How functional programming mattered

REVIEW

National Science Review
2: 349–370, 2015
doi: 10.1093/nsr/nwv042
Advance access publication 13 July 2015

COMPUTER SCIENCE

How functional programming mattered

Zhenjiang Hu[1,2,*], John Hughes[3] and Meng Wang[4]

ABSTRACT

In 1989 when functional programming was still considered a niche topic, Hughes wrote a visionary paper arguing convincingly 'why functional programming matters'. More than two decades have passed. Has functional programming really mattered? Our answer is a resounding 'Yes!'. Functional programming is now at the forefront of a new generation of programming technologies, and enjoying increasing popularity and influence. In this paper, we review the impact of functional programming, focusing on how it has changed the way we may construct programs, the way we may verify programs, and fundamentally the way we may think about programs.

Keywords: functional programming, functional languages, equational reasoning, *monad*, high order function

INTRODUCTION

Twenty-five years ago, Hughes published a paper entitled 'Why Functional Programming Matters' [1], which has since become one of the most cited papers in the field. Rather than discussing what functional programming isn't (it has no assignment, no side effects, no explicit prescription of the flow of control), the paper emphasizes what functional programming *is*. In particular, it shows that two distinctive functional features, namely higher order functions and lazy evaluation, are capable of bringing considerable improvement in modularity, resulting in crucial advantages in software development.

Twenty-five years on, how has functional programming mattered? Hughes's vision has become more widely accepted. Mainstream languages such as C#, C++, and Java scrambled one after another to offer dedicated support for *lambda expressions*, enabling programming with higher order functions. Lazy evaluation has also risen to prominence, with numerous papers on new ways to exploit its strengths and to address its shortcomings.

One way to gauge the popularity of functional programming is through its presence at conferences both in academia and industry. The ACM International Conference of Functional Programming grew to 500 participants in 2014. Developer conferences on functional programming abound—such as the Erlang User Conference/Factory in Stockholm, London and San Francisco, Scala Days and Clojure West in San Francisco, Lambda Jam in Chicago, Lambda Days in Krakow—all with hundreds of participants. Functional programming is also well represented nowadays at more general industry conferences such as GOTO in Aarhus, Strange Loop in St. Louis, and YOW! in Melbourne, Brisbane and Sydney, each with well over 1000 delegates.

Functional languages are also increasingly being adopted by industry in the real world (https://wiki.haskell.org/Haskell_in_industry). To name a few examples, Facebook uses Haskell to make news feeds run smoothly; WhatsApp relies on Erlang to run messaging servers, achieving up to 2 million connected users per server; Twitter, LinkedIn, Foursquare, Tumblr, and Klout use Scala to build their core infrastructure for sites. And while not using functional languages directly, Google's popular MapReduce model for cloud computation was inspired by the map and reduce functions commonly found in functional programming.

Generally speaking, functional programming is a style of programming: the main program is a function that is defined in terms of other functions, and the primary method of computation is the application of functions to arguments. Unlike traditional imperative programming, where computation is a sequence of transitions from states to states, functional programming has no implicit state and places its emphasis entirely on expressions (or terms). Functional programming focuses on *what* is being computed

[1]National Institute of Informatics, Tokyo 101-8430, Japan; [2]Department of Informatics, SOKENDAI (Graduate University for Advanced Studies), Tokyo 101-8430, Japan; [3]Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Gothenburg, Sweden and [4]School of Computing, University of Kent, Canterbury, Kent CT2 7NZ, UK

***Corresponding author.** E-mail: hu@nii.ac.jp

Received 18 May 2015; **Revised** 26 June 2015; **Accepted** 1 July 2015

rather than *how* it is being computed, much like the way we define mathematical functions. As a simple example, consider a mathematical definition of the factorial function:

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1)n! \end{aligned}$$

Its definition in the functional language Haskell [2] has exactly the same structure:

$$\begin{aligned} fac\ 0 &= 1 \\ fac\ (n+1) &= (n+1) * fac\ n \end{aligned}$$

In contrast, with imperative programming, we would consider a state of (n, s) representing the current counter and the partial result, and show how to compute the final result by a sequence of state transitions from the initial state of $(x, 1)$:

```
n = x;
s = 1;
while (n>0) do {
 s = s*n
 n = n-1;
}
```

The contrast in style is apparent: functional programs are more declarative and often much shorter than their imperative counterparts. This is certainly important. Shorter and clearer code leads to improved development productivity and higher quality (fewer bugs).

But it is probably a more subtle difference, well hidden behind the overloaded use of the symbol '=', that really sets the two apart. In the imperative program, '=' refers to a destructive update, assigning a new value to the left-hand-side variable, whereas in the functional program '=' means true equality: *fac* 0 is *equal* to 1 in any execution context, and can be used interchangeably. This characteristic of functional programming (known as *referential transparency* or *purity*) has a profound influence on the way programs are constructed and reasoned about—a topic that will be covered extensively in the section entitled 'Correctness of program construction'.

What this toy example does not illustrate is the use of higher order functions, a powerful abstraction mechanism highlighted in Hughes's paper [1]. We will not repeat the message here. Instead, we will describe the concept of *monads* (in the section entitled 'Structuring computation'), a design pattern for structuring computation that crucially depends on higher order functions. We will show how monads are used in developing domain-specific languages (DSLs) and taming side effects. In the section entitled 'Parallel and distributed computation', we revisit the ideas of purity and higher order functions in the context of parallel and distributed computing, again showing the unparalleled advantages that they bring. Lastly, in the section entitled 'Functional thinking in practice', we briefly sketch the impact of functional programming in terms of influence on education and other programming languages, and real-world adoption. As a note to readers, this review paper is not intended to systematically teach functional programming, nor to comprehensively discuss its features and techniques. They can be found in various textbooks on functional programming [3–5]. Rather, we aim to give a broad and yet focused view on how functional programming has mattered to software development, through showcasing advances in both academia and industry.

Since functional programming is a style, in theory one could write functional programs in any language, but of course with vastly differing levels of effort. We call a language functional if its design encourages or to some extent enforces a functional style. Prominent languages in this category include Haskell [2], Erlang [6], ML [7,8], OCaml [9], F# [10], Lisp [11,12], Scheme [13], Racket [14], Scala [15], and Clojure [16]. In this paper, we mostly use Haskell, because Haskell is not only a dominant functional language, but also covers all the most important features of functional programming.

CORRECTNESS OF PROGRAM CONSTRUCTION

Today's software systems are essential parts of our everyday lives, and their correctness is becoming ever more important; incorrect programs may not only cause inconvenience, but also endanger life and limb. A *correct* program is one that does exactly what its designers and users intend it to do.

Obvious as it sounds, guaranteeing the correctness of programs, or even defining the meaning of correctness, is notoriously difficult. The complexity of today's software systems is often to blame. But the design of many programming languages in use today—the fundamental tools we use to build software—does not help either. Programs are expressed as sequences of commands returning a final result, but also at the same time updating the overall state of the system—causing both intended and unintended *side effects*. State update is just one of the many kinds of side effect: programs may throw exceptions, send emails, or even launch missiles as side effects. For 'convenience', most languages allow such effects to be performed, without warning, anywhere in a program.

The result is that in order to specify the complete correctness of any program, one has to describe the whole state of the system, and the unlimited possibilities of interacting with the outside world—an impossible task indeed. Just consider the task of testing

part of a software system—perhaps a function called `f`. Before `f` can be executed, the tester must bring the system into the intended pre-state. After `f` has finished, the tester must check the outcome, which includes checking that the system state is as expected. But in general, the system state is only partly observable, and even identifying the parts of the state which `f` changed is problematic. Much of the work of testing imperative software consists of setting up the right state beforehand, and observing the final state afterwards.

Functional programming departs dramatically from this state of impediment by promoting purity: the result value of an execution depends on nothing other than the argument values, and no state may change as program execution proceeds. Consequently, it becomes possible to specify program behaviours independently of the rest of the system. For example, given a function that reverses a list (where $[\,]$ represents the empty list and $+\!\!+$ appends two lists), we can state the following set of properties governing the function's correctness. (As a notational convention, we use '$\equiv$' to denote semantic equality to avoid confusion with the use of '$=$' (function definition) and '$==$' (comparison for structural equality) in Haskell.)

$$\begin{array}{lcl} \mathit{reverse}\ [\,] & \equiv & [\,] \\ \mathit{reverse}\ [x] & \equiv & [x] \\ \mathit{reverse}\ (ys +\!\!+ xs) & \equiv & \mathit{reverse}\ xs +\!\!+ \mathit{reverse}\ ys \\ \mathit{reverse}\ (\mathit{reverse}\ xs) & \equiv & xs \end{array}$$

As there are neither side effects nor outside influence, these laws (the first three) completely characterize the function *reverse*. Drawing an analogy with sworn testimony, the laws specify 'the behaviour, the whole behaviour, and nothing but the behaviour'! The significance of this ability to claim that two expressions are equal (in any semantically observable way) is that one can now freely replace variables by their values, and in general any expressions by their equals—that is, programs are *referentially transparent*. This freedom makes functional programs more tractable mathematically than their conventional counterparts, allowing the use of equational reasoning in the design, construction, and verification of programs.

This is just a toy example, but the underlying idea is far-reaching. Readers who are Linux users may have come across `xmonad` (http://xmonad.org/), a tiling window manager for X11, known for its stability. `xmonad` is implemented in Haskell and relies on heavy use of semi-formal methods and program derivation for reliability; window manager properties (such as the behaviour of window focus) are specified as equational laws, similar to the ones above, and exploited for testing using QuickCheck [17].

The holy grail of program correctness is to *prove* the absence of bugs. A landmark achievement in this respect is Leroy's CompCert [18], an optimizing C compiler which is almost entirely proven correct with the Coq proof assistant. Tellingly, when John Regehr tested many C compilers using his random C program generator CSmith, he found 79 bugs in gcc, 202 in LLVM and so on, but only 6 in Compcert [19]. The middle-end bugs found in all other compilers were entirely absent from CompCert, and as of early 2011, CompCert is the only compiler tested with Csmith which lacks wrong code errors. As in the case of `xmonad`, the use of purely functional immutable data structures played a crucial role in this unprecedented achievement. This idea of immutability is also appearing in application areas where people might least expect it. Datomic (http://www.datomic.com), a fully transactional, cloud-ready and distributed database, presents the *entire database* as an immutable value, leveraging immutability to achieve strong consistency combined with horizontal read scalability.

In the rest of this section, we will see equational reasoning at work—in formal proofs of program correctness, in program testing, and in program optimization—how to associate algebraic properties to functional forms for program reasoning, how to automatically verify type properties, and how to structure and develop algebraic properties and laws for program derivation.

Equational reasoning

We have already seen the use of equational properties as specifications in the *reverse* example. Thanks to referential transparency, we are not only able to write the specifications, but also to reason with them.

Correctness proofs

One way to make use of these equations is in correctness proofs, just as in mathematics. Functional programs are often recursively defined over datatypes, lending themselves well to proofs by *structural induction*. For example, the *reverse* function we have already specified via equations can be *defined* in Haskell as follows:

$$\begin{array}{lcl} \mathit{reverse}\ [\,] & = & [\,] \\ \mathit{reverse}\ (x : xs) & = & \mathit{reverse}\ xs +\!\!+ [x] \end{array}$$

The second equation says that, to reverse a list with the first element as x and the rest of the list as xs, we reverse xs and append x to the end of it. In fact, this equation is derivable from the third law for *reverse* by replacing ys by $[x]$ and simplifying.

