Denotational Programming Strategies for Systematic Program Construction

Noel Welsh
noel@inner-product.com
Inner Product LLC
Seattle, WA, USA

ABSTRACT

We present *denotational programming strategies* blah blah blah

CCS CONCEPTS

• **Social and professional topics → Computational thinking**; *Computer science education*; Informal education; Adult education; • **Software and its engineering** → *Semantics.*

KEYWORDS

computational thinking, denotational semantics, programming strategies, design patterns

ACM Reference Format:
Noel Welsh. 2019. Denotational Programming Strategies for Systematic Program Construction. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

In our experience learning and teaching programming we've seen a consistent problem: students don't understand how language features are connected to solving problems. For the last decade we have been refining our techniques for teaching Scala, to a wide variety of students, and our curriculum has evolved to focus on *programming strategies.* A programming strategy captures a reusable building block of a program or the programming process. Programming with programming strategies gives students a systematic and repeatable process for writing code.

In this paper we present a a particular kind of programming strategy, which we call *denotational programming strategies.* These strategies draw on two ideas: the first is the relative efficiency of declarative knowldedge compared to procedural knowledge, and the second is the idea that programs and the programming process both consist of few reusable components.

Cognitive psychology distinguishes between *procedural knowledge* and *declarative knowledge.* Procedural knowledge means knowledge of *how* things work; the specific steps required to carry out a task. Declarative knowledge is knowledge of what something *is* or *why* something is the way it is. We see a similar distinction in programming language semantics, which differentiates between *operational semantics* (see, for example, [3]) and *denotational semantics* [9]. In an operational semantics the meaning of a program is explained in terms of the steps carried out by some abstract machine. In contrast an operational semantics gives meaning by relating program terms to mathematical constructs which we assume we already understand.

A short example may help illustrate the difference. Consider explaining the following Scala program.

```
def fibonacci(n: Int): Int =
  n match {
      case 0 => 1
      case 1 => 1
      case _ =>
        fibonacci(n-1) + fibonacci(n-2)
  }
```

To explain this program in an operational way we might use an abstract machine with a stack [1] and explain recursion in terms of operations on the stack. To explain this code in a denotational way we can simply say the program `fibonacci(n)` corresponds to the n^{th} Fibonacci number, assuming we already know the Fibonacci numbers.

As the example above shows, denotational or declarative knowledge can be much more compact than operational knowledge. It can be advantageous when learning as it abstracts a lot of complexity, such as stack manipulation in the case of recursion. We don't undervalue operational or proecdural knowledge; we believe that programmers should be able to explain programs in terms of some abstract machine. However we believe students can make faster progress if they are first introduced to programming in a denotational manner, with operational knowledge filled in once they are comfortable with the basic tasks of programming.

This brings us to our second idea, which is that a large part of programs themselves, and the process for constructing them, are made from reusable building blocks. We call these building blocks programming strategies. This idea is not new to us, and we discuss related work in Section 4. What is new are the strategies themselves, and our approach to teaching that gives these strategies a central role.

In our teaching we have identified ten strategies. We call three of them denotational strategies. This is for two reasons. The first is that these strategies are the inverse of denotational semantics. In denotational semantics we map programs to mathematical objects. In our denotational strategies we do the reverse: we start with some mathematical object we wish to realise, and then create the

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/1122445.1122456

[1]This kind of machine is commonly used in teaching programming, but not often in operational semantics. We use this example as we expect it will be familiar to our readers.

code that corresponds to it. The second reason is that these strategies embody denotational or declarative knowledge. The student does not need an operational understanding to write correct code when using a denotational strategy.

2 DENOTATIONAL PROGRAMMING STRATEGIES

In this section we describe our three denotational programming strategies. The majority of our students do not have a strong mathematical background so we do not emphasise this connection when teaching the strategies. However we do use functional programming jargon, such as "sum types", as our students may encounter these terms when talking to other functional programmers.

For each strategy we give:

- a *condition* that describes when the strategy is applicable;
- a *description* of the strategy and its use;
- an *example* of the strategy in use; and
- any *notes* that we feel are relevant.

We use the Scala programming language [8] to present our strategies. This is the language we teach, but the strategies are not restricted to Scala. They are often more compact in other functional programming languages, such as Haskell [7] or OCaml [6]. While our strategies can be used in object-oriented languages, such as Python or Java, in most cases they lack language and compiler support that makes them more difficult to use correctly.

2.1 Algebraic Data Types

Condition. Whenever data can be described using *logical ors* or *logical ands* it should be defined using an algebraic data type.

Description. Algebraic data types are our main way of defining data. If the data description meets the conditions for this strategy the code that defines the data follows immediately.

