

Pattern Calculus

computing with functions and structures

This draft is not for circulation.
Please do not distribute.

Barry Jay

January 21, 2008

© Barry Jay, 2007,2008

Contents

1	Introduction	1
1.1	Programming styles	1
1.2	The motivating problem	3
1.3	Pattern-matching	6
1.4	Types	8
1.5	Sub-types	9
1.6	Type inference	11
1.7	bondi	12
1.8	Summary of the main achievements	12
1.9	Overview of the contents	13
I	Terms	15
2	Functions	17
2.1	Substitution	17
2.2	Pure λ -calculus	18
2.3	β -reduction	22
2.4	Confluence	23
2.5	Fixpoints	24
2.6	Notes	26
3	Data structures	29
3.1	Constructors and operators	29
3.2	Ad hoc operators	30
3.3	Data structures as abstractions	32
3.4	Atoms and compounds	34
3.5	Pure compound calculus	35
3.6	Generic queries	35
3.7	Notes	38
4	Static patterns	39
4.1	Patterns	39
4.2	Static pattern calculus	39

4.3	Static matching	41
4.4	Extensions	42
4.5	Encoding the compound calculus	45
4.6	Extensible records	46
4.7	Object calculus	47
4.8	Notes	49
5	Dynamic patterns	51
5.1	First-class patterns	51
5.2	Dynamic pattern calculus	53
5.3	Matching	54
5.4	Confluence of matching	55
5.5	String matching	57
5.6	Encoding static patterns	59
5.7	Wild-cards	59
5.8	Views	60
5.9	Notes	61
II	Types	65
6	Monomorphism	67
6.1	Typed calculi	67
6.2	Simply-typed λ -calculus	69
6.3	Strong normalisation	72
6.4	Simply-typed compound calculus	74
6.5	Simply-typed pattern calculus	76
6.6	Notes	79
7	Data polymorphism	81
7.1	Data structures as typed abstractions	81
7.2	Quantified types	82
7.3	System F	82
7.4	Reduction of type applications	84
7.5	Lists	85
7.6	Reducibility candidates	87
7.7	Notes	90
8	Path polymorphism	91
8.1	Terminating queries	91
8.2	Combinatory types	92
8.3	Query calculus	93
8.4	Folding	93
8.5	Selecting	95
8.6	Inductive constructors	97
8.7	Notes	100

9	Pattern polymorphism	101
9.1	Local type variables	101
9.2	Unification and matching	101
9.3	Typed pattern calculus	103
9.4	Matching typed patterns	105
9.5	Generic addition	106
9.6	Wildcards and views	108
9.7	Notes	109
10	Inclusion polymorphism	111
10.1	The same, but different	111
10.2	Object types	114
10.3	Type invocation and specialisation	115
10.4	Sub-typed λ -calculus	117
10.5	Sub-unification	118
10.6	Sub-typed pattern calculus	120
10.7	Proper names and coloured circles	122
10.8	Notes	123
11	Type inference	125
11.1	Type erasure and re-construction	125
11.2	Let-terms	126
11.3	Algorithm \mathcal{W}	128
11.4	Polymorphic arguments	129
11.5	Extension calculus	130
11.6	Type inference for patterns	134
11.7	Generic arithmetic	136
11.8	The sub-typed extension calculus	137
11.9	Notes	138
12	Structure polymorphism	141
12.1	Mapping	141
12.2	Representation of data types	142
12.3	Notes	145
III	Programming in bondi	147
13	Functional programming	149
13.1	From calculus to programming language	149
13.2	Higher-order functions	149
13.3	Pattern-matching functions	151
13.4	Type declarations	153
13.5	Data polymorphic programs	154
13.6	Structure polymorphic programs	155
13.7	Notes	157

14 Query programming	159
14.1 Numerical functions	159
14.2 Adding cases to existing functions	162
14.3 Searching	164
14.4 Modifying	167
14.5 Notes	167
15 Imperative programming	169
15.1 Operational semantics	169
15.2 References	170
15.3 Stores	172
15.4 The factorial function	175
15.5 Linked lists	176
15.6 Notes	179
16 Object-oriented programming	181
16.1 Classes	182
16.2 Sub-classes	185
16.3 Specialised methods	187
16.4 Parametrised classes	190
16.5 The expression problem	193
16.6 Notes	193
16.7 Implementing the motivating example	193

List of Figures

2.1	The rewriting relation	22
2.2	General reduction	22
2.3	Parallel reduction for pure λ -calculus	23
5.1	Parallel reduction for dynamic patterns	56
6.1	The simply-typed λ -calculus	70
6.2	The simply-typed compound calculus	74
6.3	Reduction in the simply-typed compound calculus	75
6.4	The simply-typed pattern calculus	78
7.1	System F	84
8.1	The query calculus	94
8.2	Reduction in the query calculus	95
9.1	Most general unification	102
9.2	The typed pattern calculus	105
10.1	Sub-typing	114
10.2	Type specialisation	116
10.3	The sub-typed λ -calculus	118
10.4	Most general sub-unification	119
10.5	The sub-typed pattern calculus	120
11.1	Let-terms	127
11.2	Type inference	128
11.3	Poly-terms	129
11.4	Type inference for polymorphic arguments	131
11.5	Typing extensions	132
11.6	Type inference for constructors, extensions and patterns	135
11.7	Sub-typing rules for extension calculus	137
11.8	Sub-type inference for extensions	139
13.1	Generic mapping	155
13.2	Generic left fold	156