Now suppose we wish to prove that the definition actually satisfies its specification, say the property that *reverse* is its own inverse: for any finite list *xs*,

$$reverse\ (reverse\ xs) \equiv xs$$

holds. In order to prove this property for any finite list *xs*, it is sufficient to show that (1) it holds when *xs* is the empty list [ ], and (2) if it holds for *xs*, then it also holds for *x*: *xs*—this is the induction step. (1) is easy to show, and (2) can be confirmed as follows:

$$\begin{aligned}
& reverse\ (reverse\ (x : xs)) \\
\equiv\ & \{\text{def. of } reverse\} \\
& reverse\ (reverse\ xs +\!\!+ [x]) \\
\equiv\ & \{\text{law: } reverse\ (xs +\!\!+ ys) = reverse\ ys \\
& +\!\!+ reverse\ xs\} \\
& reverse\ [x] +\!\!+ reverse\ reverse\ xs \\
\equiv\ & \{\text{def. of } reverse \text{ and inductive hypothesis}\} \\
& (reverse\ [\,] +\!\!+ [x]) +\!\!+ xs \\
\equiv\ & \{\text{def. of } reverse \text{ and simplification}\} \\
& x : xs
\end{aligned}$$

Here, the law used in the above calculation can also be formally proven by induction.

Inductive proof is very commonly used in the functional setting as an effective way to verify critical software systems. The inductive proof of the inverse property of *reverse* has a strong resemblance to that of the inverse property of complex encoding and decoding in real practice [20]. More practical application examples include using the Coq proof assistant for proving the security properties of the JavaCard platform [21], certifying an optimizing compiler for C [18], and formally proving computational cryptography [22].

Property-based testing

Equational properties can also be used for *testing* functional programs. Using the QuickCheck test library [17], properties can be expressed via Haskell function definitions. For example, the property of *reverse* stated in the previous section can be written as

```
propReverseReverse :: [Integer] → Bool
propReverseReverse xs =
  reverse (reverse xs) == xs
```

If the property holds, then the corresponding function should always return *True*, so QuickCheck generates a large number of random argument values and checks that the function returns *True* for each one (the type stated for the property is needed to tell QuickCheck what kind of test data to generate, namely lists of integers). QuickCheck defines a DSL, embedded in Haskell, for expressing properties in a testable subset of predicate calculus. Quantified variables, such as *xs* above, range over 'sets' which are represented by test data generators, with fine control over the distribution of the randomly generated data.

When a test fails, QuickCheck 'shrinks' the test case to a minimal failing example. If we test the following (wrong) property,

```
propReverse :: [Integer] → Bool
propReverse xs = reverse xs == xs
```

then QuickCheck reports that [0, 1] (or, occasionally, [1, 0]) is a counterexample; the shrunk counterexample is obtained by searching for ways to simplify whatever randomly generated counterexample is first found. We obtain [0, 1] because at least two elements are needed to make this property fail, and they cannot be equal—so [0, 0] is not a counterexample. Shrinking is of critical importance to make property-based testing useful; without it, the 'signal' that causes a test to fail is drowned in the 'noise' of randomly generated data, and debugging failures is far more difficult.

Interestingly, this kind of testing finesses Dijkstra's famous objection that testing can never demonstrate the absence of bugs in software, only their presence. If we test properties that completely specify a function—such as the four properties of *reverse* stated in the introduction—and if every possible argument is generated with a non-zero probability, then property-based testing *will* eventually find every possible bug. In practice this isn't true, since we usually do not have a complete specification, and we limit the size of generated tests, but *in principle* we can find any possible bug this way—and in practice, this approach to testing can be very effective, since generated tests can explore scenarios that no human tester would think to try.

QuickCheck is heavily used in the Haskell community—it is the most heavily used testing package, and the 10th most used package of any kind, in the Hackage Haskell package database. The core of `xmonad`, discussed above, is thoroughly tested by simple equational properties on the state of the window manager, just like those we have discussed. The basic idea has been ported to many other programming languages—FsCheck for F#, ScalaCheck for Scala, test.check for Clojure, but even non-functional languages like Go and Java. There is even a commercial version in Erlang, marketed by Quviq AB, which adds libraries for defining state machine models of effectful software [23]. This version has been used to find bugs in an Ericsson Media Proxy [24], to track down notorious race conditions in the database software supplied with Erlang [25], and to formalize the basic software part of the AutoSAR automotive standard, for use in acceptance testing of vendors' code for Volvo Cars [26]. In this last project, 3000 pages of standards

documents were formalized as 20 000 lines of QuickCheck models, and used to find more than 200 different defects—many of them defects in the standard itself. A comparison with conventional test suites showed that the QuickCheck code was almost an order of magnitude smaller—despite testing more!

Automatic optimization

The ability to replace expressions with their equals and be oblivious to the execution order is a huge advantage in program optimization. The Glasgow Haskell Compiler (GHC) uses equational reasoning extensively internally, and also supports programmer-specified rewrite rules (equational transformations) as part of the source program (in pragmas) for automatic optimization. For instance, we could give GHC the inverse property of *reverse* to eliminate unnecessary double reversals of lists, and GHC will then apply the rule whenever it can. (While it is unlikely that a programmer would write a double reversal explicitly, it could well arise during optimization as a result of inlining other function calls.)

```
{-# RULES
 ''reverse-inv'' forall xs.
    reverse (reverse xs) = xs
#-}
```

In practice, people make use of this utility for serious optimizations. For example, shortcut fusion [27,28] is used to remove unnecessary intermediate data structures, and tupling transformation [29] is used to reduce multiple traversals of data.

HERMIT [30] is a powerful toolkit for developing new optimizations by enabling systematic equational reasoning inside the GHC's optimization pipeline. It provides a transformation API that can be used to build higher level rewrite tools.

Functional forms

Higher order functions are not only useful for *expressing* programs, they can be helpful in reasoning and proofs as well. By associating general algebraic (equational) laws with higher order functions, we can automatically infer properties from these laws when the higher order functions are applied to produce specific programs.

Two of the most important higher order functions are fold and unfold (also known as catamorphism and anamorphism). They capture two natural patterns of computation over recursive datatypes such as lists and trees: unfolds generate data structures and folds consume them. Here, we give them the name 'functional forms'—they can be used as design patterns to solve many computational problems, and these solutions inherit their nice algebraic properties. In this review, we focus on fold and unfold on lists. In fact, a single generic definition of fold can be given for all (algebraic) datatypes [31–33], and dually for unfold.

Algebraic datatypes

We have seen an example of an algebraic datatype, namely *List*, earlier on in the *reverse* example. *List* is the most commonly used datatype in functional programming—to such an extent that the first functional language was named Lisp [11], as in 'LISt Processing'. A list whose elements have the type α can be constructed by starting with the empty list *Nil*, and successively adding elements of type α to the list, one by one, using the data constructor *Cons*.

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

For instance, the list $[1, 2, 3, 4]$, of type *List Int*, is represented as follows:

$$as\ =\ \mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ (\mathit{Cons}\ 4\ \mathit{Nil})))$$

The notations $[\,]$ and infix : that we used above are simply shorthand for applications of *Nil* and *Cons*—*Cons x xs* can be written as $x : xs$. We will also use the 'section notation' (surrounding an operator by parentheses) to turn a binary infix operator $\oplus$ into a prefix function: $(\oplus)\ a\ b = (a\oplus)\ b = (\oplus b)\ a = a \oplus b$.

Fold

Foldr, which consumes a list and produces a value as its result, is defined as follows:

$$\begin{aligned}\mathit{foldr}\ f\ e\ \mathit{Nil}\ &= e\\ \mathit{foldr}\ f\ e\ (\mathit{Cons}\ x\ xs)\ &= f\ x\ (\mathit{foldr}\ f\ e\ xs)\end{aligned}$$

The effect of $\mathit{foldr}\ f\ e\ xs$ is to take a list *xs*, and return the result of replacing *Nil* by *e* and each *Cons* by *f*. For example, $\mathit{foldr}\ f\ e\ as$ converts the above list *as* to the value of

$$f\ 1\ (f\ 2\ (f\ 3\ (f\ 4\ e)))$$

This structurally inductive computation pattern captured by *foldr* is reusable; by choosing different *f*s and *e*s, *foldr* can perform a variety of interesting functions on lists. To take a few examples, *sum* sums up all elements of a list, *prod* multiples all elements of a list, *maxlist* returns the maximum element of a list, *reverse* reverses a list , *map f* applies function *f* to every element of a list, and *inits* computes all initial prefix lists of a list. In the definitions below, we use *partial applications* of *foldr*: defining *sum* as $\mathit{foldr}\ (+)\ 0$ (with two arguments rather than three)

is the same as defining *sum xs* to be *foldr* $(+)$ 0 *xs*.

$$
\begin{array}{ll}
sum & = foldr\ (+)\ 0 \\
prod & = foldr\ (\times)\ 1 \\
maxlist & = foldr\ max\ (-\infty) \\
reverse & = foldr\ f\ [\,]\ \textbf{where}\ f\ a\ r = r +\!\!+ [a] \\
map\ g & = foldr\ f\ [\,]\ \textbf{where}\ f\ a\ r = g\ a : r \\
inits & = foldr\ f\ [[\,]]\ \textbf{where}\ f\ a\ r = [\,] : \\
 & \quad map\ (a :)\ r
\end{array}
$$

Unfold

Unfoldr is the dual of foldr, which generates a list from a 'seed'. It is defined as follows:

$$
\begin{array}{l}
unfoldr\ p\ f\ g\ seed = \\
\quad \textbf{if}\ p\ seed\ \textbf{then}\ [\,] \\
\quad \textbf{else}\ (f\ seed) : unfoldr\ p\ f\ g\ (g\ seed)
\end{array}
$$

The functional form *unfoldr* takes a predicate *p* indicating when the seed should unfold to the empty list. When the condition fails to hold, then function *f* is used to produce a new list element as the head, and *g* is used to produce a new seed from which the tail of the new list will unfold. Like *foldr*, *unfoldr* can be used to define various functions on lists. For example, we can define the function *downfrom n* to generate a list of numbers from *n* down to 1:

$$
\begin{array}{l}
downfrom\ n = unfoldr\ isZero\ f\ g\ n \\
\quad \textbf{where} \\
\quad\quad isZero\ n\ =\ n == 0 \\
\quad\quad f\ n\ =\ n \\
\quad\quad g\ n\ =\ n - 1
\end{array}
$$

Composition of functional forms

Foldr and unfoldr can be composed to produce clear definitions of computations, which we think of as specifications in this section, because of their declarative nature. For example, the following gives a clear specification for computing the maximum of all summations of all initial segments of a list:

$$mis = maxlist\ \circ\ (map\ sum) \circ inits$$

It is defined as a composition of four functions we defined as foldrs before.

Given the duality of unfold and fold (one generating data structures and the other consuming them), compositions of an unfold followed by a fold form very interesting patterns, known as *hylomorphisms* [32,34,35]. A simple example of a hylomorphism is the factorial function:

$$factorial\ n = prod\ (downfrom\ n)$$

where *prod* is defined using *foldr* and *downfrom* using *unfoldr*.

General laws

Functional forms enjoy many nice laws and properties that can be used to prove properties of programs that are in those forms. Fold has, among others, the following three important properties [32].

First, foldr has the following *uniqueness* property:

$$foldr\ f_1\ e_1 \equiv foldr\ f_2\ e_2\ \Leftrightarrow\ f_1 \equiv f_2 \wedge e_1 \equiv e_2$$

which means that two foldrs are equivalent (extensionally), if and only if their corresponding components are equivalent. It serves as the basis for constructing other equational rules.

