How I Teach Functional Programming

Johannes Waldmann

Fakultät IMN, HTWK Leipzig, D-04275 Leipzig

Abstract. I teach a course *Advanced Programming* for 4th semester bachelor students of computer science. In this note, I will explain my reasons for choosing the topics to teach, as well as their order, and presentation. In particular, I will show how I include automated exercises, using the Leipzig `autotool` system.

1 Motivation

In my course *Advanced Programming*, I aim to show mathematical models (say, the lambda calculus) as well as their concrete realization first in its "pure" form in a functional programming language (Haskell [Mar10]), then also in some of today's multi-paradigm languages (imperative and object-oriented, actually mostly class-based) that students already known from their programming courses in their first and second semester. My course is named "advanced" because it introduces functional programming (but it's not advanced functional programming).

I will explain this in some detail now. Motivation (this section) and discussion will be somewhat opinionated, starting with the following paragraph. I still think it will be clear on what facts my opinion is based, and I hope that at least the facts (examples and exercises) are re-usable by other academic teachers of functional programming. The slides for the most recent instance of this lecture are at `https://www.imn.htwk-leipzig.de/~waldmann/edu/ss17/fop/folien/`, and online exercises can be tried here: `https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?lecture=238`

When teaching programming, the focus should really be on how to solve the task at hand (given an algorithm, produce executable code) using available tools (programming languages, libraries), and to understand the fundamentals that these tools are based on. We should not commit to teaching random quirks of programming languages of the day — although it can be instructive to compare languages, and follow their design process.

It is interesting to note that major "advances" in "mainstream" languages largely consist of the introduction of ideas that are well-known from functional programming, for decades. Java 5 (2004): generic polymorphism, first in ML (1975); Java 8 (2014): lambda expressions, first in LISP (1958); C# LINQ (2007): (higher order functions and) monads, first in Haskell (1990). Before these languages admitted functional programming, there was a whole cottage industry for simulating higher-order functions as "design patterns" [GHJV95]. And of course

JavaScript (1995) is exactly LISP (1958) — with the superficial change that lots of parentheses had been replaced with lots of braces, to fool the unsuspecting C programmer.

2 Topics

The course contains these topics, which are presented in the order given, except for those in parentheses, which will be delayed by a few weeks.

- first-order data:
  - model: trees over a signature,
  - Haskell realization: algebraic data types
  - (object-oriented simulation: the *composite* design pattern)
- first order programs:
  - model: term rewriting, equational reasoning
  - Haskell realization: oriented equations and pattern matching
- higher-order data and programs
  - model: lambda calculus
  - pure realizations: actual lambda expressions,
  - (alternative realizations: the *strategy* design pattern, functional interfaces in Java 8)
- application: patterns for systematic recursion
  - model: algebra over a signature
  - Haskell realization: fold
  - simulation: the *visitor* design pattern
- restricted polymorphism
  - Haskell: type classes,
  - simulation: Java: interfaces, bounds on type parameters
  - killer application: type-directed generation of test cases
- evaluation on demand, in particular, for infinite streams
  - model: (non) strictness of functions, lazy evaluation,
  - simulation: the iterator design pattern
- higher order functions for stream processing
- functional reactive programming
  - model: behaviours (time-dependent values) and events,
  - alternative: the *observer* design pattern

The course is mandatory for B. Sc. students of computer science (Informatik and Medieninformatik) in the 4th semester. It consists of one lecture, and one lab class, per week. I've been teaching it for roughly five years now, in its present form. Details change. The latest addition is functional reactive programming.

3 Exercises

The course also includes homework. Some exercises require proofs (well, let's say, they require argumentation) and will be discussed in class. Others are of a more mechanical nature, so they can be graded automatically (and still be discussed in class). For automation, I use the Leipzig `autotool` system [Wal17a].

Automated exercises come in two variants:

– with domain-specific syntax and semantics: e.g., for term rewriting and lambda calculus,
– with Haskell syntax and semantics: students fill in holes in a Haskell program, such that predefined tests are valid.