Algebraic data types consist of two sub-strategies, that for logical ors, also called *sum types*, and that for logical ands, also called *product types*.

For a sum type, if `A` is a `B` *or* `C` then we can immediately write the corresponding Scala

```
sealed trait A
final case class B() extends A
final case class C() extends A
```

For a product type, if `A` is a `B` *and* `C` then we can immediately write the corresponding Scala

```
final case class A(b: B, c: C)
```

Example. For our example we'll take the classic functional programming data structure the singly-linked list. For simplicity our list will hold only `Int`s.

The data description is: a `List` is either `Empty` or a `Pair`. A `Pair` has a head which is an `Int`, and a tail which is a `List`.

From this description we can immediately write the complete data definition, though here we take it step-by-step. The first part, "a `List` is either `Empty` or a `Pair`" is a logical or (a sum type). We can use the corresponding pattern to immediately write

```
sealed trait List
final case class Pair() extends List
final case class Empty() extends List
```

The second part, "a `Pair` has a head which is an `Int`, and a tail which is a `List`" is a logical and (a product type). Using the corresponding pattern we can complete the definition

```
sealed trait List
final case class Pair(head: Int, tail: List) extends List
final case class Empty() extends List
```

Notes. Scala does not support algebraic data types directly, unlike many other functional programming languages. Instead they are constructed using `sealed traits` and `final case classes` as show in the description and examples. There are some subtleties in encoding algebraic data types in Scala. For example we might use a `final case object` instead of a `final case class` if our type holds no data. We do not got into these here as they are not of interest to non-Scala programmers. Direct syntax for algebraic data types is slated for the next of version of Scala, known as Dotty [1], which will solve these issues.

In mathematics we usually do not name fields in a product type. This is also the case in languages such as Haskell and O'Caml. In Scala we must always name the fields of a type.

2.2 Structural Recursion

Condition. Whenever we want to transform an algebraic data type we can do this using structural recursion.

Description. Structural recursion provide a generic skeleton for transforming any algebraic data type. Where algebraic data types build data, structural recursion takes data apart (and possibly builds a new data structure). Unlike the algebraic data type strategy, the code given by the structural recursion strategy is not complete; there is a problem specific component that the programmer must complete.

Structural recursion has two components, one for sum types and one for product types (recall these are the two components of algebraic data types). In Scala there

For a sum type, if `A` is a `B` *or* `C` then we can immediately write the corresponding skeleton

```
anA match {
  case B() => ???
  case C() => ???
}
```

The ??? indicates code that we must complete with a problem-specific implementation.

For a product type, if `A` is a `B` *and* `C` then we can immediately write the corresponding Scala

```
anA match {
  case A(b, c) => ???
}
```

In either case, if the data is recursive then the there is a corresponding recursion on the right-hand side of the =>. This will become clearer in the example.

Example. For our example we will use the `List` data type we declared earlier, and implement a method to sum the elements of a `List`.

We start by writing the method header. This is not part of the strategy but rather basic Scala knowledge we assume the student has at this point.

```
def sum(list: List): Int =
  ???
```

The first step is to realise that `List` is an algebraic data type and therefore the structural recursion strategy can be used. From the structure of `List` we can write

```
def sum(list: List): Int =
  list match {
      case Pair(hd, tl) => ???
      case Empty() => ???
  }
```

Finally, we note that `Pair` is recursive (specifically, the `tail` is). This tells us that somewhere on the right-hand side we need to recurse on the `tail`.

```
def sum(list: List): Int =
  list match {
      case Pair(hd, tl) => ??? sum(tl)
      case Empty() => ???
  }
```

We have now gone as far as the structural recursion strategy will take us. To complete the method we can use another strategy, which we call *following the types*, but it's not a denotational strategy and hence out of scope for this paper. We give the complete code here for reference.

```
def sum(list: List): Int =
  list match {
      case Pair(hd, tl) => hd + sum(tl)
      case Empty() => 0
  }
```

Notes. There are two important teaching points when using structural recursion. The first is that we don't need to think about recursion—the recursion is given to us by the strategy. So long as we get the individual cases correct the strategy guarantees the correctness of the method. The second point is to focus just on those individual cases, which in the examples above are a base case and a recursive case. The base case usually has a straighforward solution (zero is the sum of an empty list). For the recursive case we must remember that we don't need to think about the recursion. We can just assume it's correct, so in the example above we can assume we have the sum of the tail and then the final solution is to add the head to that value.

In Scala we can implement structural recursion using pattern matching or polymorphism. We have only illustrated the pattern matching approach here, as it's more common in other functional programming languages.

2.3 Generic Types

Condition. When we have no information about some data or, alternatively, we want code to work with all data, we should represent that data with a generic type.

