

Programming the Scientific Method: Since Programming is Applied Science We should Automate the Scientific Method

Barry Jay
University of Technology, Sydney
cbj@it.uts.edu.au

Abstract

Programming is where computing engages with the world; just as computing is applied logic, programming is applied science. The role of reason is played by logic, or functionality. The role of experience is played by the inputs, which are data structures. Hence programming languages should not try to reduce everything to logic, but devote equal attention to both, as in pattern calculus and **bondi**. In the broader view, artificial intelligence has automated fragments of the scientific method, so the time is ripe to automate the scientific method as a whole, to create artificial science.

Unemployed at Last!

This cry of joy opens the novel *Such is Life* by Joseph Furphy, whose narrator, having just been fired, is free to write his stories. My joy is to have just finished my book, on pattern calculus. As well as introducing some new programming tricks, and even a new programming style, it has the grand aim of rebuilding the foundations of computing, by making pattern calculus the heir to λ -calculus. This may remind you of another collection of tales *Surely You're Joking, Mr Feynman!*.

Well, actually, I'm serious, but understand the scepticism. After all, λ -calculus is a form of applied logic, so to improve on it would seem to require a new logic, which is not on the table. Rather, logic and λ -calculus emphasise functionality over data structures. The latter are either encoded as functions, or kept isolated within databases. Pattern calculus brings functions and data back together by using patterns to describe the data, the function arguments.

Of course, the ability to combine functions and data structures cannot persuade if the focus is on pure reason, since this does not require any inputs or data. However, computers today are awash with worldly data, whose exploitation is being hampered by logical disdain. To speed things along we must recognise that if computing is applied logic then programming is applied science. That being so, we should consider how to automate the scientific method. From this viewpoint, much computing research is addressing this or that piece of the scientific method, performing experiments, building models, drawing inferences, etc. How exciting it would be to put

all of these pieces together, to automate the scientific method as a whole.

This essay began with the modest goal of creating some space for pattern calculus within the foundations of computing, but now we'll have to venture far beyond my area of expertise. So what began as a little fun, a break from routine, is in danger of becoming a burden. To keep things simple, let's stick with well known sources, and skip the formal citations. Also, since I'm not quite sure where this is going, the overview will be postponed to the conclusions.

Programs as Proofs

The strongest assertion linking programming with logic is the Curry-Howard Correspondence, relating programs to proofs. Its high point is probably Howard's connection of intuitionistic natural deduction with λ -calculus in 1969. All would agree that a proof is a function from premises to a conclusion. Indeed, it is a computable function, and so can be encoded in a program. Further, the λ -calculus, being a calculus of functions, is ideal for this purpose. Hence, the interest is in the converse claim, that a program is nothing more than a proof. Superficially, this can be true in the sense that a program demonstrates (proves) that its specification can be met. The deeper question is whether the specification can be expressed in a precise logic. In this approach, the type of the program corresponds to the proposition being proved, and so supports the emphasis upon types in programming. Whatever the future holds, the correspondence has been tremendously fruitful.

Delving a little deeper, the λ -calculus is well known as a foundation for computing. Actually, it was created in 1932 as a tool for investigating logic, to attack Russell's Paradox, which shows that naive reasoning about about sets leads to inconsistencies. It was discovered while developing Logicism, the thesis that mathematics is reducible to logic.

Meditations

From a naive viewpoint, the goal of producing knowledge by pure thought is pretty strange. After all, just as "garbage in" yields "garbage out" it would seem that "nothing in" should yield "nothing out". However, the philosopher Rene Descartes explored this question in his *Meditations on First Philosophy* (1641). Distrusting all he thought he knew, and especially his senses, he decided to start over:

Today, then, since I have opportunely freed my mind from all cares [and am happily disturbed by no passions], and since I am in the secure possession of leisure in a peaceable retirement, I will at length apply myself earnestly and freely to the general overthrow of all my former opinions.

I've heard that his meditations were begun while snowed in, and sitting within a stove. In any event, he was cut off from the world, in retirement.

From this position of extreme scepticism, he realises that he cannot doubt his own existence (someone is doing the doubting, after all) and this becomes the bedrock of his knowledge.

Cogito ergo sum.

or "I think, therefore I am." He goes on to deduce the existence of God and then, since God is good and has no wish to deceive, begins to trust his own senses.

Going back even further, the ancient Greek mathematicians thought they were doing more than pushing sand around. Pythagoras asserted that

All is number.

Plato believed that there was a world of ideal forms, including the ideal Circle, Square, Goodness, etc. of which the worldly representative were but shadows. Viewed this way, the emphasis on logic in computation emerged from a long tradition that explores the power of pure reason.

The Tabula Rasa

On the other hand, an equally long tradition, of Empiricism, has stressed the importance of worldly experience, of the senses, in acquiring knowledge. Aristotle described a newborn mind as an "uninscribed slate", also described by Avicenna as a "tabula rasa" or blank slate. In modern times, the blank slate has been used to promote the importance of culture in the nature/nurture debate. Of course, such a view requires that the newborn, unlike a slate, has the generic capacity to learn.

Scientific Method

The scientific method emerged from the Middle Ages. In his *Opus Majus* of 1267, Roger Bacon declared that

Mathematics is the gate and key of the sciences.

and William of Occam asserted in 14th Century

For nothing ought to be posited without a reason given, unless it is self-evident (literally, known through itself) or known by experience or proved by the authority of Sacred Scripture.