For the first variant, `autotool` runs an evaluation in a problem-specific monad that would typically print a lot of information along the way. For the second variant, `autotool` evaluates an actual Haskell expression, so there is no tracing. Students are encouraged to evaluate expressions from the program on their own, in a `ghci` session.

4 First Order Data and Programs

In the lecture, tree-like `data` is described first, and computation on that data (via pattern matching, `case`) next. I try to avoid built-in data types for numbers (or later, lists). When I need numbers, I write the data declaration for Peano numerals.

4.1 Exercise on Algebraic Data Types

The mathematical model for trees is terms over a signature. This is known from the course *Modeling* in the first semester. Students learn the Haskell representation, and solve exercises like this one:

– Instance: a set of `data` declarations, a type T, a number n
– Solution: n distinct elements of type T

Example instance (this is the full text of the exercise):

```
module Blueprint where
import qualified Data.Set as S
-- imported from Prelude:
-- data Bool = False | True
data C = R | G | B deriving (Eq, Ord, Show)
data T = X C | Y Bool Bool deriving (Eq, Ord, Show)
solution :: S.Set T
solution = S.fromList undefined
test :: Bool
test = S.size solution == 7
```

This is a pattern for a Haskell program text. The student is to replace `undefined` with an expression, such that `test` evaluates to `True`.

Example solution (only the relevant lines are shown):

```
solution :: S.Set T
solution = S.fromList  [ X R, X G, X B
  , Y False False, Y False True, Y True False, Y True True ]
```

For recursive data types, we can set a lower bound on the cardinality:

```
data C = R | G | B deriving (Eq, Ord, Show)
data D = U | V C deriving (Eq, Ord,Show)
data S = P | Q D S  deriving (Eq, Ord, Show)
solution :: S.Set S
solution = S.fromList undefined
test :: Bool
test = S.size solution >= 7
```

With all "complete the code" exercises, a design goal is to have them self-contained. The student can load the problem statement in a `ghci` session, and it should be correct syntactically and statically. This works since `undefined` has any type. The student can then change the source text, and evaluate `test`, or any other expression. It seems unavoidable that the code will contain items that the student cannot understand from the lecture alone (at this point). In this example, this applies to `deriving (Eq, Ord, Show)`, and the use of `Data.Set`.

4.2 Exercise on Term Rewriting

The mathematical model for processing tree-like data is *term rewriting* [BN98]. The following type of exercise helps to get a basic understanding of this model of computation.

- Instance: a term rewriting system R over signature Σ, terms s and t over Σ;
- Solution: a sequence of R-rewrite steps that transforms s to t, where a step is given by
  - (number of) rule to apply,
  - position of application (where position is sequence of natural numbers),
  - substitution for variables in rule.

This definition is copied verbatim from the definition of the rewrite relation $s \to_R t : \exists (l, r) \in R, p \in \mathsf{Pos}(s), \sigma \in \mathsf{Var} \to \mathsf{Term}(\Sigma) : s[p] = l\sigma \wedge t = s[p := r\sigma]$ that was given in the lecture.

Example instance:

```
for the system  TRS
        { variables = [ x, y, z]
        , rules = [ f (f (x, y), z) -> f (x, f (y, z))
                  , f (x, f (y, z)) -> f (f (x, y), z) ] }
```

```
give a sequence of steps
from f (f (f (a , b ), f (c , d )), e )
to f (a , f (f (b , c ), f (d , e )))
```

Example solution (attempt):

```
( f (f (f (a , b ), f (c , d )), e )
, [ Step { rule_number = 0 , position = [ 0, 1 ]
         , substitution = listToFM
            [ ( x, f (a , b )), ( y, f (c , d )), ( z, e ) ] } ] )
```

Example output of `autotool` for above input (slightly edited)

```
apply step  Step { rule_number = 0, ... }
to term
    f (f (f (a , b ), f (c , d )), e )
the rule number 0
  is f (f (x, y), z) -> f (x, f (y, z))
the subterm at position [ 0, 1]
  is f (c , d )
the instantiated lhs is
  f (f (f (a , b ), f (c , d )), e )
agrees with subterm at position?
  No.
```

This exercise type allows some modifications (not all are implemented currently):

- minimal, maximal length of sequence is given
- start term not given, target term not given, or given by a constraint (e.g., a basic term, a normal form)
- admissible steps restricted by some strategy (e.g., outermost)

It is important to note that this is an exercise on (a model of) programming, but the task is not "guess the program" but "describe the execution of a given program" (by giving its execution steps in detail).