Description. In some situations we will want our code to work with any data type. For example, if we're creating a container, such as `List` above, we usually want it to store any type the user chooses. An alternative way of phrasing this is to say we have no information, at the time we're creating the code, of the data type that will be used.

There are two components to a generic type: the declaration and the use. The declaration is given with square brackets. For example, `[A]` declares a generic type called `A`. Such a declaration can occur either immediately after the class name in a class declaration or immediately after the method name in a method declaration:

```
class ExampleClass[A] {
  def exampleMethod[B] = {  }
}
```

If a generic type is declared on a class it is in scope for any constructor parameters and the body of that class. Likewise for a method its scope is the method parameters and body.

Once a generic type has been declared, and is in scope, it can be used like any other type. Here's an example of defining the identity method: the method that returns whatever is passed to it.

```
def identity[A](a: A): A = a
```

Generic types are analogous to method parameters, which we must also first declare before we can use. For this reason generic types are also known as type parameters. There is also a similarity to construwhen extending a class that defines a generic type. In this case we must pass some type, either concrete or generic, for this generic type, similarly to how we must pass parameters to a method.

```
class ClassOne[A]
class ClassTwo[A] extends ClassOne[A]
```

Example. For our example we will extend the `List` example to store any data type the user specifies. The type will be used throughout the class, so it is scoped to the class declaration.

First we declare the generic type.

```
sealed trait List[A]
```

Now we must pass a type to this type parameter in the subclasses. Our final code is

```
sealed trait List[A]
final case class Pair[A]() extends List[A]
final case class Empty[A]() extends List[A]
```

Notes. The use of generic types (or, parametric polymorphism to use FP terminology) is familiar in containers. There are other situations where it is less familiar, and that is where this strategy is particularly useful. For example, we sometimes use a case study where we ask students to create logging infrastructure. A bit of reflection shows that we have no real knowledge of what the user wants to log, and hence a generic type is appropriate here. We find students from an OO background often reach for an interface instead, and create complicated designs as they try to encompass every possible use case.

Explaining the language machinery to use generic types is quite a bit more complicated than the core concept. We have skipped discussion of co- and contravariance, which increase both conceptual and implementation complexity.

3 EXTENDED EXAMPLE

In this section we give an extended example of the programming strategies.

4 RELATED WORK AND DISCUSSION

Our programming strategies are most similar to, and were inspired by, the *design recipes* described in How to Design Programs [2].

A more distant inspiration are *design patterns* [4]. Our denotational programming strategies differ from design patterns in the precision of the definition. Design patterns are informally defined. Our strategies are based on an underlying mathematical model.

Programmings strategies have been the subject of other research …

Design patterns have little active research, perhaps because more modern languages have subsumed them as language features (see, for example, [5]). Will the same happen to our programming strategies? One argument is that programming strategies represent ways of thinking about code, independent of implementation, and have utility even in languages that directly support them. However this same argument could be applied to design patterns. We believe our strategies will be more durable as they encode basic concepts of first order logic (logical and and logical ors for algebraic data types, and universal quantification for generic types).

Are more believable answer is that our strategies apply to functional programming, and functional programming is programming paradigm the industry is currently shifting towards.

5 CITATIONS AND BIBLIOGRAPHIES

REFERENCES

[1] Nada Amin, Karl Samuel Grütter, Martin Odersky, Tiark Rompf, Sandro Stucki, Lindley, and Sam. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Springer International Publishing, Switzerland, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

[2] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2014. *How to Design Programs, Second Edition.* MIT Press, Cambridge, MA.

[3] Matthias Felleisen and Robert Hieb. 1992. The revised report of the syntactic theories of sequential control and state. *Theoretical Computer Science* 103, 2 (1992), 235–271.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Boston, MA.

[5] Joseph Gil and David H. Lorenz. 1997. *Design Patterns vs. Language Design.* Technical Report LPCR9703. Department of Computer Science, Technion—Israel Institute of Technology, Haifa, Israel.

[6] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2019. The OCaml system release 4.08. http://caml.inria.fr/distrib/ocaml-4.08/ocaml-4.08-refman.pdf

[7] Simon Marlow. 2010. Haskell 2010 Language Report. https://www.haskell.org/definition/haskell2010.pdf

[8] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. 2006. *An Overview of the Scala Programming Language (2. Edition).* Technical Report 85634. School of Computer and Communication Sciences, École polytechnique fédérale de Lausanne. http://infoscience.epfl.ch/record/85634

[9] Dana S. Scott. 1972. Continuous Lattices. In *Toposes, Algebraic Geometry and Logic*, F. W. Lawvere (Ed.). Lecture Notes in Mathematics, Vol. 274. Springer Verlag, