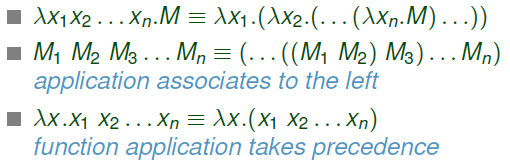
IA014 AFP – excerpts from the slides[[1]](#footnote-1)

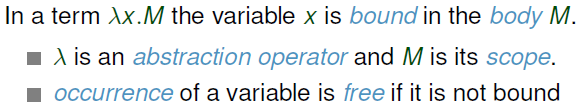
The important stuff from the slides that is not in the cheat sheet

# Untyped Lambda calculus

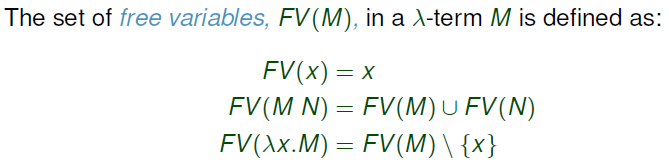
## Syntactic conventions



## Variable binding

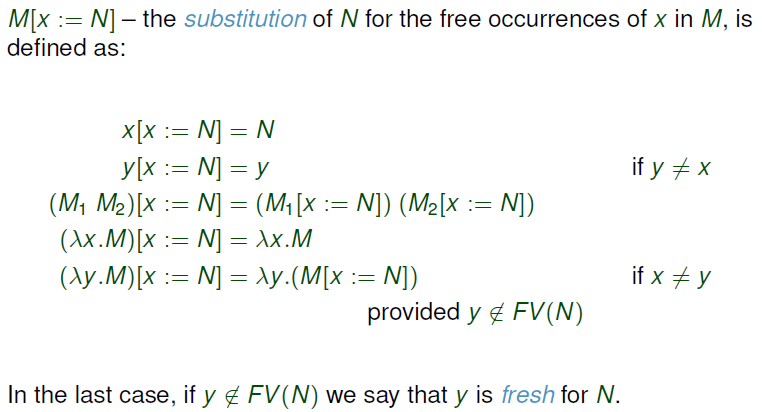


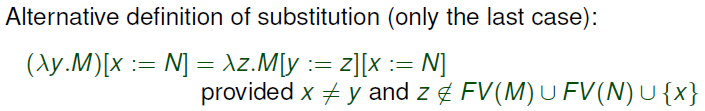
### Formally



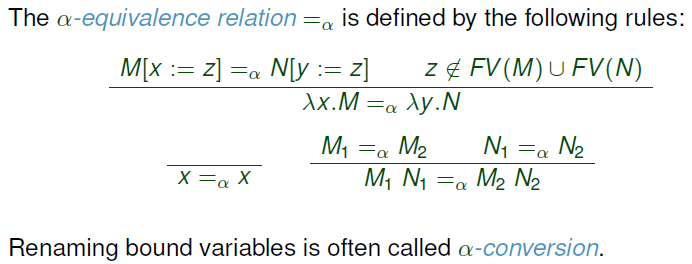


## Substitution

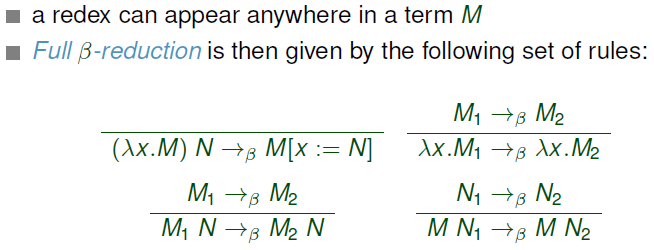




## Alpha-equivalence and Alpha-conversion



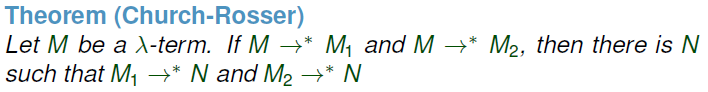
## Full Beta-reduction



### Beta-normal form



### Confluence



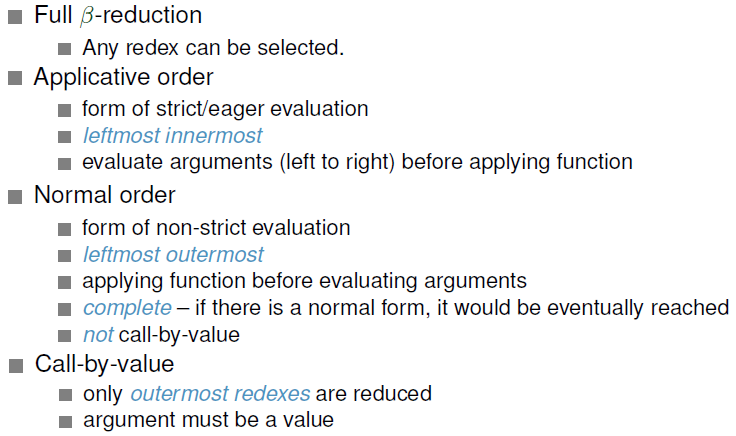


### Non-termination (strong normalizing property)

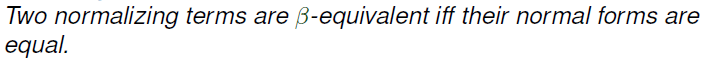
A term is normalizable iff its evaluation is guaranteed to halt in a finite number of steps. A Beta-reduction does not always terminate, therefore untyped Lambda calculus is not strongly normalizing.

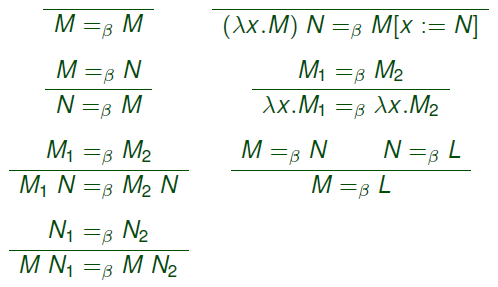
## Evaluation strategies

The choice of an evaluation (reduction) strategy matters because some strategies will evaluate a term infinitely while other strategies can evaluate the same term to a normal form in one step.

