barkhausen institut

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.

Haskell

- 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
  - strenghten your programs.

- Haskell[1] was meant for teaching and research.
- Haskell is gaining traction as **the** functional language.

[1] Paul 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.

Running Haskell

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:

```haskell
-- 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`		
`"fcplcd"`		
`True`		
`()`		unit

Values and Functions and Values and . . .

Values	Functions	
`5`		
`5.6`		
`"fcplcd"`		
`True`		
`()`		unit
	`add x y = x + y`	Named function

Values and Functions and Values and ...

Values	Functions	
`5`		
`5.6`		
`"fcplcd"`		
`True`		
`()`		unit
	`add x y = x + y`	Named function
	`\x y = x + y`	Unnamed function

Values and Functions and Values and . . .

Values	Functions	
`5`		
`5.6`		
`"fcplcd"`		
`True`		
`()`		unit
	`add x y = x + y`	Named function
	`\x y = x + y`	Unnamed function
`add`		
`\x y = x + y`		

Haskell is a functional language!

Bindings

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

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

Bindings

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

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

Some examples:

```haskell
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
```

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
```

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
```

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
```

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,"one")` `1` **and** `""`

[1] Named after Haskell Curry.

Composing Values

Products: a.k.a. tuples `(1,"one")` `1` **and** `""`

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

[1] Named after Haskell Curry.

Composing Values

Products: a.k.a. tuples `(1,"one")` `1` **and** `""`

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

Coproducts a.k.a. sums : `Left 1` / `Right "one"` `1` **or** `"one"`

[1]Named after Haskell Curry.

Composing Values

Products: a.k.a. tuples `(1,"one")` `1` **and** `""`

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

Coproducts a.k.a. sums : `Left 1` `Right "one"` `1` **or** `"one"`

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

[1] Named after Haskell Curry.

Types, types, types!

Typed values

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

Types, types, types!

Typed values

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

Typed terms

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

Functions

Types, types, types!

Typed values

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

Typed terms

Functions

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

Applications

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

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 ::          x -> x -> x
add      x y =  x + y
```

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

How about this one: `add 3.5 5`?

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	`(1,"one")`	`::`	`(Int,String)`
Coproducts	`Left 1`	`::`	`Either Int String`
	`Right "one"`	`::`	`Either Int String`

Algebraic Data Types (ADTs)

basic:

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

Algebraic Data Types (ADTs)

basic:

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

polymorphic:

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

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
```

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 ...

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:

```haskell
add :: Int -> Int -> Int       type level
add x y = x+y                  term level
```

- In fact, types supersede names: Seach for functionality; forget about names![1][2]

[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, 166–173.

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: Seach for functionality; forget about names![1][2]

```
[-> ~ 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
```

[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, 166–173.

Searching Types instead of Names

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

```haskell
add :: Int -> Int -> Int        -- type level
add x y = x+y                   -- term level
```

type level

term level

- In fact, types supersede names: Seach for functionality; forget about names![1][2]

```
[-> ~ 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:

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

```haskell
add x y = x+y
```
term level

- In fact, types supersede names: Seach for functionality; forget about names![1][2]

```
[-> ~ 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,

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: Seach for functionality; forget about names![1][2]
  Check out Coq's `Search` command!

[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, 166–173.

From Haskell to Coq

syntactic construct	**Haskell**	**Gallina (Coq)**

From Haskell to Coq

syntactic construct	Haskell	Gallina (Coq)
values	`5::Int`	`Coq.ZArith.Int`

From Haskell to Coq

syntactic construct	**Haskell**	**Gallina (Coq)**
values	`5::Int`	`Coq.ZArith.Int`
types	`add :: Int -> Int -> Int`	`add : Z -> Z -> Z`

From Haskell to Coq

syntactic construct	**Haskell**	**Gallina (Coq)**
values	`5::Int`	`Coq.ZArith.Int`
types	`add :: Int -> Int -> Int`	`add : Z -> Z -> Z`
bindings	`let ident = term in scope`	`let ident := term in scope`

