9-2009

Scientific Programming: The Promises of Typed, Pure, and Lazy Functional Programming: Part II

Konstantin Läufer  
*Loyola University Chicago*, klaeufer@gmail.com

George K. Thiruvathukal  
*Loyola University Chicago*, gkt@cs.luc.edu

Recommended Citation  

This work is licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 License.  
Copyright © 2009 Konstantin Läufer and George K. Thiruvathuka
The Promises of Typed, Pure, and Lazy Functional Programming: Part II

By Konstantin Läufer and George K. Thiruvathukal

This second installment picks up where Konrad Hinsen’s article “The Promises of Functional Programming” from the July/August 2009 issue left off, covering static type inference and lazy evaluation in functional programming languages.

In the first installment in this series on functional programming, Konrad Hinsen introduced the functional programming paradigm, which encourages recursion and higher-order functional abstraction in algorithms. In this second installment, we explore several other aspects of the functional paradigm. Using examples in the Clojure, Java, and Haskell languages, we discuss some of the benefits of static typing with type inference and pure, side-effect-free functional programming with lazy evaluation. In particular, we argue that these features make Haskell a compelling choice for general software development. The scientific computing community, however, will likely require better numeric library support and efficient functional versions of scientific algorithms before accepting Haskell more widely.

Dynamic and Static Typing

Type systems in programming languages ensure that we correctly use every program element according to its type—that is, according to the abstraction the program element represents. Typically, numeric values participate in arithmetic operations and comparisons, lists participate in element access and concatenation, and so on.

First, let’s look at type system behavior in Clojure, a modern LISP dialect that runs on the Java Virtual Machine. As Hinsen showed in the first installment, we can create a list from some elements and then prepend an element to that list:

```
user=> (cons 3 (list 4))
(3 4).
```

However, we can’t prepend an element to another element

```
user=> (cons 3 4)
java.lang.IllegalArgumentException: Don't know how to create ISeq from: Integer
```

because the predefined cons function requires its second argument to be of a list type. Clojure’s type system detected the attempt to invoke the function on an argument of the wrong type, reported a type error, and stopped program execution. Clojure uses dynamic typing: type errors don’t get detected until the program executes. Other languages with dynamic typing include Groovy, JavaScript, Lisp, Objective-C, Perl, Python, Ruby, Scheme, and Smalltalk.

Now, let’s look at the behavior of Java’s type system. We can create a list from some elements and then prepend an element to that list

```
List<Integer> l = Arrays.asList(4);
l.add(3);  
```

but we can’t prepend an element to another element

```
Integer i = new Integer(4);
i.add(3);  
```

or

```
List<Integer> l = new Integer(4);
l.add(3);  
```

In the first case, the error message is The method add(int) is undefined for the type Integer, while in the second case, it’s Type mismatch: cannot convert from Integer to List<Integer>. Java’s type system detected these errors, reported them while we were editing or compiling the program containing this code, and refused to let us run it. Java uses static typing: type errors are detected at compile time and programs with type errors won’t even compile. Other languages that use static typing include Ada, C, C++, C#, Fortran, Haskell, Java, ML, Pascal, and Scala. Among those, the ones that also support higher-order features such as first-class functions (C#, Haskell, ML, and Scala) or first-class objects (Ada, C#, Java, and Scala) are called higher-order typed, or HOT.

Assuming a sound type system, static typing represents a conservative approach: a program will be allowed to
Even in typed languages such as C++, C#, and Java, we must sometimes convert a numeric value from one type to another. When going from 16-bit integer to a 32-bit integer, for example, there’s no loss of information, and it’s safe to just let the conversion happen implicitly. In the other direction, however, part of the original number might get lost and conversion should happen only if the programmer takes responsibility. The programmer takes this responsibility using a typecast—or simply, cast—construct. In the following Java snippet, for example, the last line would cause a compile-time error without the cast:

```java
int i = 25;
float x = i;
int k = (int) f;
```

In contrast, Haskell provides such conversions using explicit conversion functions. In the equivalent situation, for example, we could choose from among `ceiling`, `floor`, `round`, and `truncate`.