Second, foldr is equipped with a general *fusion* rule to deal with composition of foldrs, saying that composition of a function and a foldr can be fused into a foldr, under the right conditions.

$$\frac{h\ (f\ a\ r) \equiv f'\ a\ (h\ r)}{h \circ foldr\ f\ e \equiv foldr\ f'\ (h\ e)}$$

Third, multiple traversals of the same list by different foldrs can be *tupled* into a single foldr, and thus a single traversal.

$$
\frac{h\ x = (foldr\ f_1\ e_1\ x,\ foldr\ f_2\ e_2\ x)}{
\begin{array}{l}
h = foldr\ f\ (e_1, e_2) \\
\quad \textbf{where}\ f\ a\ (r_1, r_2) = (f_1\ a\ r_1,\ f_2\ a\ r_2)
\end{array}
}
$$

Let us use a simple example to demonstrate how fusion is useful in the derivation of efficient programs. Assume that we have an implementation of insertion sort (into *descending* order) using foldr:

$$
\begin{array}{l}
sort = foldr\ insert\ [\,] \\
\quad \textbf{where} \\
\quad insert\ a\ [\,] = [a] \\
\quad insert\ a\ (b : x) = \textbf{if}\ a \geq b\ \textbf{then} \\
\quad a : b : x\ \textbf{else}\ b : insert\ a\ x
\end{array}
$$

Now suppose that we want to compute the maximum element of a list. This is easy to do using the existing sorting program:

$$maxList = hd \circ sort$$

where *hd* is a function to select the first element from a non-empty list ($hd\ (a : x) = a$) and return $-\infty$ if the list is empty. Though declarative and obviously correct, this program is inefficient. It is overkill to sort the whole list, just to get the head. Fusion, using the laws above, provides a standard way to solve this problem. The laws tell us that if we can calculate f' such that

$$\forall a, x.\ hd\ (insert\ a\ x) \equiv f'\ a\ (hd\ x)$$

then we can transform $hd \circ sort$ to $foldr\ f'\ (-\infty)$. By instantiating x as $b : y$ and

performing a simple calculation, we obtain f as follows:

$$\begin{aligned} f'\ a\ b &\equiv hd\ (insert\ a\ (b : y)) \\ &\equiv \textbf{if}\ a \geq b\ \textbf{then}\ a\ \textbf{else}\ b \end{aligned}$$

Thus, we have derived a definition of *maxList* using a single foldr, which is exactly the same as the definition in the section entitled 'Fold' above. This fusion improves the time complexity of *maxList* from quadratic in the length of the list to linear.

As we have seen, equational reasoning and higher order functions (functional forms) enjoy a symbiotic relationship: each makes the other much more attractive.

Types

What is the type of a function like *foldr*? In the examples above, we have already seen it used with *several different* types! Its arguments are a function that combines a list element with an accumulator, an initial value for the accumulator, and a list of elements to be combined—but those list elements may be integers, lists themselves, or indeed *any* type; the accumulator can likewise be of any type. Being able to reuse higher order functions like *map* and *foldr* for different types of data is a part of what makes them so useful: it is an essential feature of functional programming languages. Because of this, early functional languages did without a static type checker; they were *dynamically* typed, but the compiler did not attempt to discover type errors. Some, like Erlang or Clojure, still are.

Polymorphic types

In 1978, Milner introduced *polymorphic types* to ML, to solve this problem. The key idea is to allow types to include *type variables*, which can be instantiated to any type at all, provided all the occurrences of the same variable are instantiated to the same type. For example, the type of *foldr* is

$$foldr\ ::\ (\alpha \to \beta \to \beta) \to \beta \to List\ \alpha \to \beta$$

where α is the type of the list elements, and β is the type of the accumulator. When *foldr* was used to define *sum* above, then it was used with the type

$$\begin{aligned} foldr\ ::\ &(Integer \to Integer \to Integer) \to \\ &Integer \to List\ Integer \to Integer \end{aligned}$$

(in which α and β are both instantiated to *Integer*); when it was used to define *sort* then its type was

$$\begin{aligned} foldr\ ::\ &(\gamma \to List\ \gamma \to List\ \gamma) \\ &\to List\ \gamma \to List\ \gamma \to List\ \gamma \end{aligned}$$

(in which α is replaced by γ, the elements in the list to be sorted and β the accumulator type is a list of these). This allowed the flexibility of polymorphism to be combined with the security of static type checking for the first time. ML also supported *type inference*: the type of *foldr* (and indeed, every other function) could be inferred by the compiler from its definition, freeing the programmer from the need to write types in function definitions at all so ML was as concise and powerful as the untyped languages of its day, with the added benefit of static type checking.

Both these ideas have made a tremendous impact. Many functional programming languages since, Haskell among them, have borrowed the same approach to polymorphic typing that ML pioneered. Java generics, introduced in Java 5, were based directly on Odersky and Wadler's adaption of Milner's ideas to Java [36]; similar features appeared thereafter in C#. Type inference (in a more limited form) appeared in C# 3.0 in 2007, and type inference is increasingly available in Java too.

In fact, ML's polymorphic types also give us surprisingly useful semantic information about the functions they describe. For example, the *reverse* function can be applied to lists with any type of element, and so has the polymorphic type

$$reverse\ ::\ List\ \alpha \to List\ \alpha$$

This implies that it also satisfies

$$\forall f, xs.\ map\ f\ (reverse\ xs) = reverse\ (map\ f\ xs)$$

as, indeed, does any other function with the same polymorphic type! These 'free theorems', discovered by Wadler [37], are an application of Reynold's *parametricity* [38]; they are used, among other places, to justify the 'short cut deforestation' optimization in the GHC [27].

Type classes for overloading

Haskell's major innovation, as far as types are concerned, was its treatment of overloading. (Over the years, many innovations have been made in Haskell's type system; here we refer to innovations in the early versions of the language.) For example, the equality operator is overloaded in Haskell, allowing programmers to use different equality tests for different types of data. This is achieved by declaring an equality *type class*:

$$\begin{aligned} &\textbf{class}\ Eq\ \alpha\ \textbf{where} \\ &\quad (==)\ ::\ \alpha \to \alpha \to Bool \end{aligned}$$

which declares that the equality operator $(==)$ can be applied to any type α with an instance of the class *Eq*; programmers can define instances of each class, and the compiler infers automatically which instance should be used.

Right from the start, Haskell allowed instances to depend on other instances. For example, (structural) equality on lists uses equality on the list elements, expressed in Haskell by defining

$$\textbf{instance}\ Eq\ \alpha \Rightarrow Eq\ (List\ \alpha)\ \textbf{where}$$
$$xs == ys = \ldots$$

That is, given an equality on type α, the compiler can construct an equality on type $List\ \alpha$, using the definition of $==$ in the instance. Or to put it another way, the compiler can reduce a goal to find an $Eq\ (List\ \alpha)$ instance, to a goal to find an $Eq\ \alpha$ instance, for any α. This looks a lot like logic programming! Over the years, Haskell class and instance declarations have become more and more powerful, incorporating aspects of both logic programming and functional programming, with the result that the type checker is now Turing-complete (Robert Dockins, The GHC typechecker is Turing-complete, https://mail.haskell.org/pipermail/haskell/2006-August/018355.html). The important observation here is that Haskell's class system gives us a *programmable type checker*, which—while it does *not* allow us to accept programs that will generate runtime-type errors—*does* allow us to construct type systems for DSLs embedded in Haskell with exquisite precision. Scala's implicits were directly inspired by Haskell's type classes, and allow many of the same tricks to be played [39].

Types and logic

Finally, there is a deep connection between types and logic. Just consider the type of the *apply* function:

$$apply\ ::\ (\alpha \rightarrow \beta, \alpha) \rightarrow \beta$$

If we read function arrow types as logical implications, and the pair type as a conjunction, then this type reads as the (true) proposition

$$((A \Rightarrow B) \wedge A) \Rightarrow B$$

It turns out that this is not a coincidence: we can soundly regard *apply* as a proof of this property, and any type which is inhabited by some expression (in a sufficiently carefully designed functional language) corresponds to a provable property. Proof assistants such as Coq [40] and Agda [41] are based on this correspondence, the *Curry–Howard isomorphism*, and enable users to prove theorems by writing programs. A key notion here is that of a *dependent type*, a type which depends on a value. For example, if the *List* type is parameterized not only on the element type, but also on the length of the list, then *reverse* can be given a more informative type:

$$reverse\ ::\ \forall k :: Nat, \alpha :: Set.\ List\ k\ \alpha \rightarrow List\ k\ \alpha$$

representing the fact that its result has the same length as its argument. (Here *Set* is roughly speaking 'the type of types', or more precisely, the type of *small* types, i.e. excluding types such as *Set* itself.) Predicates can then be represented by types that are sometimes empty. For example, *Even k* might be a type that is empty if k is odd, and non-empty if k is even—so constructing a term of type *Even k* proves that k is even. Using these ideas, it is possible to construct large functional programs with formal proofs of correctness. Perhaps the most impressive example to date is CompCert [18], already discussed above.

Algebra of programming

In the section entitled 'Functional forms', we have seen a useful set of functional forms that capture common computation patterns while enjoying general algebraic laws for program reasoning. This attractive idea can be carried much further, leading to the development of many specific theorems as building blocks for more complex reasoning. The result is the *algebra of programming* [42,43], where functional programming provides an algebraic framework for building *programming theories* for solving various computational problems by program calculation (derivation)—a systematic way of transforming specification into efficient implementation through equational reasoning. This supports, in practice, Dijsktra's argument that programming should be considered as a discipline of a mathematical nature [44].

In this section, we review the programming theories that have been developed for constructing programs from specifications, and demonstrate how these theories can be used to derive programs in various forms for efficient sequential or parallel computation.

Programming: deriving programs from specifications

Before addressing the solution of programming problems, consider the following mathematical problem known as the Chinese Chicken–Rabbit-Cage Problem:

> An unknown number of rabbits and chickens are locked in a cage. Counting from above, we see there are 35 heads, and counting from below, there are 94 feet. How many rabbits and chickens are in the cage?

The degree of difficulty of this problem largely depends on which mathematical tools one has to hand. A preschool child may find it very difficult; he has no choice but to enumerate all possibilities. However, a middle school student equipped with knowledge of

equation solving should find it easy. He would let x be the number of chickens and y that of rabbits, and quickly set up the following problem specification:

$$\begin{aligned} x + y &= 35 \\ 2x + 4y &= 94 \end{aligned}$$

The rest is entirely straightforward. The theory of linear equations gives us a strategy for solving them systematically, to discover the values of the unknowns (i.e., $x = 23, y = 12$).

We want to solve our programming problems this way too! We would like to have an *algebra of programs*: a concise notation for problem *specification*, and a set of symbol manipulation rules with which we may calculate (derive) *programs* from specifications by *equational reasoning*. Here, by 'specification', we mean two things: (1) a naive functional program that expresses a straightforward solution whose correctness is obvious; and (2) a program in a specific form of composition of functional forms. By 'program', we mean an *efficient* functional program, which may be sequential, parallel, or distributed.

Programming theories

Just like the specific laws developed for factorization in solving equations, many laws (theorems) for deriving efficient functional programs from specifications have been developed [43,45]. They are used to capture programming principles by bridging the gap between specifications and their implementing programs. As an example, many optimization problems can be naively specified in a generate-and-test way: generating all the possibilities, keeping those that satisfy the requirements, and returning one that maximizes a certain value:

$$opt = maxlist \circ map\ value \circ filter\ p \circ gen$$

To solve this kind of optimization problems using folds, many theorems [43,46] have been developed. One example is: if (1) *gen*, *p*, and *value* can be defined as foldrs, and (2) $value = foldr\ (\oplus)\ e$, where $\oplus$ is associative and *max* is distributive over $\oplus$, then *opt* can be solved in linear time by a functional program in terms of a single foldr. With this theorem, solving optimization problems become easy: one just needs to specify the problem in the form described and the rest will follow!

