SYDDANSK UNIVERSITET
UNIVERSITY OF SOUTHERN DENMARK

Lecture 8: Type Level Programming

Søren Haagerup

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

December 11, 2016

Introduction

- **Type level programming** is an activity which can only be performed in languages with advanced type system features.
- Scala and Haskell are general-purpose languages where it is possible to do *some* Type-level programming.
- It might seem strange at first, but it is actually useful - we will also see examples of this.

Peano numbers represented by types

Recall the peano numbers

$$\textbf{data}\ Nat = Z \mid S\ Nat$$

where the numbers 0,1,2... are represented by the values $Z, S\ Z, S\ (S\ Z) \ldots$. If we were to define this on the type level instead, we could introduce empty data declarations.

$$\textbf{data}\ Z$$
$$\textbf{data}\ S\ n$$

where the numbers 0,1,2... are represented by the **types** $Z, S\ Z, S\ (S\ Z) \ldots$.
The only member of the types is $\bot$. I.e. $\bot :: S\ (S\ Z)$.

Type Classes $\approx$ Prolog Predicates

So let's say that we have defined

$$\mathbf{data}\ Z$$
$$\mathbf{data}\ S\ n$$

It turns out that the Prolog program

```
even(z).
even(s(N)):-odd(N).
odd(s(N)):-even(N).
```

corresponds quite closely to the following Haskell construction:

$$\begin{array}{ll}
\mathbf{class}\ Even\ n & \\
\mathbf{class}\ Odd\ n & \\
\mathbf{instance} & Even\ Z \\
\mathbf{instance}\ Odd\ n \Rightarrow & Even\ (S\ n) \\
\mathbf{instance}\ Even\ n \Rightarrow & Odd\ (S\ n)
\end{array}$$

Prolog/Haskell sessions

```
| ?- even(z).
yes
| ?- even(s(z)).
no
```

In the Haskell context, the type n is even, if it is an instance of the type class *Even n*.

$$> :t\ (\perp :: ((\mathit{Even}\ n) \Rightarrow n)) :: Z$$
$$(\perp :: ((\mathit{Even}\ n) \Rightarrow n)) :: Z :: Z$$
$$> :t\ (\perp :: ((\mathit{Even}\ n) \Rightarrow n)) :: S\ Z$$
$$< \mathit{interactive} >: 1 : 2 :$$
$$\mathit{No}\ \textbf{instance}\ \mathit{for}\ (\mathit{Odd}\ Z)\ \mathit{arising\ from\ an\ expression}\ \textbf{type}\ \mathit{signature}$$
$$\mathit{In\ the\ expression} : (\perp :: \mathit{Even}\ n \Rightarrow n) :: S\ Z$$

What about more advanced Prolog programs?


```
add(0,B,B).
add(s(A),B,s(C)) :- add(A,B,C).
```

We can model this relation using *multiparameter type classes*

$$
\begin{array}{lr}
\textbf{class}\ \mathit{Add}\ a\ b\ c & \\
\textbf{instance} & \mathit{Add}\ Z\ b\ b \\
\textbf{instance}\ \mathit{Add}\ a\ b\ c \Rightarrow & \mathit{Add}\ (S\ a)\ b\ (S\ c)
\end{array}
$$

What about more advanced Prolog programs?

$$\begin{array}{lll} \textbf{class} & \mathit{Add}\ a\ b\ c & \\ \textbf{instance} & & \mathit{Add}\ Z\ b\ b \\ \textbf{instance} & \mathit{Add}\ a\ b\ c \Rightarrow & \mathit{Add}\ (S\ a)\ b\ (S\ c) \end{array}$$

We can now verify an addition 1+3 = 4. To help us, define the following "values":

$$\begin{aligned} &\mathit{one} :: S\ Z \\ &\mathit{one} = \bot \\ &\mathit{three} :: S\ (S\ (S\ Z)) \\ &\mathit{three} = \bot \\ &\mathit{add} :: \mathit{Add}\ a\ b\ c \Rightarrow a \to b \to c \\ &\mathit{add} = \bot \end{aligned}$$