This synthesis of experience and reason developed into a powerful means of acquiring knowledge. The issues were also taken up by Immanuel Kant, in his *Critique of Pure Reason* (1781) who observed that the way in which we observe the world determines what we can know of it.

Turing Machines

The scientific approach can be mechanised within Turing machines. At the beginning of his seminal paper on computing machines, Alan Turing writes:

We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions q_1, q_2, \dots, q_R which will be called m -configurations. The machine is supplied with a tape, ...

This masterfully disposes of several issues at a stroke. The scale is human, so no need to consider tapes moving close to the speed of light, or quantum entanglement in the data. Also, the computer works alone, without interacting with others. The computing system is neatly divided into a finite state that contains rules (or actions or functions) and a tape that holds all the data. We may view the finite state as supporting the logic of the system, of being its rational

part, or mind, while the tape represents the observable world, or matter.

The Blank Tape

In 1936, Alonzo Church and Turing proved that λ -calculus and Turing machines support the same notion of computation. Philosophically, this seems quite strange, since λ -calculus mechanises pure thought, while Turing machines mechanise science. Does this imply that observation cannot supplement reason? To untangle this requires a closer look at the argument.

The theorem relating Turing machines and λ -calculus is quite specific. λ -calculus and Turing machines can compute the same real numbers, given no external inputs. That is, a Turing machine whose tape is initially blank mechanises the Logicist outlook, to become a mechanical meditator, without saying anything about the role of observation.

This assumption is acceptable if the initial data on the tape is finite, or produced by some other Turing machine, as when solving a finite system of equations. However, programming today is vastly more engaged with the world, being hooked up to all manner of inputs such as sensor networks, closed circuit television, weather stations and stock markets, not to mention data repositories, bio-informatics and the vast resources of the World Wide Web.

To assume that all of this information is Turing computable is to assert that the whole world is a Turing machine, as if the universe was once empty, and that the Big Bang and all the subsequent developments were of no real consequence, adding nothing in the way of information. Although logically possible, it seems unlikely. Certainly, none of this figures in Turing's initial scene with its all-too-human computer.

Universal Turing Machines

On the other hand, Turing machines can be used to support the empiricist position. A Universal Turing Machine can simulate the behaviour of any other Turing machine by encoding the latter as a program on the tape. Now, the blank tape is a tabula rasa to be inscribed by the world, and experienced by the machine.

There are two weaknesses in this argument for empiricism. First, as Kant might observe, our reasoning and sensing apparatus is not so universal. Second, and more to the point, the correct operation of a universal Turing machine presupposes an agreed representation of the program and its inputs on the initial tape. However, the world is not laid out for our convenience. For example, the incomprehensible multitude of data formats worldwide is a significant barrier to the development of web services.

Thus, while Turing machines can mechanise the acquisition of knowledge by either rational means or by experience, it is not biased one way or the other. Rather, it provides a laboratory in which to explore the options.

Polymorphism

If the tape of a Turing machine represents the world that is to be explored then, according to Occam, a Turing machine which makes few assumptions about the tape is superior to one that makes many. In other words, one program is more *polymorphic* than another if it executes correctly in more environments.

One approach, inspired by the Halting Problem, is to identify success with termination. A good computation is one that terminates. A better computation terminates sooner, while producing the same results. This the central concern of *complexity theory* of algorithms.

An alternative that is more appropriate for programming languages is to identify success with being meaningful. A program carries expectations of the environment, and guarantees certain outcomes (a form of contract, if you wish). A better program requires

fewer pre-conditions and ensures more post-conditions. One way of viewing the history of programming is as a quest for better, or alternative, approaches to polymorphism.

Programming Styles

Is programming an art or a science? We're in the habit of describing ourselves as computer scientists but, to speak precisely, this is not yet right. Of course, many computer scientists do conduct experiments using benchmarks and the like, but mostly we apply logic. For example, the book *The Science of Programming* (1981) by David Gries is devoted to programs as proofs. In this sense, programming is an art, like mathematics, as described in Donald Knuth's book *The Art of Programming* (1968).

Of course, "art" has a range of meanings, of which the most common is a form of aesthetic, or personal expression, through painting, sculpture, performance, etc. So, to describe programming as an art risks having it seen as a form of narcissism rather than as an attempt to engage with the world in a scientific manner.

Nevertheless, when it comes to programming style, it's often said that all programming languages are essentially the same (they are all Turing complete) so that the choice of programming style is merely a form of self expression, subject to the caprice of fashion. Of course, domain-specific languages may provide syntactic convenience but cannot contribute anything fundamental.

Summarising this viewpoint, computing is a science, i.e. a form of applied logic, while programming is an art, a form of self-expression.

In opposition to this, I suggest that programming is where computation engages with the world. Further, programming style is characterised by the nature of its polymorphism, by the range of environments in which can operate successfully.

Types

Program meaning can be characterised in a number of ways. If the pre- and post-conditions are expressed in a logic such as first-order predicate calculus, then the familiar Hoare triples emerge. A simpler approach, and more popular, is to use types as a proxy for these conditions.

For example, a program r that takes inputs of type S and produces outputs of type T has a function type $S \rightarrow T$ which may be written $r : S \rightarrow T$. Again, given programs $u : U$ and $v : V$ then their pair (u, v) has product type $U * V$. Now one typed program is more polymorphic than another if its type is more general, i.e. more polymorphic.