There are similar situations where programmers think they know far more about a certain object’s type than the compiler could possibly know. Such a situation might arise with heterogeneous, predefined type collections, where programmers are unable to define a specific common abstraction after the fact (another would be to define wrappers with a common interface for the predefined types):

```java
List<Object> links =
    new ArrayList<Object>();
links.add("http://www.computer.org/cise");
links.add(new URL("http://cise.aip.org/"));
...
for (final Object o : links)
    if (o instanceof String) {
        String s = (String) o;
        URL link = new URL(s);
        // do something with the link
    }
```

In this example, we mix strings and real URL objects within the same collection. We then iterate through the collection and do something with the strings. We use the `instanceof` operator to determine which objects are strings, cast them from type `Object` to the more specific type `String`, and do something string-specific, such as creating a URL from it. If we hadn’t used the `instanceof` operator, the cast would have failed as soon as the iteration reached the URL instance.

Casts from a more general to a more specific type can fail at runtime—as, for example, when an object isn’t of the expected specific type or a more specific type. Therefore, we note that only programs without casts are guaranteed to be free of type errors at runtime. By contrast, casts in the opposite direction (from a specific to a more general class) can never fail, so they don’t need a runtime check. Finally, casts between unrelated classes, such as `String` to `Integer`, can never succeed, so they always cause a compile-time type error.

Casting versus conversion functions was a huge subject of debate in the C++ community back when cycles were precious and any stealth data “transformation” (no matter how trivial) could incur a substantial performance overhead. (This was due to the copy-construction cascading effect, which is no longer an issue in most modern object-oriented languages as they typically copy references instead of values.) Today, however, casting is often a liability that actually causes runtime typing errors—even in programming languages with well-founded type systems like Java. Worse, it’s well known (especially among seasoned C/C++ programmers such as ourselves) that not all casts should be allowed. For example, there is a natural promotion of `short -> int -> long -> long long` (in C) that is lossless; however, the reverse transformation is unnatural, highly error-prone, and nonportable, especially in a heterogeneous computing world that still features 16-, 32-, and 64-bit processors. Using conversion functions (instead of casts) lets you ensure that all meaningful conversions are performed correctly (forward and backward, by making all down conversions explicit and precise) with only a slight inconvenience to the programmer (and inlining can help alleviate most performance issues).

run only if we can guarantee that it’s free of type errors. Thus, a program that passes type checking is guaranteed not to fail at runtime because of type errors (with some restrictions, as we discuss in the sidebar “On Typecasting and Conversions”). Because conditions aren’t evaluated until runtime, for example, the static type checker would still reject the following program, even though the erroneous code would always be skipped:

```java
if (3 > 4) {
    List<Integer> l = new Integer(4);
    l.add(3);
}
```

That might seem a little restrictive; dynamically typed languages would have no complaints about it. But suppose the conditional expression is much more complex and evaluates to true on some rare occasions as part of some mission-critical software—such as air-traffic control, Internet routers, and ATM machines—where every
single line of code, reachable or not, must be checked before it’s embedded and deployed on a large scale. With dynamic typing, disaster could ensue if the conditional evaluated to true. (Although this example might seem improbable, binary patching is routinely used to change compiled code in a deployed system, such as for updating operating systems, and many industries use the technique to patch equipment, such as telecommunications switches, that require near-zero downtime.) With static typing, we’d have known about the error inside the if statement long before deploying the system.

Static typing also has a significant performance benefit: because the compiler guarantees type safety, we don’t need the runtime checks found in dynamically typed languages and our code runs faster (especially when it’s all written in the same language).