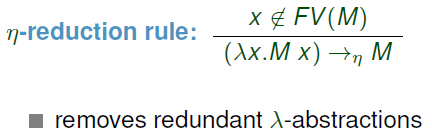


## Beta-equivalence



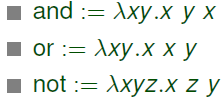


## Eta-reduction

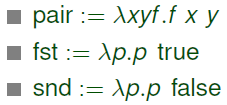


## Encoding mathematics in Lambda calculus

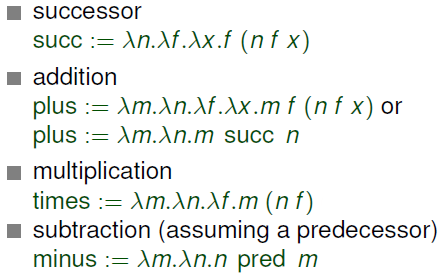




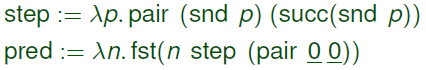
### Pairs



### Arithmetic operators



### Predecessor function



### Exercises from the slides (taken from Wikipedia[[2]](#footnote-2))

#### Multiplication using plus

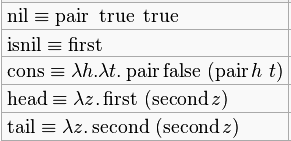
### 

Exponentiation

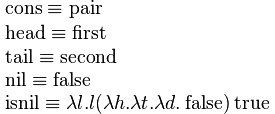


#### Lists

##### Version 1



###### Version 2



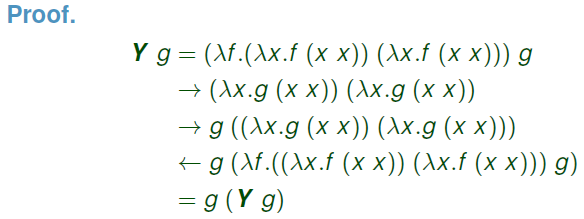
## Recursion

### Fixed point combinators

#### Y combinator (Church)

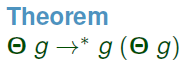


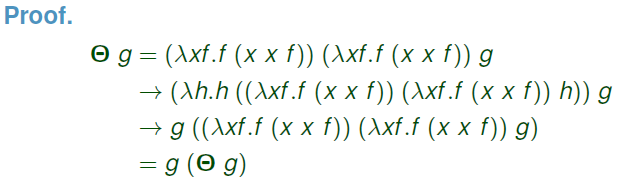




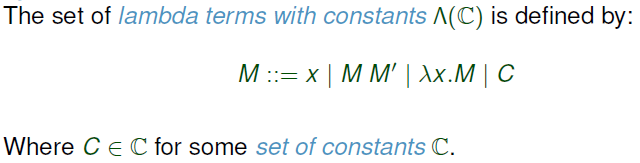
#### Theta combinator (Turing)



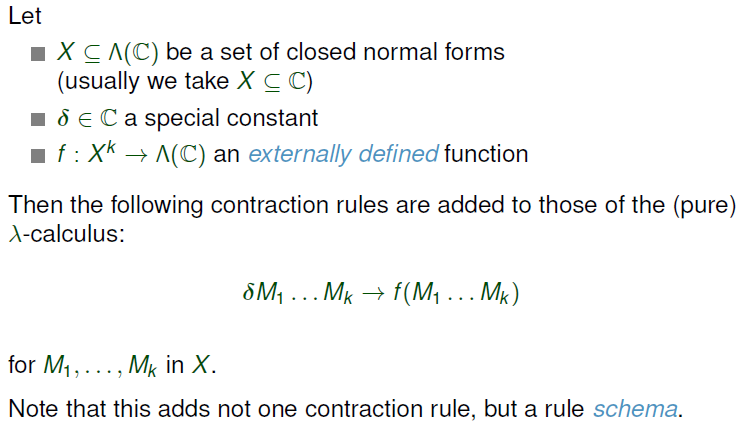


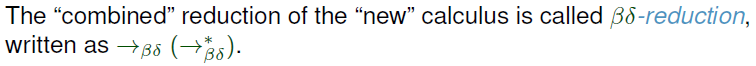


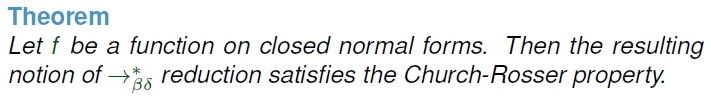
## From Lambda calculus to functional programming I



### Delta-reduction







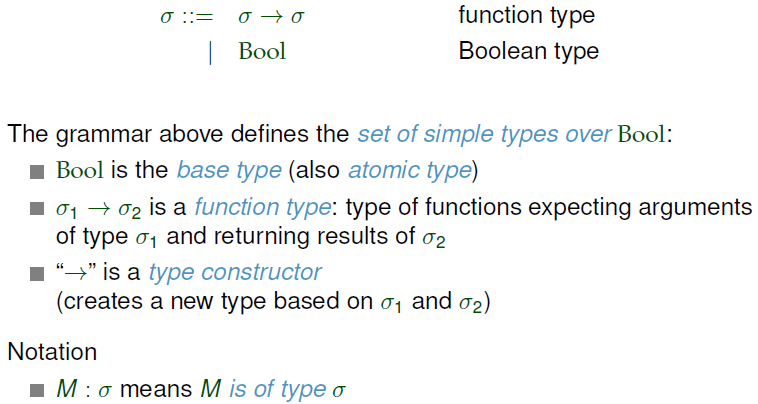
### Syntactic sugar

Programming language construct which can be removed from the language without any effect on functionality or expressive power.

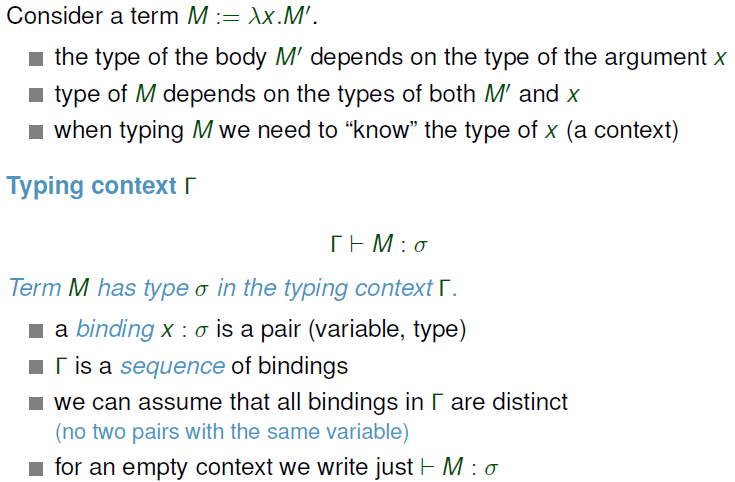
For example infix notation, local declarations (let) etc.

# Simply typed Lambda calculus

## Basic type terminology



## Typing relation, context



## Well-typed terms

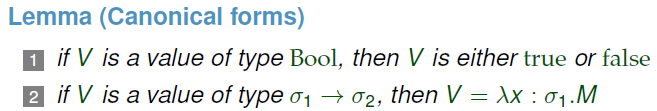


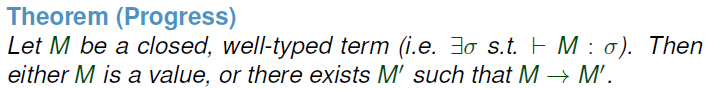
## Type safety (soundness)

Well-typed terms do not “go wrong” (e.g. do not get stuck).

