

Foundations of Certified Programming Language and Compiler Design

Dr.-Ing. Sebastian Ertel

Composable Operating Systems Group, Barkhausen Institute

Outline



Lecture	Logic	Formalisms	PL
1	Propositional and first-order logic		
2			Functional programming
3		Syntax and Semantics	
4			The untyped lambda calculus
5		Types	
6			The typed lambda calculus
7			Polymorphism
8		Curry-Howard	
9			Higher-order types
10			Dependent types

Main Goals



- Introduction to the most fundamental concepts in functional programming using Haskell.
- Show you that these are the very same as in Gallina (Coq).
- Have you write programs in Haskell and Gallina.



- At the heart of every theorem and proof is a functional language.
- Functional languages have a strong mathematical foundation.
- Step 1: Learning functional programming allows you to write programs in
 - Haskell,
 - Coq,
 - Agda, etc.
- Step 2: Learning the foundations of functional languages allows you to
 - Take full advantage of the mathematical foundation and
 - strengthen your programs.
- Haskell¹ was meant for teaching and research.
- Haskell is gaining traction as **the** functional language.

¹Paul Hudak et al. "A history of Haskell: being lazy with class". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. 2007, pp. 12–1.



Two options exist to run the examples from the lecture:

1. Starting the interpreter (GHCi):

```
[-> ~ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/    :? for help
Prelude>
```

2. Compiling (ghc foo.hs) and running (./foo) a Haskell source file:

```
-- your functions (f)/data types are defined here ...
```

```
main = return f -- ... and gets invoked here
```

3. Easiest: Try out the playground <https://play.haskell.org/>
4. There is one for Coq too: <https://coq.vercel.app/>

Values and Functions and Values and ...



Values	Functions
5	
5.6	
"fcp1cd"	
True	
()	unit
	Named function
	Unnamed function
add	
$\backslash x\ y = x + y$	

Haskell is a functional language!

Bindings



Haskell allows to bind values to names for use in a particular scope:

```
let binding = value  
in scope
```

Some examples:

```
let x = 5 in x  
let add = \x y -> x+y in add 2 3  
let add x y = x+y in add 2 3 -- syntactic convenience
```

Function Application



We always apply a function to a **single** value!

Consider the following function application: `add 1 2`

What actually happens is:

```
add 1 2 ≡      (\x y -> x+y) 1 2
           ≡      ((\x y -> x+y) 1) 2 -- application binds stronger
                                           -- to the right
           ≡      let add' = (\x y -> x+y) 1 -- partial application
                     in add' 2
           ≡      let add' = (\x -> (\y -> x+y)) 1
                     in add' 2
           ≡      let add' = (\y -> 1+y)
                     in add' 2
```




Composing Values

Products: a.k.a. tuples $(1, \text{"one"})$ 1 **and** "one"

- Currying¹: $\lambda(x,y) \rightarrow x+y \xrightarrow{\text{curry}} \lambda x \rightarrow \lambda y \rightarrow x+y$
- Comparison with lists: $[1,2,3]$ or $[\text{"one"}, \text{"two"}, \text{"three"}]$ or $\text{"one"} \equiv ['o', 'n', 'e']$

Coproducts a.k.a. sums : $\text{Left } 1$ 1 **or** "one"
 Right "one"

Enough already with the terms!
Haskell is a strongly statically typed language.
So, give me some types!

¹Named after Haskell Curry.

Types, types, types!



Typed values

```
5 :: Int
5.6 :: Float
"test" :: String
True :: Bool
() :: ()
```

Typed terms

```
add :: Int -> Int -> Int
-- Int -> (Int -> Int)
\x y -> x+y :: Int -> Int -> Int

add 2 :: Int -> Int
add 2 3 :: Int
let x = 4 in x :: Int
```

Functions

Applications



Polymorphic Types

- So far, we defined `add` for integers.
- But addition is defined for many types: `Double`, `Float`, `Int`, etc.
- We want `add` to provide addition for all types that can be added.

In order to **generalize** over a particular type we need **type variables**.

```
add :: (Num x) => x -> x -> x
add      x y =    x + y
```

(Type classes abstract over functions with the same type.)

Example terms: `add 5 5`, `add 4.5 5.5`, ...

How about this one: `add 3.5 5`?

