒ 'b list" where
 "map f [] = []" | "map f (x # xs) = f x # map f xs"
```

```
value "map ($\lambda x::\text{nat}. x + 1$) (0 # 1 # 2 # [])"
 (* 1#2#3#[] *)
```

- The effect is to apply the argument function to every element of the list.
- We can also partially apply **map** to a given function:

```
term "map ($\lambda x::\text{nat}. x + 1$)"
 (* Type: nat list ⇒ nat list *)
```

- Type inference determines the type of this function using the arguments.
- Here, it instantiates **'a** and **'b** to both be **nat**.

# Higher Order Functions and Type Inference

- **map**: higher order function takes function argument (cf.,  $\lambda x. x + 1$ ).

```
fun map :: "('a ⇒ 'b) ⇒ 'a list ⇒ 'b list" where
 "map f [] = []" | "map f (x # xs) = f x # map f xs"
```

```
value "map ($\lambda x::\text{nat}. x + 1$) (0 # 1 # 2 # [])"
 (* 1#2#3#[] *)
```

- The effect is to apply the argument function to every element of the list.
- We can also partially apply map to a given function:

```
term "map ($\lambda x::\text{nat}. x + 1$)"
 (* Type: nat list ⇒ nat list *)
```

- Type inference determines the type of this function using the arguments.
- Here, it instantiates 'a and 'b to both be nat.

# Higher Order Functions and Type Inference

- **map**: higher order function takes function argument (cf.,  $\lambda x. x + 1$ ).

```
fun map :: "('a ⇒ 'b) ⇒ 'a list ⇒ 'b list" where
 "map f [] = []" | "map f (x # xs) = f x # map f xs"
```

```
value "map ($\lambda x::\text{nat}. x + 1$) (0 # 1 # 2 # [])"
 (* 1#2#3#[] *)
```

- The effect is to apply the argument function to every element of the list.
- We can also **partially** apply **map** to a given function:

```
term "map ($\lambda x::\text{nat}. x + 1$)"
 (* Type: nat list ⇒ nat list *)
```

- Type inference determines the type of this function using the arguments.
- Here, it instantiates 'a and 'b to both be nat.

# Higher Order Functions and Type Inference

- **map**: higher order function takes function argument (cf.,  $\lambda x. x + 1$ ).

```
fun map :: "('a ⇒ 'b) ⇒ 'a list ⇒ 'b list" where
 "map f [] = []" | "map f (x # xs) = f x # map f xs"
```

```
value "map ($\lambda x::\text{nat}. x + 1$) (0 # 1 # 2 # [])"
 (* 1#2#3#[] *)
```

- The effect is to apply the argument function to every element of the list.
- We can also **partially** apply **map** to a given function:

```
term "map ($\lambda x::\text{nat}. x + 1$)"
 (* Type: nat list ⇒ nat list *)
```

- **Type inference** determines the type of this function using the arguments.
- Here, it instantiates 'a and 'b to both be nat.

# Higher Order Functions and Type Inference

- **map**: higher order function takes function argument (cf.,  $\lambda x. x + 1$ ).

```
fun map :: "('a ⇒ 'b) ⇒ 'a list ⇒ 'b list" where
 "map f [] = []" | "map f (x # xs) = f x # map f xs"
```

```
value "map ($\lambda x::\text{nat}. x + 1$) (0 # 1 # 2 # [])"
 (* 1#2#3#[] *)
```

- The effect is to apply the argument function to every element of the list.
- We can also **partially** apply **map** to a given function:

```
term "map ($\lambda x::\text{nat}. x + 1$)"
 (* Type: nat list ⇒ nat list *)
```

- **Type inference** determines the type of this function using the arguments.
- Here, it **instantiates** **'a** and **'b** to both be **nat**.

# Records

- Define a new record type with several typed fields:

```
type T1 = T1.T1; T2 = T2.T2; T3 = T3.T3; T4 = T4.T4;
```
- A record is a product type  $T_1 \times T_2 \times \dots$ , but with named selections.
- We can construct a record with the notation  $\{x_1 = v_1, \dots, x_n = v_n\}$ .
- Also support record extension (similar to inheritance).