Safety = progress + preservation.

### Progress





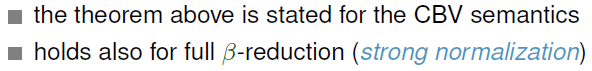
### Preservation

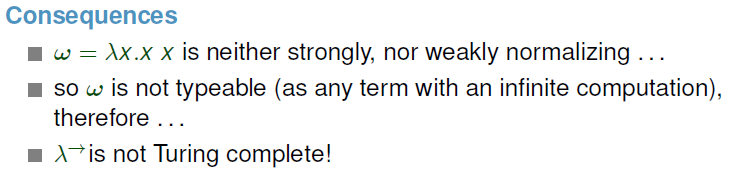




## Normalization







## From Lambda calculus to functional programming II

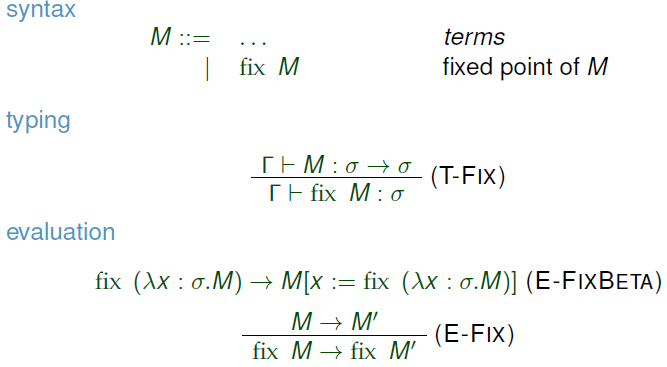
### Recursion

#### Problem

Fixed point combinators are not typable in Simply typed Lambda calculus.

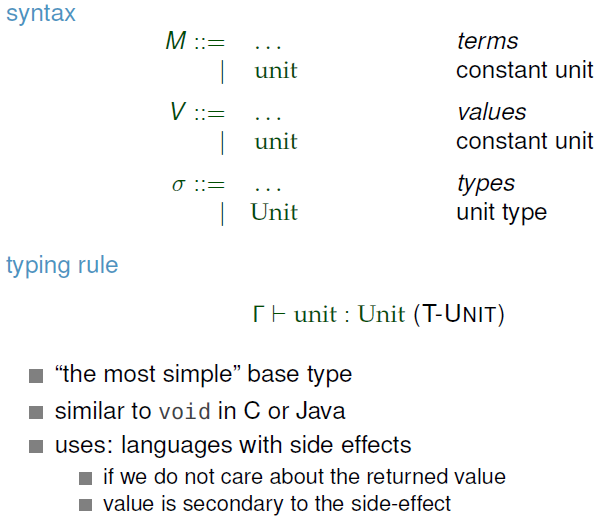
#### Solution

Extend Simply typed Lambda calculus with a new primitive *fix*.

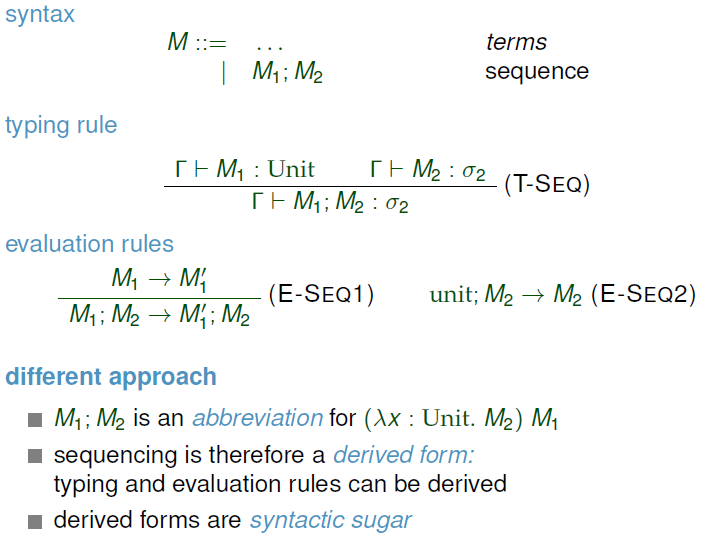


Adding *fix* breaks strong normalization.

