Utrecht University

GADTs & lambda calculus

Advanced functional programming

Wouter Swierstra

Today's lecture

Generalized algebraic data types (GADTs)

A datatype

```
data Tree a  =  Leaf
             |  Node (Tree a) a (Tree a)
```

This definition introduces:

A datatype

```haskell
data Tree a  =  Leaf
             |  Node (Tree a) a (Tree a)
```

This definition introduces:

- a new datatype `Tree` of kind * -> * .

A datatype

```haskell
data Tree a  =  Leaf
             |  Node (Tree a) a (Tree a)
```

This definition introduces:

- a new datatype `Tree` of kind * -> * .
- two constructor functions

```haskell
Leaf  ::  Tree a
Node  ::  Tree a -> a -> Tree a -> Tree a
```

A datatype

```
data Tree a  =  Leaf
             |  Node (Tree a) a (Tree a)
```

This definition introduces:

- a new datatype `Tree` of kind * -> * .

- two constructor functions

```
Leaf  ::  Tree a
Node  ::  Tree a -> a -> Tree a -> Tree a
```

- the possiblity to use the constructors `Leaf` and `Node` in patterns.

Alternative syntax

Observation

The types of the constructor functions contain sufficient information to describe the datatype.

```
data Tree :: * -> * where
  Leaf  ::  Tree a
  Node  ::  Tree a -> a -> Tree a -> Tree a
```

Algebraic datatypes

Constructors of an algebraic datatype `T` must:

- target the type `T`,
- result in a simple type of kind *, i.e., `T` $a_1 \ \ldots \ a_n$ where $a_1, \ \ldots, \ a_n$ are distinct type variables.

Another example

```
data Either :: * -> * -> *  where
  Left   ::  a ->  Either a b
  Right  ::  b ->  Either a b
```

Both constructors produce values of type `Either a b`.

Does it make sense to lift these restrictions?

Excursion: Expression language

Imagine we're implementing a small programming language in Haskell:

```haskell
data Expr =
    LitI    Int
  | LitB    Bool
  | IsZero  Expr
  | Plus    Expr Expr
  | If      Expr Expr Expr
```

Excursion: Expression language

Alternatively, we could redefine the data type as follows:

```
data Expr :: * where
  LitI    ::  Int -> Expr
  LitB    ::  Bool -> Expr
  IsZero  ::  Expr -> Expr
  Plus    ::  Expr -> Expr -> Expr
  If      ::  Expr -> Expr -> Expr -> Expr
```

Syntax: concrete vs abstract

Imagined concrete syntax:

```
if isZero (0 + 1) then False else True
```

Abstract syntax:

```
If (IsZero (Plus (LitI 0) (LitI 1)))
   (LitB False)
   (LitB True)
```

Type errors

It is all too easy to write ill-typed expressions such as:

```
If (LitI 0) (LitB False) (LitI 1)
```

How can we prevent programmers from writing such terms?

Phantom types

At the moment, *all* expressions have the same type:

```
data Expr =
    LitI    Int
  | LitB    Bool
  ....
```

We would like to distinguish between expressions of *different* types.

Phantom types

At the moment, *all* expressions have the same type:

```
data Expr =
    LitI    Int
  | LitB    Bool
  ....
```

We would like to distinguish between expressions of *different* types.

To do so, we add an additional *type parameter* to our expression data type.

Phantom types

```
data Expr a =
    LitI    Int
  | LitB    Bool
  | IsZero  (Expr Int)
  | Plus    (Expr Int) (Expr Int)
  | If      (Expr Bool) (Expr a) (Expr a)
```

Note: the type variable `a` is never actually used in the data type for expressions.

We call such type variables *phantom types*.

Constructing well typed terms

Rather than expose the constructors of our expression language, we can instead provide a *well-typed API* for users to write terms:

```haskell
litI :: Int -> Expr Int
litI = LitI

plus :: Expr Int -> Expr Int -> Expr Int
plus = Plus

isZero :: Expr Int -> Expr Bool
isZero = IsZero
```

This guarantees that users will only ever construct well-typed terms! But what about writing an interpreter...

Evaluation

Before we write an interpreter, we need to choose the type that it returns.

Our expressions may evaluate to booleans or integers:

```
data Val =
     VInt   Int
  |  VBool  Bool
```

Defining an interpreter now boils down to defining a function:

```
eval :: Expr a -> Val
```

Evaluation

```haskell
eval :: Expr a -> Val
eval (LitI n)      = VInt n
eval (LitB b)      = VBool b
eval (IsZero e)    =
  case eval e of
    VInt n ->  VBool (n == 0)
    _      ->  error "type error"
eval (Plus e1 e2)  =
  case (eval e1, eval e2) of
    (VInt n1, VInt n2)  ->  VInt (n1 + n2)
    _                   ->  error "type error"
```

Evaluation (contd.)

- Evaluation code is mixed with code for handling type errors.
- The evaluator uses *tags* (i.e., constructors) to dinstinguish values – these tags are maintained and checked at run time.
- Run-time type errors can, of course, be prevented by writing a type checker or using phantom types.
- Even if we know that we only have type-correct terms, the Haskell compiler does not enforce this.