But statically typed languages’ safety comes at a price: we must declare the type of each variable in the program, including formal arguments of methods and instance variables. This can get tedious quickly as code gets more complex, especially with higher-order programming. There are, however, several emerging languages that automatically assign types to variables. For example, in the Boo language (a type-safe version of Python designed for the .NET platform), we can write that

\[
\begin{align*}
x &= \text{new } \text{Integer}(4) \\
y &= \text{new } \text{Integer}(5) \\
z &= x + y
\end{align*}
\]

Here, \(x\) and \(y\) are both assigned the static type of \text{Integer} (on first use) and \(z\) is assigned the same static type (by inferring the expression’s result type). If \(y\) were a \text{Float} instead of \text{Integer}, we’d infer \(z\) to be a \text{Float}.

It’s important to distinguish this from what happens in languages like Python, where the static type of \(x, y,\) and \(z\) are all “object” types; the static type of each is unknown. Only the runtime type is known by examining type(\(x\)), type(\(y\)), or type(\(z\)). The C# language (from Microsoft’s .NET family) also supports automatic variable typing. So, we can write the above as

\[
\text{var } x = \text{new Integer}(4); \\
\text{var } y = \text{new Integer}(5); \\
\text{var } z = x + y;
\]

In the following typical example of higher-order programming, we use the strategy pattern to represent a custom comparison strategy as a function (in Java, encapsulated within an object) that we pass to the actual sort function:

\[
\begin{align*}
\text{List<} \text{Integer} \text{> } l &= \ldots \\
\text{Comparator<} \text{Integer} \text{> } c &= \text{new Comparator<} \text{Integer} \text{>() { \\
    \text{public int compare(} \\
    \text{    Integer l, Integer r) { \\
    \text{        ... } \\
    \text{    } \\
    \text{    } \\
    \text{    Collection.s} \text{ort(}l, c); \\
}\end{align*}
\]

In the next section, we discuss how to do away with most of this tedium.

**Type Inference and Universal Quantification**

Wouldn’t it be nice if we could have our cake and eat it, too? That is, couldn’t we have the safety of static typing without its tedium? Surprisingly, the answer is yes! There are some functional languages that have type systems as or more powerful than those of, say, Java, but that are capable of figuring out the correct, intended types of variables and functions in almost all situations.

Let’s see how this works out in Haskell (using the Glasgow Haskell interpreter, ghci; see www.haskell.org). We start with a typed version of the `makeAdder` function from the first installment. As we recall, `makeAdder` takes a numeric argument \(x\) and returns a new function on another numeric argument \(y\) that adds \(x\) and \(y\):

\[
\text{Prelude> let } \text{makeAdder } x = \\
\quad \text{\textbackslash } y \rightarrow x + y
\]

Within the interpreter, we use the keyword `let` to define new variables (the keyword isn’t necessary for top-level variable definitions in source files). Let’s now apply `makeAdder` to a suitable argument:

\[
\text{Prelude> makeAdder } 3
\]

No instance for (\text{Show} \ (t \rightarrow t))

Given that applying `makeAdder` to a single argument results in a new function that expects another argument, it’s understandable that the interpreter complains that it doesn’t know how to print this (or any other) function. Instead, let’s use the interpreter’s :\text{t} (a shorthand for :\text{type}) command to take a look at the expression’s precise type:

\[
\text{Prelude> :t makeAdder } 3
\]

\[
\text{makeAdder } 3 :: (\text{Num } t) \Rightarrow t \rightarrow t
\]

This type means that the function takes an argument of type \(t\) and returns a result of type \(t\), assuming that \(t\) is a numeric type. Indeed, we can apply this function to another argument and get the expected numeric result:

\[
\text{Prelude> (makeAdder } 3) \ 4
\]

\[
7
\]
or simply

Prelude> makeAdder 3 4
7

Why didn’t the interpreter simply type the function as, say, \texttt{Integer \rightarrow Integer}, especially given that we passed an integer constant as the first argument? Let’s look at that integer constant’s type by itself:

Prelude> :t 3
3 :: (Num t) => t

Aha! What we see is that 3 can have any required numeric type, so it’s much more general than an integer value. Typically, Haskell’s type system deals quite naturally with the overloading of numeric constants and operators, and we don’t usually have to think much about it. (To handle general operator and function overloading in a systematic yet unobtrusive way, Haskell uses \textit{type classes}: a type class \texttt{Num} includes \texttt{Int} for fixed-precision integers, \texttt{Integer} for arbitrary-precision integers, and so on. Accordingly, we can constrain the value 3 to any specific type included in the type class:

Prelude> :t 3
3 :: (Num t) => t

The details of systematic overloading in Haskell are beyond our scope here, but we offer a specific example later.)

What about the type of \texttt{makeAdder} itself?

Prelude> :t makeAdder
makeAdder :: (Num a) => a \rightarrow a \rightarrow a

Programmers (like us) who grew up on imperative programming languages usually separate the parameter types from the return type in a typical function definition. So, the Haskell syntax might initially seem a bit weird. That is, when you see an expression like \texttt{a \rightarrow a \rightarrow a}, your first inclination is to think about the first two items between arrows, \texttt{a} and \texttt{a}, as being the function’s arguments, and the third item, \texttt{a}, as being the result type. This is in fact correct if you apply the function to both arguments, but as we saw, the function can also take its arguments one after the other.

This highlights a key difference between most functional and imperative languages. Functional languages encourage higher-order functions that \textit{curry} their arguments (after the logician Haskell B. Curry, who described this technique). That is, functions are applied to some or all of their arguments one after the other, which means that the result can be either a function or a value.

For example, we can define the function \texttt{inc} (increment by 1) by partially applying the \texttt{makeAdder} function:

Prelude> let inc = makeAdder 1
Prelude> inc 4
5

Coming back to our list example, we can create a list from some elements and then prepend an element to that list. (Similar to modern scripting languages such as Python, functional languages typically offer syntactic support for common data structures such as lists and tuples.)

Prelude> 3 : [4]
[3,4]

Beyond minor syntactic differences, this looks just like the Clojure example above. But a major difference is that Haskell knows the type of the : operator, equivalent to Clojure’s \texttt{cons} function. (Haskell supports symbolic infix operators, which become prefix functions when surrounded by parentheses.)

Prelude> :t (:)
(:) :: a \rightarrow [a] \rightarrow [a]

In other words, this operator takes a value of type \texttt{a} and a list with elements of type \texttt{a}, and produces another list of type \texttt{a}.

The list’s elements can be of any type as long as that type is used consistently; this concept is known as \textit{universal quantification}: it works for all types \texttt{a}. These lists are homogeneous: all elements must be of the same type, so we can’t prepend, say, a number to a list of characters. (We discuss type-safe heterogeneous lists later.)

Prelude> 3 : ['a', 'b']
No instance for (Num Char)
arising from the literal ‘3’ at <interactive>:1:0
Possible fix: add an instance declaration for (Num Char)

As above, we can’t prepend an element to another element:

Prelude> 3 : 4
No instance for (Num [t])
arising from the literal ‘4’ at <interactive>:1:4
Possible fix: add an instance declaration for
(Num [t])

These error messages say that it found some numeric value instead of either a character value or a list value. If we narrow value 4's type down from a general number to a fixed-precision integer of type Int, then the message becomes clearer:

Prelude> 3 : (4 :: Int)
  Couldn’t match expected
type ’[t]’ against
inferred type ’Int’
In the second argument of
’(:)’, namely ’(4 ::
Int)’

As a rather contrived example of higher-order typed programming, we’ll use a Haskell version of the countdown method from Hinsen’s article, put together from predefined Haskell methods without recursion or even a formal argument:

Prelude> let countdown =
  reverse . (flip take [0 ..]) . (+ 1)

This terse definition deserves a bit of an explanation:

• the dot operator performs function composition;
• + 1 is the partial application of addition to 1, so it adds one to the other argument it receives (once countdown is applied to an actual argument);
• [0 ..] is an (infinite!) list of non-negative integers;
• take takes only a given number of elements of a list, so it makes the infinite list finite; and
• reverse does the expected.