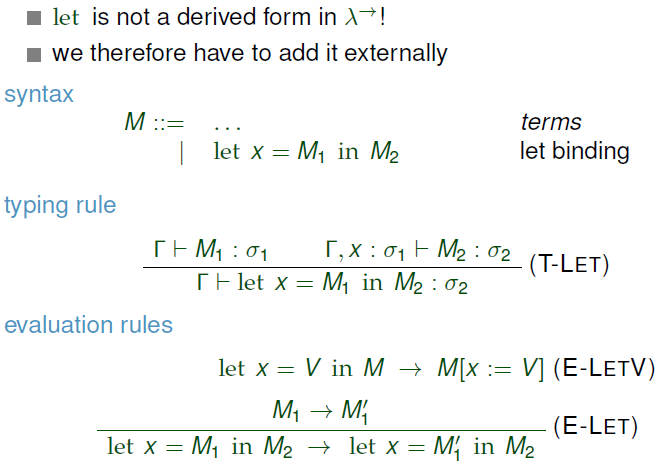
### Unit type

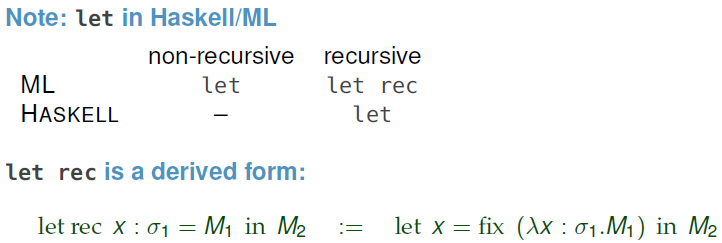


### Sequencing

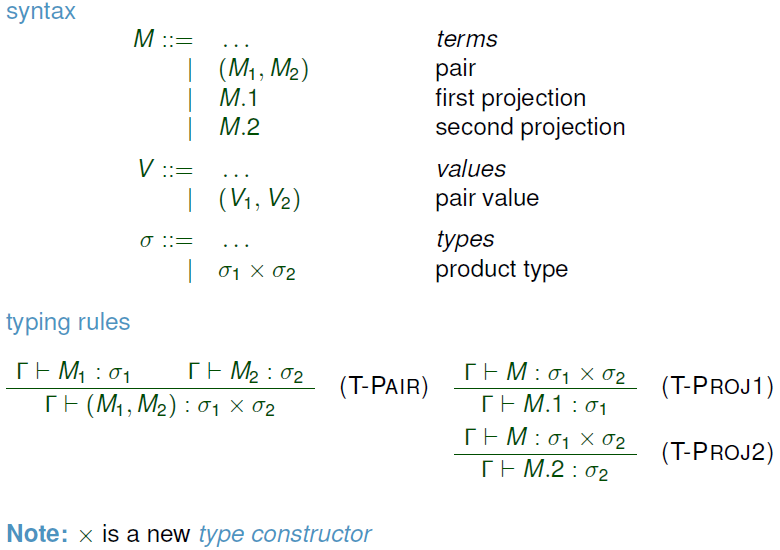


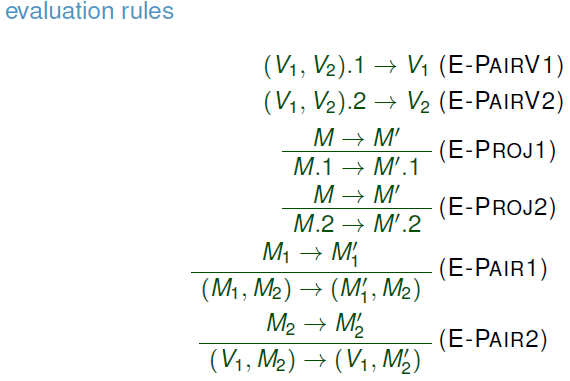
### *Let* bindings



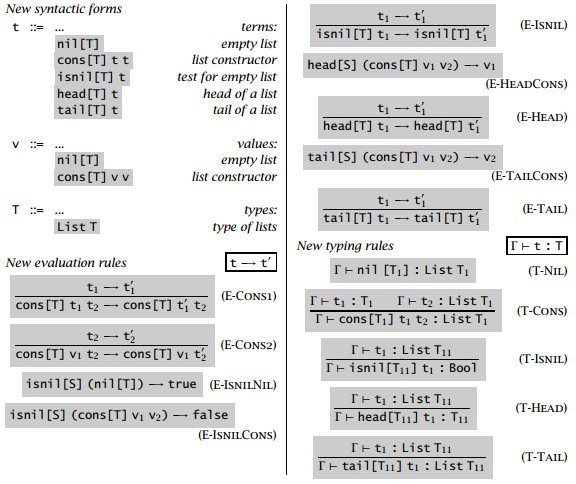


### Pairs

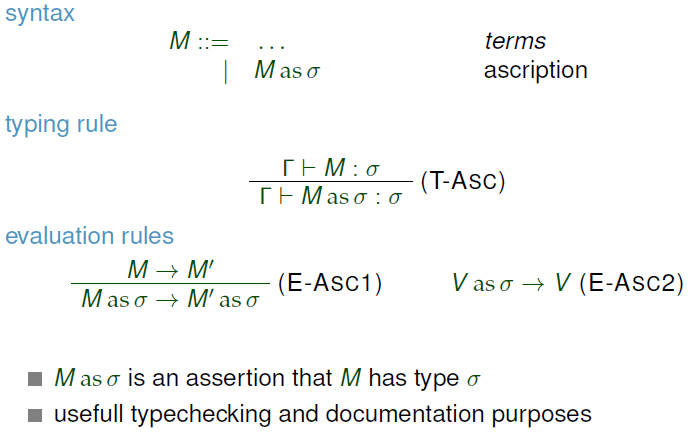




### Lists (taken from B. C. Pierce: Types and Programming Languages[[3]](#footnote-3), page 147)

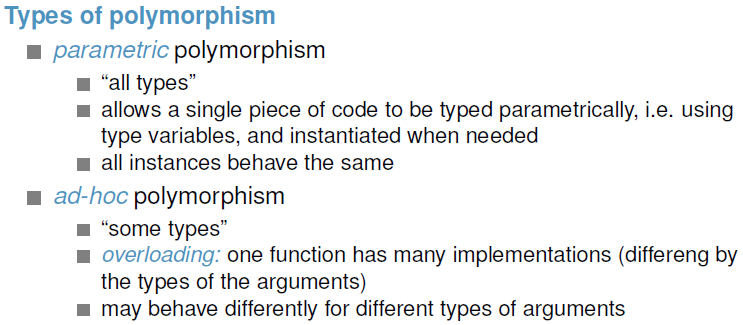


### Type ascription

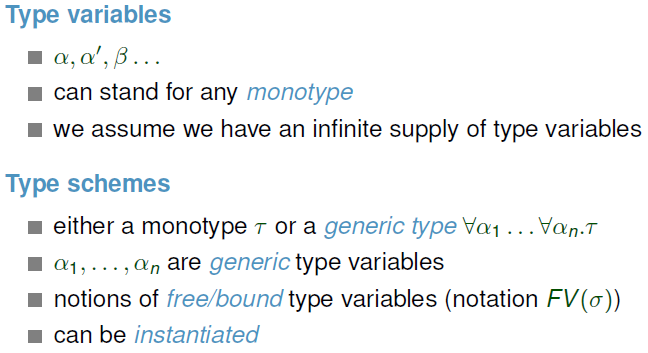


# System HM (Hindley-Milner)

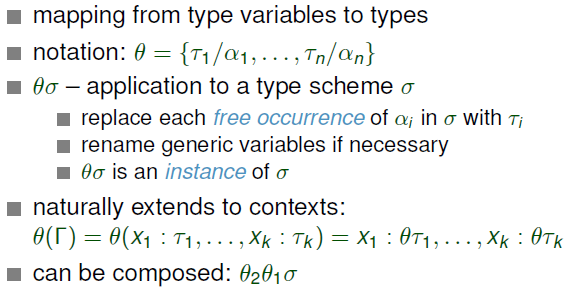
## Polymorphism



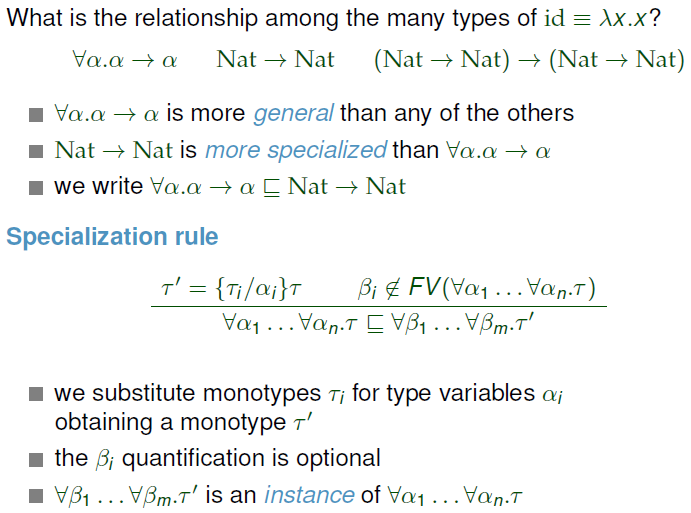
## Type variables and schemes



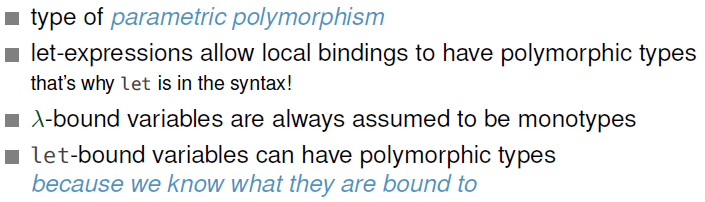
## Type substitution/instantiation



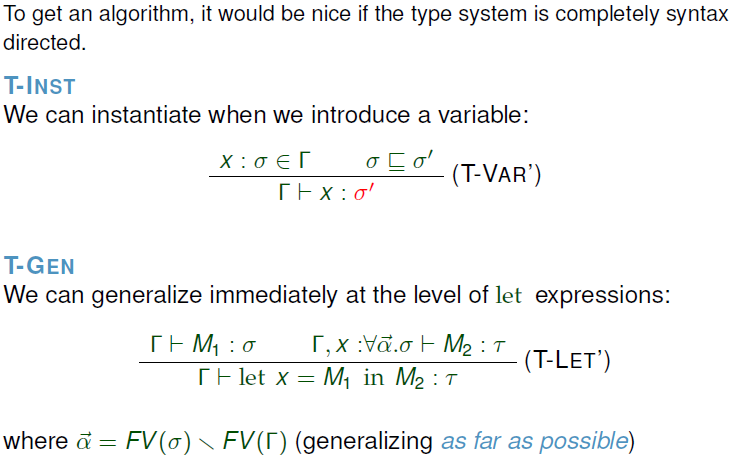
## Type ordering



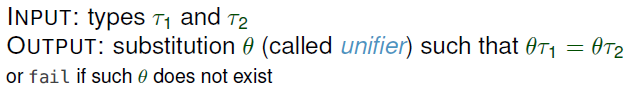
## *Let* polymorphism

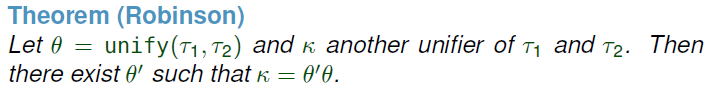


## Algorithmic type inference



### Unification



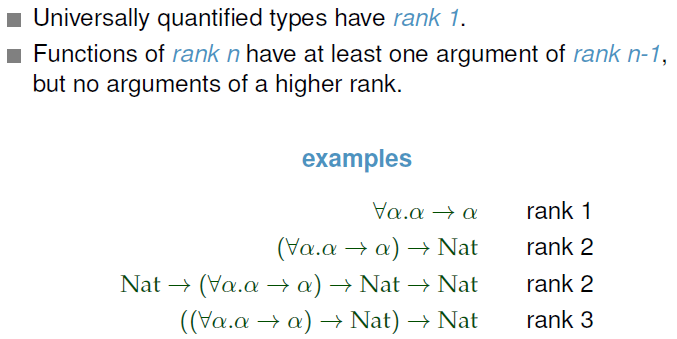