Beyond phantom types

What if we encode the type of the term in the Haskell type?

```
data Expr :: * -> * where
  LitI   :: Int -> Expr Int
  LitB   :: Bool -> Expr Bool
  IsZero :: Expr Int -> Expr Bool
  Plus   :: Expr Int -> Expr Int -> Expr Int
  If     :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Each expression has an additional *type argument*, representing the type of values it stores.

GADTs

GADTs lift the restriction that all constructors must produce values of the same type.

- Constructors can have more specific return types.
- Interesting consequences for pattern matching:
    - when case-analyzing an `Expr Int`, it could not be constructed by `Bool` or `IsZero`;
    - when case-analyzing an `Expr Bool`, it could not be constructed by `Int` or `Plus`;
    - when case-analyzing an `Expr a`, once we encounter the constructor `IsZero` in a pattern, we know that we must be dealing with an `Expr Bool`;
    - ...

Evaluation revisited

```haskell
eval :: Expr a -> a
eval (LitI n)      =  n
eval (LitB b)      =  b
eval (IsZero e)    =  (eval e) == 0
eval (Plus e1 e2)  =  eval e1 + eval e2
eval (If e1 e2 e3)
  | eval e1        = eval e2
  | otherwise      = eval e3
```

- No possibility for run-time failure; no *tags* required on our values.
- Pattern matching on a GADT requires a type signature. Why?

Type signatures are required ...

```
data X :: * -> * where
  C  ::  Int -> X Int
  D  ::  X a
  E  ::  Bool -> X Bool

f (C n)  =  [n]            -- (1)
f D      =  []             -- (2)
f (E n)  =  [n]            -- (3)
```

Type signatures are required ...

```
data X :: * -> * where
  C  ::  Int -> X Int
  D  ::  X a
  E  ::  Bool -> X Bool

f (C n)  =  [n]            -- (1)
f D      =  []             -- (2)
f (E n)  =  [n]            -- (3)
```

What is the type of `f`, with/without (3)? What is the (probable) desired type?

Type signatures are required ...

```
data X :: * -> * where
  C  ::  Int -> X Int
  D  ::  X a
  E  ::  Bool -> X Bool


f (C n)  =  [n]          -- (1)
f D      =  []           -- (2)
f (E n)  =  [n]          -- (3)
```

What is the type of `f`, with/without (3)? What is the (probable) desired type?

```
f :: X a -> [Int]          -- (1) only
f :: X b -> [c]            -- (2) only
f :: X a -> [Int]          -- (1) + (2)
```

Extending our language

Let us extend the expression types with pair construction and projection:

```
data Expr :: * -> * where
   ...
   Pair    ::  Expr a -> Expr b  -> Expr (a, b)
   Fst     ::  Expr (a,b)        -> Expr a
   Snd     ::  Expr (a,b)        -> Expr b
```

For `Fst` and `Snd`, the type of the non-projected component is ‘hidden’ – that is, it is not visible from the type of the compound expression.

Lab exercise

Extend the evaluation function accordingly. What about adding an `Either` type?

GADTs

GADTs have become one of the more popular Haskell extensions.

The ‘classic’ example for motivating GADTs is interpreters for expression languages, such as the one we have seen here.

However, these richer data types offer many other applications.

In particular, they let us *program* with types in interesting new ways.

`Prelude.head: empty list`

```
> myComplicatedFunction 42 "inputFile.csv"
*** Exception: Prelude.head: empty list
```

Can we use the *type system* to rule out such exceptions before a program is run?

Prelude.head: empty list

```
> myComplicatedFunction 42 "inputFile.csv"
*** Exception: Prelude.head: empty list
```

Can we use the *type system* to rule out such exceptions before a program is run?

To do so, we’ll introduce a new list-like datatype that records the *length* of the list in its *type*.

Natural numbers and vectors

Natural numbers can be encoded as types – no constructors are required.

```
data Zero
data Succ a
```

Vectors are lists with a fixed number of elements:

```
data Vec :: * -> * -> * where
  Nil   ::                   Vec a Zero
  Cons  ::  a -> Vec a n  -> Vec a (Succ n)
```

Type-safe `head` and `tail`

```
head :: Vec a (Succ n) -> a
head (Cons x xs)  =  x

tail :: Vec a (Succ n) -> Vec a n
tail (Cons x xs)  =  xs
```

Question

Why is there no case for `Nil` is required?

Type-safe `head` and `tail`

```haskell
head :: Vec a (Succ n) -> a
head (Cons x xs)  =  x

tail :: Vec a (Succ n) -> Vec a n
tail (Cons x xs)  =  xs
```

Question

Why is there no case for `Nil` is required?

Actually, a case for `Nil` results in a type error.

More functions on vectors

```
map :: (a -> b) -> Vec a n -> Vec b n
map f Nil          =  Nil
map f (Cons x xs)  =  Cons (f x) (map f xs)

zipWith :: (a -> b -> c) ->
           Vec a n -> Vec b n -> Vec c n
zipWith op Nil          Nil          =  Nil
zipWith op (Cons x xs)  (Cons y ys)  =
  Cons (op x y) (zipWith op xs ys)
```

We can require that the two vectors have the same length!

This lets us rule out bogus cases.

Yet more functions on vectors

```
snoc :: Vec a n -> a -> Vec a (Succ n)
snoc Nil          y  =  Cons y Nil
snoc (Cons x xs)  y  =  Cons x (snoc xs y)

reverse :: Vec a n -> Vec a n
reverse Nil          =  Nil
reverse (Cons x xs)  =  snoc xs x
```

What about appending two vectors, analogous to the `(++)` operation on lists?

Problematic functions

- What is the type of our append function?

```
vappend :: Vec a m -> Vec a n -> Vec a ???
```

Problematic functions

- What is the type of our append function?