Programming theory development

Many theories for the algebra of programming have been developed—but new ones can always be added. There is a general procedure to develop programming theories consisting of the following three major steps [47,48].

(i) Define a specific form of programs, in terms of functional forms and their composition, that can be used to describe a class of interesting computations.
(ii) Develop calculational rules (theorems) to bridge the gap between the new specific form and the existing functional forms.
(iii) Develop more theorems that can turn more general programs into the specific form to widen the application scope.

The first step plays a very important role in this development. The specific form defined should not only be powerful enough to describe computations of interest, but also manipulable and suitable for the later development of calculational laws.

Systems for program derivation

Many systems have been developed for supporting program derivation and calculation. Examples are KIDS [49], MAG [50], Yicho [47], and so on. In general, such tools (1) support interactive development of programs by equational reasoning so that users can focus on their creative steps, (2) guarantee correctness of the derived program by automatically verifying each calculation step, (3) support development of new calculation rules so that mechanical derivation steps can be easily grouped, and (4) make the development process easy to maintain (i.e. the development process should be well documented).

Proof assistants and theorem provers can provide a cheap way to implement a system for program reasoning and program calculation [51–53]. For instance, with Coq [40], a popular theorem prover, one can use Ltac, a language for programming new tactics, to build a program calculation system [53]: (1) Coq tactics can be used effectively for automatic proving and automatic rewriting, so that tedious calculation can be hidden with tactics; (2) new tactics can coexist with the existing tactics, and a lot of useful theories of Coq are ready to use for calculation, and (3) tactics can be used in a trial-and-error manner thanks to Coq's interaction mechanism.

In contrast to deriving functional programs from formal specifications, MagicHaskeller [54,55] is an interesting tool that automatically synthesizes functional programs from input/output examples. Although still under development, it has demonstrated the feasibility of the idea by synthesizing many interesting Haskell programs from just a small number of examples.

STRUCTURING COMPUTATION

In the previous section, we have experienced the liberating power of the absence of side effects.

Referential transparency is the basis of the wealth of properties that enabled us to reason, to test, to optimize, and ultimately to derive programs. On the other hand, functional programmers are by no means ascetics withdrawing from worldly pleasures. Functional programs are real programs! They do perform necessary computations with side effects such as I/Os, exceptions, non-determinism, etc.

The difference is simply a shift of perspective. Functional programmers view side effects as costly, which shall be used with care, and at the best avoided. Let us suppose that we have to pay for each pair of parentheses in a game of writing arithmetic expressions. Associativity, which removes the need of parenthesizing, becomes very valuable and players will be incentivized to write $3 \times 0.5 \times 5$ instead of $(3 \div 2) \times 5$ for example. Programming is like such writing of expressions, and the role of a programming language is to encourage good practice and at the same time discourage the opposite.

Pure functional languages do just that: side effects are encoded by a programming pattern known as *monads* [56–59]. Since it is an encoding, purity is not threatened. But monads do enjoy a clearly distinguished syntax and type to encourage thoughtful design. We will see later in this section in more details the idea of monadic effects, and the benefit of being disciplined with the uses of them. As a matter of fact, the concept of monads as a programming pattern has applications far beyond handling side effects; it is fundamentally a powerful way of structuring computations and has been adopted in languages with built-in side effects! To name a few, some readers may have heard of *async monad* in F#, *promises* in JavaScript, or *future* in Scala, which are essentially imitations of a monad used to structuring asynchronous operations. On a different front, Language INtegrated Query (LINQ) in .NET is directly inspired by monads, and a syntax sugar for them known as *monad comprehension* (following from set comprehension) in Haskell.

In the sequel of this section, we will briefly review monads and demonstrate their uses in structuring computations, in the context of programming language development, which is itself a particularly successful application of functional programming.

Monadic composition

Originally as a concept in category theory that was used by Moggi to modularize the structure of a denotational semantics [56,57], monads are soon applied by Wadler to the functional implementations of the semantics that Moggi describes, and as a general technique for structuring programs [58,59].

A monad M consists of a pair of functions *return* and $(\gg=)$ (pronounced as 'bind').

$$\begin{aligned} &\mathit{return} &&:: \alpha \to M\ \alpha \\ &(\gg=) &&:: M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta \end{aligned}$$

One shall read $M\,\alpha$ the type of a computation that returns a value of type α, and perhaps performs some side effects. An analogy is to see a value of type $M\,\alpha$ as a bank card and its pin; they are not the same as plain cash a (of type α) which can be used immediately, but contain the instruction of producing an a, and in fact any number of as.

We cannot use a value of type $M\,\alpha$ directly, but we can combine it with other instructions that use the result. For example, consider an evaluator of type $\mathit{Term} \to \mathit{Environment} \to M\ \mathit{Value}$, where M is a monad. The evaluator does not give us a value directly, but we can bind its result to be used in subsequent computations:

$$\begin{aligned} &\mathit{eval}\ u\ e \gg= (\lambda a \to \\ &\mathit{eval}\ v\ e \gg= \lambda b \to \\ &\mathit{return}\ (\mathit{Num}\ (a + b)) \end{aligned}$$

This expression evaluates the two operands of an addition and add them up. The result of the first evaluation is bound to the variable a and is passed to the second evaluation, and together with the second result (bound to b) they form parts of a new value which is again encapsulated in the monad by *return*. This pattern of structuring computation is very common and has been giving special syntax support in Haskell [58]. The above expression can be rewritten as the following equivalent form:

$$\begin{aligned} &\mathbf{do}\ a \leftarrow \mathit{eval}\ u\ e \\ &\quad\ b \leftarrow \mathit{eval}\ v\ e \\ &\quad\ \mathit{return}\ (\mathit{Num}\ (a + b)) \end{aligned}$$

which is seemingly similar to the following expression:

$$\begin{aligned} &\mathbf{let}\ a = \mathit{eval}\ u\ e \\ &\quad\ b = \mathit{eval}\ v\ e \\ &\mathbf{in}\ \mathit{Num}\ (a + b) \end{aligned}$$

Now, let us say we want to extend the evaluator with error handling as in

$$\mathit{eval} :: \mathit{Term} \to \mathit{Environment} \to \mathit{Maybe\ Value}$$

$$\mathbf{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}$$

where failures are represented by *Nothing* and successful computations return values wrapped in *Just*. Without giving up on purity and resorting to a language with side effects, the non-monadic definitions which structure computations with **let** have to be tediously rewritten: every **let** binding now has to be made to perform a case analysis, separating successes

from failures, and then decides the next step accordingly.

Whereas for the monadic version, the story is rather different. The structure of the evaluator is independent of the underlying computation that is used, and if carefully designed changing one entails minimum changes to the other. In the above, the specializing of monad type M to *Maybe* links the right instances of *return* and $\gg=$ (in fact monads forms a type class).

```
instance Monad Maybe where
  return x       = Just x
  Nothing >>= f  = Nothing
  Just x >>= f   = f x
```

There is no change in the evaluator's definition, as the handling of failure is captured as a pattern and localized in the definition of $\gg=$. In a similar manner, we can add to the evaluator real-world necessities such as I/Os, states or non-determinism in a similar manner—simply replacing the underlying monad.

Monads not only encapsulate side effects, but also make them first class. For example, the function $sequence :: [m\ \alpha] \rightarrow m\ [\alpha]$, where m is a monad, evaluates each computation in turn, and collects the results. In the evaluator example, one may rewrite the expression for addition as

$$liftM\ sum\ (sequence\ [eval\ u\ e, eval\ v\ e])$$

where *sum* adds up a list of numbers and is lifted (by *liftM*) to handle monadic values.

In addition to structuring programming languages semantics, monads also offer a nice framework for I/Os in a pure language. The idea is to view I/O operations as computations interacting with the 'outside world' before returning a value: the outside world is the state, which (represented by a dummy 'token') is passed on to ensure proper sequencing of I/O actions [60]. For example, a simple *Read-Eval-Print Loop* that reverses the command line input can be written with recursion as a function of type $IO\ ()$, where IO is the monad and $()$ is a singleton type with $()$ as its only value.

```
repl :: IO ()
repl = do
         line ← getLine
         putStrLn (reverse line)
         repl
```

Giving the stateful implementation of the IO monad, it is obvious that it could be used to support mutable states and arrays [60], though a different approach based on normal state monad, which encapsulates effectful computations inside pure functions, may be considered better [61,62].

Further extensions of the IO include the handling of concurrency [63]. The implementation of software transactional memory (STM) with monads [64] is a clear demonstration of the benefit of carefully managed side effects; two monads are used in the implementation: the *STM* monad structures tentative memory transactions, the *IO* monad commits the STM actions exposing their effects to other transactions, and a function *atomic* connects the two:

$$atomic :: STM\ \alpha \rightarrow IO\ \alpha$$

As a result of this explicit distinction of different type of effects made possible by monads, only STM actions and pure computations can be performed inside a memory transaction ruling out irrevocable actions by construction, and no STM actions can be performed outside a transaction effectively eliminating a class of bugs altogether. Moreover, since reads from and writes to mutable cells are explicit as STM actions, the large number of other (guaranteed pure) computations in a transaction are not tracked by the STM, because they are pure, and never need to be rolled back. All these guarantees make a solution based on monads very attractive indeed.

Monads are often used in combination. For example, an evaluator may need to handle errors and at the same time performing I/O, and it will be desirable to reuse existing monads instead of creating specialized ones. The traditional way of achieving such reuse is through moving up the abstraction level to build *monad transformers* [65,66], which are similar to regular monads, but instead of standing alone they modify the behaviours of underlying monads, effectively allowing different monads to stack up. The downside of monad transformers is that they are difficult to understand and are fragile to changes. Alternatives have been proposed and it is still an active area of research [67–69].

Inspired by monads, other abstract views of computation have emerged, notably *arrows* [70,71] and *applicative functors* [72]. First proposed by Hughes [70], arrows are more general than monads allowing notions of computation that may be partially static (independent of the input) or may take multiple inputs. Applicative functors are a proper superset of monads, which has weaker properties and thus more members. Similar to monads, both arrows and applicative functors can be used for structuring the semantics of *embedded* domain-specific languages (EDSLs) [73,74]. A theoretical comparison of the three can be found in [75].

Embedded domain-specific languages

So far, we have only discussed one way of developing languages that is to implement stand-alone

compilers or interpreters. Functional programming is well suited for the task and the use of monad has significant impact on modularity. However, the particular success of functional programming actually comes from another way of language development known as *embedding* [76,77].

Languages produced through embedding are known as *embedded* languages, which are libraries in host languages. Giving that such libraries usually focus on providing functionalities of a specific problem domain, the languages resulting from are seen as EDSLs, while the host languages are usually general purpose. The EDSL and its host language are one: the API of the library specifies a set of constructs of the new DSL, which at the same time shares the tool chain and the generic features of the host language, such as modules, interfaces, abstract data types, or higher order functions. Moreover, the EDSL implementation is very 'lightweight'—the EDSL designer can add features just by implementing a new function in the library, and can easily move functionality between the EDSL library and its clients. The ease of experimentation with such an EDSL helps implementors fine-tune the design, and enables (some) end users to customize the implementation with domain-specific optimizations.

Such EDSLs have appeared in a wide spectrum of application areas, including compiler development, database queries, web applications, GUIs, music, animations, parallelism, hardware descriptions, image processing, workflow, and more. See Fig. 1 for a few examples and [78] for a comprehensive listing.