### Algorithm W



## Type rank



System HM only has rank 1 types.

# System F

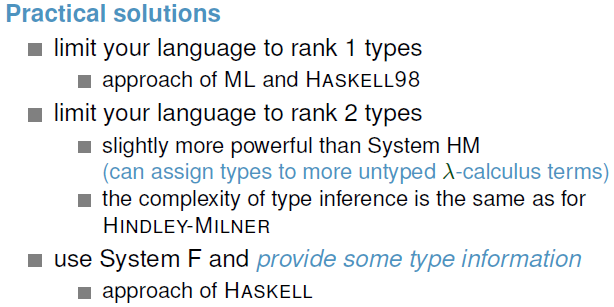
Allows types of any rank.

Well-typed System F terms are (strongly) normalizing.

## Type inference and type-checking

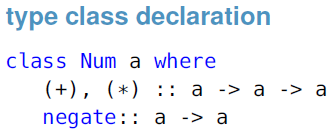
Type-checking is decidable (possible) in system F.

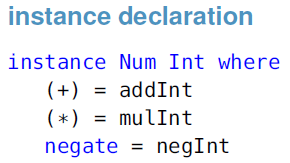
Type inference is undecidable for ranks > 2.



# Classes

## Type classes





We can then define a function *square* which takes types that are instances of the class *Num*.

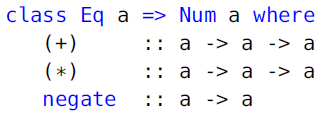


### Subclasses

For testing membership of a square, we need both equality and arithmetic.



We can also create subclasses. For example the class *Num* is a subclass of *Eq*.



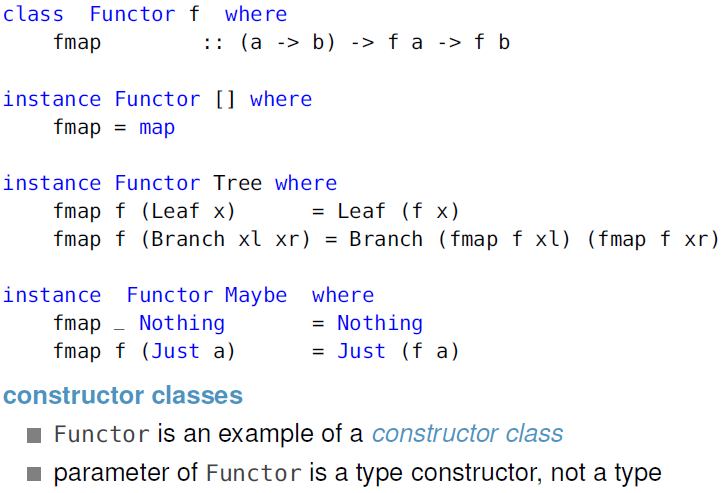
### Default implementation

We can define a default implementation for methods in classes, instances can then redefine the behaviour.

