Points of View

a tribute to Alan Kay

Edited by
Ian Piumarta &
Kimberly Rose

Designed and assembled by the editors in Los Angeles, California.

Published by Viewpoints Research Institute, Inc., Glendale, California.
http://www.vpri.org

Printed and bound by Typecraft Wood & Jones, Inc., Pasadena, California.
http://www.typecraft.com

ISBN 978-0-9743131-1-5

11 10 9 8 7 6 5 4 3 2 1

Butler Lampson

Declarative Programming: The Light at the End of the Tunnel

Ah, but a man's reach should exceed his grasp,
Or what's a heaven for?
— Robert Browning, *Andrea del Sarto*

Goals

I started out to write about declarative programming, which seemed like a good topic because in a way it is the opposite of the kind of programming that Alan Kay introduced in Smalltalk and has been working on ever since, and also because Alan Borning's ThingLab [53], one of the first examples of general-purpose declarative programming, was developed in Alan's group. As I thought about it, though, I realized that I don't really know what declarative programming is. In fact, it seems to be an umbrella term for "the kind of programming we wish we had."

What kind of programming do we wish we had? We want to be able to tell the computer what to do, in a way that is easy for us and that reliably and promptly gets us the result we want (or an intelligible explanation of why we can't have it, and how we might change the program to get it).

The problem, of course, is that what the computer natively knows how to do is very far removed from this ideal. It *knows* how to perform very small, very precisely-defined state changes on a state space whose main component is an array of a few billion eight-bit numbers. We *want* it to look through a few attached cameras and drive a car through New York city traffic, or find integers x, y, z and $n > 2$ such that $x^n + y^n = z^n$. This is a gap too great to be bridged in any way we can currently imagine, so we must lower our aspirations.

There are two things we know how to do that make the gap smaller. One is to make the machine operate on more interesting datatypes than bytes—for example, on arrays of floating point numbers, on relations, or on images—and to do big operations on these datatypes, such as finding the eigenvalues of a matrix, or a list of all the registered voters in electoral precincts that went Republican in the last presidential election but are in cities that went Democratic, or the faces of women in a picture. The other is to make the machine optimize some function subject to a set of constraints, perhaps approximately. The challenge is to use these two methods (and anything else we can think of) to come closer to our goal.

The most common banners under which people have tried to do this carry the labels *domain-specific languages* (DSLs) and *declarative programming*. The first is fairly easy to understand. The idea is to restrict the scope of programming enough that the machine can do a good job, albeit within narrow boundaries. Parser generators and MATLAB are examples at the two poles of this approach. A parser generator meets our ideal perfectly, as long as the only thing that can vary in what we ask for is the language to be parsed. MATLAB is very good at handling problems that can be solved with standard operations on vectors and matrices; if you want to do something non-standard, it pushes you closer to programming in FORTRAN. The most common fate of a DSL is to be absorbed into a general-purpose programming language as a library, perhaps with a little additional syntax as in Linq;[1] this happens because when the DSL is successful people try to stretch its boundaries, adding more and more

[1] `http://msdn.microsoft.com/netframework/future/linq`

general-purpose facilities, and you don't have to go very far down this path before you have a clumsy general-purpose language that is hard to understand and hard to support.

> *By relieving the brain of all unnecessary work,*
> *a good notation sets it free to concentrate on more advanced problems,*
> *and in effect increases the mental power of the race.*
> — Alfred North Whitehead, *An Introduction to Mathematics*

Declarative programming

Declarative programming is more puzzling, and it is the main topic of this paper. The fact that no one knows what it is gives me free rein to reflect on a wide range of ideas and techniques.

Two somewhat unrelated goals seem to motivate the idea of declarative programming:

1. Make it easier to get from a precise specification of what the program is supposed to do to a working program. Two examples of this are SQL queries and parser generators.

2. Make it easier for a non-programmer to get from a fuzzy idea of what they want to a working program. Two examples of this are spreadsheets and search folders.

This paper is mostly about the first goal, though it has some things to say about the second.

It's easier to say what declarative programming is not than to say what it is. Certainly programming in the native instruction set of the computer is not declarative programming, and neither is programming in C, Visual Basic, or Smalltalk. In fact, any program that explicitly gives the computer a long sequence of small steps to carry out is not declarative; this means that a program with loops or recursion is not declarative. One consequence is that there's not

much hope for using declarative programming all the way down to the bare machine, as one can do in Smalltalk: it's not turtles all the way down.

