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

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

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

Fundamental Types of HOL

- Booleans (bool or $\mathbb{B}$): True or False, $\wedge$, $\vee$, $\neg$, $\longrightarrow$.
- Natural numbers (nat or $\mathbb{N}$): $0, 1, 2, 3, 4 \cdots$ 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: $\lambda\, x :: nat.\, x + 1$ has the type $nat \Rightarrow nat$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $fst(x, y) = x$ and $snd(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \cdots \times A_n$.
- Explicitly assign type to term: type coercision, `x :: T`.

Fundamental Types of HOL

- Booleans (bool or $\mathbb{B}$): True or False, $\wedge$, $\vee$, $\neg$, $\longrightarrow$.
- Natural numbers (nat or $\mathbb{N}$): $0, 1, 2, 3, 4 \cdots$ 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: $\lambda\, x :: nat.\, x + 1$ has the type $nat \Rightarrow nat$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $fst(x, y) = x$ and $snd(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \cdots \times A_n$.
- Explicitly assign type to term: type coercision, `x :: T`.

Fundamental Types of HOL

- Booleans (bool or $\mathbb{B}$): True or False, $\wedge$, $\vee$, $\neg$, $\longrightarrow$.
- Natural numbers (nat or $\mathbb{N}$): $0, 1, 2, 3, 4 \cdots$ 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: $\lambda\, x :: nat.\, x + 1$ has the type $nat \Rightarrow nat$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $fst(x, y) = x$ and $snd(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \cdots \times A_n$.
- 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 $\mathbb{N}$): $0, 1, 2, 3, 4 \cdots$ 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: $\lambda\, x :: nat.\, x + 1$ has the type $nat \Rightarrow nat$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $fst(x, y) = x$ and $snd(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \cdots \times A_n$.
- Explicitly assign type to term: type coercision, `x :: T`.

Fundamental Types of HOL

- Booleans (bool or $\mathbb{B}$): True or False, $\wedge$, $\vee$, $\neg$, $\longrightarrow$.
- Natural numbers (nat or $\mathbb{N}$): $0, 1, 2, 3, 4 \cdots$ 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: $\lambda\, x :: nat.\, x + 1$ has the type $nat \Rightarrow nat$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $fst(x, y) = x$ and $snd(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \cdots \times A_n$.
- 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 $\mathbb{N}$): $0, 1, 2, 3, 4 \cdots$ 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: $\lambda\, x :: nat.\, x + 1$ has the type $nat \Rightarrow nat$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $fst(x, y) = x$ and $snd(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \cdots \times A_n$.
- 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 $\mathbb{N}$): $0, 1, 2, 3, 4 \cdots$ 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: $\lambda\, x :: nat.\, x + 1$ has the type $nat \Rightarrow nat$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $fst(x, y) = x$ and $snd(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \cdots \times A_n$.
- Explicitly assign type to term: type coercision, `x :: T`.

Fundamental Types of HOL

- Booleans (bool or $\mathbb{B}$): True or False, $\wedge$, $\vee$, $\neg$, $\longrightarrow$.
- Natural numbers (nat or $\mathbb{N}$): $0, 1, 2, 3, 4 \cdots$ 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: $\lambda\, x :: nat.\, x + 1$ has the type $nat \Rightarrow nat$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $fst(x, y) = x$ and $snd(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \cdots \times A_n$.
- 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 $\mathbb{N}$): $0, 1, 2, 3, 4 \cdots$ 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: $\lambda\, x :: nat.\, x + 1$ has the type $nat \Rightarrow nat$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $fst(x, y) = x$ and $snd(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \cdots \times A_n$.
- Explicitly assign type to term: type coercision, `x :: T`.

Fundamental Types of HOL

- Booleans (bool or $\mathbb{B}$): True or False, $\wedge$, $\vee$, $\neg$, $\longrightarrow$.
- Natural numbers (nat or $\mathbb{N}$): $0, 1, 2, 3, 4 \cdots$ 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: $\lambda\, x :: nat.\, x + 1$ has the type $nat \Rightarrow nat$.
- Pairs ($A \times B$): (x, y) for $x :: A$ and $y :: B$:
- Selectors: $fst(x, y) = x$ and $snd(x, y) = y$.
- Can be nested for n-tuples, e.g. $A_1 \times A_2 \times A_3 \times \cdots \times A_n$.
- Explicitly assign type to term: type coercion, `x :: T`.

Example: Evaluating Terms

```
value "True ∧ False" (* Returns False *)

value "True ∨ False" (* Returns True *)

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

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

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

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

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

Example: Evaluating Terms

```
value "True ∧ False" (* Returns False *)

value "True ∨ False" (* Returns True *)

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

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

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

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

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

Example: Evaluating Terms

```
value "True ∧ False" (* Returns False *)

value "True ∨ False" (* Returns True *)

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

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

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

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

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

Example: Evaluating Terms

```
value "True ∧ False" (* Returns False *)

value "True ∨ False" (* Returns True *)

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

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

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

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

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

Example: Evaluating Terms

```
value "True ∧ False" (* Returns False *)

value "True ∨ False" (* Returns True *)

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

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

value "(λ x::nat. x^2 + 1) 6" (* Returns 37 *)

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

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

Example: Evaluating Terms

```
value "True ∧ False" (* Returns False *)

value "True ∨ False" (* Returns True *)

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

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

value "(λ x::nat. x^2 + 1) 6" (* Returns 37 *)

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

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

Example: Evaluating Terms

```
value "True ∧ False" (* Returns False *)

value "True ∨ False" (* Returns True *)

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

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

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

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

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

Example: Evaluating Terms

```
value "True ∧ False" (* Returns False *)

value "True ∨ False" (* Returns True *)

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

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

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

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

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

Basic Commands

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 etc.
- Instantiated to ground types (i.e., types without variables):

```
bag = " ⇒ nat"
```

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

```
empty_bag :: " bag"
  "empty_bag = (λ x. 0)"

add_bag :: " bag ⇒ 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 ⇒ 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 ⇒ 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)"
```

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.

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

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

```
type 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)`.

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

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

- 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 ⇒ 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 ⇒ 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 ⇒ 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"
```

Overview

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

Recursive Functions

Recursive Functions

- Define a function by recursion over the algebraic datatype T1

```
fun rf :: "T1 ⇒ 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 ⇒ nat ⇒ nat" where
  "plus Zero n = n" |
  "plus (Suc m) n = Suc (plus m n)"
```

- Length of a list:

```
fun len :: "'a list ⇒ 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 ⇒ 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 ⇒ nat ⇒ nat" where
  "plus Zero n = n" |
  "plus (Suc m) n = Suc (plus m n)"
```

- Length of a list:

```
fun len :: "'a list ⇒ 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 ⇒ 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 ⇒ nat ⇒ nat" where
  "plus Zero n = n" |
  "plus (Suc m) n = Suc (plus m n)"
```

- Length of a list:

```
fun len :: "'a list ⇒ 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 ⇒ 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 ⇒ nat ⇒ nat" where
  "plus Zero n = n" |
  "plus (Suc m) n = Suc (plus m n)"
```

- Length of a list:

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

Example: Non-Termination

- This function does not terminate: it's an infinite loop.

```
bad :: "nat ⇒ nat"
  "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 [illegible]

Example: Non-Termination

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

```
fun bad :: "nat ⇒ 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 ⇒ 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 ⇒ 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 ⇒ 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 ⇒ 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

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 precendence; 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 ≡).

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 precendence; 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 ≡).

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 precendence; 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 ≡).

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 precendence; 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 ≡).

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 precendence; 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 ≡).

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 precendence; 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 ≡).

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 precendence; 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 ≡).

Higher Order Functions and Type Inference

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 (λ x::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 (λ x::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 (λ x::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 (λ x::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 (λ x::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 (λ x::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 (λ x::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 (λ x::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 (λ x::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 (λ x::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

Records

- Define a new record type with several typed fields:

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

- Isomorphic to a product type $T_1 \times T_2 \times \cdots$, but with named selectors.
- We can construct a record with the notation $(\!| f_1 = v_1, f_2 = v_2, \cdots |\!)$.
- 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 $T_1 \times T_2 \times \cdots$, but with named selectors.
- We can construct a record with the notation $(\!| f_1 = v_1, f_2 = v_2, \cdots |\!)$.
- 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 $T_1 \times T_2 \times \cdots$, but with named selectors.
- We can construct a record with the notation $(\!| f_1 = v_1, f_2 = v_2, \cdots |\!)$.
- 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 $T_1 \times T_2 \times \cdots$, but with named selectors.
- We can construct a record with the notation $(\!| f_1 = v_1, f_2 = v_2, \cdots |\!)$.
- 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 $T_1 \times T_2 \times \cdots$, but with named selectors.
- We can construct a record with the notation $(\!| f_1 = v_1, f_2 = v_2, \cdots |\!)$.
- 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

- define a new type name as a synonym
- define a simple function or constant
- check type and evaluate a term
- define an algebraic datatype
- define a recursive function
- define a record type
- 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

- 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.

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

```haskell
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 area)
- 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)
}
```

Example: Isabelle/HOL → Haskell

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

Haskell Code

```haskell
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).
- This code can be compiled with GHC or interpreted with GHCi.

Example: Isabelle/HOL → Haskell

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

Haskell Code

```haskell
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 → 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

- 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

- 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.