4.3 Exercises on First Order Programs

The main point is case distinctions on algebraic data types. In the lecture, I emphasize that it is best if a set of patterns in a case distinction is complete (it covers all values) and disjoint, and this is achieved easily by writing one pattern for each constructor of the data type of the discriminant. I strongly recommend to write these patterns in order of declaration, e.g., Zero before Successor, and Nil before Cons (but this needs polymorphism, which happens only later).

Here is an example for an exercise that I use in class: Given

```
data Bool = False | True
data T = F T | G T T T | C
```

answer for each of the following expressions:

– is it syntactically correct
– is statically correct
– what is its result (its dynamic semantics)
– is the pattern match complete? disjoint?

```
 1. case False of { True -> C }
 2. case False of { C -> True }
 3. case False of { False -> F F }
 4. case G (F C) C (F C) of { G x y z -> F z }
 5. case F C of { F (F x) -> False }
 6. case F C of { F (F x) -> y }
 7. case F C of { F x -> False ; True -> False }
 8. case True of { False -> C ; True -> F C }
 9. case True of { False -> C ; False -> F C }
10. case C of { G x y z -> False; F x -> False; C -> True }
```

For self-study exercises, I use "complete this Haskell code", cf. Subsection 4.1.

– instance: a pattern for Haskell program (`undefined` can be replaced by arbitrary expression) that contains a definition of an expression `test::Bool`
– solution: a program that is an instance of the pattern such that `test` evaluates to `True`.

The specification can be given by a concrete test case, but it is much better to give it as a *property*: a function $p : D \to$ `Bool` that encodes $\forall x \in D : p(x)$. Students should learn from the start to use property-based testing, e.g., `smallcheck` [Che13], and later they can also learn how automatic enumeration of test cases works (with Haskell type classes).

Care must be taken to not reveal the answer when writing the property. As an example, the task is to implement the `min` operation for Peano numbers, and we can specify it using a correct implementation of addition

```
import Prelude hiding (min)

data N = Z | S N deriving (Show , Eq)

min :: N -> N -> N
min x y = undefined

spec1 = \ x y -> min x y == min y x
spec2 = \ x y -> min (plus x y) x == x

plus :: N -> N -> N
plus x y = case x of { Z -> y ; S x’ -> S (plus x’ y) }
```

For being self-contained, exercises should only use libraries that are standard (`base`), or that are easily installable (`smallcheck`). There should be no hidden test cases: all of the tests (the specifications) are visible. This takes some pressure off `autotool` since all computations should be done on the student's side.

4.4 A Side Remark on Automated Testing

I think that `smallcheck` is too complicated already, as its implementation uses `LogicT`, which cannot be explained easily. For that reason, I am inclined to switch to `leancheck` [Mat16] because they essentially have

```
class Serial a where series :: [[a]]
```

which *can* be explained later, cf. Subsection 6.3. To get instances of `Serial` for user-defined data types, I can write down an instance

```
instance Monad m => Serial m N where series = cons0 Z \/ cons1 S
```

or I can have them derived

```
data N = Z | S N deriving (Eq, Generic) ; instance Serial m N
```

Both involve magic, i.e., concepts (Monads, Generic) that the student cannot grasp at this point.

Also I found that `smallcheck` uses IO too often in its API. To work around this, I need (some lines of) boilerplate, and copy it in each exercise, like this:

```
test :: Bool
test = and [ null $ failures 10 1000 $ spec1
           , null $ failures 10 1000 $ spec2 ]
-- | first f failures from t testcases for property p
failures f t p = take f $ filter ( \ x -> not $ p x )
               $ take t $ do d <- [ 0 .. ] ; list d series
```

5 Polymorphism

A main motivation for polymorphic data (that is, type constructors), are container types (lists, trees). Again, I prefer to write all data declarations, and not use types from the Prelude. Students should definitely *not* get the impression that lists are somehow intrinsic to Haskell [Wal17b].