From Haskell to Coq

syntactic construct	**Haskell**	**Gallina (Coq)**
values	`5::Int`	`Coq.ZArith.Int`
types	`add :: Int -> Int -> Int`	`add : Z -> Z -> Z`
bindings	`let ident = term in scope`	`let ident := term in scope`
unnamed functions	`\x y -> x + y`	`fun (x y : Z) => x + y`

From Haskell to Coq

syntactic construct	**Haskell**	**Gallina (Coq)**
values	`5::Int`	`Coq.ZArith.Int`
types	`add :: Int -> Int -> Int`	`add : Z -> Z -> Z`
bindings	`let ident = term in scope`	`let ident := term in scope`
unnamed functions	`\x y -> x + y`	`fun (x y : Z) => x + y`
named functions	`add x y = x + y`	`Definition add (x y : Z) : Z := x + y.`

From Haskell to Coq

syntactic construct	**Haskell**	**Gallina (Coq)**
values	`5::Int`	`Coq.ZArith.Int`
types	`add :: Int -> Int -> Int`	`add : Z -> Z -> Z`
bindings	`let ident = term in scope`	`let ident := term in scope`
unnamed functions	`\x y -> x + y`	`fun (x y : Z) => x + y`
named functions	`add x y = x + y`	`Definition add (x y : Z) : Z := x + y.`
function application	`add 1 2`	`add 1 2`

From Haskell to Coq

syntactic construct	**Haskell**	**Gallina (Coq)**
values	`5::Int`	`Coq.ZArith.Int`
types	`add :: Int -> Int -> Int`	`add : Z -> Z -> Z`
bindings	`let ident = term in scope`	`let ident := term in scope`
unnamed functions	`\x y -> x + y`	`fun (x y : Z) => x + y`
named functions	`add x y = x + y`	`Definition add (x y : Z) : Z := x + y.`
function application	`add 1 2`	`add 1 2`
algebraic data types	`data Bool = True | False`	`Inductive bool : Set := True | False.`

From Haskell to Coq

syntactic construct	**Haskell**	**Gallina (Coq)**
values	`5::Int`	`Coq.ZArith.Int`
types	`add :: Int -> Int -> Int`	`add : Z -> Z -> Z`
bindings	`let ident = term in scope`	`let ident := term in scope`
unnamed functions	`\x y -> x + y`	`fun (x y : Z) => x + y`
named functions	`add x y = x + y`	`Definition add (x y : Z) : Z := x + y.`
function application	`add 1 2`	`add 1 2`
algebraic data types	`data Bool = True | False`	`Inductive bool : Set := True | False.`
	`data List = Nil | Cons a (List a)`	`Inductive list (A:Set) : Set := Nil | Cons A (list A).`

From Haskell to Coq

syntactic construct	**Haskell**	**Gallina (Coq)**
values	`5::Int`	`Coq.ZArith.Int`
types	`add :: Int -> Int -> Int`	`add : Z -> Z -> Z`
bindings	`let ident = term in scope`	`let ident := term in scope`
unnamed functions	`\x y -> x + y`	`fun (x y : Z) => x + y`
named functions	`add x y = x + y`	`Definition add (x y : Z) : Z := x + y.`
function application	`add 1 2`	`add 1 2`
algebraic data types	`data Bool = True | False`	`Inductive bool : Set := True | False.`
	`data List = Nil | Cons a (List a)`	`Inductive list (A:Set) : Set := Nil | Cons A (list A).`

pattern matching	`case b of`
	`  True -> 1`
	`  False -> 0`
		`match b with`
		`| True -> 1`
		`| False -> 0 end`