13.3	Generic right fold and zipwith	157
14.1	Generic equality	161
14.2	Generic addition	162
14.3	Complex numbers with special cases	163
14.4	Generic search	164
14.5	Apply to all	166
15.1	Operational semantics of the extension calculus	170
15.2	Typing imperatives	171
15.3	Evaluation in the extension calculus with state	174
15.4	Linked lists	176
15.5	Doubly-linked lists	178
15.6	The generic iterator	179
16.1	The Name class	182
16.2	The Person class	183
16.3	The ProperPerson class	185
16.4	The employee class	187
16.5	Points	188
16.6	Coloured points	189
16.7	Circles and coloured circles	189
16.8	The Node class	190
16.9	The DNode class	192
16.10	Managers	194
16.11	Departments	195

Preface

The pattern calculus is a new foundation for computation, in which the expressive power of functions and of data structures are fruitfully combined within pattern-matching functions. The best of existing foundations focus on either functions (in the λ -calculus) or on data structures (Turing machines) or compromise on both (as in object-orientation). By contrast, a small typed pattern calculus supports all the main programming styles, including functional, imperative, object-oriented and query-based styles. Indeed, it supports web-services, to the extent that these are generic functions applied to partially specified data structures.

This book is an elaboration of the idea that computation is pattern-matching. As with all such ideas, it must be made to work hard. New sorts of patterns and polymorphism are developed, culminating in the possibility that any term can be a pattern. Types and sub-typing must be construed in novel ways, too. The development is incremental, using over a dozen calculi. Finally, the practical potential of pattern calculus is illustrated in a programming language **bondi**. While the results could be spread over a series of papers, this would certainly diminish the overall impact, as experience shows that no one paper can be both formal enough to be convincing and comprehensive enough to be motivating. Rather, the book has been conceived as a whole, starting with an initial, motivating problem and ending with its implemented solution.

The primary audience for this book is thus the experts in fields related to the foundations of computation, including λ -calculus, rewriting theory, type theory and programming language design. The formal development is created with them in mind. The book should also be useful for research students, senior undergraduates and others interested in novel approaches to computation: the many examples and the introduction to the **bondi** programming language are developed for them. Some care has been taken to allow exploration of **bondi** without having to address the formalisms.

In addition, the book may come to be of interest to some other communities interested in patterns. For example, it may be possible to represent some *design patterns* using the powerful patterns here. Also, since patterns can now be *computed* and not just written by a programmer, it should be possible to automate the process by which the patterns generated by pattern recognition software in, say, image processing or data mining, are used to produce pattern-matching functions.

My interest in pattern-matching arose from the realisation that the *shapes* of data structures were best described as patterns. This led to a series of calculi of increasing sophistication: the *constructor calculus* (2001); the *higher-order pattern calculus* and the first version of **bondi** (2004); and, with Delia Kesner, the *pure pattern calculus* (2005-6). In the Fall of 2004, I gave a series of seminars on pattern calculus in North America and Europe, at which point the need for a book became clear to me. After some false starts, the initial draft of Parts I and II was produced while working with Simon Peyton Jones at Microsoft Research (Cambridge) in the second half of 2006. Part III was produced in 2007

at University of Technology, Sydney. Most of the material was presented to the pattern calculus seminar in Sydney, in 2007. The current version will provide the textbook for a course at the University of Technology, Sydney in 2008.

This document is a first draft and so is likely to have many imperfections, especially in relating this work to that of others. All comments will be gratefully received.

Assumed knowledge

For the most part, the book is self-contained, aside from some rudimentary knowledge of set theoretical notation. Of greater importance is some mathematical or logical maturity.

Acknowledgments

This section is the most incomplete of all. Initial thanks to: Luca Cardelli, Marcelo Fiore, Thomas Given-Wilson, Daniele Gorla, Bob Harper, Ryan Heise, Freeman Huang, Neil Jones, Simon Peyton Jones, Jean-Pierre Jouannaud, Delia Kesner, Robin Milner, Eugenio Moggi, Clara Murdaca, Tony Nguyen, Jens Palsberg, Richard Raban, Matt Roberts, Don Sannella, David Skillicorn, Tony Sloane, Eelco Visser, Joost Visser and members of the pattern calculus seminar.

I am particularly indebted to Microsoft Research (Cambridge) for their support while writing this book, and to University of Technology, Sydney for giving me the freedom to pursue my thoughts.