At the opposite extreme, "do what I mean" is not declarative programming either. In other words, a declarative program is not magic, and it doesn't make wild guesses about the user's intent. It is just as precise as any other program.

It is common to classify programs as imperative (with object-oriented as an important case) or declarative (with functional and logic as important cases). In practice, however, these categories are not strict. Imperative programs often have large parts that are functional, and functional programs in systems like MapReduce and Dryad usually have computational kernels that are written imperatively, though their external behavior must be functional.

Successful declarative systems usually have a few things in common:

1. They give you a way to write the program that is a **good match** to the user's view of the problem. Another way of saying this is that the system synthesizes a program that the computer can run efficiently from a specification that the user writes, which may have a very different form. The purest version of this is the planning that has been part of robotic systems for many years [63]. It used to be called *automatic programming*, but that term has fallen out of favor. An important aspect of a good match is that the user can employ a familiar vocabulary. Thus declarative systems often involve a DSL, or a database schema that has been worked out by someone else. Another important aspect is that you can debug the program at the same level that you write it; macro recorders generally fail this test.

2. They are **compositional**, which means that you can write a program in small pieces that are fairly independent, and the system will put them together automatically. A spreadsheet is a simple example of this, and a solver for an optimization problem with constraints such as linear programming is a more sophisticated example. Functional programming is the most basic composition mechanism.

3. They give you **big primitives**, so that you can get a lot of work done without having to write a lot of code, and your program only needs to have a few steps. A primitive can be big by operating on *big data* (arrays, graphs, relations), by *solving* a nontrivial system of equations or constraints (such as linear programming or Boolean satisfiability [57]), or by embodying a *powerful algorithm* (such as scale-invariant feature transforms in computer vision [60]) or a *powerful data structure* (such as a balanced tree for storing ordered data).

4. They have clean **escape hatches**, so that you can fall back to boring old imperative programming when efficiency, familiarity, or legacy code demands that. An escape hatch may be internal, allowing the declarative program to invoke a primitive written in some other way, or external, allowing an imperative program such as a shell script to invoke a declarative program.

Another characteristic of most declarative systems is that you can get started (do the equivalent of "Hello world") with very little effort, though certainly other systems like Python have this property too.

Don't ask what it means, but rather how it is used.

— Ludwig Wittgenstein, unknown source

Examples

Another way to characterize declarative programming is to look at some examples of successful declarative systems:

Spreadsheets such as Excel. A spreadsheet is functional programming with a human face, without recursion, and with powerful primitives for tabular layout, for charts and graphs, and for aggregating data (pivot tables). Excel has a rather clumsy escape hatch to Visual Basic. Hundreds of millions of people

have learned how to make a spreadsheet do useful work, though only a few can use more than a small fraction of its capabilities.

SQL queries. This is functional programming with big arguments (relations), powerful primitives (for aggregation), and good optimization. It has also been enormously successful, though it's a tool for professionals—the general user needs a front end to generate SQL, such as a form to fill in, and these front ends only expose a small fraction of SQL's power.

Parser generators such as `yacc` are a successful example at the opposite pole from these two. They produce a parser for a context-free language from a grammar. Where Excel and SQL share an expression language with ordinary imperative languages, and have escape hatches to general imperative programming, a parser generator is as domain-specific and declarative as possible. It takes a specification that *is* the user's intent (the grammar defining the sentences to be recognized), often produces a parse tree by default, and usually has a very stylized escape hatch that just allows you to write patterns to define what the output tree should be (though some let you attach to each grammar rule some arbitrary code that runs in a context where the results of the parse are accessible).

Streaming data flow systems like DryadLINQ [64] (which grew out of Unix pipes and the AVS graphics system [62]) are an interesting variation on functional programming. They let you write arbitrary kernels that take a set of input streams and produce a set of output streams (some of which might be much smaller if the kernel does aggregation). Then you can compose many such kernels into a dataflow graph that can be deployed automatically over a number of CPU cores or cluster nodes. Dryad (and MapReduce, which is less general) can automatically partition the computation, run it on thousands of compute nodes in a cluster, and recover from detected failures of some of the nodes. Here the main value is that you can easily express complex operations on very large data sets, and the system handles partitioning, scheduling, concurrency and fault tolerance automatically. This kind of composition and scaling is similar to what you get from a transaction processing system. Dryad has escape

hatches both above and below: you can program the kernels any way you like as long as their only communication with the rest of the world is through Dryad's streams, and you can take the result streams and process them with ordinary .NET programs; this works because Dryad's datatypes (collections) are also .NET datatypes.