Composing Types



Products	<code>(1, "one")</code>	<code>:: (Int, String)</code>
Coproducts	<code>Left 1</code>	<code>:: Either Int String</code>
	<code>Right "one"</code>	<code>:: Either Int String</code>

Algebraic Data Types (ADTs)



basic:

```
data Bool = True
          | False
```

polymorphic:

```
data Either a b = Left a
                | Right b
```

recursive:

```
data List a = Nil
            | Cons a (List a)
```



Pattern Matching

```
zeros :: List Int -> Int
```

```
zeros Nil          = 0
zeros (Cons x xs)  = ( if x == 0
                        then 1
                        else 0 ) + zeros xs
```

... is syntactic sugar for ...

```
zeros l = case l of
  Nil      -> 0
  Cons x xs -> ( case x == 0 of
                  True  -> 1
                  False -> 0 ) + zeros xs
```



Searching Types instead of Names

- Haskell has a very clear distinction between terms and types:

`add :: Int -> Int -> Int` type level

`add x y = x+y` term level

- In fact, types supersede names: Search for functionality; forget about names!¹²

```
[-> ~ hoogle search --count=5 "Int -> Int -> Int"
GHC.Arr badSafeIndex :: Int -> Int -> Int
GHC.Base quotInt :: Int -> Int -> Int
GHC.Base remInt :: Int -> Int -> Int
GHC.Base divInt :: Int -> Int -> Int
GHC.Base modInt :: Int -> Int -> Int
```

¹Mikael Rittri. "Using Types as Search Keys in Function Libraries". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, 174–183.

²Colin Runciman and Ian Toyn. "Retrieving Re-Usable Software Components by Polymorphic Type". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, 166–173.



Searching Types instead of Names

- Haskell has a very clear distinction between terms and types:

`add :: Int -> Int -> Int` type level

`add x y = x+y` term level

- In fact, types supersede names: Search for functionality; forget about names!¹²

```
[-> ~ hoogle search --count=5 "a -> a -> a"
Prelude asTypeOf :: a -> a -> a
GHC.Base asTypeOf :: a -> a -> a
GHC.IO.SubSystem conditional :: a -> a -> a
GHC.IO.SubSystem (<!>) :: a -> a -> a
Data.ByteString.Builder.Prim.Internal caseWordSize_32_64 :: a -> a -> a
```

¹[Mikael Rittri](#). "Using Types as Search Keys in Function Libraries". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, 174–183.

²[Colin Runciman and Ian Toyn](#). "Retrieving Re-Usable Software Components by Polymorphic Type". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London,



Searching Types instead of Names

- Haskell has a very clear distinction between terms and types:

`add :: Int -> Int -> Int` type level

`add x y = x+y` term level

- In fact, types supersede names: Search for functionality; forget about names!¹²

```
[-> ~ hoogle search --count=5 "Num a => a -> a -> a"]
```

```
Prelude (+) :: Num a => a -> a -> a
```

```
Prelude (-) :: Num a => a -> a -> a
```

```
Prelude (*) :: Num a => a -> a -> a
```

```
Prelude subtract :: Num a => a -> a -> a
```

```
GHC.Num (+) :: Num a => a -> a -> a
```

¹[Mikael Rittri](#). "Using Types as Search Keys in Function Libraries". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, 174–183.

²[Colin Runciman](#) and [Ian Toyn](#). "Retrieving Re-Usable Software Components by Polymorphic Type". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, 166–170.

Searching Types instead of Names



- Haskell has a very clear distinction between terms and types:

`add :: Int -> Int -> Int` type level

`add x y = x+y` term level

- In fact, types supersede names: Search for functionality; forget about names!¹²

Check out Coq's `Search` command!

¹Mikael Rittri. "Using Types as Search Keys in Function Libraries". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, 174–183.

²Colin Runciman and Ian Toyn. "Retrieving Re-Usable Software Components by Polymorphic Type". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, 166–173.