```
vappend :: Vec a m -> Vec a n -> Vec a ???
```

How can we add two *types*, `n` and `m`?

Problematic functions

- What is the type of our append function?

```
vappend :: Vec a m -> Vec a n -> Vec a ???
```

How can we add two *types*, `n` and `m`?

- Suppose we want to convert from lists to vectors:

```
fromList :: [a] -> Vec a n
```

Where does the type variable `n` come from? What possible values can it have?

Writing vector append

There are multiple options to solve that problem:

- construct explicit evidence,
- use a type family (more on that in the next lecture).

Explicit evidence

Given two 'types' `n` and `m`, what is their sum?

We can define a GADT describing the *graph* of the addition function:

```
data Sum :: * -> * -> * -> * where
  SumZero :: Sum Zero n n
  SumSucc :: Sum m n s ->
             -> Sum (Succ m) n (Succ s)
```

Explicit evidence

Given two ‘types’ `n` and `m`, what is their sum?

We can define a GADT describing the *graph* of the addition function:

```
data Sum :: * -> * -> * -> * where
  SumZero :: Sum Zero n n
  SumSucc :: Sum m n s ->
           -> Sum (Succ m) n (Succ s)
```

Using this function, we can now define `append` as follows:

```
append :: Sum m n s -> Vec a m -> Vec a n -> Vec a s
append SumZero     Nil         ys =  ys
append (SumSucc p) (Cons x xs) ys =
  Cons x (append p xs ys)
```

Passing explicit evidence

This approach has one major disadvantage: we must construct the evidence, the values of type `Sum n m p`, by hand every time we wish to call `append`.

We could use a multi-parameter type class with functional dependencies to automate this construction...

Converting between lists and vectors

It is easy enough to convert from a vector to a list:

```
toList :: Vec a n -> [a]
toList Nil          =  []
toList (Cons x xs)  =  x : toList xs
```

This simply discards the type information we have carefully constructed.

Converting between lists and vectors

It is easy enough to convert from a vector to a list:

```
toList :: Vec a n -> [a]
toList Nil          =  []
toList (Cons x xs)  =  x : toList xs
```

This simply discards the type information we have carefully constructed.

Converting between lists and vectors

Converting in the other direction, however is not as easy:

```
fromList :: [a] -> Vec a n
fromList []      =  Nil
fromList (x:xs)  =  Cons x (fromList xs)
```

Question

Why doesn't this definition type check?

Converting between lists and vectors

Converting in the other direction, however is not as easy:

```
fromList :: [a] -> Vec a n
fromList []      =  Nil
fromList (x:xs)  =  Cons x (fromList xs)
```

Question

Why doesn’t this definition type check?

The type says that the result must be polymorphic in `n`, that is, it returns a vector of *any* length, rather than a vector of a specific (unknown) length.

From lists to vectors

We can

- specify the length of the vector being constructed in a separate argument,
- hide the length using an *existential* type.

From lists to vectors (contd.)

Suppose we simply pass in a regular natural number, `Nat`:

```
fromList :: Nat -> [a] -> Vec a n
fromList Zero       []     =  Nil
fromList (Succ n)  (x:xs)  =  Cons x (fromList n xs)
fromList _           _     =  error "wrong length!"
```

From lists to vectors (contd.)

Suppose we simply pass in a regular natural number, `Nat`:

```
fromList :: Nat -> [a] -> Vec a n
fromList Zero       []     =  Nil
fromList (Succ n)  (x:xs)  =  Cons x (fromList n xs)
fromList _          _      =  error "wrong length!"
```

This still does not solve our problem – there is no connection between the natural number that we are passing and the `n` in the return type.

Singletons

We need to reflect type-level natural numbers on the value level.

To do so, we define a (yet another) variation on natural numbers:

```
data SNat :: * -> * where
  SZero  ::  SNat Zero
  SSucc  ::  SNat n -> SNat (Succ n)
```

This is a *singleton type* – for any `n`, the type `SNat n` has a single inhabitant (the number `n`).

From lists to vectors

```haskell
data SNat :: * -> * where
  SZero  ::  SNat Zero
  SSucc  ::  SNat n -> SNat (Succ n)

fromList :: SNat n -> [a] -> Vec a n
fromList SZero     []     = Nil
fromList (SSucc n) (x:xs) = Cons x (fromList n xs)
fromList _         _      = error "wrong length!"
```

Question

This function may still fail dynamically. Why?

From lists to vectors

We can

- specify the length of the vector being constructed in a separate argument,
- hide the length using an *existential* type.

What about the second alternative?

From lists to vectors

We can define a wrapper around vectors, *hiding* their length:

```
data VecAnyLen :: * -> * where
  VecAnyLen :: Vec a n -> VecAnyLen a
```

A value of type `VecAnyLen a` stores a vector of *some* length with values of type `a`.

From lists to vectors

We can convert any list to a vector of some length as follows:

```
fromList :: [a] -> VecAnyLen a
fromList []      =  VecAnyLen Nil
fromList (x:xs)  =
  case fromList xs of
    VecAnyLen ys -> VecAnyLen (Cons x ys)
```

From lists to vectors

We can combine the two approaches and include a `SNat` in the packed type:

```
data VecAnyLen :: * -> * where
  VecAnyLen :: SNat n -> Vec a n -> VecAnyLen a
```

Question

How does the conversion function change?

Calling functions on vectors

Given two vectors `xs : Vec a n` and `ys : Vec a m`.

Suppose I want to zip these vectors together using:

```
zipVec :: Vec a n -> Vec b n -> Vec (a,b) n
```

Question

What happens when I call `zip xs ys`?

Comparing the length of vectors

We can define a boolean function that checks when two vectors have the same length

```
equalLength :: Vec a m -> Vec b n -> Bool
equalLength Nil Nil = True
equalLength (Cons _ xs) (Cons _ ys) =
  equalLength xs ys
```

Comparing the length of vectors

Such a function is not very useful…

Suppose I want to use this to check the lengths of my vectors:

```
if equalLength xs ys
  then zip xs ys
  else error "Wrong lengths"
```

Question

Will this type check?

Comparing the length of vectors

Such a function is not very useful…

Suppose I want to use this to check the lengths of my vectors:

```
if equalLength xs ys
  then zip xs ys
  else error "Wrong lengths"
```

Question

Will this type check?

No! Just because `equalLength xs ys` returns True, does not guarantee that `m` and `n` are equal…

How can we enforce that two types are indeed equal?

Equality type

Just as we saw for the `Sum` type, we can introduce a GADT that represents a 'proof' that two types are equal:

```
data Equal :: * -> * -> * where
  Refl  ::   Equal a a
```

Equality type

Just as we saw for the `Sum` type, we can introduce a GADT that represents a 'proof' that two types are equal:

```
data Equal :: * -> * -> * where
  Refl  ::   Equal a a
```

We can even 'prove' properties of our equality relation:

```
refl :: Equal a a
sym :: Equal a b -> Equal b a
trans :: Equal a b -> Equal b c -> Equal a c
```

Equality type

Just as we saw for the `Sum` type, we can introduce a GADT that represents a ‘proof’ that two types are equal:

```
data Equal :: * -> * -> * where
  Refl  ::   Equal a a
```

We can even ‘prove’ properties of our equality relation:

```
refl :: Equal a a
sym :: Equal a b -> Equal b a
trans :: Equal a b -> Equal b c -> Equal a c
```

Question

How are these functions defined? What happens if you don’t pattern match on the `Refl` constructor?

Equality type

Instead of returning a boolean, we can now provide evidence that the length of two vectors is equal:

```
eqLength :: Vec a m -> Vec b n -> Maybe (Equal m n)
eqLength Nil          Nil          =   Just Refl
eqLength (Cons x xs)  (Cons y ys)  =
  case eqLength xs ys of
    Just Refl   ->  Just Refl
    Nothing     ->  Nothing
eqLength _            _            =   Nothing
```

Using equality

```haskell
test :: Vec a m -> Vec b (Succ n) -> Maybe (a,b)
test xs ys =
  case eqLength xs ys
    Just Refl -> head (zip xs ys)
    _         -> Nothing
```

Question

Why does this type check?

Expressive power of equality

The equality type can be used to encode other GADTs.

Recall our expression example using phantom types:

```
data Expr a =
    LitI    Int
  | LitB    Bool
  | IsZero  (Expr a)
  | Plus    (Expr a) (Expr a)
  | If      (Expr a) (Expr a) (Expr a)
```

Expressive power of equality

We can replace this with a phantom type

```
data Expr a =
    LitI  (Equal a Int) Int
  | LitB  (Equal a Bool) Bool
  | IsZero (Equal a Bool) (Equal b Int) (Expr b)
  | Plus (Equal a Int) (Expr a) (Expr a) (Expr a)
  ...
```

Safe vs unsafe coercions

Using our equality function we can safely coerce between types:

```haskell
coerce :: Equal a b -> a -> b
coerce Refl x = x
```

Question

Why does this type check?

Safe vs unsafe coercions

Using our equality function we can safely coerce between types:

```haskell
coerce :: Equal a b -> a -> b
coerce Refl x = x
```

Question

Why does this type check?

Question

What about this definition:

```haskell
coerce :: Equal a b -> a -> b
coerce p x = x
```

Aside: irrefutable patterns

Haskell also allows irrefutable patterns:

```haskell
lazyHead ~(x:xs) = x
```

This does not force the list to weak head normal form.

Aside: irrefutable patterns

In tandem with GADTs this is particularly dangerous:

```
coerceL :: Equal a b -> a -> b
coerceL ~Refl x = x
```

Question

How could this cause well-typed program to crash with a type error?

Aside: irrefutable patterns

In tandem with GADTs this is particularly dangerous:

```
coerceL :: Equal a b -> a -> b
coerceL ~Refl x = x
```

Question

How could this cause well-typed program to crash with a type error?

```
foo :: Bool -> Int
foo b = coerceL undefined b
```

Apparently unrelated language features may interact in unexpected ways!

Outlook generic programming: Reflecting types

We can even use GADTs to *reflect* types themselves as data:

```
data Type :: * -> * where
  INT  :: Type Int
  BOOL :: Type Bool
  LIST :: Type a -> Type [a]
  PAIR :: Type a -> Type b -> Type (a,b)
```

Safe dynamically typed values

We can define dynamically typed values by packing up a type representation with a value:

```
data Dynamic :: * where
  Dyn  ::  Type a -> a -> Dynamic
```

Safe dynamically typed values

We can define dynamically typed values by packing up a type representation with a value:

```
data Dynamic :: * where
  Dyn  ::  Type a -> a -> Dynamic
```

To unwrap these values safely, we check whether the types line up as expected:

```
coerce :: Type a -> Dynamic -> Maybe a
coerce t (Dyn t' x) =
  case eqType t t'
    Just Refl -> Just x
    _         -> Nothnig
```

Generic programming

We can also define new functions *by induction on the type structure*:

```
f :: Type a -> ... a ...
```

In this way, we can define our own versions of functions such as show, read, equality, etc.

Summary

- GADTs can be used to encode advanced properties of types in the type language.
- We end up mirroring expression-level concepts on the type level (e.g. natural numbers).
- GADTs can also represent data that is computationally irrelevant and just guides the type checker (equality proofs, evidence for addition).
  Such information could ideally be erased, but in Haskell, we can always cheat via `undefined :: Equal Int Bool`…
- All complicated case expressions using GADTs can be compiled to a simple language – GHC Core.

The Lambda Calculus

- Introduced by Church 1936 (or even earlier).
- Formal language based on variables, function abstraction and application (substitution).
- Allows to express higher-order functions naturally.
- Equivalent in computational power to a Turing machine.
- Is at the basis of functional programming languages such as Haskell.

What and why?

- A simple language with relatively few concepts.
- Easy to reason about.
- Original goal: reason about expressiveness of computations.
- Today more: core language for playing with all sorts of language features.
- Many flavours: untyped, typed, added constants and constructs.

Lambda calculus: definition

There are only three constructs:

```
e  ::=  x           (variables)
   |    e e         (application)
   |    λ x → e     (abstraction)
```

Conventions

- Note: application associates to the left:
  `a b c = (a b) c`
- Note: only unary functions and unary application – but we write
  λ `x y` $\rightarrow$ `e` for λ `x` $\rightarrow$ `(`λ `y` $\rightarrow$ `e)`.
- Note: the function body of a lambda extends as far as possible to the right:
  λ `x` $\rightarrow$ `e f` should be read as λ `x` $\rightarrow$ `(e f)`

Definitions

- We usually consider terms *equal up to renaming* (alpha equivalence);
- The central computation rule is *beta reduction*:

`(`λ `x` $\rightarrow$ `e) (a)` reduces to `e [x/a]`

That is, the body of the lambda `e` where all the variables `x` are replaced with the term `a`.

Applications?

It seems as if we can do nothing useful with the lambda calculus.

There are no constants – no numbers, for instance.

But it turns out that we can **encode** recursion, numbers, booleans, and just about any other data type.

Church numerals

```
zero  ≡ λ s z → z
one   ≡ λ s z → (s z)
two   ≡ λ s z → (s (s z))
three ≡ λ s z → (s (s (s z)))
...
```

So far, so good, but can we calculate with these numbers?

Addition

$$\texttt{suc} \equiv \lambda\ \texttt{n} \longrightarrow \lambda\ \texttt{s z} \longrightarrow \texttt{(s (n s z)))}$$

$$\texttt{add} \equiv \lambda\ \texttt{m n} \longrightarrow \texttt{m suc n}$$

Does this work as expected?

Addition

$$\texttt{suc} \equiv \lambda\ \texttt{n} \rightarrow \lambda\ \texttt{s z} \rightarrow \texttt{(s (n s z)))}$$

$$\texttt{add} \equiv \lambda\ \texttt{m n} \rightarrow \texttt{m suc n}$$

Does this work as expected?

```
suc two
```

Addition

$$\texttt{suc} \quad \equiv \quad \lambda\ \texttt{n} \rightarrow \lambda\ \texttt{s z} \rightarrow \texttt{(s (n s z)))}$$

$$\texttt{add} \quad \equiv \quad \lambda\ \texttt{m n} \rightarrow \texttt{m suc n}$$

Does this work as expected?

```
suc two
```

$$(\lambda\ \texttt{n} \rightarrow (\lambda\ \texttt{s z} \rightarrow \texttt{(s (n s z)))) two}$$

Addition

$\texttt{suc} \equiv \lambda\ \texttt{n} \rightarrow \lambda\ \texttt{s z} \rightarrow \texttt{(s (n s z)))}$

$\texttt{add} \equiv \lambda\ \texttt{m n} \rightarrow \texttt{m suc n}$

Does this work as expected?

```
suc two
```

$\texttt{(}\lambda\ \texttt{n} \rightarrow \texttt{(}\lambda\ \texttt{s z} \rightarrow \texttt{(s (n s z)))) two}$

$\lambda\ \texttt{s z} \rightarrow \texttt{(s (two s z))}$

Addition

$\texttt{suc} \quad \equiv \quad \lambda\ \texttt{n} \rightarrow \lambda\ \texttt{s z} \rightarrow \texttt{(s (n s z)))}$

$\texttt{add} \quad \equiv \quad \lambda\ \texttt{m n} \rightarrow \texttt{m suc n}$

Does this work as expected?

```
suc two
```

$(\lambda\ \texttt{n} \rightarrow (\lambda\ \texttt{s z} \rightarrow \texttt{(s (n s z)))) two}$

$\lambda\ \texttt{s z} \rightarrow \texttt{(s (two s z))}$

$\lambda\ \texttt{s z} \rightarrow \texttt{(s (s (s z)))}$

Other data types?

This illustrates how we can represent *numbers* as *functions* – but it turns out we can also represent booleans using lambda terms, or just about any (simple) Haskell data type.

Other data types?

This illustrates how we can represent *numbers* as *functions* – but it turns out we can also represent booleans using lambda terms, or just about any (simple) Haskell data type.

But what about recursion?

Fixed-point combinators

Many fixed-point combinators can be defined in the untyped lambda calculus.

Here is one of the smallest and most famous ones, called `Y`.

$$\texttt{Y} \quad \equiv \quad \lambda\ \texttt{f} \rightarrow (\lambda\ \texttt{x} \rightarrow (\texttt{f (x x)}))\ (\lambda\ \texttt{x} \rightarrow (\texttt{f (x x)}))$$

Verification that `Y` is a fixed-point combinator