5.1 Exercise on polymorphic types

– instance: set of data declarations, some of them for type constructors, a number n, a type T
– solution: a set of n expressions of type T with distinct values

Example instance:

```
{- using these types and type constructors from Prelude:
data () = ()
data Bool = False | True
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
-}

data C = R | G | B deriving (Eq, Ord, Show)
data Pair a b = Pair a b deriving (Eq, Ord, Show)

solution :: S.Set (Either (Pair Bool (Maybe ())) (Maybe (Maybe C)))
solution = S.fromList undefined

test :: Bool
test = S.size solution >= 9
```

5.2 Exercise on polymorphic functions

They are of the type "complete this Haskell program", and earlier remarks apply. Now that have polymorphism, we can use it to emphasize some extra points:

We can specify a polymorphic type for what the student should write

```
reverse :: List a -> List a
reverse xs = undefined
```

and give a monomorphic test case

```
reverse (Cons True (Cons False Nil)) == Cons False (Cons True Nil)
```

The polymorphic type will prevent the student from writing

```
reverse xs = Cons False (Cons True Nil)
```

This makes it harder to cheat. Of course, one test is not enough, think of

```
reverse xs = append (tail xs) (Cons (head xs) Nil)
```

Then we can add another property like

```
prop xs = reverse (reverse xs) == xs
```

and it seems then we reached the threshold of "sufficient specification" where the intended solution is smaller than any cheating "solution".

6 Higher Order Data and Functions

The main motivation for higher order features is that they allow to abstract over programs, as a higher order function can be seen as a schema for a program. The underlying mathematical model is the lambda calculus [Bar84]. Of course, in the lecture I just use the basic computational model, and don't do any "theory". I mention termination and confluence.

6.1 Exercises on Lambda Calculus

This exercise is similar to the one on term rewriting (Subsection 4.2).

- instance: lambda terms s, t, possibly some restrictions
- solution: sequence of α and β steps that transforms s to t, and conforms to restrictions.

Example instance:

```
give a derivation that transforms  (\ x y -> y x) (y y) (\ x -> x)
                                 to  y y
```

Here, I use the standard abbreviations for nested lambdas, and nested application. To be absolutely sure, `autotool` will display the abstract syntax tree.

Example solution (attempt):

```
[ Step { position = [ 0 ]
       , action = Reduce
          { formal = x , body = \ y ->  y x , argument = y y } } ]
```

Example output trace (slightly edited):

```
Current step is  Step { position = [ 0], ... }
Subterm at position [ 0] ist
    (\ x y -> y x) (y y)
Error: when replacing x with y y:
       in \ y -> y x ,
       a free occurence of a variable would become bound
```

In a correct solution, this step should be prefixed with

```
Step  { position = [ 0,0,0]
      , action = Rename { from = y , to = z } }
```

The necessity of local renamings is also discussed in the lecture. It may appear as a technical shortcoming of the calculus, but similar problems will appear in any (programming) language that has local names, when the program is modified (refactored) by expanding function definitions. This can be shown by comparing the following Java snippets (that can be executed in `jshell` of JDK 9):

```
int x = 3;
int f(int y)
  { return x + y; }
int g(int x)
  { return (x + f(8)); }
// g(5) => 16
```

```
int x = 3;
int g(int x)
  { return (x + (x+8)); }
// g(5) => 18
```

As a side remark, I also show to the students that JavaScript [TWB17] meanwhile does almost behave like a real programming language. There is really not much remaining difference (well, except for lack of static typing) between

```
$ ghci                              $ node
GHCi, version 8.0.2                 > let d = f => x => f (f (x))
Prelude> let d f x = f (f x)        undefined
Prelude> d d d (\x -> x+1) 0        > d(d)(d)(x=>x+1)(0)
16                                  16
```

6.2 Exercises on Fold

A prime motivation for higher-order functions is that they can be used to express recursion schemes over recursive algebraic data types. The point is that "instantiating the schema" is just function application, because the schema *is a* function, and it needs to be higher-order for this to work.