The addition is verified, since the following query type checks:

$$\begin{aligned} &> :t\ \mathit{add}\ \mathit{one}\ \mathit{three} :: S\ (S\ (S\ (S\ Z))) \\ &\mathit{add}\ \mathit{one}\ \mathit{three} :: S\ (S\ (S\ (S\ Z))) :: S\ (S\ (S\ (S\ Z))) \end{aligned}$$

Multiparameter type classes + Functional dependencies ≈ Type level Functions

To define type level **functions** and not just **relations**, we need a way to tell the compiler, that the "result" can be computed from the input. In Haskell, one can write:

$$
\begin{array}{ll}
\textbf{class } \mathit{Add}\ a\ b\ c \mid a\ b \rightarrow c & \\
\textbf{instance} & \mathit{Add}\ Z\ b\ b \\
\textbf{instance } \mathit{Add}\ a\ b\ c \Rightarrow & \mathit{Add}\ (S\ a)\ b\ (S\ c)
\end{array}
$$

$$
\begin{array}{l}
> :t\ \mathit{add}\ \mathit{three}\ \mathit{three} \\
\mathit{add}\ \mathit{three}\ \mathit{three} :: S\ (S\ (S\ (S\ (S\ (S\ Z)))))
\end{array}
$$

Practical Example: Length-indexed lists

We would like to introduce a data type $Vec\ a\ n$ which represents lists containing elements of type a, which is of length n. For example:

$$
\begin{aligned}
&[\,] :: Vec\ Char\ Z \\
&[\texttt{'a'}] :: Vec\ a\ (S\ Z) \\
&[\texttt{'a'},\texttt{'b'}] :: Vec\ a\ (S\ (S\ Z)) \\
&[\texttt{'a'},\texttt{'b'},\texttt{'c'}] :: Vec\ a\ (S\ (S\ (S\ Z)))
\end{aligned}
$$

We could then define

$$
\begin{aligned}
&head :: Vec\ a\ (S\ n) \rightarrow a \\
&tail :: Vec\ a\ (S\ n) \rightarrow Vec\ a\ n \\
&safeZip :: Vec\ a\ n \rightarrow Vec\ b\ n \rightarrow Vec\ (a,b)\ n
\end{aligned}
$$

Notice that *head* and *tail* would then be *total* functions, since the type checker discards the empty list.

Why is it useful?

1. At compile time, we can make sure that the program will never fail due to list length errors.
2. When implementing a matrix multiplication algorithm, where a matrix of size $m \times n$ should be multiplied by a matrix of size $n \times k$, we do not need error handling in our code, if we are just working with types like the one presented here.
3. There is no runtime overhead incurred by the extra type-level information - it is only used at compile time.

Introducing
Generalized Algebraic Data Types (GADTs)

It is not obvious how to define a type *Vec* with the properties from the previous slide, from what we have learned in this course.

The modern way to achieve it in Haskell is to use a **Generalized Algebraic Data Type** (GADTs).

Introducing Generalized Algebraic Data Types (GADTs)

The traditional ADT for lists

$$\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

can be written in GADT-notation by

$$\begin{aligned}&\textbf{data}\ List\ a\ \textbf{where}\\&\quad Nil :: List\ a\\&\quad Cons :: a \rightarrow List\ a \rightarrow List\ a\end{aligned}$$

We specify the complete type signature of the data constructors. Referring to our types Z and $S\ n$ from earlier, we can define

$$\begin{aligned}&\textbf{data}\ Vec :: * \rightarrow * \rightarrow *\textbf{where}\\&\quad Nil :: Vec\ a\ Z\\&\quad Cons :: a \rightarrow Vec\ a\ n \rightarrow Vec\ a\ (S\ n)\end{aligned}$$

head, *tail* and *safeZip*

Now we can define the functions announced earlier:

$$
\begin{aligned}
&\mathit{head} :: \mathit{Vec}\ a\ (S\ n) \rightarrow a \\
&\mathit{head}\ (\mathit{Cons}\ x\ \_) = x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{tail} :: \mathit{Vec}\ a\ (S\ n) \rightarrow \mathit{Vec}\ a\ n \\
&\mathit{tail}\ (\mathit{Cons}\ \_\ \mathit{xs}) = \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{safeZip} :: \mathit{Vec}\ a\ n \rightarrow \mathit{Vec}\ b\ n \rightarrow \mathit{Vec}\ (a, b)\ n \\
&\mathit{safeZip}\ \mathit{Nil}\ \mathit{Nil} = \mathit{Nil} \\
&\mathit{safeZip}\ (\mathit{Cons}\ x\ \mathit{xs})\ (\mathit{Cons}\ y\ \mathit{ys}) = \mathit{Cons}\ (x, y)\ (\mathit{safeZip}\ \mathit{xs}\ \mathit{ys})
\end{aligned}
$$

Implementing *replicate*

We would like to implement *replicate* for our length-indexed vectors.

$$replicate :: n \to a \to Vec\ n\ a$$

What is wrong with this type?
The previously mentioned functions only used $S\ n$ and Z on the type level. But sometimes it is useful to give $S\ n$ and Z as arguments to functions.

Singleton types: Chaining the type and the value level together

But we can add a singleton-type *Nat* which connects the value level and the type level:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Nat}\ \mathit{nat}\ \textbf{where} \\
\quad Z :: \mathit{Nat}\ Z \\
\quad S :: \mathit{Nat}\ n \to \mathit{Nat}\ (S\ n)
\end{array}
$$

Then we can define

$$
\begin{array}{l}
\mathit{replicate} :: \mathit{Nat}\ n \to a \to \mathit{Vec}\ n\ a \\
\mathit{replicate}\ Z\ \_ = \mathit{Nil} \\
\mathit{replicate}\ (S\ n)\ x = \mathit{Cons}\ x\ (\mathit{replicate}\ n\ x)
\end{array}
$$

What would be the type of $(+\!\!+)$?

Let's say we give input lists $xs :: Vec\ a\ n$ and $ys :: Vec\ a\ m$.
What could be a proper type for

$$
\begin{aligned}
&(+\!\!+) :: ?\\
&Nil \qquad\qquad\quad +\!\!+\ ys = ys\\
&(Cons\ x\ xs)\ +\!\!+\ ys = Cons\ x\ (xs +\!\!+ ys)
\end{aligned}
$$

What would be the type of $(+\!\!+)$?

Let's say we give input lists $xs :: Vec\ a\ n$ and $ys :: Vec\ a\ m$.
We can use our type class from earlier:

$$
\begin{aligned}
&(+\!\!+) :: (Add\ n\ m\ r) \Rightarrow Vec\ a\ n \to Vec\ a\ m \to Vec\ a\ r \\
&Nil \qquad\qquad\quad +\!\!+\ ys = ys \\
&(Cons\ x\ xs) +\!\!+\ ys = Cons\ x\ (xs +\!\!+\ ys)
\end{aligned}
$$

We can end up with a function $(+\!\!+)$ with the type given above, but we need to write the definition a bit differently.

Defining $(+\!\!+)$ as a part of the *Add* type class

$$
\begin{array}{l}
\textbf{class}\ \mathit{Add}\ a\ b\ c \mid a\ b \rightarrow c\ \textbf{where} \\
\quad (+\!\!+) :: \mathit{Vec}\ t\ a \rightarrow \mathit{Vec}\ t\ b \rightarrow \mathit{Vec}\ t\ c \\
\textbf{instance}\ \qquad\qquad\qquad\quad \mathit{Add}\ Z\ b\ b\ \textbf{where} \\
\quad \mathit{Nil} +\!\!+ \mathit{ys} = \mathit{ys} \\
\textbf{instance}\ \mathit{Add}\ a\ b\ c \Rightarrow \mathit{Add}\ (S\ a)\ b\ (S\ c)\ \textbf{where} \\
\quad (\mathit{Cons}\ x\ \mathit{xs}) +\!\!+ \mathit{ys} = \mathit{Cons}\ x\ (\mathit{xs} +\!\!+ \mathit{ys})
\end{array}
$$

Making life easier again

Haskell Programmers have not been so happy about the solution from before:

$$(+\!\!+) :: (Add\ n\ m\ r) \Rightarrow Vec\ a\ n \rightarrow Vec\ a\ m \rightarrow Vec\ a\ r$$

They asked:

1. Why do we need to add a type class constraint?
2. Why do we have to do our type level programming using the *logical programming* paradigm?