Mashups are a DSL that exploits two powerful features of HTML and XML: a hierarchical namespace that extends all the way down to the smallest elements (even to single characters if you like) and fairly elaborate two-dimensional layout of text and graphics. When you combine these with the web's ability to fetch information from anywhere in the Internet and the existence of more or less functional web services for search, mapping, financial and demographic information, etc., you can easily produce nice-looking displays that integrate a lot of disparate information. The escape hatch is JavaScript.

Mathematica is a DSL that deals with symbolic mathematical expressions. It gets its power by embodying sizable pieces of mathematics (polynomials, differential equations, linear algebra, etc.) so that it can solve a wide range of equations. In addition, it can evaluate expressions numerically and solve equations numerically if symbolic methods fail, and you can easily turn numerical results into two- and three-dimensional graphics. It incorporates its own general-purpose imperative programming language, so it doesn't need an escape hatch. MATLAB is a similar system that specializes in numerical linear algebra and digital signal processing. Numerical computation is steadily becoming more declarative.[2]

Security policy is not very declarative today; usually you have to specify the access controls for each object individually, which is time-consuming and error-prone. Experimental systems such as Chang's [55] show the possibilities for expressing policies in a way that is very close to the user's intent.

Table 1 summarizes these examples.

[2] `http://www.nag.co.uk/market/trefethen_future.asp`

Example	Themes / composition	Data model	Algorithms / primitives	Escape hatch
Excel	FP	2-D tables	Incremental evaluation	Imperative (Visual Basic)
SQL	FP, scaling	Relations	Queries, aggregation	Imperative (TSQL)
yacc	DSL for context-free languages	Context-free grammar	Context-free parsing	Output patterns
DryadLINQ	FP for streams, scaling	Data streams	Partition, fault tolerance	Arbitrary kernels; embed in .NET
Mashup	Path names	Labeled tree	Graphical layout	Imperative (JavaScript)
Mathematica	DSL, recursive FP	Math expressions	Math	Native imperative
MATLAB	DSL	Matrices	Linear algebra	Native imperative
Web site security	DSL	Relations	SAT solver	None

FP = Functional Programming, without recursion unless otherwise noted.
DSL = Domain-Specific Language. They come with powerful built-in algorithms that operate on the domain.

Table 1: Examples of declarative systems

Time is nature's way of keeping everything from happening at once.

— variously attributed

Failures

Many attempts have been made to do less domain-specific declarative programming. I think it's fair to say that all of these have been failures: they are based on powerful ideas and can do some very impressive toy examples, but so far at least, they all turn out to have limitations that keep them from being widely adopted. The basic problem seems to be that these systems are solving a system of equations or constraints, and it's too hard

- to write down everything needed to avoid undesired solutions,
- to keep the solver's problem from becoming intractable, and
- to make the program modular, which is essential for large problems.

These systems fall into three main classes: constraint programming, logic programming, and algebraic specifications for datatypes.

Constraint programming, as in ThingLab, is very appealing, since it's often easy to obtain constraints directly from a specification, and a constraint solver is a very powerful primitive. A variation is to add a goal function of the variables to be optimized subject to the constraints. The difficulty is that the only general solution method is some kind of search of the solution space, and you have to choose between very stylized constraints for which there's an efficient search algorithm, as in linear programming, and more general constraints for which the only known method is exponential. If the goal function is differentiable then hill climbing sometimes works, but usually there are many local maxima.

Logic programming, as in Prolog, has the same intractability problem, even though the domain is just Boolean values rather than reals, lists, or whatever, and usually this simplification is counterbalanced by wanting to deal with many more variables. There are periodic waves of enthusiasm for Prolog or for

the closely related rule systems that underlay the expert systems of the 1980s, but they don't last.

Algebraic datatypes are rather different, since they are a way of writing a specification, not a program, but their failure illustrates some of the same points very clearly. The idea is that you can specify the behavior of a datatype such as a queue by specifying the primitives (*put* an item on the end, *get* an item from the front, *test* for empty) and a set of axioms that they satisfy, given in the form of equations. This strategy falls foul of the fact that it's amazingly difficult to write down a set of consistent axioms for even a simple datatype that doesn't allow all kinds of undesired behavior.

All of this is not to say that constraint solvers, optimizers and theorem provers are useless. On the contrary, they are very valuable primitives, just not able to bear all the burden of expressing a program. Whether it's a linear equation solver, a polynomial root finder, a linear programming package, a regular expression matcher, a polymorphic type inference system, or a SAT solver, it can be a good tool as long as it works on a closed system whose interactions with the rest of the world are the responsibility of the programmer rather than themselves being governed by automatic search. There are a few examples of solvers that can be extended cleanly, such as SMT theorem provers [56], but they are conspicuous by their rarity and don't eliminate the fundamental intractability.