Barry Jay
Sydney,
January, 2008

Chapter 1

Introduction

1.1 Programming styles

Why are there so many programming styles? Imperative, relational, functional and object-oriented styles are all in widespread use; they run on the same hardware; and yet there is little to bind them together. Since a large project typically employs several programming styles, it has been necessary to build fragile middleware to link them together. The resulting machinery grows in complexity and cost over time. The most recent manifestation of this arises when combining internet-based search with existing styles, as required for powerful web services. Thus, any progress in answering this abstract question may have substantial practical implications.

Although each style has its own idiosyncrasies, especially of syntax, it is productive to identify a programming style with its central concept, whose manipulation supports a distinctive approach to program re-use, or polymorphism. For example: imperative languages such as FORTRAN and C assign to locations holding data; relational languages such as SQL access fields; functional languages such as ML and Haskell apply functions; object-oriented languages, such as C++ and Java invoke methods. In this light, the original question can be re-expressed more positively as: how compatible are the central concepts of programming?

There are three tempting reasons to dismiss the question. It might seem that it was answered at the very beginning, in the 1930s, when Alonzo Church and Alan Turing showed the equivalence of λ -calculus (for functions) and Turing machines (for imperative programming). However, this equivalence says merely that both approaches support the same functions on natural numbers, without in any way addressing programming style or program re-use. Indeed, the latter concepts did not properly arise until in the late 1950s with the advent of the high-level languages FORTRAN (imperative) and Lisp (functional).

Perhaps stylistic differences arise only in practice. However, the differences are well established in theory, too. Over and above the communities devoted

to each style, there is the more fundamental division between the study of program complexity, as measured by Turing machines, and program meaning, as emphasised in λ -calculus.

Finally, it might seem that the fragmentation of styles is caused by differences between type systems, but types did not become important in programming until after the stylistic differences were well established.

The oldest source of stylistic tension appears to be that between functions and data structures. At one extreme, functional programming considers everything to be a function, even a number or a pair. At the other extreme, imperative programming considers everything, even a function, to be a data structure built from assignable components, such as squares on a tape. In between are various compromises or mixtures. For example, relational programming combines a limited range of data structures (tables of records) with a limited range of functions (queries and updates). Again, object-orientation wraps a mixture of data and functionality into a self-contained object.

Thus, the original question can be further narrowed to ask if there is a single concept able to support a uniform treatment of both functions and data structures.

The desired concept is pattern-matching, as embodied in pattern calculus. Pattern-matching functions combine the variable binding that is central to functionality with patterns that can describe data structures. By making the class of patterns sufficiently generous, all the main programming styles can be expressed within a single small calculus.

Although pattern matching is a very well-known programming technique, it has generally employed a narrow class of patterns that add convenience without increasing expressive power beyond that of the λ -calculus. By contrast, the pure pattern calculus allows any term to be a pattern, so that patterns are dynamic: they can be generated by a pattern recognition algorithm, passed as parameters, and then applied in a pattern-matching function. This new expressivity is enough to support the existing programming styles, and to suggest some new ones, based on new forms of polymorphism.

Of course, such bold claims require substantial justification. Without detailed technical development, these claims may appear too good to be true. On the other hand, the proofs of a few theorems may not be sufficient to demonstrate relevance to computing practice. Of course, it is impossible to address both kinds of concerns within a single paper, but a completely comprehensive account would exhaust everyone before the merits of the approach can be judged. To find some middle ground, this book has been organised around a single problem intended both to motivate the reader and to limit its scope. The following section states the problem, and its solutions sketched out in various styles. The book ends with the full implementation of the most general solution.

1.2 The motivating problem

The problem is to award a salary increase to all employees of a large organisation. Salaries are represented by floating point numbers, of type `Float`, and are to be increased by applying to them a given function `f:Float -> Float`.

Let us consider how various programming styles can address this problem. Each solution will be discussed and then written in **bondi**, a programming language which implements the pattern calculus. The **bondi** syntax is not spelt out until Part III but as it is drawn from existing programming styles, it is hoped that the programs written here will aid understanding.

In the simplest model, there is a data type `Salary` of salaries, and a function

```
incrSalary : (Float -> Float) -> Salary -> Salary
```

for increasing them, so that if `s` is a salary then `incrSalary f s` is the salary obtained by applying `f` to the float within `s`. These salaries are used in defining another data type `Employee` of employees for which can be defined a function

```
incrEmpSalary: (Float -> Float) -> Employee -> Employee
```

for increasing employee salaries. If the employees are stored in a list `es` then their salaries can be increased by the function

```
(incrEmpSalaryList: (Float -> Float) ->
                        List Employee -> List Employee) f es =
    mapList (incrEmpSalary f) es
```

where `mapList` is given by the pattern-matching function

```
let rec (mapList : (a -> b) -> List a -> List b) f =
  | Nil -> Nil
  | Cons x xs -> Cons (f x) (mapList f xs)
```

which applies its first argument `f` to each entry of its second, list argument. Each case of the pattern-matching function begins with a vertical bar. The first case maps `Nil` to `Nil` while the second case maps a list of the form `Cons h t` to `Cons (f h) (mapList f t)`. In the type `(a -> b) -> List a -> List b` the expression `List a` is the type of lists whose entries are of type `a`. Here `a` and `b` are type variables which may be replaced by any types. Hence `mapList` can be applied to lists of floats or salaries or employees, etc. That is, it is polymorphic in the types used to represent the list data, or *data polymorphic*. This is more commonly known as *parametric polymorphism* because of the presence of type parameters, but such parameters turn out to be common to several forms of polymorphism, so a narrower name has been adopted here. To emphasise the polymorphism, the typing may sometimes be written as

```
mapList: all a. all b. (a -> b) -> List a -> List b
```

where `all a` indicates that the type variable `a` is universally quantified, so that it can be instantiated in several different ways within a single program.