Here is the whole story, step by step:

Prelude> countdown 5
  ==>(reverse . (flip take [0 ..]) . (+ 1)) 5
  ==>(reverse (flip take [0 ..] (1 + 5))
  ==>(reverse (flip take [0 ..] 6)
  ==>(reverse (take 6 [0 ..])
  ==>[reverse [0,1,2,3,4,5]
  ==>[5,4,3,2,1,0]

Of course, any sane Haskell programmer would simply write

Prelude> let countdown n =
  reverse [0 .. n]

**Pure Functional Programming and Lazy Evaluation**

Pure functional languages such as Haskell are side-effect free, except within special language constructs for actions, such as input/output, that by definition entail side-effects. The absence of side-effects guarantees referential transparency: we can replace any expression in a program with its resulting value without changing the program’s meaning. This makes it possible to reason about programs and their correctness, similar to the way we’d reason about mathematical formulas. In practice, this means that every variable is defined exactly once and can’t be modified later.

Although side effects are common in many modern scientific and high-performance computing codes, minimizing side effects actually makes work easier for the compiler, especially for parallel-language compilers. Because compilers have a notoriously difficult time with side effects, we can’t use many optimization techniques—such as common-subexpression elimination or invariant hoisting. In theory and practice, typed functional programming languages compile well, and the same compilation techniques work well with imperative languages like Java, Clean, OCaml, and F# offer other, modern examples. Another advantage could become even more prominent in the multicore and novel computing era: if multiple expressions are ready to be evaluated, you can throw any number of cores at them.

With referential transparency, it’s possible to delay the evaluation of a complex expression’s subexpressions until it’s required for computing the final top-level result. This evaluation strategy is called lazy evaluation—as opposed to the more familiar eager evaluation—where all subexpressions are evaluated inside-out, regardless of whether they matter for the final result. We can think about lazy evaluation as a generalization of Boolean short-circuit evaluation, but without the pitfalls of missing skipped subexpressions’ side effects.

Lazy evaluation generally leads to the inevitable overhead of keeping track of computations that are ready to be evaluated but haven’t been required yet. Consequently, the memory footprint of programs in lazy languages should be higher than equivalent programs in eager languages. Data from the Computer Languages Benchmarks Game (http://shootout.alioth.debian.org/) confirms this expectation. Nevertheless, the Glasgow Haskell Compiler now produces such effectively optimized code that those same benchmarks run within a factor of at most three of equivalent C and Fortran programs, and even up to three times faster in some cases. Recent work on Haskell runtime support on multicore hardware has shown promising initial results.2 As we’ve already seen, lazy evaluation gives us the seemingly magic ability...
to express infinite structures such as [0 ..] naturally and concisely. By contrast, in conventional languages with eager evaluation, we might represent infinite structures by wrapping the delayed computation within a function or method. For example, in Java, we can express infinite lists using the `Iterator` abstraction:

```java
class Naturals implements Iterator<Integer> {
    private int value = 0;
    public boolean hasNext() {
        return true; }
    public Integer next() {
        return value++; }
}
```

We can take this idea further and develop a full-fledged library for programming with infinite streams, as others have already done for various languages. The lucid functional data-flow language by Edward Ashcroft and William Wadge is an early example of streams and infinitary programming.3

### Algebraic Datatypes

Returning to our types discussion, algebraic datatypes let us define our own nonrecursive and recursive structures. Formally speaking, an algebraic datatype is a (possibly recursive) sum type of product types; sum and product types are formalizations of union and record/tuple types, respectively (see http://blog.lab49.com/archives/3011 for more details). As a simple nonrecursive example, we define a datatype representing the union of integers and strings as

```haskell
data IntOrString = AnInt Int | AString String
```

We then define a function that adds all the integers and concatenates all the strings it finds in a list of `IntOrStrings`. Functions on data-types usually employ pattern matching to perform the case distinction among the possible variants and to bind substructures to variables:

```haskell
add [] = (0,"")
add (AnInt i : xs) =
    let (k, t) = add xs in (i + k, t)
add (AString s : xs) =
    let (k, t) = add xs in (k, s ++ t)
```

This list is homogeneous in the sense that all of its elements are of type `IntOrString`, but heterogeneous in the sense that `IntOrString` is a discriminated union type.