## Constructor classes

### Functor

#### Functor class



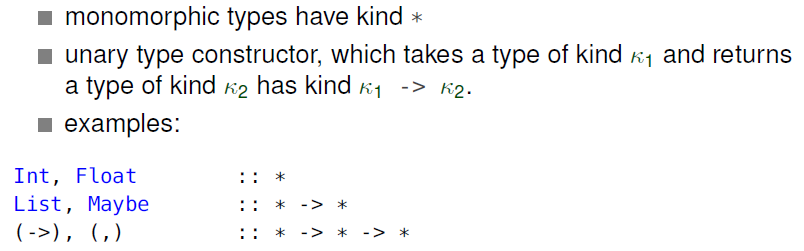
#### Functor laws



#### Functor lifting

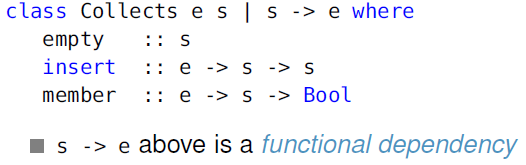
If we partially apply *fmap* to a function *g*, we get a function *(fmap g)* of type *f a -> f b*. So the “normal” function *g* is transformed into a function operating on containers. This transformation is called lifting.

### Kinds (“types of types”)



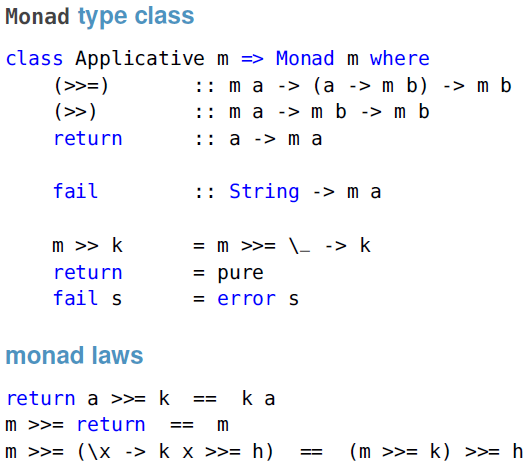
Kinds can be inferred.

### Functional dependencies

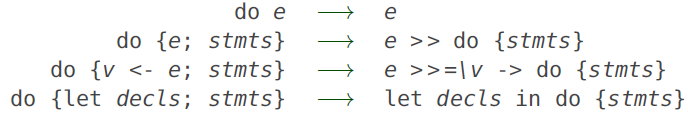


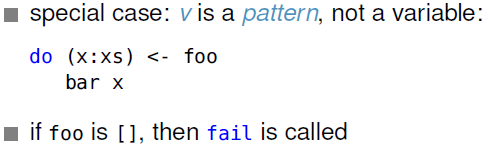
Type *s* uniquely determines type *e*.