Although the polymorphism of `mapList` is useful, there are several limitations to this approach. First, the employees may be stored within some structure other than a list. For example, if employees are stored in a binary tree then mapping requires some function `mapTree: (a -> b) -> Btree a -> Btree b` to define

```
incrEmpSalaryTree: (Float -> Float) ->
    Btree Employee -> Btree Employee .
```

In this way, new mapping programs are required for each new structure. This repetition is avoided by introducing a *structure polymorphic* function

```
map: (a -> b) -> c a -> c b
```

in which the variable `c` represents an arbitrary structure, such as `List` or `Btree`. Now define

```
incrAllEmpSalary: (Float -> Float) -> c Employee -> c Employee
```

to handle all structures holding employees.

Second, the employees are unlikely to form a single structure, but rather to be distributed among departments, divisions, etc. in which employees do not have a distinguished status. That is, it is unlikely that the company has type `c Employee` since it could as easily have type `c Product` where `Product` types the items the company sells. In general, it will be necessary to search for employees within an arbitrary structure whose type does not give any clues as to their whereabouts. This requires a *path polymorphic* function able to traverse all paths through a data structure. The solution employs the generic function

```
let rec (apply2all : (all a. (a -> a)) -> b -> b) f =
  | Ref x as y -> f y
  | x y -> f ((apply2all f x) (apply2all f y))
  | x -> f x
```

that applies its first argument `f` to components of its second argument. Its first case is used to avoid copying (assignable) references, which need not concern us here. The second case will cause `apply2all` to act on both components of any *compound* `x y` be it a pair or list or tree. The third case terminates the recursion at an *atom*. Since the function `f` will be applied to components of unknown type, it must have a universal type `all a. (a -> a)`. Thus `f` cannot be `incrEmpSalary` since its type is not universal. To make the latter more general, combine it with a generic default function, namely the identity, to get

```
let (incrAnyEmpSalary: (Float -> Float) -> a -> a) f =
  | Employee y -> incrEmpSalary f (Employee y)
  | x -> x
```

so that `incrAllEmpSalary` can be redefined as

```
let (incrAllEmpSalary: (Float -> Float) -> a -> a) f =
  apply2all (incrAnyEmpSalary f).
```

It traverses the whole data structure, applying `incrEmpSalary f` to any employees it finds, and applying the identity function otherwise.

Having considered how employees are placed within the company, now consider the various classes of employees, such as managers, permanent or temporary employees. Since different kinds of employees have different sorts of remuneration, such as bonuses, it is likely that the algorithm for increasing salaries is different in each class. The standard approach to this difficulty is to shift to a class-based object-oriented language. Now there is a class `Employee2` of employees, with a method `incrSalary2` for increasing salaries by assigning to some field. This class has sub-classes for temporary employees and managers which may specialise the method for increasing salaries. This is all handled using sub-typing, and the associated *inclusion polymorphism*.

Now redefine `incrAnyEmpSalary` by

```
let (incrAnyEmpSalary2 : (Float -> Float) -> a -> Unit) f =
  | (x : Employee2[b]) -> x.incrSalary2(f)
  | x -> ()
```

in which the special case for employees (of type `Employee2[b]`) invokes the method `incrSalary2` while the default does nothing, as represented by `() : Unit`. Finally, define

```
let incrAllEmpSalary2 f = iter (incrAnyEmpSalary2 f);;
```

to iterate `incrAnyEmpSalary2` across an arbitrary data structure or object, using the *generic iterator*

```
iter: (all a.(a -> Unit)) -> b -> Unit .
```

The resulting solution is able to handle arbitrary employee sub-classes within an arbitrary company structure. This solution uses data and path polymorphism to search for employee objects and inclusion polymorphism to customise the salary increase method for each class of employees. All of which are implemented using pattern-matching functions.

This solution goes a long way towards unifying the various existing programming styles within **bondi**. Still, the solution is capable of further generalisation by allowing dynamic patterns. Any organisation which has grown through mergers, or developed subsidiaries in foreign cultures, is likely to have more than one representation of salaries or employees, leading to ontological problems. These difficulties may be resolved by allowing a free variable in a pattern to represent unknown pattern information. For example, a free variable may represent a function which converts a floating point number into a salary. The value of this variable in each context could then be computed from a static ontology.

This deceptively simple problem shows how each conventional programming style is tempted to artificially constrain the polymorphism of its solution. Conversely, the pattern calculus promises a new standard of generality, but how does it work?