In the lecture I use the phrase "when a fold is applied, each constructor gets replaced by a corresponding function", and from that, it is quite clear that each algebraic data type has exactly on recursion schema, and its type and implementation can be derived mechanically. I do this for Peano numbers, lists, trees; but it is also a nice exercise to have students construct and discuss the "fold" for `Bool`, `Maybe`, and `Either`. After this, I have students look up in `https://www.haskell.org/hoogle/` whether functions of that type are already in the standard library. Of course they are. This shows the value of static types for documentation.

Here is a programming exercise example: for Peano numbers, `fold` and `plus` are given, `minus` should be implemented.

```
data N = Z | S N deriving ( Eq, Show )

fold :: b -> (b -> b) -> (N -> b)
fold z s x = case x of
    Z -> z
    S x' -> s (fold z s x')

plus :: N -> N -> N
plus x y = fold y S x

minus :: N -> N -> N
minus x y = ( fold undefined undefined ) y

spec1 :: (N,N) -> Bool
spec1 = \ (x,y) -> minus (plus x y) y == x

spec2 :: (N,N) -> Bool
spec2 = \ (x,y) -> minus x (plus x y) == Z
```

In the lecture, I describe how to solve this kind of exercise "write function f as fold". The student should draw an example input tree (for the argument of the fold) and write at each node the required result of f on that subtree. Then they can read off test cases for the unknown arguments of fold. With the above example, let `x = S (S (S Z))` and consider `minus x (S (S Z))`. According to specification, the result is `S Z`. Subtrees to be evaluated are `minus x (S Z)` with result `S (S Z)` and `minus x Z` with result `S (S (S Z))`.

So if `minus x y = fold z s y`, then we see that `n = x` and, for example, `s (S (S (S Z))) = S (S Z)`. So quite clearly `s` is the predecessor function, with a modification (`pre Z = Z`).

Now I also make a point of showing that some functions are not folds. For instance, if we would have a presentation `pre = fold z s`, then on the one hand `Z = pre (S Z) = s (pre Z) = s Z` but on the other hand we have `S Z = pre (S (S Z)) = s (pre (S Z)) = s Z`, which is a contradiction, since `s` is a function. Note that this reasoning only works for pure functional programs.

This leads the way for an extension of the above exercise: `pre` can be defined as the projection of a helper function, which *is* a fold.

```
pre :: N -> N
pre x = case pre' x of
    Pair y z -> z

spec3 :: N -> Bool
spec3 = \ x -> pre (S x) == x

data Pair a b = Pair a b deriving ( Eq, Show )

pre' :: N -> Pair N N
pre' = fold undefined ( \ x -> case x of Pair y z -> undefined )

spec4 :: N -> Bool
spec4 = \ x -> pre' (S x) == Pair (S x) x
```

6.3 Type Classes

There are cases where full generic polymorphism does not capture the intention of the programmer. For instance, the type of `sort` cannot be

```
sort :: List a -> List a
```

because we need to be able to compare elements. So it is

```
sort :: Ord a => List a -> List a
```

I mention the standard type classes `Show, Eq, Ord`,

The “killer example” for type classes is automated type-directed generation of arguments for property tests. The essential classes (for leancheck) are

```
check :: Testable a => a -> IO ()
class Testable a where
  results :: a -> [([String],Bool)]
instance Testable Bool where
```

```
  results p = [([],p)]
instance (Listable a, Testable b)
  => Testable (a -> b) where ...
class Listable a where tiers :: [[a]]
instance Listable Int where ...
```

If an expression `check (\ x y -> x + y == y + x :: Int)` is type-checked, the compiler notes that

```
\ x y -> x + y == y + x :: Int -> (Int -> Bool)
```

and then goes on to prove that

```
instance Testable (Int -> (Int -> Bool))
```

can be derived from the axioms given above. This is a nice showcase of formal deduction put to very practical use.