```
Y f
```

Verification that `Y` is a fixed-point combinator

$$\texttt{Y f}$$

$$\equiv$$

$$\texttt{(}\lambda\ \texttt{x} \rightarrow \texttt{(f (x x))) (}\lambda\ \texttt{x} \rightarrow \texttt{(f (x x)))}$$

Verification that `Y` is a fixed-point combinator

```
Y f
```

$\equiv$

`(`λ` x `$\rightarrow$` (f (x x))) (`λ` x `$\rightarrow$` (f (x x)))`

$\equiv$

`f ((`λ` x `$\rightarrow$` (f (x x))) (`λ` x `$\rightarrow$` (f (x x))))`

Verification that `Y` is a fixed-point combinator

```
Y f
```

$\equiv$

```
(λ x → (f (x x))) (λ x → (f (x x)))
```

$\equiv$

```
f ((λ x → (f (x x))) (λ x → (f (x x))))
```

$\equiv$

```
f (Y f)
```

What else?

There is still plenty missing to define a full programming language.

But the heart of Haskell – the lambda calculus – is similar in computing power to Turing machines.

What else?

There is still plenty missing to define a full programming language.

But the heart of Haskell – the lambda calculus – is similar in computing power to Turing machines.

But we can represent lambda terms using an even more simple language – namely that of *combinatory logic*.

Even Simpler

A straightforward implementation of the lambda calculus may give rise to abitrary large reduction steps. We can represent all lambda expressions using only three combinators with the following reduction behaviour:

```
S f g  x = (f x) (g x)
K y    x = y
I      x = x
```

Translation

Given a lambda term of the form – how can we translate this to an expression using SKI?

```
data SKI = Var String | S | K | I | App SKI SKI


toSKI :: Lambda -> SKI
toSKI (Var x)      = Var x
toSKI (App t1 t2) = (toSKI t1)  `App`(toSKI t2)
toSKY (Lam x t)   = remove x (toSKI t)
```

The auxiliary function `remove` does the actual work...

Bracket abstraction

```
remove :: Var -> Lambda -> SKI
remove x (Var y)
  | x == y = I
  | otherwise = K `App` y
remove x (App t1 t2) =
  S `App` (remove (App t1 x))
    `App` (remove (App t2 x))
```

This is sometimes called *bracket abstraction*.

Note: there is no case for lambdas – why?

Intuition

What's going on?

```
S f g  x = (f x) (g x)
K y    x = y
I      x = x
```

`S` is *duplicating* a variable; `K` is discarding a variable; `I` is using a variable.

Intuition

What's going on?

```
S f g  x = (f x) (g x)
K y    x = y
I      x = x
```

`S` is *duplicating* a variable; `K` is discarding a variable; `I` is using a variable.

Bracket abstraction simply explains *how* to route the argument of a function to the variable's occurrences in the lambda's body.

Alternatives

Haskell Curry proposed the following combinators:

```
B x y z = x (y z)
C x y z = x z y
K x y = x
W x y = x y y
```

Here `B` ‘routes arguments’ to the left only; `C` ‘routes arguments’ to the right; and `W` duplicates its inputs.

Even Simpler

The combinator `I` is superfluous:

```
S K K x  ⟶ (K x) (K x) ⟶ x
```

and hence

```
I = S K K
```

Even Simpler

In 1989 Jeroen Fokker invented:

```
X = λ f → (f S f3)
f3 = λ p _ _ → p        -- first of three
```

with which we can define K as follows:

```
K y x   →  X   X                    y  x
```

Does it reduce as expected?

Even Simpler

In 1989 Jeroen Fokker invented:

```
X = λ f ⟶ (f S f3)
f3 = λ p _ _ ⟶ p        -- first of three
```

with which we can define K as follows:

```
K y x   ⟶  X   X                    y  x
```

Does it reduce as expected?

```
        ⟶  X   S   f3               y  x
        ⟶  S   S   f3          f3   y  x
        ⟶  S   f3  (f3 f3)     y       x
        ⟶  f3  y   (f3 f3 y)   x
        ⟶  y
```

Even simpler

Check for yourself:

```
S = X (X X)
```

Even simpler

Check for yourself:

```
S = X (X X)
```

Labs

In the labs, we have several exercises about representing lambda terms using GADTs and translating lambda terms to SKI combinators.

Back to Haskell

This is nice – but what does this have to do with Haskell?

Back to Haskell

This is nice – but what does this have to do with Haskell?

GHC translates to an intermediate language: *GHC Core*.

GHC Core is really little more than a (typed) lambda calculus.

You can read the spec on GitHub.

Core sketch

GHC Core is based on System Fc – a typed lambda calculus extended with type coercions.

- variables, lambdas, and application;
- literals;
- let bindings;
- case expressions;
- coercions – used to implement GADTs amongst other things;
- ‘ticks’ – used for HPC to track program coverage.

Inspecting core can be useful to see how code is generated and optimized.

Generating core

```
alias ghci-core="ghci -ddump-simpl \
-dsuppress-idinfo -dsuppress-coercions \
-dsuppress-type-applications \
-dsuppress-uniques -dsuppress-module-prefixes"
```

The following Haskell code and corresponding Core:

```haskell
f :: Int -> Int
f x = x + 1
```

```
f :: Int -> Int
f = \ (x :: Int) ->
      case x of _ { I# x1 -> I# (+# x1 1) }
```

What we haven’t discussed yet

Types

Compare

$$\texttt{false} \equiv \lambda\ \texttt{t f -> f}$$
$$\texttt{zero} \equiv \lambda\ \texttt{s z -> z}$$

We can easily write terms that do not make sense in lambda calculus; Haskell has types to prevent that.

Overloading

In Haskell, functions can be overloaded using type classes. How can such overloading be resolved and desugared?

What we haven't discussed yet – contd.

Laziness

Haskell makes use of a particular evaluation strategy called lazy evaluation. We have not looked at evaluation strategies at all so far.

Side effects

The lambda calculus has no notion of effects, not even encapsulated effects such as Haskell offers with `IO`. So the behaviour of `IO` cannot be described by reduction to the lambda calculus.

What we haven’t discussed yet – contd.

Laziness

Haskell makes use of a particular evaluation strategy called lazy evaluation. We have not looked at evaluation strategies at all so far.

Side effects

The lambda calculus has no notion of effects, not even encapsulated effects such as Haskell offers with `IO`. So the behaviour of `IO` cannot be described by reduction to the lambda calculus.

Yet the lambda calculus is powerful enough to describe almost all of Haskell!

Bonus slides about Church encoding, recursion & pattern matching

Pairs

$$\texttt{pair} \equiv \lambda\ \texttt{x y} \rightarrow (\lambda\ \texttt{p} \rightarrow (\texttt{p x y}))$$

$$\texttt{fst} \equiv \lambda\ \texttt{p} \rightarrow (\texttt{p}\ (\lambda\ \texttt{x y} \rightarrow \texttt{x}))$$
$$\texttt{snd} \equiv \lambda\ \texttt{p} \rightarrow (\texttt{p}\ (\lambda\ \texttt{x y} \rightarrow \texttt{y}))$$

The function `pair` remembers its two parameters and returns them when asked by its third parameter.

How do you come up with these definitions?

Church encoding for arbitrary datatypes

There is a correspondence between the so-called *fold* (or *catamorphism* or *eliminator*) for a datatype and its Church encoding.

Haskell:

```
data Nat = Suc Nat | Zero


foldNat Zero     s z  =  z
foldNat (Suc n)  s z  =  s (foldNat n s z)
```

Lambda calculus:

```
zero   ≡  λ s z → z
suc n  ≡  λ s z → (s (n s z))
```

Church encoding for arbitrary datatypes – contd.

Haskell:

```haskell
data Bool = True | False


foldBool True   t f  =  t
foldBool False  t f  =  f
```

Lambda calculus:

$$\texttt{true} \equiv \lambda\ \texttt{t}\ \texttt{f} \to \texttt{t}$$
$$\texttt{false} \equiv \lambda\ \texttt{t}\ \texttt{f} \to \texttt{f}$$

Note that `foldBool` is just `ifthenelse` again.

Church encoding for arbitrary datatypes – contd.

Haskell:

```haskell
data Pair x y = Pair x y
```

```haskell
foldPair (Pair x y) p = p x y
```

Lambda calculus:

$$\texttt{pair} \equiv \lambda\ x\ y \rightarrow (\lambda\ p \rightarrow (p\ x\ y))$$

Encoding vs. adding constants

The fact that we can encode certain entities in the lambda justifies that we can add them as constants to the language without changing the nature of the language.

Example (adding Booleans)

```
e  ::=  true
    |    false
    |    if e then e else e
```

Once we have new forms of expressions, we need more than just beta-reduction:

```
if true   then e1 else e2 ⟶  e1
if false  then e1 else e2 ⟶  e2
```

Church Booleans

$$\texttt{true} \equiv \lambda\ \texttt{t}\ \texttt{f} \rightarrow \texttt{t}$$

$$\texttt{false} \equiv \lambda\ \texttt{t}\ \texttt{f} \rightarrow \texttt{f}$$

$$\texttt{ifthenelse} \equiv \lambda\ \texttt{c}\ \texttt{t}\ \texttt{e} \rightarrow \texttt{c}\ \texttt{t}\ \texttt{e}$$

Church Booleans

$$\begin{aligned}
\texttt{true} &\equiv \lambda\ \texttt{t f} \longrightarrow \texttt{t}\\
\texttt{false} &\equiv \lambda\ \texttt{t f} \longrightarrow \texttt{f}\\
\texttt{ifthenelse} &\equiv \lambda\ \texttt{c t e} \longrightarrow \texttt{c t e}
\end{aligned}$$

The function `ifthenelse` is almost the identity function.

$$\begin{aligned}
\texttt{and} &\equiv \lambda\ \texttt{x y} \longrightarrow \texttt{ifthenelse x y false}\\
\texttt{and} &\equiv \lambda\ \texttt{x y} \longrightarrow \texttt{x y false}\\
\texttt{or} &\equiv \lambda\ \texttt{x y} \longrightarrow \texttt{ifthenelse x true y}\\
\texttt{or} &\equiv \lambda\ \texttt{x y} \longrightarrow \texttt{x true y}
\end{aligned}$$

Church Booleans

$$
\begin{aligned}
\texttt{true} &\equiv \lambda\ \texttt{t f} \rightarrow \texttt{t} \\
\texttt{false} &\equiv \lambda\ \texttt{t f} \rightarrow \texttt{f} \\
\texttt{ifthenelse} &\equiv \lambda\ \texttt{c t e} \rightarrow \texttt{c t e}
\end{aligned}
$$

The function `ifthenelse` is almost the identity function.

$$
\begin{aligned}
\texttt{and} &\equiv \lambda\ \texttt{x y} \rightarrow \texttt{ifthenelse x y false} \\
\texttt{and} &\equiv \lambda\ \texttt{x y} \rightarrow \texttt{x y false} \\
\texttt{or} &\equiv \lambda\ \texttt{x y} \rightarrow \texttt{ifthenelse x true y} \\
\texttt{or} &\equiv \lambda\ \texttt{x y} \rightarrow \texttt{x true y}
\end{aligned}
$$

The function `isZero` takes a number and returns a Bool.

$$
\texttt{isZero} \equiv \lambda\ \texttt{n} \rightarrow \texttt{(n (}\lambda\ \texttt{x} \rightarrow \texttt{false) true)}
$$

Towards Haskell

These definitions are somewhat magical – but all form examples of a more general encoding of data types into lambda terms, known as the *Church encoding*.

Haskell is based on a typed lambda calculus, extended with constructs for type-equalities and conversions necessary to account for GADTs.

Yet, so far it seems hard to believe that we can desugar Haskell to some form of lambda calculus.

Binding names with `let`

Haskell allows us to bind identifiers to expressions in the language using `let`.

We have only introduced informal abbreviations for lambda terms so far such as `true` or `isZero`.

Binding names with `let` – contd.

In fact, `let` can simply be desugared to a lambda binding.