We have said that EDSLs are libraries. But obviously not all libraries are languages. So what is it that elevates EDSLs from their humble origin as libraries to the level of languages?

For a language to be worth its name, it must allow code fragments manipulating primitive data to be shared through defining reusable procedures. In an EDSL setting, the data (also known as the domain concepts) are themselves functions. For example, in modelling 3D animation, the domain concept of *behaviour*, representing time-varying reactive values, is a function from time to a 'normal' value. To have a language, we need to provide constructs that manipulate such domain concepts. This is easy in functional languages: we can simply define higher order functions called *combinators*, taking domain concepts as inputs and combining them into more complex ones, just like what procedures do to primitive data. For this reason, EDSLs are also known as combinator libraries—libraries that not just offer domain-specific operations, but also more importantly combinators that manipulate them. We illustrate this idea by developing an example EDSL.

A classic example: parser combinators

Our aim is to develop a parsing EDSL for the following BNF grammar:

$$\mathit{float} ::= \mathit{sign}^? \ \mathit{digit}^+ \ ('.' \ \mathit{digit}^+)^?$$

Like any language, an EDSL has syntax and semantics. In this embedded setting, the syntax is simply the interface of the library containing the

```
query = do  cust ← table customers
            restrict (cust!city . == . "London")
            project (cust!customerID)
```

(a) HaskellDB [121] for generating and executing SQL statements

```
htmlPage content =
    (header ≪ ((thetitle ≪ "Mypage")
      +++ (script ! [thetype "text/javascript",
                     src "http://..."] ≪ ""))
    )) +++ (body ≪ content)
```

(b) (X)HTML [119] for producing XHTML

```
mouseTurn g u =
  turn3 xVector3 y (turn3 zVector3 (−x) g)
  where
    (x, y) = vector2XYCoords (π̂*mouseMotion u)

mouseSpinningPot u =
  mouseTurn (withColorG green teapot) u
```

(c) Fran [120] for composing interactive, multi-media animations.

Figure 1. Code fragments in three EDSLs.

representation of the domain concept, and the operations on it; and the semantics is the implementation of the operations in the host language.

A parser is a program that receives a piece of text as the input, analyses the structure of the text, and produces an output usually in the form of trees that can be more conveniently manipulated by other parts of a compiler. A parser can be represented as a function:

$$\textbf{newtype}\ \mathit{Parser}\ \alpha = \mathit{MkP}\ (\mathit{String} \rightarrow [(\alpha, \mathit{String})])$$

The type *Parser* is a new type, distinct but isomorphic to its underlying type of a function from strings to lists of parsing results, depending on how many ways the parse could succeed, or fail with an empty list. The parameterized type α is the tree produced by parsing, and is paired with the remaining unparsed string. In DSL terminology, *Parser* is the domain concept we are trying to model and as usual it has an underlying representation as a function.

Primitive Parsers: With a parser representation at hand, we can start building a library to manipulate it. To start with, we define some basic parsers.

```
item :: Parser Char
item = MkP f
  where
    f []       = []
    f (c : cs) = [(c, cs)]
```

The parser *item* consumes a character of the input string and returns it. And a second parser is a parser that always fails.

```
zero :: Parser α
zero = MkP f where f _ = []
```

These two are the only basic parsers that we will ever need, and all the power of the resulting language comes from its combinators.

Parser Combinators: For the grammar we have in mind, our language consists of the following set of basic combinators:

```
sat      :: (Char → Bool) → Parser Char
plus     :: Parser α → Parser α → Parser α
optional :: Parser α → Parser (Maybe α)
```

In the above, *sat* enriches *item* with a predicate so that only characters satisfying the predicate will be parsed. By providing different predicates, it can be specialized to a number of different parsers:

```
char x = sat (== x)
digit = sat(λx → '0' ≤ x ∧ x ≤ '9')
```

For example, *digit* succeeds with '*123*' and produces [('*1*', '*23*')] as the outcome, but fails (returning [ ]) with '*A23*'. Similarly, *char* only recognizes the character that is passed to it as input and fails on all others.

Parser *plus p q* combines the outcomes of applying *p* and *q*. In the case when one of the two fails, the outcome will be the same as using the other one:

```
sign :: Parser Char
sign = (char '+') `plus` (char '-')
```

In the above, the prefix function *plus* is turned into an infix one by the surrounding left single quotes.

The combinator *optional* corresponds to the $(^?)$ notation in our grammar allowing an optional field to be parsed into *Nothing* when it is not filled. Note that here *Nothing* represents success in parsing, not failure which is represented by the empty list.

So far, what we have seen are combinators that are useful for constructing parsers recognizing individual components of a grammar. A crucial missing part is a way to sequence the individual components, so that individual parsers are chained together and applied in sequence. For example, our *float* grammar is a chain of sign, digits, and fraction parsers. In another words, we need a way to structure the parsing computation. Sounds familiar? Monads introduced in the previous subsection fit the bill very well.

Just like *Maybe*, *Parser* α is a monad with its own bind and return:

```
(≫=) :: Parser α → (α → Parser β)→Parser β
return  :: α → Parser α
```

Now we are ready to define combinators that repeatedly apply a parser:

```
zeroOrMore :: Parser α → Parser [α]
zeroOrMore p =
  do {x ← p;
      xs ← zeroOrMore p;
      return (x : xs)}
  `plus` return []

oneOrMore :: Parser α → Parser [α]
oneOrMore p =
  do x ← p
     xs ← zeroOrMore p
     return (x : xs)
```

Parser *zeroOrMore p* returns each application of *p* as a list. For example, applying (*zeroOrMore digit*) to '*12a*' results in [('*12*', '*a*'), ('*1*', '*2a*'), ('', '*12a*')]. Parser *oneOrMore p* is similar to *zeroOrMore p* except requiring *p* to apply at least once.

Just to show that the use of monad has not prevented us from doing the same kind of equational reasoning as before, we list a few arbitrarily selected

laws as an example:

$$\begin{aligned} &zero \gg\!= p && \equiv p \\ &zero \text{ `}plus\text{` } p && \equiv p \\ &p \text{ `}plus\text{` } (q \text{ `}plus\text{` } r) && \equiv (p \text{ `}plus\text{` } q) \text{ `}plus\text{` } r \\ &(p \text{ `}plus\text{` } q) \gg\!= r && \equiv (p \gg\!= r) \text{ `}plus\text{` } (q \gg\!= r) \end{aligned}$$

Finally, we can directly translate the BNF grammar at the beginning of this subsection to the following executable parser:

```
float = do
  sgn ← optional sign
  in ← oneOrMore digit
  frac ← optional (do {char '.'; oneOrMore digit})
  return (mkFloat sgn in frac)
```

We stop here and look at what we have learnt from this exercise. Readers interested in knowing more about parser combinators may start with Hutton and Meijer's tutorials [79,80] for more combinators and techniques for improving efficiency, and several more papers on the subject [81–85].

(i) *Combinators are the key*. The parser library we have developed is a language. It is not so much about what specific operations are provided, but the unlimited possibilities through creative programming. As we can see, the very small number (two in our case) of basic parsers, which is typical in EDSLs, can be combined in different ways to produce highly complex programs. This power comes from the combinators which hinge on higher order functions.

(ii) *Monads are very useful*. The monadic sequential combination is an excellent formulation of a recurring pattern: a sequence of operations are performed in turn and have their results combined in the end. We have relied on it to program parsers that goes beyond simply consuming a single character.

When the semantics of the domain concepts is obvious, directly encoding the operations that can be performed on them often results in an elegant EDSL implementation. This is exactly what we did for the parser example: the set of combinators implements what we can do to parsers and the set is easily extensible by defining new combinators. On the other hand, adding new domain concepts usually requires a complete reimplementation. Moreover, with this direct approach, performance of the domain-specific programs relies on the optimization of the host language compiler, which is hard to predict and control.

An alternative embedding technique is to generate code from the EDSL programs: the domain concepts are represented as an abstract syntax tree, and a separate interpretation function provides the semantics of the syntax constructs, resulting in a so called *embedded domain-specific compiler* [86]. In this case, the host language is used only at 'compile time'; the EDSL programmer can use the full power of the host language to express the program, but at runtime, only the generated code needs be executed. This approach is sometimes referred as *deep* embedding in contrast to the above more direct *shallow* embedding. The separation of phases in deep embedding provides opportunities for domain-specific optimizations as part of the interpretation function and adding new domain concepts simply means additional interpretation functions. On the other hand, extending the set of operations involves modifying the abstract syntax and existing interpretation functions.

The two embedding approaches are dual in the sense that the former is extensible with regard to adding operations while the latter is extensible with regard to adding concepts [87]. The holy grail of embedded language implementation is to be able to combine the advantages of the two in a single implementation—a manifestation of the *expression problem* [88]. Current research addresses the problem at two levels: exploiting sufficiently expressive host languages for a modular representation of the abstract syntax tree [89–92], or combining the shallow and deep embedding techniques [93]. Notably in [91], the technique also addresses another source of inefficiency with embedded languages namely the tagging and untagging required for manipulating the abstract syntax trees represented as datatypes.

Lastly, the connection between EDSLs and monads may extend beyond the implementation level. It has become popular to include the monadic constructs as part of the EDSL surface language, which sparks interesting interactions with the underlying type system [94–96].

PARALLEL AND DISTRIBUTED COMPUTATION

For a long time, parallel and distributed programming was seen as a specialized activity by special experts. This situation is currently changing extremely rapidly with pervasive parallel and distributed environments such as multicore/manycore hardware and cloud computing. With the Google's MapReduce [97], one can now easily write a program to process and generate large data sets with a parallel, distributed algorithm on a cluster. The MapReduce model is actually inspired by the *map* and *reduce* functions commonly used in functional programming.

Functional programming languages offer a medium where programmers can express the features of parallel algorithms, without having to detail the low-level solutions [98–100]. The high level of programming abstraction of function composition and higher order functions simplifies the task of programming, fosters code reuse, and facilitates the development of substantially architecture-independent programs. The absence of side effects avoids the unnecessary serialization which is a feature of most conventional programs.

Parallel functional programming

Functional languages have two key properties that make them attractive for parallel programming: they have powerful abstraction mechanisms (higher order functions and polymorphism) for supporting explicit parallel programming known as *skeleton parallel programming* that abstracts over both computation and coordination and achieves the architecture-independent style of parallelism, and they have no side effect that can eliminate unnecessary dependences for *easy parallelization*.

Skeleton parallel programming

Parallel primitives (also called parallel skeletons [101,102]) intend to encourage programmers to build a parallel program from ready-made components for which efficient implementations are known to exist, making the parallelization process easier.

The higher order functions discussed in the section entitled 'Functional forms' can be regarded as parallel primitives suitable for parallel computation over parallel lists, if we impose some properties on the argument operators. Three known data parallel skeletons *map*, *reduce,* and *scan* can be defined as special instances of $foldr\ op\ e$. The definition of *map* has been given in the section entitled 'Functional forms', and the definitions of *reduce* and *scan* are as follows:

$$\begin{aligned} reduce\ op\ e &= foldr\ op\ e \\ scan\ op\ e &= map\ (foldr\ op\ e) \circ inits \end{aligned}$$

where *op* is an associative operator, and *scan* is defined as a composition of $map\ (foldr\ op\ e)$ and *inits*. Note that *reduce* is different from *foldr* in that the operator *op* it can accept must be associative, which restricts the type of *reduce* (the operator must combine operands of the same type), but allows the implementation greater freedom to choose an order of combination (*foldr* always combines from right to left). The definitions above just define the *semantics* of *reduce* and *scan*, not necessarily their implementation.