```
type Person =
 surname : string
 forename : string
```

## • Address books

```
type Employee = Person +
 identifier : int
 salary : float
```

# Records

- Define a new record type with several typed fields:

```
record RT = f1::T1 f2::T2 ... fn::Tn
```

- Isomorphic to a product type  $T_1 \times T_2 \times \dots$ , but with named selectors.
- We can construct a record with the notation  $(f_1 = v_1, f_2 = v_2, \dots)$ .
- Also support **record extension** (similar to inheritance).

```
record Person =
 surname :: string
 forename :: string
```

- Add new fields:

```
record Employee = Person +
 ident :: nat
 paygrade :: nat
```

# Records

- Define a new record type with several typed fields:

```
record RT = f1::T1 f2::T2 ... fn::Tn
```

- Isomorphic to a product type  $T_1 \times T_2 \times \dots$ , but with named selectors.
- We can construct a record with the notation  $(f_1 = v_1, f_2 = v_2, \dots)$ .
- Also support **record extension** (similar to inheritance).

```
record Person =
 surname :: string
 forename :: string
```

- Add new fields:

```
record Employee = Person +
 ident :: nat
 paygrade :: nat
```

# Records

- Define a new record type with several typed fields:

```
record RT = f1::T1 f2::T2 ... fn::Tn
```

- Isomorphic to a product type  $T_1 \times T_2 \times \dots$ , but with named selectors.
- We can construct a record with the notation  $(f_1 = v_1, f_2 = v_2, \dots)$ .
- Also support **record extension** (similar to inheritance).

```
record Person =
 surname :: string
 forename :: string
```

- Add new fields:

```
record Employee = Person +
 ident :: nat
 paygrade :: nat
```

# Records

- Define a new record type with several typed fields:

```
record RT = f1::T1 f2::T2 ... fn::Tn
```

- Isomorphic to a product type  $T_1 \times T_2 \times \dots$ , but with named selectors.
- We can construct a record with the notation  $(f_1 = v_1, f_2 = v_2, \dots)$ .
- Also support **record extension** (similar to inheritance).

```
record Person =
 surname :: string
 forename :: string
```

- Add new fields:

```
record Employee = Person +
 ident :: nat
 paygrade :: nat
```

# Records

- Define a new record type with several typed fields:

```
record RT = f1::T1 f2::T2 ... fn::Tn
```

- Isomorphic to a product type  $T_1 \times T_2 \times \dots$ , but with named selectors.
- We can construct a record with the notation  $(f_1 = v_1, f_2 = v_2, \dots)$ .
- Also support **record extension** (similar to inheritance).

```
record Person =
 surname :: string
 forename :: string
```

- Add new fields:

```
record Employee = Person +
 ident :: nat
 paygrade :: nat
```

# Command Summary

|                                               |                                      |
|-----------------------------------------------|--------------------------------------|
| <code>def</code>                              | define a new type name as a synonym  |
| <code>defconst</code>                         | define a single function or constant |
| <code>check-type</code> and <code>eval</code> | check type and evaluate a term       |
| <code>defstruct</code>                        | define an algebraic datatype         |
| <code>defrec</code>                           | define a recursive function          |
| <code>defrecord</code>                        | define a record type                 |
| <code>defmacro</code>                         | assign optional syntax to a constant |

# Command Summary

- **type\_synonym**      define a new **type name** as a synonym
- **definition**      define a simple **function** or **constant**
- **term** and **value**    check **type** and evaluate a **term**
- **datatype**      define an algebraic **datatype**
- **fun**              define a recursive **function**
- **record**          define a **record** type
- **notation**        assign optional **syntax** to a constant

# Command Summary

- **type\_synonym**      define a new **type name** as a synonym
- **definition**        define a simple **function** or **constant**
- **term** and **value**    check **type** and evaluate a **term**
- **datatype**          define an algebraic **datatype**
- **fun**                define a recursive **function**
- **record**            define a **record** type
- **notation**          assign optional **syntax** to a constant

# Command Summary

- **type\_synonym**      define a new **type name** as a synonym
- **definition**        define a simple **function** or **constant**
- **term** and **value**    check **type** and evaluate a **term**
- **datatype**            define an algebraic **datatype**
- **fun**                    define a recursive **function**
- **record**                define a **record** type
- **notation**             assign optional **syntax** to a constant

# Command Summary

- **type\_synonym**      define a new **type name** as a synonym
- **definition**        define a simple **function** or **constant**
- **term** and **value**    check **type** and evaluate a **term**
- **datatype**            define an algebraic **datatype**
- **fun**                    define a recursive **function**
- **record**                define a **record** type
- **notation**             assign optional **syntax** to a constant

# Command Summary

- **type\_synonym**      define a new **type name** as a synonym
- **definition**        define a simple **function** or **constant**
- **term** and **value**    check **type** and evaluate a **term**
- **datatype**          define an algebraic **datatype**
- **fun**                define a recursive **function**
- **record**            define a **record** type
- **notation**          assign optional **syntax** to a constant

# Command Summary

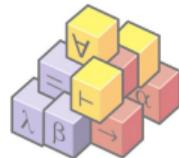
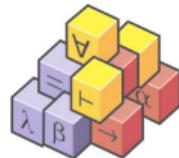
- **type\_synonym**      define a new **type name** as a synonym
- **definition**        define a simple **function** or **constant**
- **term** and **value**    check **type** and evaluate a **term**
- **datatype**          define an algebraic **datatype**
- **fun**                define a recursive **function**
- **record**            define a **record** type
- **notation**          assign optional **syntax** to a constant

# Command Summary

- **type\_synonym**      define a new **type name** as a synonym
- **definition**        define a simple **function** or **constant**
- **term** and **value**    check **type** and evaluate a **term**
- **datatype**          define an algebraic **datatype**
- **fun**                define a recursive **function**
- **record**            define a **record** type
- **notation**         assign optional **syntax** to a constant

# Overview

- 1 HOL as a Functional Programming Language
- 2 Type System
- 3 Algebraic Datatypes
- 4 Recursive Functions
- 5 Code Generator**



# Code Generation

- Isabelle produces code for datatypes and functions in SML, OCaml, Haskell, Scala.
- Tom's Isabelle is a verification tool for functional programs.

```
fun f x => x + 1
val x = 1
val y = f x
val z = f y
val w = f z
```
- This generates code for each of the functions in the target language. Also creates my explicit signature datatypes, etc.

# Code Generation

- Isabelle produces code for datatypes and functions in

SML, OCaml, Haskell, Scala.

- Turns Isabelle into a verification tool for functional programs.

```
export_code <functions>
 in <language>
 module_name <name>
```

- This generates code for each of the functions in the target language.
- Also creates any requisite algebraic datatypes, etc.

# Code Generation

- Isabelle produces code for datatypes and functions in **SML, OCaml, Haskell, Scala**.
- Turns Isabelle into a **verification tool** for functional programs.

```
export_code <functions>
 in <language>
 module_name <name>
```

- This generates code for each of the functions in the target language.
- Also creates any requisite algebraic datatypes, etc.

# Code Generation

- Isabelle produces code for datatypes and functions in  
SML, OCaml, Haskell, Scala.
- Turns Isabelle into a **verification tool** for functional programs.

```
export_code <functions>
 in <language>
 module_name <name>
```

- This generates code for each of the functions in the target language.
- Also creates any requisite algebraic datatypes, etc.

# Code Generation

- Isabelle produces code for datatypes and functions in  
SML, OCaml, Haskell, Scala.
- Turns Isabelle into a **verification tool** for functional programs.