Wouldn't it be better to introduce a way to write *type level functions* using *functional programming* paradigm, and end up with a type like

$$(+\!\!+) :: Vec\ a\ n \rightarrow Vec\ a\ m \rightarrow Vec\ a\ (n :+ m)$$

where $(:+)$ is a function on types?

Type families ≈ Type level functions

The following is a quite new feature of Haskell:

$$
\begin{array}{l}
\textbf{infixl } 6 \mathbin{:+} \\
\textbf{type } \textit{family } n1 \mathbin{:+} n2 :: * \\
\textbf{type instance } Z \mathbin{:+} n2 \qquad = n2 \\
\textbf{type instance } S\ n1 \mathbin{:+} n2 = S\ (n1 \mathbin{:+} n2) \\
\textbf{infixr } 5 +\!\!+ \\
(+\!\!+) :: Vec\ a\ n1 \to Vec\ a\ n2 \to Vec\ a\ (n1 \mathbin{:+} n2) \\
Nil \qquad\quad +\!\!+ ys = ys \\
Cons\ x\ xs +\!\!+ ys = Cons\ x\ (xs +\!\!+ ys)
\end{array}
$$

Practical example: Function with variable number of arguments

Say, we would like to implement the following family of functions that sum up their integer arguments:

$$sum_n :: Int_1 \to \cdots \to Int_n \to Int$$
$$sum_n = \lambda i_1 \to \cdots \to \lambda i_n \to i_1 + \cdots + i_n$$

Solution using a Type Family

$$\textbf{type}\ \textit{family MultiArgs}\ n\ a :: *$$
$$\textbf{type instance}\ \textit{MultiArgs}\ Z\ a = a$$
$$\textbf{type instance}\ \textit{MultiArgs}\ (S\ n)\ a = a \to \textit{MultiArgs}\ n\ a$$

$$\begin{aligned}
&\textit{sum}' :: \textit{Nat}\ n \to \textit{Int} \to \textit{MultiArgs}\ n\ \textit{Int}\\
&\textit{sum}'\ Z\ \textit{acc} \qquad\ = \textit{acc}\\
&\textit{sum}'\ (S\ n)\ \textit{acc} = \lambda i \to \textit{sum}'\ n\ (\textit{acc} + i)
\end{aligned}$$

$$\begin{aligned}
&\textit{sum} :: \textit{Nat}\ n \to \textit{MultiArgs}\ n\ \textit{Int}\\
&\textit{sum}\ n = \textit{sum}'\ n\ 0
\end{aligned}$$

Solution using a Type Family

Example GHCi session:

$$> sum\ Z$$
$$0$$
$$> sum\ (S\ Z)$$
$$< interactive >: 3 : 1 :$$
$$\quad No\ \textbf{instance}\ for\ (Show\ (Int \to Int))\ arising\ from\ a\ use\ \textbf{of}\ `\ print\ '$$
$$\quad In\ a\ stmt\ \textbf{of}\ an\ interactive\ GHCi\ command : print\ it$$
$$> sum\ (S\ Z)\ 3$$
$$3$$
$$> sum\ (S\ (S\ Z))\ 3\ 4$$
$$7$$

Dependently Typed Programming

- A **dependent type** is a type that depends on a value.
- A dependent function's return type may depend on the value of an argument
- The examples we have just seen with length-indexed vectors and functions with a variable number of arguments, are examples of **dependently typed programming**.