1.3 Pattern-matching

The functions above illustrate several approaches to pattern matching, which can be supported by various calculi. Their starting point is the λ -calculus. In brief, the terms of the pure λ -calculus are given by

$$t ::= x \mid t t \mid \lambda x.t$$

in which every term t is either a variable, application or λ -abstraction, respectively. The sole evaluation rule is β -reduction, given by

$$(\lambda x.s) u \longrightarrow \{u/x\}s$$

where $\{u/x\}s$ is the *substitution* of u for x in s .

The patterns of `mapList` are of the most common form, namely those headed by a constructor such as `Nil` or `Cons`. In principle, the resulting pattern-matching functions can be translated back into the λ -calculus so that, while convenient, they do not increase expressive power.

By contrast, the pattern `x y` in `apply2all` is headed by a binding variable. It is able to match against any compound data structure be it a pair or a `Cons`-list or a node of a binary tree. In this setting the constructors must be considered as primitives, not abstractions, since they are used to define the data structures. The compounds considered here are essentially the *pairs* in Lisp. Corresponding to a case of the form `x y -> t(x,y)` is the Lisp program

```
if pair? z then t(car z, cdr z) ...
```

where `z` is the function argument that corresponds to the pattern `x y`. Just as in the pattern-matching approach, the use of the operators `pair?`, `car` and `cdr` is completely uniform. That is, although Lisp is often considered as viewed as an implementation of untyped λ -calculus, it does actually try to balance the roles of functions and data structures through its treatment of pairs.

The relationships can be clarified by introducing the *compound calculus* in which the *symbols* of Lisp are divided into (free variables) and constructors. Its term forms

$$t ::= x \mid c \mid t t \mid \lambda x.t \mid t =_c t \mid \text{pair? } t \mid \text{car } t \mid \text{cdr } t.$$

are those of the λ -calculus augmented by some collection of constructors c and the operators for constructor equality, the pair (or compound) test, and the first and second components, respectively. In a sense, the compound calculus is just a λ -calculus with some extra constants, but this is a fairly superficial view, that downplays the importance of data structures in computation.

The expressive power of the compound calculus is essentially the same as that of the *static pattern calculus* whose terms are given by

$$t ::= x \mid c \mid t t \mid p \rightarrow t$$

where $p \rightarrow t$ is a *case* built from a *pattern* p and a *body* t . In turn, the patterns p are given by

$$p ::= x \mid c \mid p p$$

consisting of (binding) variables, constructors and applications. For example, the familiar λ -abstraction $\lambda x.s$ is given by $x \rightarrow s$. Like the pure λ -calculus, there is only one reduction rule, the *match* rule, given by

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

where the substitution applied to s is the *match* $\{u/p\}$ of p against u . The rules that define the match correspond to the rules for the operators of the compound calculus. The static pattern calculus is expressive enough to support all of the pattern-matching in the motivating example.

By compressing all of the operators into a single rule, the static pattern calculus begins to compete with the λ -calculus as a foundation for computation, but its main weakness is the need for syntactic classes of constructors and patterns in addition to those for variables and terms required by the λ -calculus.

The most general approach is the *pure* or *dynamic pattern calculus* whose terms

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

consist of variables, constructors, applications and cases. Now constructors are built from variables, just like binding variables, and patterns are simply terms. Since patterns are now reducible, binding variables must be considered as arguments of abstractions and pattern-matching functions. It can easily happen that binding variables are lost during reduction, so these must be recorded separately, in the sequence θ . In matching, binding variables behave just like constructors, so that the economical solution is to identify them. For example, the λ -abstraction $\lambda x.s$ is now given by $[x] \hat{x} \rightarrow s$. The sole reduction rule is

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

where matching now tracks binding variables explicitly.

Patterns may now contain a mix of free and binding variables. For example, the *generic eliminator* is given by

$$[x] \hat{x} \rightarrow ([y] x \hat{y} \rightarrow y)$$

whose pattern $x \hat{y}$ contains both a free variable x and a binding variable y .

1.4 Types

One way of characterising the polymorphism of a program is the variety of programming environments in which it can be meaningfully employed. Perhaps the most common way to measure this variety is with a type system, so that any well-typed program of type T can be employed in any environment expecting a program of type T . When working with a fixed collection of terms, such as λ -terms, it is natural to identify the nature of the polymorphism with that of the type system, but in general polymorphism need not depend upon typing. For example, the path polymorphism of `apply2all` is already a new form of polymorphism, not expressible in untyped λ -calculus, even though its type is of an old, familiar form.

Once again, the starting point is within the λ -calculus, this time using Girard's System **F**, whose types T are given by

$$T ::= X \mid T \rightarrow T \mid \forall X.T$$

where X is a type variable, $S \rightarrow T$ is a function type and $\forall X.T$ quantifies the type variable X in the type T . This compact type system is remarkably expressive. In particular, it is able to represent all the inductively defined data types, such as lists and trees. Further, evaluation of its terms is guaranteed to terminate. This captures the important intuition that a query of an inductive data structure (no cycles) should always terminate.