Things get more exciting and useful when we introduce recursion; here’s a very simple yet general tree type:

```haskell
data Tree a = Empty
           | Node a [Tree a]
```

We can now define some typical tree functions. The pattern underscore is a pseudo-variable for substructures that we don’t need to reference on the right-hand side. The type declarations are optional, but often helpful for documenting intent and usage: we’ll get an error if our implementation doesn’t match our stated intent:

```haskell
size Empty = 0
size (Node _ ts) = foldl (+)
    1 (map size ts)
```

However, we can’t yet print values of these types:

```haskell
Prelude> t1
No instance for (Show (Tree Integer))
```

To solve this problem, we use Haskell’s systematic overloading mechanism to define the missing `show` function used by the interpreter’s main loop to print values:

```haskell
instance (Show a) => Show (Tree a) where
    show Empty = "Empty"
    show (Node x ts) =
        "Node " ++ (show x) ++
        " " ++ (show ts)
```

This instance definition means that we can print a type a tree as long as we already know how to print a. Haskell already knows how to print its predefined basic types, such as `Integer`, so we can print our tree exactly the way we originally coded it:

```haskell
Prelude> t1
Node 1 [Node 2 [Node 4 [], Node 3 [Node 5 []]]]
```

We can now define some typical tree functions. The pattern underscore is a pseudo-variable for substructures that we don’t need to reference on the right-hand side. The type declarations are optional, but often helpful for documenting intent and usage: we’ll get an error if our implementation doesn’t match our stated intent:
rootValue :: Tree a -> a
rootValue (Node x _) = x

traverse :: Tree a -> [Tree a]
traverse Empty = []
traverse (t & (Node x ts)) =
    t : concat (map traverse ts)

In these functions, map applies a
function to each element in a list (in
this case, it recursively applies size to
the children, ts), while concat flat-
tens a list of lists to a simple list. Tree
traversal converts a (nonlinear) tree to
a linear sequence of references to all
of the tree's subtrees. As a result, we
no longer need a dedicated size func-
tion because a tree's size is simply the
length of its linearization:

Prelude> rootValue t1
5
Prelude> size t1
5
Prelude> traverse t1
[Node 1 [...],Node 2 [...],Node 3 [...],Node 4 [...],Node 5 [...]]

A limitation of our traverse
function is that the traversal order is
hard-coded: the function performs a
depth-first tree traversal, descend-
ing as far down the leftmost path as
possible before visiting the rest of the
tree. This is typical for recursive im-
plementations of tree traversal, where
the implicit function-call stack serves
as a last-in-first-out work queue that
stores the children yet to be visited.

We can make this function
more flexible by parameterizing it
differently. In this new version, the
first argument is the list represent-
ing the work queue, and the second
argument is a function for adding
another list of elements to the queue;
this function determines the order in
which we traverse the tree's subtrees:

traverseUsingList [] _ = []
traverseUsingList (Empty : rest) addAll =
    traverseUsingList rest addAll
traverseUsingList
    ((t & (Node x ts)) : rest)
    addAll =
    t : traverseUsingList
        (addAll rest ts) addAll

We now have a functional version
of the usual tree traversal algorithm.
Initially, the only subtree in the work
queue is the tree itself. We then re-
peatedly take the first subtree from the
work queue, add it to the result-
ing linear structure, and add the cur-
rent subtree's children to the work
queue. Given that our trees are finite,
this process continues until the work
queue is empty. We can now invoke
this function using two different strat-
egies to add items to the work queue:

Prelude> map rootValue
    (traverseUsingList [t1] (++)
    [1,2,3,4,3,5])

In the first case, addAll is list concat-
ation with the order of arguments
flipped, so that the children get added
to the queue's beginning, and we have
our last-in-first-out discipline back, re-
sulting in a depth-first tree traversal.