```
let x = e1 in e2  ≡ (λ x → (e2)) e1
```

Note that this does not work if `x` is a recursive binding or if you want to preserve sharing.

What about recursion, then?

Recursion

Haskell example

```
fac = \ n → if n == 0 then 1 else n * fac (n - 1)

fac = fix
  (\ fac n → if n == 0 then 1
                     else n * fac (n - 1))
```

The desired function `fac` can be viewed as a fixed point of the related non-recursive function `fac’`.

Fixed points

A *fixed-point combinator* is a combinator `fix` with the property that for any `f`,

```
-- Using recursion directly
fix f = f (fix f)
```

In particular,

```
fix fac' = fac' (fix fac')
```

thus `fix fac’` is a fixed point of `fac`.

Recap

It is thus possible to desugar a recursive Haskell definition into the lambda calculus by translating recursion into applications of `fix`.

Conversely, we can justify adding recursion as a construct to the lambda calculus without changing its essential nature.

General vs. structural recursion

Note that most recursive functions can actually be defined without a fixed-point combinator. We have already defined `add`:

$$\texttt{add} \quad \equiv \quad \lambda\ \texttt{m n} \rightarrow (\texttt{m suc n})$$

In Haskell, `add` would be recursive

```
data Nat = Suc Nat | Zero


add (Suc m)  n  =  Suc (add m n)
add Zero     n  =  n
```

but can also be defined in terms of `foldNat`:

```
add m n = foldNat m Suc n
```

General vs. structural recursion – contd.

Functions defined in terms of a fold function are called *structurally recursive*.

Recursion using the fixed-point combinator is called *general recursion*.

Writing functions using general recursion is often perceived as simpler or more direct.

Structural recursion is often more well-behaved. For instance, for many datatypes it can be proved that if the arguments to the fold terminate, the structurally recursive function also terminates.

Pattern matching

In Haskell we can define functions using pattern matching:

```haskell
data Nat = Suc Nat | Zero

pred (Suc m)  = m
pred Zero     = Zero
```

Question

How can we define `pred` for the Church numerals?

Case function

Alternatively, pattern matching via case on a natural number can be captured as a function:

```
caseNat :: Nat → (Nat → r) → r → r
caseNat (Suc n)  s z  =  s n
caseNat Zero     s z  =  z

pred = \ m → caseNat  m
               (\ m' → m')
               Zero
```

The case function can be expressed in terms of the fold for that datatype, and hence the Church encoding.

Case function – contd.

Haskell:

```
caseNat :: Nat → (Nat → r) → r → r
caseNat (Suc n)  s z  =  s n
caseNat Zero     s z  =  z

foldNat :: Nat → (s → s) → s → s
foldNat (Suc n)  s z  =  s (foldNat n s z)
foldNat Zero     s z  =  z
```

Case via fold

We call `foldNat` choosing `s ≡ (r, Nat)` – that is pairing the return type and natural number:

```
caseNat n s z  ≡
  fst (foldNat n (\ (_,r) → (s r, Suc r))
                            (z, zero))
```

The second component of the pair just constructs the natural number again. This is how we can access the predecessor!

Nested patterns

Haskell allows nested patterns, too:

```
fib Zero            =  Zero
fib (Suc Zero)      =  Suc Zero
fib (Suc (Suc n))   =  add (fib n) (fib (Suc n))
```

These can easily be desugared to nested applications of `case` using only flat patterns (and hence to applications of `caseNat`):

```
fib n = case n of
          Zero    → Zero
          Suc n'  → case n' of
                      ...
```

Recap

We have seen how most Haskell constructs can be desugared to the lambda calculus:

- constructors of datatypes using the Church encoding,
- non-recursive `let` using lambda abstractions,
- general recursion using a fixed-point combinator,
- pattern matching using possibly nested applications of case functions.

Recap – contd.

Many other Haskell constructs can be expressed in terms of the ones we have already seen – for instance:

- `where`-clauses can be transformed into `let`
- `if-then-else` can be expressed as a function
- list comprehensions can be transformed into applications of `map`, `concat` and `if-then-else`
- monadic `do` notation can be transformed into applications of a limited number of functions