# Monads



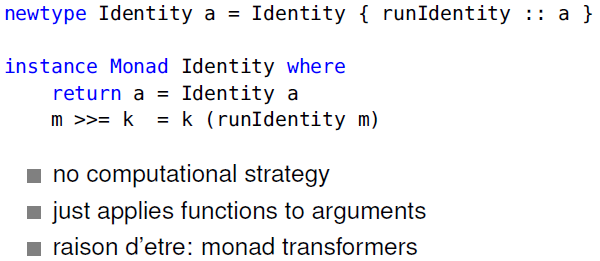
## *Do* notation



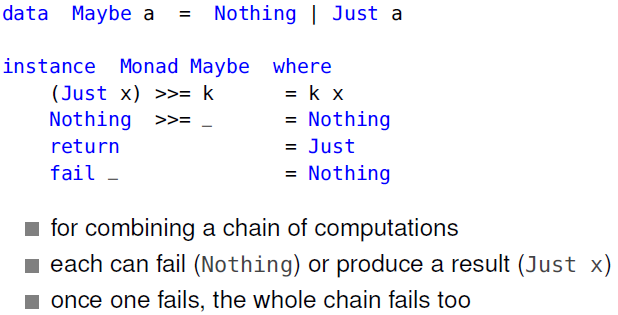


## Monads examples

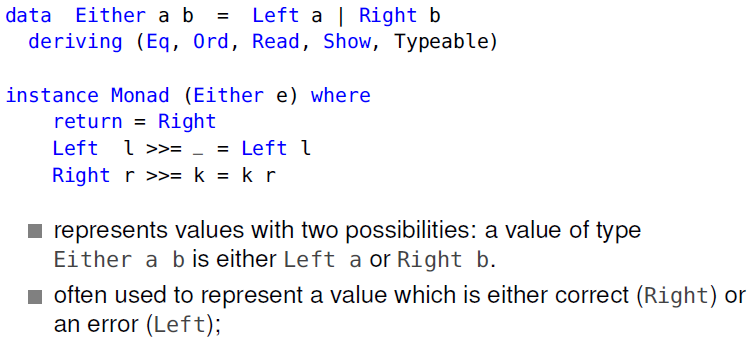
### The *Identity* monad



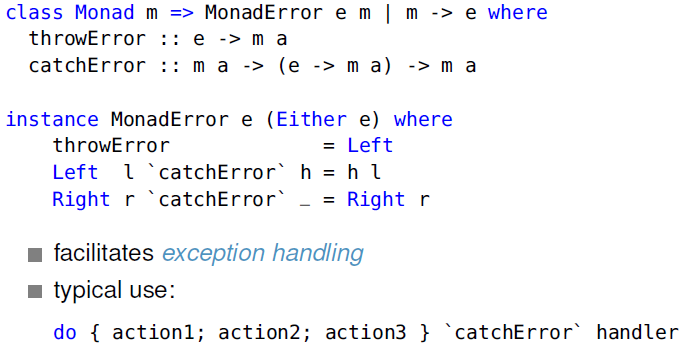
### The *Maybe* monad



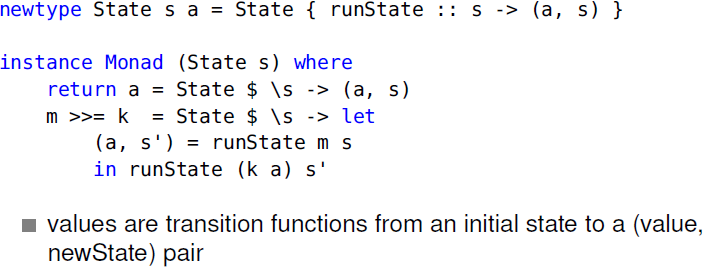
### The *Either* monad



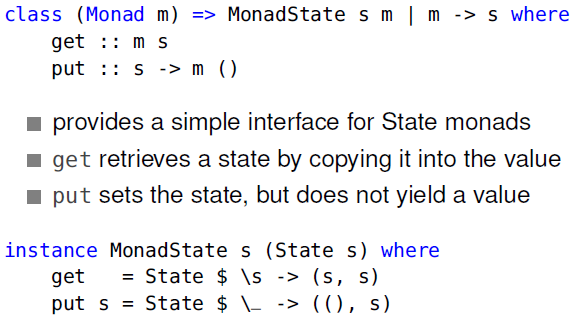
### The *MonadError* class



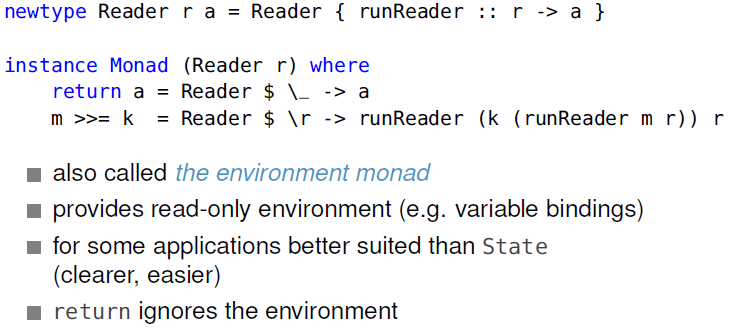
### The *State* monad



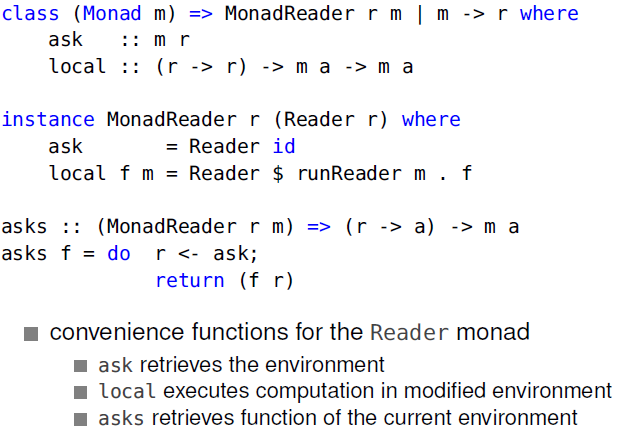
### The *MonadState* class



### The *Reader* monad

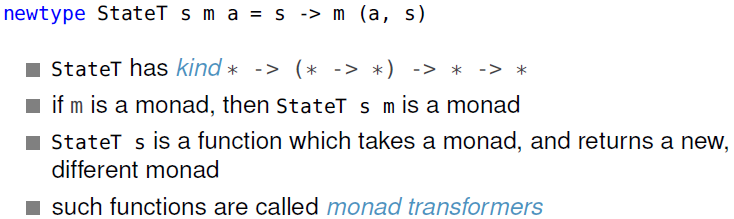


### The *MonadReader* class

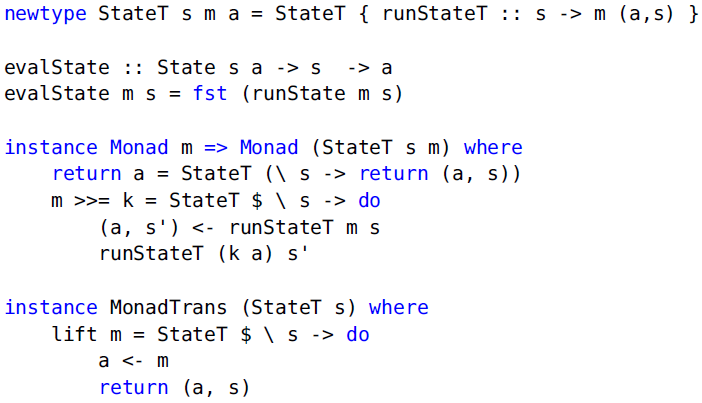


## Monad transformers

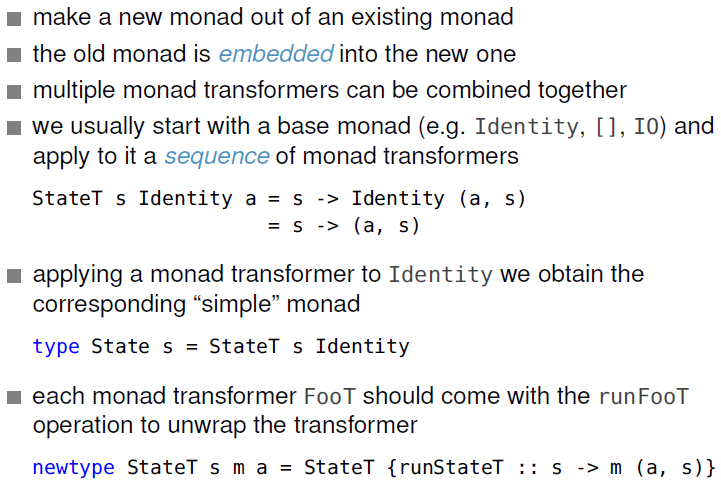
### StateT



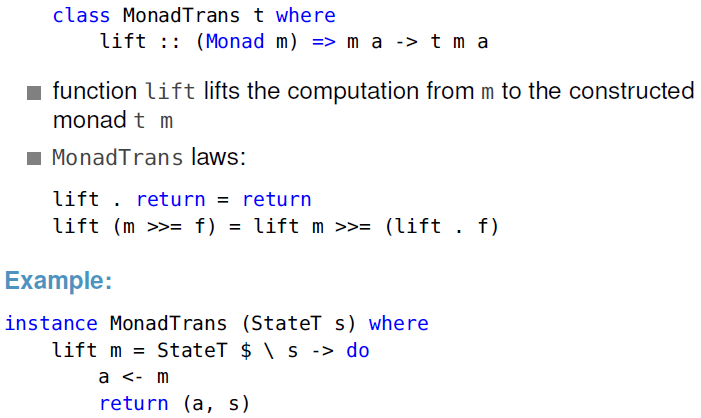
### The *StateT* transformer monad



### About monad transformers



### The *MonadTrans* class (in Haskell)

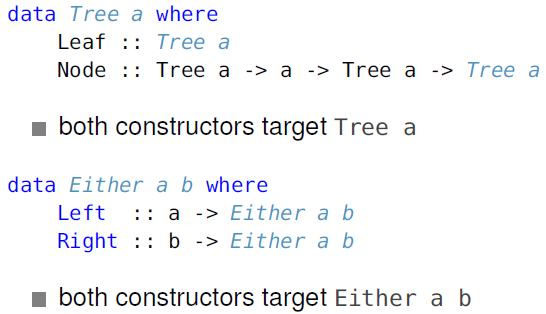


# Data types

## ADTs (Algebraic Data Types)

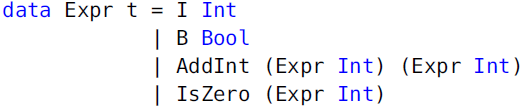
### Range restriction of ADTs

In ADTs, all constructors have identical range types.