There are two main approaches to comparing types. One is to use type variables X and Y that can be instantiated by type parameters. For example, the polymorphic identity function has type $\forall X. X \rightarrow X$. When X is instantiated to a type T then this yields the identity function on T of type $T \rightarrow T$. The other comparison is to introduce a subtyping relation $S < T$ so that any program of type S can be considered to be of type T as well. For example, that coloured points of type CPoint are also points of type Point can be expressed by the subtyping $\text{CPoint} < \text{Point}$.

The use of a type system reduces the task of exploring the computational environment to that of type checking or type inference. The application of a program $r : R$ to an input $u : U$ is well-typed if R is more polymorphic than a type of the form $U \rightarrow S$. When type polymorphism uses type variables then this amounts to solving type equations. When type polymorphism uses subtyping then this amounts to checking (or solving) type inequalities.

Functions

These twin challenges, of establishing correctness and increasing polymorphism, have been central to the development of programming languages. Interestingly, the division between program and

inputs, between functions and data structures, that first arose with universal Turing machines, has continued throughout, so that it is convenient to consider them separately.

The treatment of functions has been a great success. Function types give an accurate and uniform account of functions and their applications. They are used to type the abstractions of λ -calculus, and also to type the routines and procedures of imperative languages. In particular, a command requiring a sub-routine that takes an integer argument can be represented by a higher-order function of type $(\text{Int} \rightarrow \text{command}) \rightarrow \text{command}$.

The Next 700 Programming Languages

This success led Peter Landin to propose that λ -calculus, with its approach to functions and scope, should be the basis of future language development, using his SECD machine as the basis for implementation. It avoided the jumps and goto's in the imperative language FORTRAN, and improved upon the functional language Lisp, whose scoping rules are more dynamic than those of λ -calculus. Given that there were already 700 programming languages in 1966, his conceit was to sketch the development of the next 700.

This worldview gained further support from the Curry-Howard Correspondence, since functions are so easily aligned with proofs, their arguments corresponding to premises and their results corresponding to conclusions.

Data Structures

Although Landin's approach gives a unified account of functionality, problems with new data representations remained. For example, the success in unifying procedures and λ -abstractions as functions was not replicated by a unification of arrays and lists. Arrays rely upon physical contiguity to move from one entry to another, much as one moves from one square of the tape to the next, while lists rely on more abstract relationships, which must be implemented using some addressing mechanism. Since the computing environment is ever more distributed, the focus has been on the more abstract approaches. For the sake of argument, let us put arrays to one side and consider the more abstract approaches.

Lisp supports a very flexible means of building data structures. These are constructed by pairing, and deconstructed by taking components, using the operations `car` and `cdr`. However, there are two difficulties with this approach.

First, the atoms used to build the compounds are not stable in Lisp, in that a symbol may be confused with its value. One way of fixing this is to introduce pointers, as in C, which are clearly distinguished from their values.

The second difficulty is that the operations `car` and `cdr` are not typable in any useful manner. That is, there is no general mechanism for determining the type of a component from that of the compound. For example, a component of a tree may be either a subtree or a leaf value. This ambiguity is reflected in Lisp terminology, where lists are also pairs.

The Next Four Data Representations

To a large extent, the subsequent history of programming languages can be viewed as a series of attempts to work with data structures in a manner that is both correct and polymorphic. Here are four examples.

Records, as in Pascal, provide a more abstract treatment of data structures, in which the access paths themselves (e.g. a sequence of pointers) become more abstract, as fields. Pascal supports strong typing, but not type polymorphism. Records are also the subject of queries in languages such as SQL. Record types lend themselves to subtyping, as measured by inclusion of their sets of fields.

Abstract data types characterise data structures by their interfaces, as described in a type declaration. This ties in very well with the functional approach, since the interface describes the behaviour, which can be described using functions.

Data structures themselves can be represented by functions, using Church encodings. For example, a pair is a function that can evaluate a function requiring two arguments. Remarkably, such encodings can be typed using quantified function types in Jean-Yves Girard's System **F** (1972). For example, the product type $U * V$ can be represented by $\forall Z.(U \rightarrow V \rightarrow Z) \rightarrow Z$.

Objects, as in Simula or C++, provide a mix of records and functionality (methods) in a relatively autonomous package. That all of these approaches (and yet more!) remain in active use today suggests that there is more to be understood.

Atoms and Compounds

All of these competing alternatives arise from the lack of simple characterisation of data structures comparable to that of functions in λ -calculus. The central characteristic of data structures that distinguishes them from functions is that their internal structure may be examined. In particular, every data structure is either divisible into components or is not, is either a compound or an atom. Hence, to define a function that acts on arbitrary structures, it is enough to provide two rules; one for (binary) compounds and one for atoms.

For example, the size of a compound data structures is the sum of the sizes of its components: the size of an atom is one. It is easy to write such a generic function in Lisp, but not in the other approaches above: records require knowledge of the fields; abstract data types require that the size function be provided in the interface; functions cannot be decomposed into components; and each class of objects would require its own size method.

This ability to examine all the components, stopping only at the atoms, is called path polymorphism. In my book it is expressed in the compound calculus, obtained by adding to λ -calculus the Lisp operations for compounds, plus a collection of constructors to play the role of atoms. This separation of the constructors from the variables resolves the ambiguity of the Lisp symbols.