I also compare to means of restricting polymorphism in Java. The situation is a bit messed up because of Java's historical baggage. For instance, `equals` is in `class Object` and thus has the wrong type, and it needs Eclipse's "derive (hashCode and) equals" (from the "source" menu) to implement it correctly.

But — Haskell has some historical baggage as well: notice the unnatural appearance of `:: Int` in the example above. Of course the natural place for this type declaration would be right where `x` is declared:

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

If I try this, ghc tells me that this is illegal (wat?) and that it can only be enabled with `ScopedTypeVariables`. WAT? Where's the type variable?

7 Strictness, Laziness, Streams

There are two motivations for demand-driven evaluation. One is that it may help to save work. The other, more important, is that it is needed for modularity of programming [Hug89]. The classical example is `if-then-else` as a function, which will not work as expected with strict evaluation, since it forces the values of both branches.

7.1 Strictness

In lecture, I give the usual definition that f is strict in a specific argument position iff $f(\dots,\bot,\dots) = \bot$. This definition can be applied directly in `ghci`:

```
False && undefined  -- does not raise an exception
```

This short-cut evaluation of Boolean connectives also shows that other languages have some non-strict operations, but only as special cases.

I go on to discuss reasons for evaluation. Where does the "demand" in "on-demand" evaluation come from? Without giving a full account of the dynamic semantics of Haskell, I teach the principle that: if the value of a `case` expression is demanded, then the value of its discriminant will be demanded, to the extent that is necessary for deciding which pattern matches, and then the value of the chosen branch is demanded.

Students can use this to prove that

```
f :: Bool -> Bool -> Bool
f x y = case y of { False -> x ; True  -> y }
```

is strict in the second argument, and they can evaluate `f undefined True` to prove that it's not strict in the first. Another question of this type is: analyze strictness of

```
g :: Bool -> Bool -> Bool -> Bool
g x y z =
  case (case y of False -> x ; True -> z) of
    False -> x
    True  -> False
```

and, assuming standard definitions of Boolean connectives,

```
f x y z = case y && z of
  False -> case x || y of
    False -> z
    True  -> False
  True -> y
```

Actually, it would be nice to have automation for this kind of exercise.

7.2 Streams

In particular, laziness is helpful for separating the processing of data streams into producer, transformer, and consumer. Often, the transformer is written using higher order functions (map, bind), and the consumer is a fold.

This is the only place in the lecture where students actually use Haskell's built-in list type and operations. In the lecture, I emphasize the difference between `foldr` (the "right" fold) and `foldl`. A nice exercise is: write a function

```
fromBits :: [Bool] -> Integer
fromBits [True,False,False,True,False] = 18
```

with one of the folds. Is it `foldl`, or is it `foldr`? Prove that one of these works, and that the other one does indeed not.

I also compare to the corresponding stream processing libraries LINQ for C# and `java.util.stream` for Java 8. I will not go into detail here, and just list the correspondence:

Haskell	LINQ	java.util.stream
[e]	IEnumerable<E>	Stream<E>
map	Select	map
(>>=)	SelectMany	flatMap
filter	Where	filter
foldl	Aggregate	reduce

7.3 Behaviours and Events

To conclude the lecture with a "real life" application of (ideas from) functional programming, I briefly discuss functional-reactive programming [EH97].

The approach is to give a mathematical model of behaviours as values that depend on the time, and events, which can be thought of as behaviours that are piece-wise constant, and can be represented as streams of pairs of time and value, with increasing time.