While we pass a queue representa-
tion and an associated operation openly
to the traverseUsingList function,
we could instead define a proper ab-
stract datatype for queues in two ways:
through a tuple of one or more functions
that share and operate on a hidden data
representation, or through a nonstan-
dard extension that provides existential
quantification of type variables (in con-
trast to universal quantification).

In this second installment on func-
tional programming, we’ve expand-
ed on the HOT languages notion and
presented a case that having at least
a basic knowledge of such languages
can be part of a healthy programming
diet. Even if you’re not intent on us-
ning functional programming anytime
soon, the approach can help you get
better results in any language, be-
cause many functional programming
ideas are used to implement optimiz-
ing compilers. Also, many languages
in one form or another (such as C#)
are introducing the ideas, and Micro-
soft recently introduced a completely
functional programming language,
F#, into its languages suite. Given that
Microsoft’s in the business of selling
languages and tools, this is a big deal.
We’re equally convinced that clearly
understanding typing systems can be
helpful when writing programs and
avoiding the pitfalls entailed when the
decision is entirely left up to runtime.
Our intention, however, is not to dis-
miss languages lacking well-founded
type systems outright. We all use a vari-
ety of languages in our work (Java, C#,
Haskell 101

In addition to www.haskell.org, the official Haskell community wiki, we found the following resources quite useful:

- At the 2008 Object-Oriented Programming, Systems, Languages, and Applications (Oopsla) Conference, Mark Dominus, a leading Perl developer, gave an invited talk on the Haskell Type system. You can read his notes (and find a link to his slides) at http://blog.plover.com/talk/ataypical-typing.html.
- The Learn You a Haskell for Great Good site offers a gentle Haskell tutorial at http://learnyouahaskell.com/chapters.

C/C++, Python) for reasons sometimes beyond our control or for value beyond the language (such as the ecosystem).

Haskell itself has long been an excellent choice for general software development, though its use in scientific and high-performance computing has been hindered by a lack of library support and absence of efficient, purely functional versions of common scientific algorithms. Nevertheless, Haskell has a large, vibrant, diverse community that has created an ecosystem ranging from a distributed version-control system (DARCS) to the HAppS Web framework to the Haskore computer music system. The Glasgow Haskell Compiler is very mature and produces highly optimized, fast-running code (it also supports separate compilation through Haskell’s relatively simple module system). The Haskell folks have shown that syntax does matter: Haskell’s intuitive, concise, and adaptable syntax should make mathematically inclined programmers feel right at home and is particularly effective for creating domain-specific languages. If you’d like to learn more, the links in the “Haskell 101” sidebar will bring you up to speed.

Acknowledgments
Thanks to Konrad Hinsen for his excellent comments on various iterations of this article and to Sergio Fanchiotti for his helpful suggestions on useful Haskell resources.

References

Konstantin Läufer is a professor of computer science at Loyola University Chicago. His research interests include programming languages, software architecture and frameworks, distributed systems, mobile and embedded computing, human-computer interaction, and educational technology. Läufer has a PhD in computer science from the Courant Institute at New York University. Contact him via www.cs.luc.edu/laufer.

George K. Thiruvathukal is an associate professor of computer science at Loyola University Chicago and is now an associate editor in chief of this magazine. His technical interests include parallel/distributed systems, programming language design/implementation, and computer science across the disciplines. Thiruvathukal has a PhD in computer science from the Illinois Institute of Technology. Contact him via http://gkt.etl.luc.edu.