Indeed, the compound calculus is so simple and so close to Lisp that I'm surprised that was not identified around 1960. However, several factors made it hard to recognise that data structures are as fundamental as functions, including: the lack of constructors; the confusion of lists and pairs; and the obscurity of the names `car` and `cdr` compared to that of the operator `lambda`.

And of course, there is the problem of typing. The key issue is that the type of a compound does not determine the types of its components. For example the components of a tree may be other trees or leaf values. Hence, the types of the components must remain local to the computation that uses them. For example, the size of a data structure is an integer that does not reveal any information about the types of the components being sized. This constraint is violated by the primitives `car` and `cdr` but can be formalised when the identification of the components and their use are combined within a single construction, such as a pattern-matching function.

Patterns

Talk of patterns conjures up many different notions. It seems that the word "pattern" has its roots in the weaving trade, where it described the figure, or design, that was to appear in the cloth. From there, it came to denote the instructions given to a Jacquard loom in 19th Century, a step toward automation of the weaving process. In the 20th Century, it became clear that, to a significant extent, our ability to make sense of the world is driven by pattern matching. Further, a significant aspect of learning, or creativity, is the ability to recognise, or create, new patterns.

One of the first efforts to compute with patterns was in the 1960s, through string matching, in which regular expressions describe patterns within strings of characters, or words in languages such as SNOBOL or Perl. Another use is to identify patterns in images, e.g. a barcode or face. More generally, data mining can be characterised as the search for patterns in large data sets. Again, design patterns are informal descriptions of solutions to common programming problems. Doubtless, there are many other uses. Here, the emphasis is on pattern matching as reduction in some calculus.

Algebraic Data Types

The interface to an abstract data type typically consists of some constructors for building data structures, and some destructors (also known as selectors or eliminators) for retrieving information; other operations are built from these primitives. These are analogous to the introduction and elimination rules of sequent calculus in logic.

For example, an abstract data type of lists might have constructors `Nil` and `Cons` for creating lists, and destructors `head` and `tail` for dismantling them, along with equations

$$\begin{aligned} \text{head } (\text{Cons } h \ t) &= h \\ \text{tail } (\text{Cons } h \ t) &= t \end{aligned}$$

which could, of course, be oriented from left to right to become reduction rules.

Then it was noticed that the destructors could all be described uniformly, by pattern matching. For lists, `head` and `tail` can be given by

$$\begin{aligned} \text{head} &= | \text{Cons } h \ t \rightarrow h \\ \text{tail} &= | \text{Cons } h \ t \rightarrow t \end{aligned}$$

The vertical bar `|` begins the case, given by a pattern and body separated by an arrow `->`.

Similarly, if binary trees are constructed using `Leaf` and `Node` then the eliminator for `Leaf` is given by

$$\text{elimLeaf} = | \text{Leaf } y \rightarrow y$$

Of course, omitting the destructors in this way breaks the correspondence with the sequent calculus, which weakens the Curry-Howard Correspondence. The connection to logic is further weakened by the possibility of match failure, as when evaluating `head Nil`. Some attempts were made to link pattern matching to logic, notably the typed pattern calculus of Breazu-Tannen et al, but these were rather remote from programming practice.

These concerns, both theoretical and practical, underpinned research in which pattern matching was an adjunct to the manipulation of algebraic data types, which were themselves incidental to the λ -calculus. Thus, even though they became central to the treatment of data types in typed functional programming languages, they never gained any status within the foundations of computing.

In the main, pattern matching was regarded as no more than a syntactic convenience, a programming style in the sense of a fashion statement, rather than as a source of polymorphism. The research questions, if any, were linked to parametric polymorphism (using type variables to represent the type of the list data), compiler construction, convenient syntax for the user, and efficient implementation. No one appears to have noticed any deeper significance.

bondi

From the viewpoint of atoms and compounds, this approach to pattern matching is rather limited. Why not allow a pattern for an arbitrary compound? These are supported in **bondi** by allowing a binding symbol at the head of a pattern, not just a constructor. For example, the generic size function is given by the recursive function

```
let rec size =
```

```
| x y -> (size x) + (size y)
| _ -> 1 .
```

The pattern `x y` is used to represent an arbitrary compound, in which `x` corresponds to its `car` and `y` to its `cdr`. If the argument is not a compound then the default case uses the wildcard `_` to yield a size of 1.

Pattern Polymorphism

Again, why not allow patterns to be dynamically created? **bonDi** supports this pattern polymorphism by allowing a mix of free and binding symbols in patterns. For example, the generic eliminator is given by

```
let elim x = | {y} x y -> y .
```

The syntax `{y}` indicates that `y` is a binding symbol, and so `x` is a free variable of the pattern. Thus `elim Leaf` reduces to `elimLeaf`. More generally, the tail of a list can be given by `elim (Cons _)` as this reduces to `| Cons _ y -> y`. Here the argument to `elim` is not a constructor but a data structure. More generally, the argument can be a function. For example, the head of a list can be given by `elim (fun z -> Cons z _)` as this reduces to `| Cons y _ -> y`.