It has been shown that *map*, *reduce,* and *scan* have nice massively parallel implementations on many architectures [102,103]. If k and an associative $\oplus$ use $O(1)$ parallel time, then $map\ k$ can be implemented using $O(1)$ parallel time, and both $reduce\ op\ e$ and $scan\ op\ e$ can be implemented using $O(\log N)$ parallel time (N denotes the size of the list). For example, $reduce\ op$ can be computed in parallel on a tree-like structure with the combining operator *op* applied in the nodes, while $map\ k$ is computed in parallel with k applied to each of the leaves. The study on efficient parallel implementation of $scan\ op\ e$ can be found in [102], which plays an important role in the implementation of parallel functional language NESL [104].

Just like *foldr* being the most important higher order function for manipulating lists sequentially, *reduce* plays the most important role in parallel processing of lists. If we can parallelize *foldr* as a parallel program using *reduce*, we can parallelize any sequential function that is defined in terms of *foldr* (This again shows an advantage of structuring programs using general functional forms.). This has attracted a lot of work in developing programming theories to parallelize *foldr* [105–108]. One known theorem for this parallelization is the so-called third homomorphism theorem [109], which shows that $foldr\ op\ e$ can be parallelized as a composition of a *map* and a *reduce* if and only if there exists op' such that the following holds:

$$foldr\ op'\ e = foldr\ op\ e \circ reverse$$

In other words, this theorem says that a foldr is parallelizable if and only if can be written as a foldr on its reverse list. With this theorem, we can see that many of the functions defined in the section entitled 'Functional forms', such as *sum*, *sort*, *maxlist*, and *reverse*, can be parallelized as a composition of a map and a reduce. Importantly, this parallelization is not just a guide to programmers but also can be done automatically [110].

Easy parallelization

Purely functional languages have advantages when it comes to (implicit) parallel evaluation [111,112]. Thanks to the absence of side effects, it is always safe to execute computations of subexpressions in parallel. Therefore, it is straightforward to identify the parallel task in a program, which would require complex dependency analysis when parallelizing imperative programs.

Parallel Haskell provides two operators *pseq* and *par* for parallelization: $pseq\ e_1\ e_2$ evaluates e_1 then e_2 in a sequential order, and $par\ e_1\ e_2$ is a kind of *fork* operation, where e_1 is started in parallel with e_2 and the result of e_2 is returned. Consider the

following normal Haskell function that implements the well-known Quicksort algorithm:

$$
\begin{aligned}
&sort\ [\,] = [\,]\\
&sort\ (x : xs) = less +\!\!+ [x] +\!\!+ greater\\
&\quad \textbf{where}\\
&\qquad less = sort\ [y \mid y \leftarrow xs, y < x]\\
&\qquad greater = sort\ [y \mid y \leftarrow xs, y \geq x]
\end{aligned}
$$

The following parallel version is just a little more complicated (with addition of the underlined codes); *greater* is computed in parallel with *less* by wrapping the original expression $less +\!\!+ [x] +\!\!+ greater$ with *par greater* and *pseq less*.

$$
\begin{aligned}
&parSort\ [\,] = [\,]\\
&parSort\ (x : xs) =\\
&\quad \underline{par\ greater}\ (\underline{pseq\ less}\\
&\qquad (less +\!\!+ [x] +\!\!+ greater))\\
&\quad \textbf{where}\\
&\qquad less = parSort\ [y \mid y \leftarrow xs, y < x]\\
&\qquad greater = parSort\ [y \mid y \leftarrow xs, y \geq x]
\end{aligned}
$$

We can further control the granularity of the parallel task by switching to the normal *sort* when the number of elements of list is shorter enough.

$$
\begin{aligned}
&parSort\ [\,] = [\,]\\
&parSort\ l@(x : xs)\\
&\quad \mid \underline{shorter\ l\ =\ sort\ l}\\
&\quad \mid otherwise\ =\\
&\qquad\quad \underline{par\ greater}\ (\underline{pseq\ less}\\
&\qquad\qquad (less +\!\!+ [x] +\!\!+ greater))\\
&\quad \textbf{where}\\
&\qquad less = parSort\ [y \mid y \leftarrow xs, y < x]\\
&\qquad greater = parSort\ [y \mid y \leftarrow xs, y \geq x]
\end{aligned}
$$

It is worth noting that parallel functional programs are easy to debug. Regardless of the order in which computations are executed, the result of the program will always be the same. Specifically, the result will be identical to that obtained when the program is run sequentially. This implies that programs can be debugged sequentially, which represents a huge saving in effort.

Distributed functional programming

Distributed systems, by definition, do not share memory between nodes (computers)—which means that the imperative approach to parallel programming, with shared mutable data structures, is inappropriate in a distributed setting. Instead, nodes communicate by copying messages from sender to receiver; copying a *mutable* data structure changes semantics, because mutation of one copy is not reflected in the other, but *immutable* data can be copied transparently. This makes functional programming, with its immutable data structures, a natural fit. For systems which are designed to be *scalable*, in which overall system performance can be increased just by adding more nodes, it makes sense to use the same share-nothing communication between processes running on the *same* node, so that they may easily be deployed across multiple nodes as new nodes are added. Distributed systems can also offer high reliability, by using redundancy to tolerate failures, and in-service upgrade, by upgrading one node at a time while the others continue to deliver service.

```
qsort([]) -> [];
qsort([X|Xs]) ->
  Parent = self(),
  Less = [ Y || Y <- Xs, Y < X ],
  Grtr = [ Y || Y <- Xs, Y >= X ],
  spawn(fun() -> Parent ! {less, qsort(Less)} end),
  spawn(fun() -> Parent ! {grtr, qsort(Grtr)} end),
  receive {less, SortedLess} ->
    receive {grtr, SortedGrtr} ->
      SortedLess ++ [X] ++ SortedGrtr
    end
  end.
```

Figure 2. Parallel QuickSort in Erlang.

Erlang was designed at Ericsson in the late 1980s for building systems of this kind, originally for the telecom domain [113]. Later, Erlang proved to be ideal for building scalable internet services, and many start-ups have used it as their 'secret sauce'. The first of these was Bluetail AB, founded in 1998 to develop among others an SSL accelerator in Erlang, and sold for $140 million less than 18 months later; the most spectacular to date is WhatsApp, whose back-end servers are built in Erlang, and sold to Facebook in 2014 for $22 billion.

Erlang is a simple functional language with a slightly wordier syntax than Haskell; the factorial function defined in the introduction would be written in Erlang as follows:

$$
\begin{aligned}
&fac(0) \rightarrow 1;\\
&fac(N)\ \textbf{when}\ N > 0 \rightarrow N * fac(N-1)
\end{aligned}
$$

Erlang provides immutable lists and tuples, and LISP-like atoms, but no user-defined datatypes. Erlang lacks a static type system (Although many developers use Dialyzer [114], a static analysis tool that can detect many type errors.)—a reasonable choice since dynamic code loading, necessary for in-service upgrades, is difficult to type statically to this day.

To this functional core, Erlang adds features for concurrency and message passing. For example, Fig. 2 presents a (not very efficient) parallel version of QuickSort in Erlang. This function uses pattern matching on lists to select between the case of an empty list and a cons ($[X|Xs]$ means $x : xs$), and

in the latter case uses list comprehensions to select the elements less than or greater than the pivot, then *spawn*s two new processes to sort each sublist recursively. Spawning a process calls the function provided (as an Erlang λ-expression, **fun**() → . . . **end**) in the new process. Each of these processes sends the result of its recursive sort back to the parent process (*Parent* ! . . .), using the parent's *process identifier*, which is obtained by calling *self*(). Each result is tagged with an *atom* (*less* or *grtr*), which allows the parent process to **receive** the results in the correct order—messages wait in the recipient's 'mailbox' until a matching **receive** removes them from it, so it doesn't matter in which order the messages from the child processes actually arrive. Erlang processes share no memory—they each have their own heap—which means that the lists to be sorted must be copied into the new process heaps. This is why we filter *Xs* to extract the less and greater elements *before* starting the child processes: it reduces the costs of copying lists. The advantage of giving each process its own heap is that processes can garbage collect independently while other processes continue working, which avoids long garbage collection pauses and makes Erlang suitable for soft real-time applications.

Erlang adds mechanisms for one process to monitor another, turning a crash in the monitored process into a message delivered to the monitoring one. These mechanisms are used to support fault tolerance, with a hierarchy of supervisor processes which are responsible for restarting subsystems that fail; indeed, Erlang developers advocate a 'let it crash' approach, in which error-handling code (which is often complex and poorly tested) is omitted from most application code, relying on the supervisors for fault tolerance instead. Common patterns for building fault-tolerant systems are provided in the 'Open Telecom Platform' libraries—essentially higher order functions that permit fault-tolerant systems to be constructed just by instantiating the application-dependent behaviour.

Erlang's approach to concurrency and distribution has been very successful, and has been widely emulated in other languages—for example, Cloud Haskell [115] provides similar features to Haskell developers. One of the best known 'clones' is the Akka library for Scala [15], which is used among others to build Twitter's back-end services.

FUNCTIONAL THINKING IN PRACTICE

Functional programming has had big influences on education and other language design, and seen significant uses in industry.

Education

The style of teaching functional languages as first languages was pioneered by MIT in the 1980s, where functional language scheme was taught in the first course using the famous textbook *Structure and Interpretation of Computer Programs* [116]. Now many universities such as Oxford (Haskell) and Cambridge (ML) follow this functional-first style. In a recent survey (http://www.pl-enthusiast.net/2014/09/02/who-teaches-functional-programming/), 19 out of top 30 American universities in the US News 2014 Computer Science Ranking give their undergraduate students a serious exposure to functional languages. Compared to the other programming-first implementations, the functional-first approach has the advantages of reducing the effect of diversity of students in background, letting students focus on more fundamental issues, think more abstractly, and touch ideas of recursion, data structure, functions as first class data earlier. In fact, one of the explicit goals of Haskell's designers was to create a language suitable for teaching. Indeed, almost as soon as Haskell was defined, it was being taught to undergraduates at Oxford and Yale.

For learning the latest advanced functional programming techniques, there has been an excellent series of International Summer Schools on Advanced Functional Programming since 1995. Five such summer schools have been held so far in 1995, 1996, 1998, 2002, and 2004, with all lecture notes published in Lecture Notes in Computer Science by Springer. For the new applications of functional programming, there has been another series of Summer Schools on Applied Functional Programming organized by Utrecht University since 2009. The two-week course covers applications of functional programming, concentrating on topics such as language processing, building graphical user interfaces, networking, databases, and programming for the web.

For exchanging ideas of functional programming in education, there is a series of International Workshops on Trends in Functional Programming in Education since 2012, where novel ideas, classroom-tested ideas, and work in progress on the use of functional programming in education are discussed among researchers and teachers. They have previously been held in St Andrews, Scotland (2012); Provo Utah, USA (2013); and Soesterberg, the Netherlands (2014).

Influences on other languages

Ideas originated from functional programming such as higher order functions, list structure, type

inference, etc. have made their way into the design of many modern programming languages, unleashing influences in an unostentatious manner. It is well conceivable that 'main stream' programmers may be using functional features in their code without realizing.

One of the early examples of such is *blocks* in *Smalltalk*—a way of expressing lambda expressions and therefore higher order functions. More recently, the development of C# is influenced by functional programmers working in Microsoft. The LINQ features are directly inspired by monads and functional lists. Lambda expressions are introduced in C# 3.5, but higher order programming had been possible earlier on through delegates. Type inference is part of C#'s design and generics (polymorphism) are added in C# 2.0.