Make no little plans. They have no magic to stir men's blood.

— Daniel Burnham, as quoted by Charles Moore

Big Primitives

Most of our steadily increasing ability to use computers to solve increasingly large and complex problems is based on the availability of more and more powerful primitives that you can use in a program with confidence that they will deliver what they promise. I have mentioned some of these already, but it's instructive to see a (necessarily partial) catalog:

- Matrix operations
- Linear programming
- Symbolic mathematics
- SAT solvers
- Synthetic graphics, both two- and three-dimensional; games show what is routinely possible
- Image processing; striking examples of what's possible are Street View and Streetside, Photosynth [61], and the World Wide Telescope [58]
- Vision, still much inferior to human vision but able to extract three-dimensional models of buildings from video
- Relational queries
- Full text search over corpora of many terabytes
- Typesetting and layout from HTML (or other forms of text, such as TEX)
- Graph algorithms such as PageRank [54]
- Views on relations, and especially two-way mappings between relations and forms
- Machine learning (a specialized form of program synthesis)

In addition, there are techniques that make it easier to get a computer to do our will and are more broadly applicable than individual primitives, but that are not instances of declarative programming:

- Transactions, which make it easy to do complicated operations atomically, and to abandon them when necessary without having to worry about side effects
- Undo and versions, which make it easy to experiment in more general settings than transactions and to keep track of the evolution of a big project
- Static analysis of programs to infer their properties
- Lazy and speculative execution, powerful general methods for matching the work that a computer does to the needs of the problem

- Indirect references and incremental execution, which make it much easier to adapt a program to a changing environment

And the users exclaimed with a laugh and a taunt:
"It's just what we asked for but not what we want."
— unknown

Non-programmers

A non-programmer is someone who is uncomfortable with precision and abstraction, which seems to cover most people. For the most part they can only tell a computer what to do by pushing buttons. Sometimes one button push does a lot, but if there's not a button (perhaps with a few accompanying form fields) that does what they want, they are reduced to leading the computer by the hand with a sequence of manual button pushes. Except for spreadsheets, we have not been very successful in finding better ways for them to adapt the computer to their needs. Macro recorders help a little, but very often a recorded macro needs to be edited to make it useful, and in every system that I know about this is too hard for a non-programmer. Because most declarative systems depend on precision and abstraction, they are not much help.

I can only speculate on how we might improve this situation: by making it possible for a person to engage in a dialog with the computer, explaining either in natural language or by example what they want the computer to do (change all the references in this paper into PMLA form, or turn on the heat when a person gets up but ignore the dog, or tell me which of my Facebook friends knows fewer than two of the others). The computer's side of the dialog is its expression, in terms meaningful to the person, of what it's supposed to do or of what it doesn't understand. The person then can give corrections or more examples. This is obviously a form of program synthesis, and it's declarative in the sense that it's not a long sequence of small steps. For it to work, the

computer and the user have to share a conceptual model of the problem domain. A small step in this direction is Miller's keyword programming [59].

Conclusion

For forty years people have been working to make programming easier, faster, and more reliable. For non-programmers it's also important for the machine to help the users state their needs precisely. So far the biggest successes have come from domain-specific imperative languages and from providing powerful primitives that you can invoke from imperative languages. Declarative programming seeks to go further, allowing you to state what you want from the program and have the computer synthesize it, or less ambitiously, to explicitly give the machine only a few steps for it to take. This works to some extent, and it works best for specific domains and when you have big primitives.

As machines get better at reasoning, as computers are integrated more deeply into application areas, and as we build bigger primitives, surely declarative programming will get better. Two things that will help are codifying more information in a form the machine can understand, and building primitives in a form that declarative programming can easily use.

Butler Lampson holds a B.A. from Harvard and a Ph.D. from the University of California at Berkeley. He has been a member of the faculty at Berkeley, the Computer Science Laboratory at Xerox PARC, and the Systems Research Center at Digital.

He is a member of the National Academies of Sciences and Engineering, and a Fellow of both the ACM and the American Academy of Arts & Sciences. He has received numerous awards including the Turing Award in 1992 and the NAE's Draper Prize in 2004.

Butler is currently a Technical Fellow at Microsoft and Adjunct Professor of Computer Science and Electrical Engineering at MIT.