```
export_code <functions>
 in <language>
 module_name <name>
```

- This generates code for each of the functions in the target language.
- Also creates any requisite algebraic datatypes, etc.

# Code Generation

- Isabelle produces code for datatypes and functions in  
SML, OCaml, Haskell, Scala.
- Turns Isabelle into a **verification tool** for functional programs.

```
export_code <functions>
 in <language>
 module_name <name>
```

- This generates code for each of the functions in the target language.
- Also creates any requisite algebraic datatypes, etc.

## Example: Isabelle/HOL $\rightarrow$ Haskell

```
export_code append in Haskell module_name List
```

### Haskell Code

```
module List(List, append) where {
 data List a = Nil | Cons a (List a)

 append :: List a -> List a -> List a
 append Nil ys = ys
 append (Cons x xs) ys = Cons x (append xs ys)
}
```

Code appears in a virtual file system in file browser / Output panel

The code can be compiled with GHC or interpreted with GHCi

## Example: Isabelle/HOL $\rightarrow$ Haskell

```
export_code append in Haskell module_name List
```

### Haskell Code

```
module List(List, append) where {
 data List a = Nil | Cons a (List a)

 append :: List a -> List a -> List a
 append Nil ys = ys
 append (Cons x xs) ys = Cons x (append xs ys)
}
```

Code appears in a virtual file system in the browser (Output panel)

The code can be compiled with GHC or interpreted with GHCi

## Example: Isabelle/HOL $\rightarrow$ Haskell

```
export_code append in Haskell module_name List
```

### Haskell Code

```
module List(List, append) where {
 data List a = Nil | Cons a (List a)

 append :: List a -> List a -> List a
 append Nil ys = ys
 append (Cons x xs) ys = Cons x (append xs ys)
}
```

## Example: Isabelle/HOL $\rightarrow$ Haskell

```
export_code append in Haskell module_name List
```

### Haskell Code

```
module List(List, append) where {
 data List a = Nil | Cons a (List a)

 append :: List a -> List a -> List a
 append Nil ys = ys
 append (Cons x xs) ys = Cons x (append xs ys)
}
```

- Code appears in a **virtual file system** in file browser (Output pane).
- This code can be compiled with **GHC** or interpreted with **GHCi**.

## Example: Isabelle/HOL $\rightarrow$ Haskell

```
export_code append in Haskell module_name List
```

### Haskell Code

```
module List(List, append) where {
 data List a = Nil | Cons a (List a)

 append :: List a -> List a -> List a
 append Nil ys = ys
 append (Cons x xs) ys = Cons x (append xs ys)
}
```

- Code appears in a **virtual file system** in file browser (Output pane).
- This code can be compiled with **GHC** or interpreted with **GHCi**.

# Conclusion

## This Lecture

- Functional programming in Isabelle/HOL.
- Algebraic data types and recursive functions.
- Code generation.

## Next Lecture

- How we can start to prove things about these programs.

# Conclusion

This Lecture

Next Lecture

- How we can start to prove things about these programs.

# Conclusion

## This Lecture

- Functional programming in Isabelle/HOL.
- Algebraic data types and recursive functions.
- Code generation.

## Next Lecture

- How we can start to prove things about these programs.

# Conclusion

## This Lecture

- Functional programming in Isabelle/HOL.
- Algebraic data types and recursive functions.
- Code generation.

## Next Lecture

- How we can start to prove things about these programs.

# Conclusion

## This Lecture

- Functional programming in Isabelle/HOL.
- Algebraic data types and recursive functions.
- Code generation.

## Next Lecture

- How we can start to prove things about these programs.

# Conclusion

## This Lecture

- Functional programming in Isabelle/HOL.
- Algebraic data types and recursive functions.
- Code generation.

## Next Lecture

- How we can start to prove things about these programs.

# Conclusion

## This Lecture

- Functional programming in Isabelle/HOL.
- Algebraic data types and recursive functions.
- Code generation.

## Next Lecture

# Conclusion

## This Lecture

- Functional programming in Isabelle/HOL.
- Algebraic data types and recursive functions.
- Code generation.

## Next Lecture

- How we can start to prove things about these programs.