From Haskell to Coq

syntactic construct	**Haskell**	**Gallina (Coq)**
values	`5::Int`	`Coq.ZArith.Int`
types	`add :: Int -> Int -> Int`	`add : Z -> Z -> Z`
bindings	`let ident = term in scope`	`let ident := term in scope`
unnamed functions	`\x y -> x + y`	`fun (x y : Z) => x + y`
named functions	`add x y = x + y`	`Definition add (x y : Z) : Z := x + y.`
function application	`add 1 2`	`add 1 2`
algebraic data types	`data Bool = True | False`	`Inductive bool : Set := True | False.`
	`data List = Nil | Cons a (List a)`	`Inductive list (A:Set) : Set := Nil | Cons A (list A).`
pattern matching	`case b of`	`match b with`
	`True -> 1`	`| True -> 1`
	`False -> 0`	`| False -> 0 end`
recursive functions	`zeros :: List Int -> Int`	`Fixpoint zeros (l:list Z) : Z : =`
		`...`

From Haskell to Coq

syntactic construct	**Haskell**	**Gallina (Coq)**
values	`5::Int`	`Coq.ZArith.Int`
types	`add :: Int -> Int -> Int`	`add : Z -> Z -> Z`
bindings	`let ident = term in scope`	`let ident := term in scope`
unnamed functions	`\x y -> x + y`	`fun (x y : Z) => x + y`
named functions	`add x y = x + y`	`Definition add (x y : Z) : Z := x + y.`
function application	`add 1 2`	`add 1 2`
algebraic data types	`data Bool = True | False`	`Inductive bool : Set := True | False.`
	`data List = Nil | Cons a (List a)`	`Inductive list (A:Set) : Set := Nil | Cons A (list A).`

pattern matching	`case b of`
	`True -> 1`
	`False -> 0`
		`match b with`
		`| True -> 1`
		`| False -> 0 end`

recursive functions	`zeros :: List Int -> Int`	`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]

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

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.

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

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.

[1] Robin 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.

The Syntax of Terms

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, $\texttt{succ}\ 0$, $\texttt{succ}\ (\texttt{succ}\ 0)$, $(\texttt{succ}\ 0) + 0$, $\ldots$

Concrete Syntax
BNF

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

$$
\begin{array}{lcll}
v & \in & \{0\} & \text{values (a.k.a. constants)} \\
t & ::= & & \textbf{terms:} \\
 & | & v & \text{values} \\
 & | & \texttt{succ}\ t & \text{successor} \\
 & | & t_1 + t_2 & \text{addition}
\end{array}
$$

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 $\texttt{succ}\ t \in \mathcal{T}$;
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:

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

Concrete Syntax
Inference Rules

Definition (Terms by Inference Rules)

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

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

- 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

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

Finally, let $S = \bigcup_i S_i$.

Abstract Syntax

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

Abstract Syntax

- 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:

Abstract Syntax

- 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

- 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.

Abstract Syntax

- 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 :=`	a type for terms
`| Zero: term`	values
`| Succ: term -> term`	unary function
`| Add: term -> term -> term.`	binary function

Abstract Syntax

- 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 :=`	a type for terms
`| Zero: term`	values
`| Succ: term -> term`	unary function
`| Add: term -> term -> term.`	binary function

- The according inference rules are:

Abstract Syntax

- 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 :=`	a type for terms
`| Zero: term`	values
`| Succ: term -> term`	unary function
`| Add: term -> term -> term.`	binary function

- The according inference rules are:

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

What we have learned today

- Our formal meta-language:

What we have learned today

- Our formal meta-language:
  - BNF

What we have learned today

- Our formal meta-language:
  - BNF
  - Inference rules

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.

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.

Inference Rules Recap

So far, we used inference rules to specify:

Inference Rules Recap

So far, we used inference rules to specify:

- the syntax of terms

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

Inference Rules Recap

So far, we used inference rules to specify:

- the syntax of terms

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

- the "preservation" of a predicate

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

Inference Rules Recap

- That is, we used inference rules to specify relations:

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 $\mathcal{T}$

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

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

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 $\mathcal{T}$

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

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

- the property P of terms t

$$\boxed{P(t)}$$

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

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 $\mathcal{T}$

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

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

- the property P of terms t

$$\boxed{P(t)}$$

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

- A function is a type of relation that associates one input with exactly one output.