Pattern Calculus

Underpinning **bonDi** is the dynamic pattern calculus, a variant of the pure pattern calculus developed with Delia Kesner. Its terms are given by the Backus-Naur Form

$$t ::= x \mid \hat{x} \mid t t \mid [\theta] t \rightarrow t .$$

Each symbol x may appear as either a *variable symbol* x or as a *matchable symbol* \hat{x} . A case $[\theta] p \rightarrow s$ has binding symbols θ , pattern p and body s . Occurrences of a binding symbol x in p are given by the matchable symbol \hat{x} , and in s are given by variable symbol x . For example, the identity function is given by $[x] \hat{x} \rightarrow x$.

There is no need to specify a separate class of term constructors as these are given by matchable symbols that are not bound. Then, since binding symbols and constructors are both inert under reduction, it is convenient to identify them. This is quite different from **bonDi**, whose tight control over evaluation allows free and binding occurrences of symbols have the same syntax x, y etc, distinct from that for constructors such as `Leaf`.

The sole reduction rule is

$$([\theta] p \rightarrow s) u \longrightarrow \{u/[\theta] p\}s$$

where $\{u/[\theta] p\}$ is the result of matching p against u to find values for the binding symbols in θ . For example, the case

$$[x, y] \hat{x} \hat{y} \rightarrow x$$

represents `car` in the compound calculus.

Just as the power of λ -calculus is in the implicit rules for substitution, the additional power of the pattern calculus lies in the implicit rules for matching. These can be made explicit by interpreting pattern calculus in, say, compound calculus or Lisp.

Generic Queries

The usual database queries, of selecting and updating can be expressed as generic functions in **bonDi**, using path polymorphism to traverse the structure, and pattern polymorphism to determine when interesting data has been found, and then acting on it accordingly.

Generic selection can be defined by the program

```
let rec (select:(all a.a->List b) -> c -> List b)=
  fun f ->
    | z y -> append (f (z y))
                (append (select f z) (select f y))
```

```
| y -> f y .
```

When applied to some parametrically polymorphic function f that is able to act on any argument and to a data structure x then `select f x` collects a list of the results of applying f to each component of x . A common scenario is to find all the components of type b in a data structure of arbitrary type c . Then f is itself a pattern-matching function, with a default case of type $c \rightarrow \text{List } b$ that returns the empty list, and a special case of type $b \rightarrow \text{List } b$ that returns a singleton list. A similar approach can be used to support generic updating

```
update : (all a. a -> a) -> b -> b .
```

Note that these queries are fully generic, in that they can be applied to lists or trees of any kind, not just records. Additional cases can be added to handle arrays etc.

Local Type Variables

All of this expressive power is very attractive, but is it reasonable? In particular, can it be typed? The core type system for pattern calculus is a variant of that of System **F**. For example, an update able to act on an arbitrary data structure will have type $\forall X. X \rightarrow X$ where X represents the type of the unknown data structure. This example shows that pattern calculus supports more terms than System **F**, whose only value of the type above is the polymorphic identity function. However, pattern calculus only produces more terms for types that are already inhabited by values in System **F**. That is, pattern calculus does not “prove” any more propositions than System **F**, which makes it hard to distinguish or acknowledge within the Curry-Howard worldview.

Actually, it is an open question as to whether there is a viable logic corresponding to pattern calculus. We have seen that even pattern matching for ADTs is problematic. Intuitively, path polymorphism corresponds to a generic form of structural induction, which has been a meta-principle of logic till now. Perhaps data structures are built using *cuts* (modus ponens) that cannot be eliminated.

Even though the types are familiar, their relationship to the terms is not quite trivial. A key question is how to type cases so that the size function is typable even though `cdr` is not. The solution is to require that the types of generic components appearing in patterns be local to the enclosing case. That is, in a case $[\theta] p \rightarrow s$ the binding symbols θ now contain both binding type symbols as well as binding term symbols. Further, if the case has type $P \rightarrow S$ then the binding type symbols may not be free in either P or S . For example, the size function has type $X \rightarrow \text{Int}$ that does not expose the type of any components while the candidate type $X \rightarrow (Y \rightarrow X)$ for `car` exposes the type Y of the second component.

Typing Special Cases

The collection of generic functions that act uniformly on all data structures is, of course, quite small. However, great expressive power arises by adding some special cases. For example, the generic addition in **bonDi** adds compounds using the case

```
| (x1 x2,y1 y2) -> (plus (x1,y1)) (plus(x2,y2))
```

Since the types of `x1` and `x2` need not agree, the type of `plus` is $a*b \rightarrow a$ or, in System **F** style, $\forall X. \forall Y. X * Y \rightarrow X$. To this can be added a special case for integers, of type $\text{Int} * \text{Int} \rightarrow \text{Int}$. In standard accounts of pattern matching, all the cases refer to a single data type, and so all cases must have the same type. Here, special cases may have special types. This is type-safe since if $a*b$ is $\text{Int} * \text{Int}$ then the actual result type Int is also the expected result type a . More generally, a special case of type $P \rightarrow S$ can be added to a default of type $Q \rightarrow T$ if any solution of $P = Q$ also solves $S = T$. Similar remarks apply to floating point numbers,

etc. Then, for example, `plus((1,2.2),(3,4.4))` evaluates to `(4,6.6)`.