From Haskell to Coq

syntactic construct

values

types

bindings

unnamed functions

named functions

function application

algebraic data types

pattern matching

recursive functions

Haskell

```
5 :: Int
add :: Int -> Int -> Int
let ident = term in scope
\x y -> x + y
add x y = x + y
add 1 2
data Bool = True | False
data List = Nil | Cons a (List a)
case b of
  True -> 1
  False -> 0
zeros :: List Int -> Int
```

Gallina (Coq)

```
Coq.ZArith.Int
add : Z -> Z -> Z
let ident := term in scope
fun (x y : Z) => x + y
Definition add (x y : Z) : Z := x + y.
add 1 2
Inductive bool : Set := True | False.
Inductive list (A:Set) : Set := Nil | Cons A (list A).
match b with
| True -> 1
| False -> 0 end
Fixpoint zeros (l:list Z) : Z :=
...
```

Functions in Coq need to be total!



Type Checking vs. Type Inference

“Well-typed programs cannot go wrong!”¹

Type checking The algorithm to **verify** that the program **preserves** its type during execution(/evaluation).

Type inference The algorithm to find the *principal type* for programs of a polymorphic type system.

```
Int -> Int -> Int
Float -> Float -> Float
(Num a) => a -> a -> a  principle type
```

- The focus of this lecture is on type checking.

¹Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.

Outline



Lecture	Logic	Formalisms	PL
1	Propositional and first-order logic		
2			Functional programming
3		Syntax and Semantics	
4			The untyped lambda calculus
5		Types	
6			The typed lambda calculus
7			Polymorphism
8		Curry-Howard	
9			Higher-order types
10			Dependent types

Main Goals



As a foundation to talk about logic and programming languages, we will learn

- a formal (meta-)language to define the syntax of a language.



Definition (Well-formedness)

Let \mathcal{T} be the set of all terms of a language then every $t \in \mathcal{T}$ is well-formed, i.e., t is a term of the language.

- We study a language for simple arithmetic.
- Terms include:
 $0, \text{succ } 0, \text{succ}(\text{succ } 0), (\text{succ } 0) + 0, \dots$



This is the most common form of defining the syntax of a language.

v	\in	$\{0\}$	values (a.k.a. constants)
t	$::=$		terms:
		v	values
		$\text{succ } t$	successor
		$t_1 + t_2$	addition



Definition (Terms by Induction)

The set of terms is the smallest set \mathcal{T} such that

1. $\{0\} \subseteq \mathcal{T}$;
2. if $t \in \mathcal{T}$ then $\text{succ } t \in \mathcal{T}$;
3. if $t_1 \in \mathcal{T}$ and $t_2 \in \mathcal{T}$ then $t_1 + t_2 \in \mathcal{T}$.



Definition (Terms by Inference Rules)

The set of terms is defined by the following inference rules:

$$\frac{}{0 \in \mathcal{T}} \quad \frac{t \in \mathcal{T}}{\text{succ } t \in \mathcal{T}} \quad \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T}}{t_1 + t_2 \in \mathcal{T}}$$

- Inference rules are the de-facto standard to define the type system and the semantics of programming languages.
- We will make heavy use of them throughout this lecture.



For completeness, there is also the concrete representation of terms:

Definition (Terms, Concretely)

For each natural number i , define a set S_i as follows

$$\begin{aligned} S_0 &= \emptyset \\ S_{i+1} &= \{0\} \cup \\ &\quad \{\text{succ } t \mid t \in S_i\} \cup \\ &\quad \{t_1 + t_2 \mid t_1, t_2 \in S_i\} \end{aligned}$$

Finally, let $S = \bigcup_i S_i$.



- Let's connect the concrete syntax definitions to the (future) implementations of programming languages and compilers that we wish to write.

- For compiler authors think:

Concrete Syntax is what the parser sees.

Abstract Syntax is what the parser emits, i.e., an internal representation (IR) such as an AST.

```
Inductive term : Set :=
| Zero: term           values
| Succ: term -> term    unary function
| Add: term -> term -> term. binary function
```

- The according inference rules are:

$$\frac{}{\text{Zero} \in \text{term}} \quad \frac{t \in \text{term}}{\text{Succ } t \in \text{term}} \quad \frac{t_1 \in \text{term} \quad t_2 \in \text{term}}{\text{Add } t_1 \ t_2 \in \text{term}}$$

What we have learned today



- Our formal meta-language:
 - BNF
 - Inference rules
- An algebraic/inductive data type is the direct connection between concrete and abstract syntax.
- Hence, a function `eval` is an interpreter of a term ... for now.



So far, we used inference rules to specify:

- the syntax of terms

$$\frac{}{0 \in \mathcal{T}} \quad \frac{t_1 \in \mathcal{T}}{\text{succ } t_1 \in \mathcal{T}} \quad \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T}}{t_1 + t_2 \in \mathcal{T}}$$

- the “preservation” of a predicate

$$\frac{}{P(\text{Zero})} \quad \frac{t_1 \in \text{Term} \quad P(t_1)}{P(\text{Succ } t_1)} \quad \frac{t_1 \in \text{Term} \quad P(t_1) \quad t_2 \in \text{Term} \quad P(t_2)}{P(\text{Add } t_1 \ t_2)}$$

Inference Rules Recap

- That is, we used inference rules to specify relations:
- the relation \in of terms(/words) t and the set of all terms \mathcal{T}

$$t \in \mathcal{T}$$

$$\frac{}{0 \in \mathcal{T}} \quad \frac{t_1 \in \mathcal{T}}{\text{succ } t_1 \in \mathcal{T}} \quad \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T}}{t_1 + t_2 \in \mathcal{T}}$$

- the property P of terms t

$$P(t)$$

$$\frac{}{P(\text{Zero})} \quad \frac{t_1 \in \text{term} \quad P(t_1)}{P(\text{Succ } t_1)} \quad \frac{t_1 \in \text{term} \quad P(t_1) \quad t_2 \in \text{term} \quad P(t_2)}{P(\text{Add } t_1 \ t_2)}$$

- A function is a type of relation that associates one input with exactly one output.
- A binary relation is a set of (ordered) pairs where one input maybe related to more than one output.
- Here is one important relation for the definition of programming languages:



Mathematically: For programming languages such a relation is the evaluation of a term:

Definition (One-Step Evaluation Relation)

The *one-step* evaluation relation \longrightarrow is the smallest binary relation that relates a term t of a language to another term t' .

Logically: This relation defines a term rewriting system.

Semantically: We talk about evaluating a term, i.e., we reduce it to another term of a smaller size.

Example: Booleans



Syntax

$t ::=$
| true
| false
| if t then t else t

terms:
constant true
constant false
conditional

$v ::=$
| true true value
| false false value

Evaluation

$\frac{}{\text{if true then } t_2 \text{ else } t_3 \longrightarrow t_2} \text{E-IFTRUE}$

$\frac{}{\text{if false then } t_2 \text{ else } t_3 \longrightarrow t_3} \text{E-IFFALSE}$

$\frac{t_1 \longrightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3} \text{E-IF}$

Computations with Booleans

Derivation Trees



$s \stackrel{\text{def}}{=} \text{if true then false else false}$
 $t \stackrel{\text{def}}{=} \text{if } s \text{ then true else true}$
 $u \stackrel{\text{def}}{=} \text{if false then true else true}$

$$\frac{\frac{\text{if true then false else false} \longrightarrow \text{false}}{\text{if } s \text{ then true else true} \longrightarrow \text{if false then true else true}} \text{E-IfTrue} \quad \text{E-If}$$
$$\frac{\text{if } t \text{ then false else false} \longrightarrow \text{if } u \text{ then false else false}}{\text{if } t \text{ then false else false} \longrightarrow \text{if } u \text{ then false else false}} \text{E-If}$$



Theorem (Determinacy of One-Step Evaluation)

If $t \longrightarrow t'$ and $t \longrightarrow t''$ then $t' = t''$.

Proof.

The proof is by induction **on the derivation** $t \longrightarrow t'$, i.e., on the structure of t .

Case

$t \longrightarrow t'$

$t \longrightarrow t''$

E-IfTRUE

$$\frac{\vdots}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow t_2} \text{ E-IfTRUE}$$

where: $t_1 = \text{true}$

E-IfFALSE:

$t_1 = \text{false} \neq t_1 = \text{true}$

E-If:

$t_1 \longrightarrow t'_1$ but $t_1 = \text{true}$ such that $\text{true} \longrightarrow ???$

E-IfFALSE

analogous with $t_1 = \text{false}$

E-If

E-If :

By induction hypothesis $t'_1 = t''_1$
for $t_1 \longrightarrow t'_1$ and $t_1 \longrightarrow t''_1$