Phase distinction: Types are not values

- In Haskell, types and values live in different scopes. We can use *singleton types* to chain types and values together.
- Very recent versions of GHC have *automatical promotion* of value level constructions to the type level.
- In general, features from the value level are being ported to the type level (functions $\rightarrow$ type families),
- In general, features from the type level are being ported to the kind level (type polymorphism $\rightarrow$ kind polymorphism)
- Other languages, like **Agda** and **Coq** have an infinite stack of *levels*. And concepts from lower levels can be used on higher leves. In Haskell, for now, we are restricted to the small hierarchy

  $$value : type : kind : sort$$

  which - for practical purposes - seems to be enough.

A taste of the
Curry-Howard Isomorphism using Haskell

- A deep result from Computer Science, is the direct relationship between **computer programs** and **mathematical proofs**.
- This correspondance is the foundation of programming languages like **Agda** and **Coq**
- Essentially there is a bijection such that
  - Mathematical Propositions $\approx$ Types
  - Mathematical Proofs $\approx$ Programs

A taste of the Curry-Howard Isomorphism using Haskell

The *reverse* function is essentially the following:

$$
\begin{aligned}
&\mathit{reverse} :: \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ n\\
&\mathit{reverse}\ \mathit{xs} = \mathit{go}\ \mathit{Nil}\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{go} :: \mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ k \to \mathit{Vec}\ a\ (k :+ m)\\
&\mathit{go}\ \mathit{acc}\ \mathit{Nil} = \mathit{acc}\\
&\mathit{go}\ \mathit{acc}\ (\mathit{Cons}\ x\ \mathit{xs}) = \mathit{go}\ (\mathit{Cons}\ x\ \mathit{acc})\ \mathit{xs}
\end{aligned}
$$

But it does not type check out of the box... It comes up with two quite clear error messages, though:

1. In def. of *reverse*: Couldn't match type n with $n :+ Z$
2. In def. of *go*: Could not deduce $((n :+ S\ m) {\sim} S\ (n :+ m))$ from the context $(k {\sim} S\ n)$ bound by a pattern with constructor

$$\mathit{Cons} :: a \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (S\ n)$$

A taste of the Curry-Howard Isomorphism using Haskell

We essentially need to prove the following statements about our Peano numbers defined on type level

$$\forall n . n :+ Z \equiv n$$

and

$$\forall n, m . (n :+ (S\ m)) \equiv (S\ (n :+ m))$$

As indicated earlier, a proof corresponds to *program code*.

A taste of the C-H Iso. using Haskell

Below, we have extended our reverse definition with the needed proofs:

$$
\begin{aligned}
&go :: Vec\ a\ m \rightarrow Vec\ a\ k \rightarrow Vec\ a\ (k :+ m)\\
&go\ acc\ Nil = acc\\
&go\ acc\ (Cons\ x\ xs) = subst\ (plus\_suc\ (size\ xs)\ (size\ acc))\\
&\qquad\qquad\qquad\qquad \$\ go\ (Cons\ x\ acc)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&reverse :: Vec\ a\ n \rightarrow Vec\ a\ n\\
&reverse\ xs = subst\ (plus\_zero\ (size\ xs))\\
&\qquad\qquad \$\ go\ Nil\ xs
\end{aligned}
$$

I have not explained how *subst*, *size*, *plus_suc* and *plus_size* is defined[1] - this could be a subject for another course.

[1] The details can be found on
`http://dev.stephendiehl.com/hask/#advanced-proofs`

This is it

- This was the last lecture of the Programming Languages course.
- Your exam is **The obligatory assignment**, due 9th of January at 23.59.
- My exam is **The course evaluation**, to be filled out by you now. To remind you, this was what we covered in each of the 8 lectures:
  1. Expressions, Types and Patterns
  2. List algorithms, explicit recursion and list comprehensions
  3. List algorithms using higher order functions
  4. Algebraic Data Types
  5. Lazy Evaluation and Infinite Data Structures
  6. Proving and Testing Program Properties
  7. Monads and IO
  8. Type Level Programming