However, System **F** has no notion of a compound (everything is a function) and so cannot express a uniform approach to data structures. For example, it cannot support a function that computes the size of both a list and a tree. To support such uniformity requires that data types be distinguished from function types. Indeed, making this distinction is common in practice. The *combinatory type system* adds type constants C such as `List` and `Int`, and type applications such as `List Int` to get

$$T ::= X \mid C \mid T T \mid T \rightarrow T \mid \forall X.T.$$

It is used by the typed pattern calculus, whose matching algorithm matches types as well as terms. The latter supports more polymorphism than System **F** but, since pattern-matching can be used to define fixpoints, its reduction need not terminate. A closer analogue of System **F** is given by a typed version of compound calculus, called *query calculus*. It supports a polymorphic notion of folding or primitive recursion to define queries able to act on arbitrary data structures. Hence query calculus is more polymorphic than System **F** since its queries apply uniformly to all its data types. To the extent that records are supported it is more polymorphic than SQL, too. Further, by limiting the types of constructors (to be inductive), evaluation can be guaranteed to terminate while still being able to represent all the datatypes modeled in System **F**.

Since the terms of the query calculus are so similar to those of Lisp, and the combinatory types have been used in λ -calculus for at least twenty years, it is natural to wonder why this typed calculus has not previously appeared.

Perhaps the fundamental barrier was that quantified types were imported from logic, using the Curry-Howard correspondence. This fruitful and influential approach relates function types to implications, and type quantifiers to logical quantifiers so that functions are related to proofs. In this setting, data structures are just abstractions, so that the path polymorphism of Lisp becomes illogical and hence invisible.

Another barrier to typing concerns the relationship between the types of the cases of a pattern-matching function. When such a function is converted to a λ -term it becomes a conditional of the form `if b then s else t` where `b` is a boolean that represents the success or failure of some matching. When typing conditionals it is essential that both branches have the same type, which suggests that each of the original cases must have the same type. For example, `mapList` has two cases of type `List a -> List b`. In general, however, this is too restrictive. For example, consider `incrAnyEmployeeSalary` in the previous section. Its default case is the identity of type `a -> a` but its special case is of type `Employee2[b] -> Employee2[b]`. No danger arises from this since if the employee pattern matches then `a` must be of the form `Employee[b]`. That is, matching yields type information that is lost in the translation to conditionals.

This example illustrates the general principle that special cases may have special types. This principle will be realised in several different ways, according to the circumstances in each calculus. In the query calculus, cases may be combined using a typecase, in which the choice of case is determined by the typing. Later, more subtle approaches will be employed, especially in the presence of sub-typing.

1.5 Sub-types

Sub-typing is another mechanism by which types can express polymorphism. If S is a sub-type of T (written $S < T$) then any term of type S is also a term of type T . It follows that a term of type T may evaluate to one that also has type S . This is subtly different from the polymorphism supported by type variables since their instantiation cannot, in general, be reversed. Hence, both type mechanisms will be required to support a full range of polymorphic styles.

In general, sub-typed calculi have proved to be less polymorphic than one might wish. Various approaches have been considered, with increasing expressive power being bought at the price of ever increasing complexity. The resolution proposed here is based on the following observation: the introduction of type variables imposes severe constraints upon sub-typing, and in particular conflicts with the identification of data types with function types. However, since this identification has been rejected anyway, the constraints show how to develop a simple account of sub-typing that is more expressive than most.

The argument proceeds by considering the nature of function (or method) specialisation, in which a special case, of type $P \rightarrow S$ is added to a default function of type $Q \rightarrow T$ to get a function f . The risk of type errors arises if a single term can be of type P and also of type Q . Let us consider some

possibilities.

Suppose that there are no type variables. Further, suppose that P is a sub-type of Q . Let $t : Q$ be a term that evaluates to a term $t' : P$. Then type checking will show that $f t$ has type T but $f t'$ has type S so that type safety requires $S < T$. That is, if $P < Q$ then it must be that $S < T$. More generally, if there is a type U which is a sub-type of both P and Q then the possibility that t' has type U implies that $S < T$.

Now suppose that there are type variables but that the sub-typing relation is just equality. Clearly, $P = Q$ implies that $S = T$. More generally, any solution of $P = Q$ must also solve $S = T$. Since there is no bound on the collection of type substitutions that unify P and Q it is impractical to check them all. Fortunately, if there is a solution of $P = Q$ then there is a most general solution, their most general unifier. Hence it suffices to prove that if P and Q have a most general unifier then it also unifies S and T .

Now consider the general situation in which sub-typing is non-trivial and there are type variables. The solution would be straightforward if there is a most general solution to the problem of finding a greatest lower bound for P and Q but this is not so easy. Even to define a most general solution to $P < Q$ is a challenge. For example, if this inequality turns out to be equivalent to the pair of inequalities $Q_1 < X$ and $X < P_2$ then X could be any type between Q_1 and P_2 . However, to avoid such difficulties imposes severe constraints upon the sub-typing relation.