Java's generic type system introduced in Java 5 is based on ML's Hindley–Milner type systems. Subsequent releases gradually introduced type inference, another feature that is usually associated with functional languages. Java 8 embraced functional programming by releasing a wealth of features that specifically aimed at facilitating such a programming style. It includes dedicated support for lambda expressions and the passing of functions as method arguments (i.e. higher order functions), which is further made easy by the new feature of *method references*.

C++ aboards the lambda expression train in C++11. A particular merit of this C++ feature, as opposed to lambdas in other imperative languages, is that it offers fine-grained control over variable capture. Programmers are able to declare in their definitions whether the lambda bodies may refer to external variables (variables that are not formally declared as parameters) by value, by reference, or not at all—a step towards purity.

Modern multiparadigm languages often have good provision of functional features. Ruby is admitted by its inventor as having Lisp as its origin and blocks at its core. Additional lambda syntax is added in Ruby 1.9. Python adopted the list comprehension notation and has support for lambda expressions. The Python standard library includes many functional tools imported from Haskell and Standard ML. Scala is an object-functional language that has full support for functional programming. In addition to a strong static type system, it also features currying, type inference, immutability, lazy evaluation, and pattern matching.

Meijer's *reactive extensions* ('RX') [117] enable .NET developers to manipulate asynchronous data streams as immutable collections with purely functional operations defined over them. RX simplifies the construction of event-driven reactive programs dramatically. The design has been copied in many other languages, perhaps most notably at Netflix, where RxJava is heavily used in the Netflix API.

Uses in industry

It has often proven easier to adopt functional programming in small, new companies, rather than in large, existing software development organizations. By now, there have been many, many start-ups using functional programming for their software development. One of the first was Paul Graham's Viaweb, which built a web shop system in Lisp, and was later sold to Yahoo. Graham's well-known article 'Beating the Averages' (http://www.paulgraham.com/avg.html) discusses Lisp's importance as Viaweb's 'secret weapon'. He writes of studying competitors' job adverts: 'The safest kind were the ones that wanted Oracle experience. You never had to worry about those. You were also safe if they said they wanted C++ or Java developers. ...If I had ever seen a job posting looking for Lisp hackers, I would have been really worried.'

The first company to use Haskell was Galois, who develop 'high assurance' software, primarily under government contracts. Galois develop their code in Haskell—which in itself helps to ensure quality—but also use property-based testing, and the theorem provers we have discussed in this paper. More recently, Facebook boasts its spam detecting and remediating system being the largest Haskell deployment currently in existence, actively and automatically fighting off vast amounts of undesirable content from reaching its users. At the heart of the system is a DSL for writing the detection logic, firstly implemented in FXL (an in-house functional language) and is being migrated to Haskell at the moment.

Erlang, with its roots in industry, has a strong start-up culture. The first Erlang start-up, Bluetail AB, was set up after Ericsson decided to discourage the use of Erlang internally. Many key members of the Erlang team left Ericsson to found a company focussing on scalable and highly reliable internet products; within 18 months Bluetail was sold to Alteon Web Systems for $150 million. The founders were soon starting new companies, including Klarna AB (providing invoicing services to web shops in seven European countries) and Tail-f (sold to Cisco in 2014 for $175 million). The highest profile Erlang start-up to date, though, is WhatsApp, bought by Facebook in 2014 for $22 *billion*. A tech blog at the time asked 'How do you support 450 million users with only 32 engineers? For WhatsApp,

acquired earlier this week by Facebook, the answer is Erlang'. This illustrates nicely the benefits of productivity, scalability, and reliability that functional programming delivers.

Functional programming has also found many users in the financial sector, thanks not least to Peyton-Jones et al's seminal work on modelling *financial contracts* in Haskell [118]. Traders need to evaluate all kinds of financial derivatives quickly, so they can decide at which price to buy or sell them. But new, ingenious derivatives are being invented constantly, forcing traders to update their evaluation software continuously. By providing combinators for modelling contracts, this task can be accelerated dramatically, bringing a substantial advantage to traders who use them—the first trader able to evaluate a new derivative correctly stands to make a considerable profit. Credit Suisse was the first to use this technology, with the help of Augustsson, who now plays a similar role at Standard and Chartered—but these are far from the only examples. Functional programming is also used for automated trading at Jane Street, whose systems are built in OCaml. The clarity and quality of OCaml code helps Jane Street ensure that there are no bugs, which might rapidly lose large sums of money.

Languages such as Scala and Clojure (which run on the JVM) and F# (which runs on .NET) aim to be less disruptive, by integrating with an existing platform. They are enjoying wide adoption; for example, Apache Spark, a popular open-source framework for Big Data analytics, is largely built in Scala. Nowadays, there are successful companies whose business idea is to support the adoption of functional programming by their customers: Erlang Solutions (for Erlang), Typesafe (for Scala), Cognitect (for Clojure), Well-typed and FP Complete (for Haskell).

New applications of functional programming are appearing constantly. A good way to follow these developments is via developer conferences such as the Erlang Factory, Scala Days and Clojure/conj, and also via the annual conference on Commercial Applications of Functional Programming (http://cufp.org) , held in association with ICFP.

CONCLUSION

Twenty-five years ago, functional programming was high in the sky, favoured only by researchers in the ivory tower. Twenty-five years on, it has touched down on the ground and had a wide impact on the society as a new generation of programming. Where will functional programming be twenty-five years from now?

REFERENCES

1. Hughes, J. Why functional programming matters. *Comput J* 1989; **32**: 98–107.
2. Hudak, P, Jones, SLP and Boutel, WP *et al*. Report on the programming language Haskell a non-strict purely functional language. *SIGPLAN Not* 1992; **27**: 1.
3. Bird, R. *Introduction to Functional Programming*, 2nd edn. Upper Saddle River, New Jersey: Pearson Education, 1998.
4. Hudak, P. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. New York, NY, USA: Cambridge University Press, 2000.
5. Paulson, LC. *ML for the Working Programmer*, 2nd edn. New York, NY, USA: Cambridge University Press, 1996.
6. Armstrong, J. *Programming Erlang: Software for a Concurrent World*. Raleigh: Pragmatic Bookshelf, 2007.
7. Milner, R, Tofte, M and Macqueen, D. *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997.
8. Minsky, Y, Madhavapeddy, A and Hickey, J. *Real World OCaml: Functional Programming for the Masses*. O'Reilly Media, 2013.
9. Minsky, Y. Ocaml for the masses. *Commun ACM* 2011; **54**: 53–8.
10. Hansen MR Rischel, H. *Functional Programming Using F#*. New York, NY, USA: Cambridge University Press, 2013.
11. Steele, GL, Jr. *Common LISP: The Language*. Newton, MA, USA: Digital Press, 1984.
12. Seibel, P. *Practical Common Lisp*, 1st edn. Berkely, CA, USA: Apress, 2012.
13. Dybvig, RK. *The Scheme Programming Language*, 4th edn. Cambridge: The MIT Press, 2009.
14. Felleisen, M, Findler, RB and Flatt, M *et al*. Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. Cambridge, MA, USA: MIT Press, 2001.
15. Wampler, D and Payne, A. *Programming Scala: Scalability = Functional Programming + Objects*, 1st edn. O'Reilly Media, 2009.
16. Halloway, S. *Programming Clojure*, 1st edn. Raleigh, NC: Pragmatic Bookshelf, 2009.
17. Claessen, K and Hughes, J. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00. New York, NY, USA: ACM, 2000, 268–79.
18. Leroy, X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*. New York, NY, USA: ACM Press, 2006, 42–54.
19. Yang, X, Chen, Y and Eide, E *et al*. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11. New York, NY, USA: ACM, 2011.
20. Hinze, R. Generics for the masses. *J Funct Progr* 2006; **16**: 451–83.
21. Chetali, B. Security testing and formal methods for high levels certification of smart cards. In *Proceedings of the 3rd International Conference on Tests and Proofs*, TAP '09. Berlin: Springer, 2009, 1–5.

22. Barthe, G, Grégoire, B and Zanella Béguelin, S. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09. New York, NY, USA: ACM, 2009, 90–101.
23. Hughes, J. QuickCheck testing for fun and profit. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*, PADL'07. Berlin: Springer, 2007, 1–32.
24. Arts, T, Hughes, J and Johansson, J *et al.* Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06New York, NY, USA: ACM, 2006, 2–10.
25. Hughes JM Bolinder, H. Testing a database for race conditions with QuickCheck: none. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11. New York, NY, USA: ACM, 2011, 72–7.
26. Arts, T, Hughes, J and Norell, U *et al.* Testing autosar software with QuickCheck. In *Testing: Academic and Industrial Conference—Practice and Research Techniques, Part of the ICST Workshop Proceedings*. Graz, Austria: IEEE Digital Library, 2015.
27. Gill, A, Launchbury, J and Peyton Jones, SL. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93. New York, NY, USA: ACM, 1993, 223–32.
28. Takano, A and Meijer, E. Shortcut deforestation in calculational form. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95. New York, NY, USA: ACM, 1995, 306–13.
29. Hu, Z, Iwasaki, H and Takeichi, M *et al.* Tupling calculation eliminates multiple data traversals. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97. New York, NY, USA: ACM, 1997, 164–75.
30. Farmer, A, Gill, A and Komp, E *et al.* The hermit in the machine: a plugin for the interactive transformation of GHC core language programs. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12. New York, NY, USA: ACM, 2012, 1–12.
31. Fegaras, L and Sheard, T. Revisiting catamorphisms over datatypes with embedded functions. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '96*. New York, NY, USA: ACM, 1996, 284–94.
32. Meijer, E, Fokkinga, M and Paterson, R. Functional programming with bananas lenses envelopes and barbed wire. In *Proceddings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture* (LNCS 523). Cambridge, MA: ACM, 1991, 124–44.
33. Sheard, T and Fegaras, L. A fold for all seasons. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93. New York, NY, USA: ACM, 1993, 233–42.
34. Hinze, R, Wu, N and Gibbons, J. Conjugate hylomorphisms—or: the mother of all structured recursion schemes. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15. New York, NY, USA: ACM, 2015, 527–38.
35. Hu, Z, Iwasaki, H and Takeichi, M. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96. New York, NY, USA: ACM, 1996, 73–82.
36. Odersky, M and Wadler, P. Pizza into java: translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97. ACM: New York, NY, USA, 1997, 146–59.
37. Wadler, P. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89. New York, NY, USA: ACM, 1989, 347–59.
38. Reynolds, JC. Types abstraction and parametric polymorphism. In: Mason, R.E.A (ed.). *Proceedings of IFIP 9th World Computer Congress*. Amsterdam: Elsevier Science Publishers B. V. (North-Holland), 1983, 513–23.
39. Oliveira BCS Moors, A and Odersky, M. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10. New York, NY, USA: ACM, 2010, 341–60.
40. Bertot, Y and Castéran, P. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*, 1st edn. Springer Publishing Company, Incorporated, 2010.
41. Norell, U. Dependently typed programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08. Berlin: Springer, 2009, 230–66.
42. Bird, R. Constructive functional programming. In *STOP Summer School on Constructive Algorithmics*. Ameland, 1989. http://www.staff.science.uu.nl/~swier101/STOP/.
43. Bird, R and de Moor, O. *Algebra of Programming*. Upper Saddle River, NJ, USA: Prentice Hall, 1997.
44. Dijkstra, EW. Programming as a discipline of mathematical nature. *Am Mathemat Mon Comp J* 1974; **81**: 608–12.
45. Bird, R. *Pearls of Functional Algorithm Design*, 1st edn. New York, NY, USA: Cambridge University Press, 2010.
46. Bird RS Moor, O. Solving optimisation problems with catamorphism. In *Proceedings of the Second International Conference on Mathematics of Program Construction*. London, UK: Springer, 1993, 45–66.
47. Hu, Z, Yokoyama, T and Takeichi, M. Program optimizations and transformations in calculational form. In *Summer School on Generative and Transformational Techniques in Software Engineering* (LNCS 4043). Braga, Portugal: Springer, 2006, 139–64.
48. Takano, A, Hu, Z and Takeichi, M. Program transformation in calculational form. *ACM Comput Surv* 1998; **30**.
49. Smith, DR. KIDSa knowledges-based software development system. In: Lowry, MR and McCartney, RD (eds) *Proceedings of AAAI'88 Workshop on Automating Software Design*. The MIT Press, 1991, 483–514.
50. de Moor, O and Sittampalam, G. Generic program transformation. In *Advanced Functional Programming, Lecture Notes in Computer Science*. Berlin: Springer, 1998, 116–49.
51. Mu, S, Ko, H and Jansson, P. Algebra of programming in Agda: dependent types for relational program derivation. *J Funct Progr* 2009; **19**: 545–79.
52. Naiman, L and Hehner, E. Netty: a prover's assistant. In *COMPUTATION TOOLS 2011: The Second International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking*, Rome, 2011 September 25–30.
53. Tesson, J, Hashimoto, H and Hu, Z *et al.* Program calculation in Coq. In *Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology*, AMAST'10. Berlin: Springer, 2011, 163–79.
54. Katayama, S. Recent improvements of MagicHaskeller. In *Proceedings of Workshop on Approaches and Applications of Inductive Programming*, AAIP '10(LNCS 5812). Berlin: Springer, 2010, 174–93.
55. Katayama, S. An analytical inductive functional programming system that avoids unintended programs. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12. New York, NY, USA: ACM, 2012, 43–52.

56. Moggi, E. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. Piscataway, NJ, USA: IEEE Press, 1989, 14–23.
57. Moggi, E. Notions of computation and monads. *Inf Comput* 1991; **93**: 55–92.
58. Wadler, P. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90. New York, NY, USA: ACM, 1990, 61–78.
59. Wadler, P. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92. New York, NY, USA: ACM, 1992, 1–14.
60. Peyton Jones SL Wadler, P. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93. ACM: New York, NY, USA, 1993, 71–84.
61. Launchbury, J and Peyton Jones SL, . State in Haskell. *Lisp Symb Comput* 1995; **8**: 293–341.
62. Launchbury, J and Sabry, A. Monadic state: axiomatization and type safety. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97. New York, NY, USA: ACM, 1997, 227–38.
63. Peyton Jones, S, Gordon, A and Finne, S. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96. New York, NY, USA: ACM, 1996, 295–308.
64. Harris, T, Marlow, S and Peyton-Jones, S *et al.* Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05. New York, NY, USA: ACM, 2005, 48–60.
65. Liang, S, Hudak, P and Jones, M. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95. New York, NY, USA: ACM, 1995, 333–43.
66. Steele, GL, Jr. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94. New York, NY, USA: ACM, 1994, 472–92.
67. Kiselyov, O, Sabry, A and Swords, C. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13. New York, NY, USA: ACM, 2013, 59–70.
68. Schrijvers, T and Oliveira, BCS. Monads zippers and views: virtualizing the monad stack. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11. New York, NY, USA: ACM, 2011, 32–44.
69. Snyder, M and Alexander, P. Monad factory: type-indexed monads. In *Proceedings of the 11th International Conference on Trends in Functional Programming*, TFP'10. Berlin: Springer, 2011, 198–213.
70. Hughes, J. Generalising monads to arrows. *Sci Comput Progr* 2000; **37**: 67–111.
71. Paterson, R. Arrows and computation. In: Gibbons, J and de Moor, O (eds) *The Fun of Programming*. Palgrave Macmillan, 2003, 201–22.
72. Mcbride, C and Paterson, R. Applicative programming with effects. *J Funct Progr* 2008; **18**: 1–13.
73. Hudak, P, Courtney, A and Nilsson, H *et al.* Arrows robots and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University* (LNCS 2638). Berlin: Springer, 2003, 159–87.
74. Devriese, D, Sergey, I and Clarke, D *et al.* Fixing idioms: a recursion primitive for applicative DSLs. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13. New York, NY, USA: ACM, 2013, 97–106.
75. Lindley, S, Wadler, P and Yallop, J. Idioms are oblivious arrows are meticulous monads are promiscuous. *Electron Notes Theor Comput Sci* 2011; **229**: 97–117.
76. Hudak, P. Building domain-specific embedded languages. *ACM Comput Surv.* 1996; **28**, 4es, Article 196.
77. Hudak, P. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98. Washington, DC, USA: IEEE Computer Society, 1998, 134–42.
78. Hudak, P, Hughes, J and Peyton Jones, S *et al.* A history of Haskell: being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III. New York, NY, USA: ACM, 2007, 12-1-; 12-55.
79. Hutton, G and Meijer, E. Monadic parser combinators. *Technical Report NOTTCS-TR-96-4*. Department of Computer Science, University of Nottingham, 1996.
80. Hutton, G and Meijer, E. Monadic parsing in Haskell. *J Funct Progr* 1998; **8**: 437–44.
81. Baars, Al, Löh, A and Swierstra, SD. Parsing permutation phrases. *J Funct Progr* 2004; **14**: 635–46.
82. Chakravarty, MMT. Lazy lexing is fast. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, FLOPS '99. London, UK: Springer, 1999, 68–84.
83. Claessen, K. Parallel parsing processes. *J Funct Progr* 2004; **14**: 741–57.
84. Ford, B. Packrat parsing: simple powerful lazy linear time functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02. New York, NY, USA: ACM, 2002, 36–47.
85. Swierstra SD Duponcheel, L. Deterministic error-correcting combinator parsers. In *Advanced Functional Programming, Second International School-Tutorial Text*. London, UK: Springer, 1996, 184–207.
86. Elliott, C, Finne, S and De Moor, O. Compiling embedded languages. *J Funct Progr* 2003; **13**: 455–81.
87. Gibbons, J and Wu, N. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14. New York, NY, USA: ACM, 2014, 339–47.
88. Wadler, P. The expression problem. Java Genericity Mailing list 1998. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (12 November 1998, date last accessed).
89. Axelsson, E. A generic abstract syntax model for embedded languages. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12. New York, NY, USA: ACM, 2012, 323–34.
90. Bahr, P and Hvitved, T. Compositional data types. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, WGP '11. New York, NY, USA: ACM, 2011, 83–94.
91. Carette, J, Kiselyov, O and Shan, C. Finally tagless partially evaluated: tagless staged interpreters for simpler typed languages. *J Funct Progr* 2009; **19**: 509–43.
92. Swierstra, W. Data types à la carte la carte. *J Funct Progr* 2008; **18**: 423–36.
93. Svenningsson, J and Axelsson, E. Combining deep and shallow embedding for EDSL. In: Loidl, H-W and Peña, R (eds) *Trends in Functional Programming* (LCNS 7829). Berlin: Springer, 2013, 21–36.
94. Persson, A, Axelsson, E and Svenningsson, J. Generic monadic constructs for embedded languages. In: Gill, A and Hage, J (eds) *Implementation and Application of Functional Languages* (LCNS 7257). Berlin: Springer, 2012, 85–99.

95. Sculthorpe, N, Bracker, J and Giorgidze, G *et al*. The constrained-monad problem. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13. New York, NY, USA: ACM, 2013, 287–98.
96. Svenningsson, JD and Svensson, BJ. Simple and compositional reification of monadic embedded languages. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13. New York, NY, USA: ACM, 2013, 299–304.
97. Dean, J and Ghemawat, S. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*, OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, 10.
98. Backus, J. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun ACM* 1978; **21**: 613–41.
99. Jones, SP. Parallel implementations of functional programming languages. *Comput J* 1989; **32**: 175–86.
100. Loidl HW Rubio, F and Scaife, N *et al*. Comparing parallel functional languages: programming and performance. *Higher Order Symb Comput* 2003; **16**: 203–51.
101. Cole, M. Higher-order functions for parallel evaluation. In: Hall, CV, Hughes, RJM and O'Donnell, JT (eds). *Proceedings of Computing Science Department Research Report 89/R4*. Isle of Bute, Scotland: Glasgow University, 1989, 8–20.
102. Cole, M. *Algorithmic skeletons : a structured approach to the management of parallel computation*. Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.
103. Skillicorn DB, . Architecture-independent parallel computation. *IEEE Comput* 1990; **23**: 38–51.
104. Blelloch, GE. NESL: a nested data parallel language. *Technical Report CMU-CS-92-103*. School of Computer Science, Carnegie-Mellon University, 1992.
105. Chin, Wn, Takano, A and Hu, Z. Parallelization via context preservation. In *IEEE Computer Society International Conference on Computer Languages*. Chicago, USA: Loyola University Chicago, 1998.
106. Hu, Z, Takeichi, M and Chin, W-N. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*. San Diego, CA, USA: ACM Press, 1998, 316–28.
107. Hu, Z, Takeichi, M and Iwasaki, H. Diffusion: calculating efficient parallel programs. In *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (BRICS Notes Series NS-99-1). San Antonio, TX, 1999, 85–94.
108. Matsuzaki, K, Hu, Z and Takeichi, M. Towards automatic parallelization of tree reductions in dynamic programming. In *18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2006)*. Cambridge, MA, USA: ACM Press, 2006, 39–48.
109. Gibbons, J. The third homomorphism theorem. *J Funct Progr* 1996; **6**: 657–65.
110. Morita, K, Morihata, A and Matsuzaki, K *et al*. Automatic inversion generates divide-and-conquer parallel programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07. New York, NY, USA: ACM, 2007, 146–55.
111. Chakravarty MMT Leshchinskiy, R and Jones, SP *et al*. Data parallel Haskell: a status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07. New York, NY, USA: ACM, 2007, 10–8.
112. Hammond, K. Why parallel functional programming matters: panel statement. In *Proceedings of the 16th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe'11 Berlin: Springer, 2011, 201–5.
113. Armstrong, J. A history of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III. New York, NY, USA: ACM, 2007, 6–1–6–26.
114. Sagonas, K. Detecting defects in Erlang programs using static analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '07. New York, NY, USA: ACM, 2007, 37.
115. Epstein, J, Black, AP and Peyton-Jones, S. Towards Haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11. New York, NY, USA: ACM, 2011, 118–29.
116. Abelson, H, Sussman, GJ and Sussman, J. *Structure and Interpretation of Computer Programs*. Cambridge, MA, USA: MIT Press, 1985.
117. Meijer, E. Reactive extensions (rx): curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10. New York, NY, USA: ACM, 2010, **11**: 1.
118. Peyton Jones, S, Eber, J-M and Seward, J. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00. New York, NY, USA: ACM, 2000, 280–92.
119. Bringert, B. xhtml: an XHTML combinator library. https://hackage.haskell.org/package/xhtml (9 May 2012, date last accessed).
120. Elliott, C and Hudak, P. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97. New York, NY, USA: ACM, 1997, 263–73.
121. Leijen, D, Andersson, C and Andersson, M *et al*. haskelldb: a library of combinators for generating and executing SQL statements. https://hackage.haskell.org/package/haskelldb (29 September 2014, date last accessed).
122. Blelloch, GE. Scans as primitive operations. *IEEE Trans Comp* 1989; **38**: 1526–38.