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 $\mathcal{T}$

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

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

- the property P of terms t

$$\boxed{P(t)}$$

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

- 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.

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 $\mathcal{T}$

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

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

- the property P of terms t

$$\boxed{P(t)}$$

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

- 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:

Evaluation

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'.

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	$::=$		terms:
	$\mid$	`true`	constant true
	$\mid$	`false`	constant false
	$\mid$	`if` t `then` t `else` t	conditional

v	$::=$		values:
	$\mid$	`true`	true value
	$\mid$	`false`	false value

Example: Booleans

Syntax

t	::=		terms:
	|	`true`	constant true
	|	`false`	constant false
	|	`if` t `then` t `else` t	conditional

v	::=		values:
	|	`true`	true value
	|	`false`	false value

Evaluation

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

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

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}\ \text{E-If}$$

Computations with Booleans

Derivation Trees

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

Computations with Booleans

Derivation Trees

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

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

Computations with Booleans

Derivation Trees

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

Computations with Booleans

Derivation Trees

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

$$\frac{}{\texttt{if } t \texttt{ then false else false} \longrightarrow \texttt{if } u \texttt{ then false else false}}\ \text{E-IF}$$

Computations with Booleans

Derivation Trees

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

$$\cfrac{\cfrac{}{\texttt{if } s \texttt{ then true else true} \longrightarrow \texttt{if false then true else true}}\ \textsc{E-If}}{\texttt{if } t \texttt{ then false else false} \longrightarrow \texttt{if } u \texttt{ then false else false}}\ \textsc{E-If}$$

Computations with Booleans

Derivation Trees

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

$$\dfrac{\dfrac{\dfrac{}{\texttt{if true then false else false} \longrightarrow \texttt{false}}\ \textsc{E-IfTrue}}{\texttt{if } s \texttt{ then true else true} \longrightarrow \texttt{if } \texttt{false} \texttt{ then true else true}}\ \textsc{E-If}}{\texttt{if } t \texttt{ then false else false} \longrightarrow \texttt{if } u \texttt{ then false else false}}\ \textsc{E-If}$$

Determinism

Theorem (Determinacy of One-Step Evaluation)

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

Determinism

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.

Determinism

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	$t \longrightarrow t'$	$t \longrightarrow t''$

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 $\quad t \longrightarrow t' \quad t \longrightarrow t''$

E-IfTrue

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

where: $t_1 = \texttt{true}$

E-IfFalse: $t_1 = \texttt{false} \neq t_1 = \texttt{true}$

E-If: $t_1 \longrightarrow t_1'$ but $t_1 = \texttt{true}$ such that $\texttt{true} \longrightarrow$???

Determinism

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	$$\dfrac{\vdots}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow t_2}\ \text{E-IfTrue}$$ where: $t_1 = \texttt{true}$	E-IfFalse: E-If:	$t_1 = \texttt{false} \neq t_1 = \texttt{true}$ $t_1 \longrightarrow t_1'$ but $t_1 = \texttt{true}$ such that $\texttt{true} \longrightarrow$???
E-IfFalse	analogous with $t_1 = \texttt{false}$		

Determinism

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	$$\dfrac{\vdots}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow t_2}\ \text{E-IfTrue}$$ where: $t_1 = \texttt{true}$	E-IfFalse:	$t_1 = \texttt{false} \neq t_1 = \texttt{true}$
		E-If:	$t_1 \longrightarrow t_1'$ but $t_1 = \texttt{true}$ such that $\texttt{true} \longrightarrow$???
E-IfFalse	analogous with $t_1 = \texttt{false}$		
E-If		E-If :	By induction hypothesis $t_1' = t_1''$ for $t_1 \longrightarrow t_1'$ and $t_1 \longrightarrow t_1''$