With this model, interactive software can be written in a modular way, by using combinators for behaviours and events. This is in contrast to the method of event processing via the observer pattern, which quickly leads to *callback hell.*

In class, I use `threepenny-gui` [Apf13]. One of this library's examples is

```
dollar <- UI.input ; euro   <- UI.input
getBody window #+ [
  column [ grid [[string "Dollar:", element dollar]
                ,[string "Euro:"  , element euro  ]]  ]]
euroIn   <- stepper "0" $ UI.valueChange euro
dollarIn <- stepper "0" $ UI.valueChange dollar
let rate = 0.7 :: Double
    withString f = maybe "-" (printf "%.2f") . fmap f . readMay
    dollarOut = withString (/ rate) <$> euroIn
    euroOut   = withString (* rate) <$> dollarIn
element euro   # sink value euroOut
element dollar # sink value dollarOut
```

This uses combinators for behaviours and events like

```
stepper :: MonadIO m => a -> Event a -> m (Behavior a)
```

and general combinators like

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

The point is that the `Functor` abstraction is standard (not specific to this library) and makes it easier to understand and to use. Behaviours also instantiate `Applicative`, so we can use

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

to combine them. A quick exercise for the above code is to add another output element that shows the concatenation of `dollar` and `euro` behaviours. This can be done with

```
(++) <$> dollarOut <*> euroOut
```

I admit that this is on the border from introductory to advanced functional programming, and a full explanation is outside the scope of the lecture. On the other hand, this may just be a gut reaction by the seasoned Haskell programmer ("they destroyed Haskell with `Applicative`, `Foldable`, and whatnot") that sees his familiar environment (e.g., the type of `foldl`) change.

8 Discussion

Regarding evolution of programming languages, I find it amusing to observe current developments towards, and also against, static typing.

Static types document design decisions in a way that can be checked by machine. So it detects software errors at compile time, when they are cheaper to repair (instead of runtime), and it allows for more efficient execution (by the removal of runtime type checks). So, alongside higher order functions and generic polymorphism mentioned above - there can be no doubt that static typing is the correct thing to do, and thus, the right thing to teach. In Dijkstra's words, "It is not the task of the University to offer what society asks for, but to give what society needs." [Dij00]. I claim that society needs static typing.

But what do market forces in the software industry think? On the one hand, there are serious proposals to introduce gradual type systems [Sav14] for untyped languages (which are marketed as "dynamic"). On the other hand, there are proposals to open up JVM and C# for "dynamics". I think all of this is mostly marketing, and it should not influence teaching.

Let us hope that TypeScript [typ17] (or MyPy [Leh17]) wins before departments succumb and throw out statically typed languages from introductory courses.

References

Apf13. Heinrich Apfelmus. threepenny-gui: GUI framework that uses the web browser as a display. `https://wiki.haskell.org/Threepenny-gui`, 2013.

Bar84. Henk Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North Holland, Amsterdam, 1984.

BN98. Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

Che13. Roman Chelplyaka. smallcheck: A property-based testing library. `https://hackage.haskell.org/package/smallcheck`, 2013.

Dij00. Edsger W. Dijkstra. Answers to questions from students of Software Engineering. `https://www.cs.utexas.edu/~EWD/transcriptions/EWD13xx/EWD1305.html`, 2000.

EH97. Conal Elliott and Paul Hudak. Functional Reactive Animation. In *International Conference on Functional Programming*, 1997. `http://conal.net/papers/icfp97/`.

GHJV95. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

Hug89. J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

Leh17. Jukka Lehtosalo. Mypy—an experimental optional static type checker for Python. `http://mypy-lang.org/`, 2017.

Mar10. Simon Marlow. Haskell 2010 Language Report. `https://www.haskell.org/onlinereport/haskell2010/`, 2010.

Mat16. Rudy Matela. leancheck: Cholesterol-free property-based testing. `https://hackage.haskell.org/package/leancheck`, 2016.

Sav14. Neil Savage. Gradual evolution. *Commun. ACM*, 57(10):16–18, September 2014.

TWB17. Brian Terlson and Allan Wirfs-Brock. ECMAScript 2017 Language Specification. `http://www.ecma-international.org/ecma-262/8.0/`, 2017.

typ17. TypeScript—A Typed Superset of JavaScript. `http://www.typescriptlang.org/`, 2017.

Wal17a. Johannes Waldmann. Leipzig autotool. `https://gitlab.imn.htwk-leipzig.de/autotool/all0`, 2017.

Wal17b. Johannes Waldmann. When You Should Use Lists in Haskell (Mostly, You Should Not). `https://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/list-or-not-list/`, 2017.