Subtyping and Object Orientation

Functional accounts of subtyping do not describe object-oriented programming languages very well. Complexity grows with expressive power, at the cost of intelligibility. For example, most theorists argue that subtyping of function argument types should be contravariant (so that $P \rightarrow S < Q \rightarrow T$ implies that $Q < P$) but Java makes them invariant (so that $P = Q$). Again, if one combines subtyping with type parameters then type equations $S = T$ become type inequalities $S < T$ which are much harder, if not impossible, to solve. Even the very nature of an object is obscure. In theory they are self-contained but in practice they usually belong to a class that manages their methods.

In a sense, all these difficulties are artificial, being caused by the drive to represent everything as functions, e.g. as collections of methods. Once the correct relationship between functions and data structures is established there is no need for a separate notion of objects. These can be treated as data structures, with methods given by pattern-matching functions. Now the slogan that special cases have special types takes on new meaning, influenced by subtyping. A special case of type $P \rightarrow S$ can be added to a default (method) of type $Q \rightarrow T$ if any solution of $P \sim Q$ (in which P and Q are related by subtyping) also solves $S < T$. Inheritance is achieved by making the pattern for an object in a subclass be a refinement of the pattern for an object in the superclass.

Now consider contravariance. if data types are encoded as function types then the latter must be contravariant in their argument types. However, this encoding is actually harmful, just as it is for terms. By supporting a separate class of data types, the need for contravariance is eliminated, so that invariant argument types are expressive enough, and much simpler to handle.

Finally, most accounts of subtyping allow a single inequality to reduce to a pair of inequalities, so that general solutions do not always exist. For example, if function types are contravariant then the inequality $X \rightarrow X < S \rightarrow T$ reduces to $S < X < T$ where it is not clear if X should be mapped to S or to T or to some intermediate type. However, when function argument types are invariant then the original inequality above reduces to $S = X < T$ which is solved by mapping X to S and then solving $S < T$. More generally, subtyping can be defined so that the supertypes of any type form a finite linear order, just like the class hierarchy in Java. Then type inequalities cannot increase in number during their solution, whose algorithm becomes a variation of that for type unification.

The usual object-oriented concepts of object, class, self, invocation, etc. can all be encoded within **bondi**. Objects are represented by stateful data structures, methods become pattern-matching functions, whose cases are related by both type unification and subtyping. Thus, one may define a parametrised class of linked links by

```
class Node <a> {
  value : a ;
  next : Node <a>;
  getValue = { |() -> !this.value }
  setValue = { fun v -> this.value = v }
  getNext = { |() -> !this.next }
  setNext = { fun (n: Node <a>) -> this.next = n }
  insert = { fun (n: Node<a>) ->
    n.setNext (this.getNext());
    this.setNext n }
}
```

and then a subclass of doubly-linked lists by

```
class DNode <a> extends Node {
```

```
  previous : ref (DNode <a>);
  getPrev = { |() -> !(this.previous) }
  setPrev = { fun (p: DNode <a>) ->
    this.previous = Ref p }
  setNext = {
    let (aux: Node<a> -> Unit) =
      | (n:DNode<a>) ->
        super.setNext n;
        n.setPrev this
    | n -> super.setNext n
    in fun (x: Node<a>) -> aux x
  }
}
```

Note how type inference is used to infer the types of most symbols.

The Expression Problem

Past difficulties in combining object orientation with functional programming led to speculation that there is a fundamental tension between them. One difficulty is the tension between referential transparency and dynamic dispatch. The former is the requirement that the meaning of a symbol be determined by its declaration, while dynamic dispatch delays the binding until the point of use. The former emphasises correctness, while the latter emphasises polymorphism. **bondi** supports both programming styles. That is, by working solely within the functional fragment of the language one can ensure referential transparency. Similarly, **bondi** supports object-oriented classes with dynamic dispatch.

In the Landin worldview, the types can be the only source of such difficulties. In turn, these are introduced by declarations of ADTs and classes, so perhaps these two are incompatible.

Andrew Appel's *Expression Problem* was to allow arbitrary interleavings of declarations of abstract data types and of classes. The difficulty is that the former fix the data structures, but allow new functions to be defined upon them at will, while the latter fix the functionality (the methods) but allow new data structures to be introduced later (through subclassing).

Behavioural Continuity

From the viewpoint of pattern calculus, however, the various declarations are not fundamental, but merely a means of introducing new symbols to the standard framework. In particular, pattern-matching functions suggest a new approach to combining correctness and expressivity. The issues are exemplified by considering the arithmetic of a newly defined type of complex numbers.

The standard approach introduces these by a declaration of the form

```
datatype Complex = Complex of Float and Float
```

then `plus (Complex 1.1 2.2, Complex 3.3 4.4)` evaluates to `Complex 4.4 6.6` using the given generic addition. That is, the complex numbers inherit the generic addition, as desired.

However, the inherited multiplication has the wrong semantics, as multiplication of complex numbers is driven by the understanding that `Complex x y` is $x + iy$ where i is the square root of minus one.

Here, the traditional functional programming style is stuck. By contrast, in an object-oriented approach, one would like to define a class of complex numbers as a subclass of some class of numbers, and then specialise the method for multiplication, but there may be problems handling this binary method. Also, this commits complex numbers to a particular place in the class hierarchy, so that they cannot be arguments of global functions, or inherit from other classes.