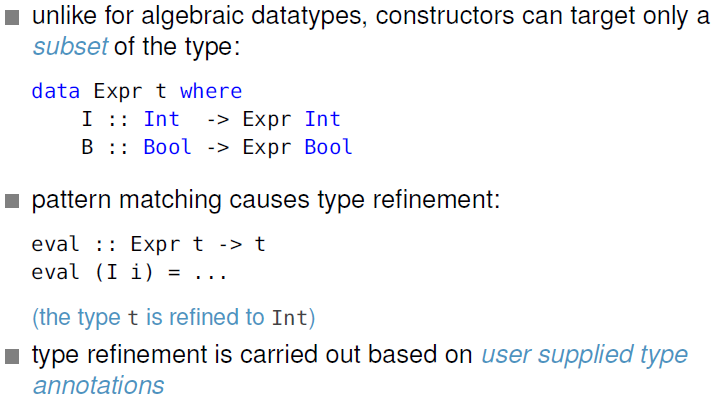
## Phantom types

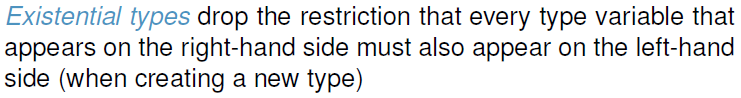
A phantom type is a parametrised type whose parameters do not all appear on the right-hand side of its definition.



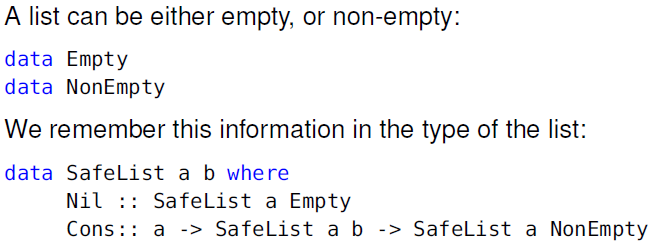
Allows malformed expressions to typecheck.

## GADTs (Generalized Algebraic Data Types)



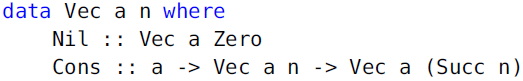


### Safe lists



### Vectors

Vectors are lists of a fixed length. To express the length, we need to encode natural numbers on the type level.



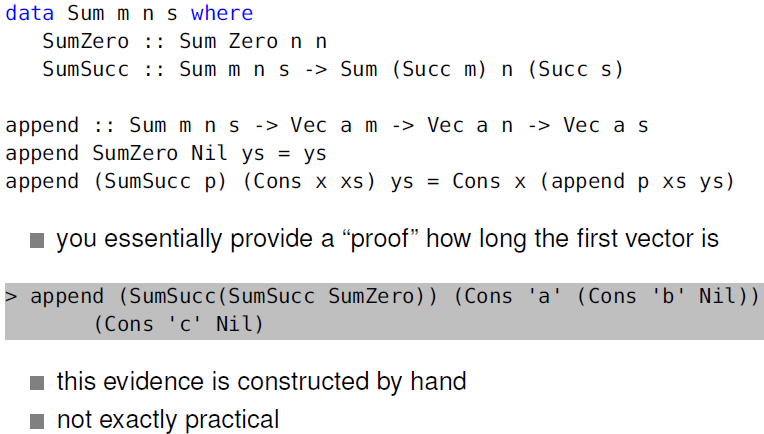
A problem shows up when we want to write a function which should join two vectors such as:

*append :: Vec a n -> Vec a m -> Vec a (m + n)*

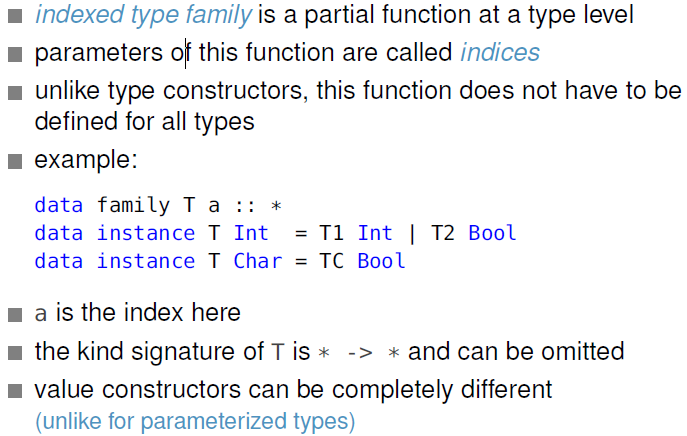
Doing *(m + n)* on a type level is not possible.

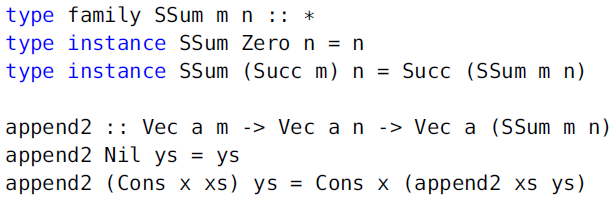
#### Possible solutions

##### Solution 1: Encoding the addition as GADT



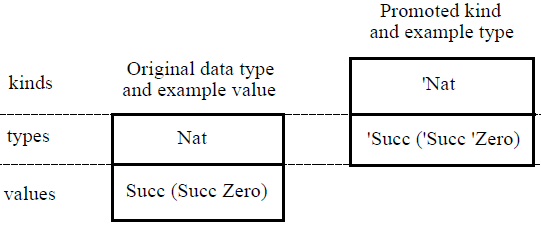
##### Solution 2: Type families





## Data type promotion

Using *DataKinds*, value constructors also become type constructors. So types are promoted to kinds, so now we can for example have a kind *\* -> Nat -> \**.

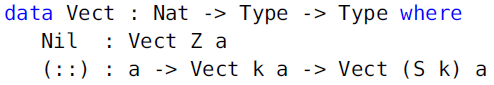


## Dependent types

In dependently typed languages, types can contain (depend on) arbitrary values, and appear as arguments and results of arbitrary functions.

Haskell does not support dependent types, the following examples are taken from Idris.

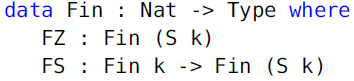
#### Vectors

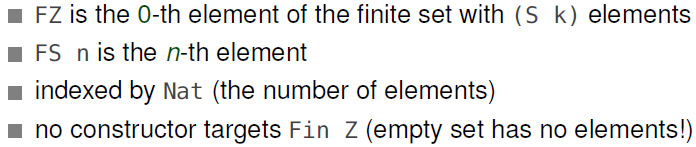


#### Matrices



#### Finite sets





1. As of January 2016, this document does not contain excerpts about Generic programming with GADT, I/O and Concurrency as these topics weren’t covered in the Autumn 2015 semester. [↑](#footnote-ref-1)
2. https://en.wikipedia.org/wiki/Church\_encoding [↑](#footnote-ref-2)
3. http://port70.net/~nsz/articles/book/pierce\_types\_and\_programming\_languages\_2002.pdf [↑](#footnote-ref-3)