When all types are function types then there is only one viable approach to their sub-typing: $P \rightarrow S < Q \rightarrow T$ if $Q < P$ and $S < T$. Note that the sub-typing on P and Q is the reverse of that considered above when specialising. That is, sub-typing here is *contravariant* in function argument types. However, this implies that the inequality $X \rightarrow P_2 < Q_1 \rightarrow X$ is equivalent to $Q_1 < X$ and $X < P_2$ which has no general solution.

Once again, the difficulty is resolved by recognising that data structures and functions are different, and have different types. In practice, sub-typing is driven by the desire to hide some data within a structure. Since this has nothing to do with functions, there is considerable freedom in deciding how to sub-type the latter. The solution adopted is to make function types *invariant* in their argument types, so that $P \rightarrow S < Q \rightarrow T$ implies that $P = Q$ and $S < T$.

Further constraints along these lines lead to a particularly simple approach to sub-typing with some novel characteristics. First, the super-types of a type form a finite linear order. While quite unusual in a calculus, this reflects the nature of class hierarchies in programming languages such as Java. Second, most types are minimal, in the sense that they have no proper sub-types. This can be used to limit the argument types P and Q in specialisation so that any substitution which causes P and Q to have a lower bound can be replaced by one that identifies them. Now the general form of specialisation relation can be defined so that 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 $S < T$.

Of course, there are other issues to consider besides specialisation, such as inheritance, and the typing of “self” but these all become easier to handle

when sub-typing is complemented by type variables. In brief, sub-typing is structural, determined by the existence of a top type `Top` which is a super-type of all others. The typing of self employs a standard type variable, used to represent the type of any additional fields. For example, rather than declaring a sub-type relation `Manager < Employee` the type of a generic employee is `Employee2 [[X]]` for some type variable X . Then the type of a generic manager is `Employee2 [[Manager2[[Y]]]]` which may be abbreviated to `Manager2[Y]`. That is, a manager is an employee with some extra information. Corresponding to `Manager < Employee` above is `Manager2[Top] < Employee2[Top]`. This use of type variables provides a fine level of control over re-use in general and of inheritance in particular.

1.6 Type inference

Having developed some typed calculi, the next step is to try and reduce the burden of explicit types in writing and evaluating programs. In λ -calculus, erasure of all types from a term of System **F** produces a term of the pure λ -calculus. Further, this translation preserves the reduction rules. Hence, the challenge of type inference for it is to reverse the type erasure: given a term t of the pure λ -calculus, try to find a typed term whose type erasure is t . In System **F** there is no general algorithm for inferring or checking types. However, there is a sub-system, the Hindley-Milner type system, in which every well-typed term has a most general, or principal, type. The essential point is to describe two sub-classes of types, the *mono-types* T and the *poly-types* τ . This underlies programming languages such as ML, in which types are inferred or checked during compilation, but discarded during evaluation.

For the pattern calculus, the situation is more complex. To begin, some generics such as `apply2all` require arguments that have a quantified type. This can be addressed by modifying the definition of mono-types. In the Hindley-Milner system a function type $U \rightarrow S$ is built from mono-types U and S . Here, a function type $\tau \rightarrow S$ may employ a poly-type τ as its argument type. The resulting definitions are

$$\begin{aligned} T &::= T \mid C \mid T T \mid \tau \rightarrow T \\ \tau &::= T \mid \forall X. \tau. \end{aligned}$$

Type inference proceeds as before, except that argument poly-types must be given explicitly. Since generics such as `apply2all` are relatively rare and complex, this burden does not seem onerous.

Another difficulty is that types play a crucial role in matching and in reduction of typecases, so that, in general, type erasure does not preserve reduction. The use of types in matching can be avoided by restricting the patterns so that term matching *implies* type matching, as will occur if every pattern has its most general type. However, this is incompatible with pattern reductions that lose type information, so patterns must be restricted. Various possibilities have been considered, but the simplest approach is adopted here, namely to make

all patterns *static*. This will suffice for the motivating problem, but not for the dynamic examples such as the generic eliminator.

Typecases are avoided by combining them with cases to form *extensions* of the form $p \rightarrow s \mid r$ in which the typing of the pattern p exploits both the most general type of the pattern, but also the typing of the default r . The ultimate default of an extension is typically given by a λ -abstraction, so the latter must be re-introduced as a term form. Thus, the patterns and terms of the *extension calculus* are given by

$$\begin{aligned} p &::= x \mid c \mid p p \\ t &::= x \mid c \mid t t \mid \lambda x.t \mid t \rightarrow t|t. \end{aligned}$$

Now types of programs can be inferred and checked, and then discarded prior to evaluation.

1.7 bondi

The programming language **bondi** is based on the sub-typed version of the extension calculus described above. To this is added machinery for declaring data types, imperative features, and object-oriented classes. Some care is required with the imperatives, which are introduced using references. The standard difficulty is to safely combine assignment and type variables. The novel difficulty is to manage the interaction between path polymorphic functions and cyclic data structures. For example, if a department and manager refer to each other then naively traversing all paths will automatically produce an infinite loop. Such issues are handled by using references to references for the “backward links” and avoiding these during traversal.