In **bondi**, this is solved by specialising existing functions, such as multiplication for new data types. For example, the declaration

```
datatype Complex = Complex of Float and Float
with times += | (Complex x1 y1, Complex x2 y2) ->
  Complex (x1*x2 - y1*y2) (x1*y2 + x2*y1) .
```

Within **bondi**, the pattern-matching function for `times` has been changed by the introduction of the new case above. This will slow the multiplication of existing data structures, but cannot change the results, since existing data structures cannot involve the new constructor `Complex`. The resulting behavioural continuity combines most of the reasoning associated with referential transparency with most of the expressive power associated with dynamic dispatch. Similar techniques can be applied to pretty printing, and perhaps to parsing.

A Universal Programming Language?

Given that **bondi** is able to support most, if not all, of the main programming styles, perhaps it can serve as a universal typed programming language.

bondi demonstrates how imperative, functional, query-based and object-oriented styles can all be supported within a simple, small language. Support for other styles, such as logic programming, has yet to be confirmed. Assuming there are no surprises, such a language may become the common core for all the main programming styles. Applications written in such a style would communicate freely, without the need for cumbersome middleware that forces communication to the lowest common denominator. Further, since there is a common type system, able to support both type parameters and subtyping, it may be possible to support typed inter-process communication.

Of course, this would never diminish the need for domain-specific languages, or of syntax that allows free expression to the personal style of the programmer. Rather, it is a matter of eliminating the technical obstacles that have isolated the various approaches to polymorphism.

Artificial Intelligence

This digression into polymorphism and programming styles shows their importance as tools, but now let us return to the acquisition of scientific knowledge. Our earlier observation that Turing machines combine experience and reason (or action) is to say that they are agents in the current sense of Artificial Intelligence (AI). For example, in the preface to *Artificial Intelligence: A Modern Approach*, Stuart Russell and Peter Norvig write:

We define AI as the study of agents that receive percepts from the environment and perform actions.

Unfortunately, AI research has not fully explored this domain. The main emphasis is placed on modes of reasoning that are less algorithmic, less reliant on branching and looping, such as proof-search, neural networks and genetic algorithms. By contrast, the environment is rather restricted in areas such as constraint satisfaction, search and planning. A little freer is the study of natural language; the inputs are still constrained by fixed rules, it's just that the rules are obscure. Polymorphism plays a small role here.

Chaotic Environments

Many of the most troublesome computing tasks arise from the failure of pure reason, so that humans are forced to fiddle with the parameters before things work as they should. That is, humans still do the science. Here are some examples.

Despite our best attempts to define interfaces precisely, mismatches are fairly common when integrating software modules or components. Integrators conduct experiments, build mental models of the issues and use these to guide repairs, whose correctness is checked by further experiments. Once integration is accomplished,

it may happen that some components need to be upgraded to a new version, so that integration must be repeated. Similar remarks apply to configuration management for printers, email clients, servers, etc.

When allocating computing resources, on a chip, in a parallel computer, or a distributed system, strenuous efforts have been devoted to developing the algorithms, or reasoning tools for determining the allocation. However, each new application requires some tuning to maximise performance. This may be done by hand, or by automated profiling that makes dynamic adjustments to loads, etc.

Semi-structured data has a clearly specified structure at the lower levels, but some uncertainty at the upper levels. The challenge is to navigate through to the low-level data of interest. Data formats such as XML provide some help with structuring the data, but the programming model is not stable.

Even more chaotic is the world-wide web. When attempting to find a web service, the best-case scenario, is that the desired service and the actual service are described in the same way, using types or schemas. More likely is that the services are in an unfamiliar format, or not given at all, so that the challenges are similar to, or worse than, those of integrating known components.

Data mining is like reverse engineering except that there is no guarantee that there is any pattern to be found in, say, stock indices or crime statistics, or the distribution of galaxies. Once patterns (rules) have been found in the data then humans are required to make sense of them.

Artificial Science

Of course, polymorphic programs will play a vital role in simplifying such problems. For example, path polymorphism can dramatically simplify access to semi-structured data. Again, pattern polymorphism supports a novel approach to problem decomposition, in that it is easier to determine missing pattern parameters than to determine the behaviour of some object.

Nevertheless, the challenge of discovering unknown pattern parameters is still significant. Data mining will be essential to discovering such patterns, but the results will be imprecise and produce unsatisfactory results. What is required is a process that can be iterated, to improve first approximations. The natural candidate for this is the scientific method.

In brief, the scientific method is a cyclic activity, of observation, modelling, inference, prediction, experiment. Experimental results provide evidence for or against the model that led to the prediction, and also the new observations from which more refined models can be built. Each of these steps has been studied in isolation, and most have been automated, at least in part.

It is now routine for experimental data to be freely available over the Internet, be it from physical particle detectors, or DNA analysis, meteorology, or maps of the sky or the sea bed. Other observations come from stock market indices, newsfeeds, blogs, twits, etc. We are awash with on-line data.

The basis of modelling is data mining, which detects patterns in the data. Exploiting these patterns to develop theories is the goal of knowledge representation. Simplest is to discover numerical relationships between known parameters, as in Galileo's account of pendulum's (just now automated at Cornell University). Rather more abstract, and harder to formulate, would be situations where the nature of the parameters is not yet known, such as the concepts of mass and force developed within Newton's Laws.

Once the model is developed then inferring consequences of the laws will draw on existing systems for manipulating equations by rewriting, and for proving theorems.

Running experiments is similar to validating the results of data mining, except that the patterns being validated may not be the same as those originally discovered.

Of course, science is a collaborative activity, involving many scientists, so that some concurrency theory of communicating agents will be required to organise the community.

A Grand Challenge

Developing artificial science is surely a grand challenge. Even though most of the machinery exists, it will be pushed to support the goals of science, and integration to support iteration of the scientific method will be a major challenge. Although this may seem daunting, observe that scientists collaborate, whereas traders compete. Hence, in principal, it should be easier to build a network of collaborating scientists than to build a network of competing traders. E-science is easier than e-business.

The goal above is rather abstract compared to other grand challenges such as putting a man on the moon, curing cancer (failed), decoding the human genome, developing cheap fusion power (failed) or building a dome for the Baptistry in Florence. It would help to have a particular goal of artificial science that captures the popular imagination, has clearly defined criteria for success, and is achievable in a decade. Finding such a project will not be easy. For example: integrating quantum theory and relativity may take quite a while; modelling a bacterium as a process has no well-defined goal; and automating software integration is unlikely to capture the public imagination. Perhaps, this last could be expanded to supporting the next generation of web services. More precisely, the grand challenge is to produce a universal adaptor that will use scientific method to link any collection of components, using success criteria provided by the components themselves and/or the user. Imagine the adaptor being able to link up a new printer to laptop, DVD player and mobile phone without any human intervention.

Program Optimisation

The next few paragraphs outline some possibilities for artificial science in **bondi**.

Program optimisation treats the program as a data structure, whose traversal reveals nodes suitable for optimisation. This is usually expressible as an update, that traverses the structure looking for suitable sub-terms to optimise. This approach is adopted by tools such as Stratego.

Data Parallel Computing

Crucial to data parallel programming is to customise the data distribution according to the resources available and the nature of the problem. To a large extent, this reduces to a problem of shape analysis, as I claimed in 1990s. In turn, shapes can be represented by patterns, so the results of dynamic shape analysis may be exploited as pattern parameters. On another level, a central technique of data parallel computing is the divide-and-conquer strategy, here encoded as path polymorphism.

Memory Management

The problems arising in data parallel computing are also familiar in managing a single workstation, with a hierarchy of caches, etc. and multi-core chips. Perhaps the pattern calculus can be made sufficiently fine-grained to support assembly-level programming, all the way down to bytes and bits. It would be interesting to translate the C language into pattern calculus.

Concurrency

Concurrency theory has been hampered by the lack of a good model for exchanging information, especially structured information. A concurrent pattern calculus may help here, by making the match process symmetrical, to support the trade of information rather than a one-way flow. For example, successful matching may lead

to the exchange of a credit card number for a booking number. Now variables and matchable symbols play the roles of input and output channels in the sense of pi-calculus. Matching produces a substitution, an output, while substitution application creates an input.

Web Services

Web services require high levels of both functionality and the ability to work with a diverse collection of data structures. Pattern calculus has just the necessary elements to make this work. The actions to be performed (the service) can now be separated from the description of the relevant data, as the latter can be described by a pattern parameter. For example, a services that compares products can be described independently of the format used to represent products.

Data Mining

Data mining can be viewed as a process of pattern generation. More precisely, mining seeks rules, or cases, in which some discovered pattern predicts a desired conclusion. Till now, such patterns are typically numerical relationships involving, say linear algebra, or the arrow of time. However, the collection of patterns may be greatly increased with the ability to handle the structures holding the data, as well as the data itself.

The larger strategy

Perhaps the biggest gap in the development of artificial science will be in the development of models from the results of data mining. In science, this is the most creative step, and most likely to require interaction with humans. Similar remarks apply to the development of testable hypotheses from theories. That said, there are many low-level forms of science in which automated tactics are likely to prove useful, much as in theorem-proving. Another way of looking at this is to combine data mining with theorem-proving.

Although this may look hard, it is certainly easier than e-business, since science is a collaborative activity, whereas business is competitive. Indeed artificial science may be a necessary step towards the development of business agents.

Summary

The time has come to automate the scientific method as a whole, and not just its logical fragment, to generalise from computing with well-formed inputs to programming in a chaotic world.

The 20th century theory of computing arose in trying to mechanise mathematics and logic. This emphasis on reason was a natural starting point when the inputs to computation are small, but grows ever more limiting as our access to realworld data increases. The time has come to embrace a scientific outlook, in which reason and experience play complementary roles.

Turing machines provide a suitable mechanism for representing reason and experience, but to program them using λ -calculus is to downplay the significance of experience, as represented by data structures. Pattern calculus provides a closer analogue to Turing machines, since the tape can be represented directly as a data structure.

Pattern calculus provides a clean solution to many vexing problems in the theory of programming, such as generic arithmetic, or the typing of object orientation, and suggest solutions to outstanding problems, such as developing generic queries or discovering web services.

More generally, it opens up the possibilities for artificial intelligence, by allowing it to be applied in domains where the environment is less structured, and more chaotic than before. A good example of this is the goal of defining software that can explore ways

of connecting disparate software components, a universal software adaptor.

Such artificial science may have an even greater impact in exposing other branches of science to automatic or semi-automatic approaches. This will be hard, but even partial successes could be transformative, in identifying hypotheses and models of interest.

Whether or not you embrace this vision, it is clear that programming based on logic alone is no longer enough. New sorts of polymorphism are required for programs to make sense of the world.

Back to Work!

Well! I was hoping to take a break after the book, but there is so much to do.