A novel feature of **bondi** is the ability to both define new functions on existing types (in functional style) and to add new cases to existing functions (in object-oriented style). This is achieved without ever changing the behaviour of existing programs, so that one has, *behavioural continuity*. The difficulty of achieving this has been seen as evidence that one must choose between a data-centric or a function-centric programming style, but **bondi** is good evidence that the two styles are compatible.

This theory and implementation is enough to show that new expressive power, and new combinations of existing programming styles are both theoretically sound and implemented. Much more appears to be possible, with potential applications to developing web services, data mining, pattern recognition, etc.

1.8 Summary of the main achievements

- Compound calculus provides a new foundation for Lisp that describes its structures (pairs) as well as its functions.
- Dynamic pattern calculus allows arbitrary terms to appear in patterns, while retaining confluence, and hence uniqueness of results.

- Query calculus generalises queries from records to arbitrary data structures. When all constructors are inductive then reduction always terminates.
- Typed pattern calculus holds promise as a basis for developing web services, pattern recognition, etc.
- Sub-typing becomes tractable by making function types invariant in their argument types, while expressive power is actually increased by introducing type variables.
- The extension calculus supports type erasure and type inference.
- Pattern calculus can support five forms of polymorphism, in data, path, pattern, inclusion and structure, even though all reduction is based on a single notion of pattern-matching.
- **bondi** supports functional, imperative, query, and object-oriented programming styles.
- **bondi** allows new functions to be defined on existing data types, and also new method specialisations to be defined in sub-classes, while ensuring behavioural continuity.
- **bondi** supports a general solution to the motivating problem, of generically increasing salaries of different classes of employees, as represented within company data of unknown structure.

Perhaps surprisingly, the proofs have all been produced by adapting techniques already familiar from λ -calculus and rewriting theory. The chief contribution lies in the novelty of the definitions, which in turn derive from a single observation, that functions and data structures are not interchangeable, but can be related through pattern-matching.

1.9 Overview of the contents

The rest of the book is divided into three parts, on terms, types and programs. As much as possible, each part is intended to be self-contained. For example: the reduction rules for the various typed calculi are always presented in full; the intuitive meaning of the **bondi** programs and their execution should be clear from the context in Part III. However, a deeper understanding of later parts will depend upon those that precede them. As far as possible, the story is told in the figures, which have already been listed at the front.

Part I considers terms without types. It starts with a brief introduction to the pure λ -calculus in Chapter 2. Chapter 3 introduces data structures. They are built from *constructors*, a class of inert term constants. While commonly viewed as syntactic sugar for the corresponding λ -terms, the compound calculus

shows how they may support new expressive power, such as the ability to compute the size of a data structure. The Chapter 4 introduces pattern-matching through a class of static patterns. The *static pattern calculus* supports a pattern that matches an arbitrary compound so that it is able to support algorithms that traverse arbitrary data structures. In passing from λ -calculus to compound calculus to static patterns, the size of the calculus has expanded to include new term constants, and then a new syntactic class of patterns. The final chapter in this part, Chapter 5, simplifies the syntax while further increasing expressive power, by allowing *any term* to be a pattern. The main theorems in this part are to show that reduction is confluent, and that various translations preserve reduction.

Part II develops type systems that can support the program re-use introduced in Part I. Chapter 6 introduces simple types (without variables) for the various calculi of Part I. Subsequent chapters strengthen the type system to support the various forms of polymorphism. Chapter 7 introduces type variables (or parameters) to support System **F** and its data polymorphism. Chapter 8 and Chapter 9 use the combinatory type system to type the compound calculus (as the query calculus) and the pattern calculus.

Chapter 10 introduces *sub-typing*, which is structural. Various type-level operations are required for type specialisation and invocation. A key aspect of this approach is that unknown fields are represented by a type variable. Chapter 11 considers type inference for the λ -calculus and the pattern calculus. Just as let-terms are used to link pure λ -calculus to System **F**, the *extension calculus* is used to link the pure pattern calculus with the typed pattern calculus. Chapter 12 introduces *structure polymorphism* in which arbitrary data structures are represented by a small class of primitive structures. Then structure polymorphic functions can be defined by providing cases for the primitives only. In this way, the need for arbitrary constructors can be replaced by a finite set of primitives plus a collection of *names*. This chapter was delayed to take advantage of type inference in presenting the examples. The main theorems in this part show that reduction is confluent, preserves typing and progresses, in the sense that matching cannot get “stuck”. Also, System **F** and the inductive query calculus are strongly normalising.

Part III presents the main examples within the **bondi** programming language. Chapter 13 introduces the language, as based on the extension calculus, and illustrates polymorphism in data and structure. Chapter 14 introduces path polymorphism and shows how to add new cases to existing functions. Chapter 15 adds state in the form of references. Chapter 16 adds classes to support *object-orientation*. Behavioural continuity is discussed. The final section returns to the motivating problem of increasing salaries, and presents an object-oriented solution combining data, path and inclusion polymorphism in a state-based solution.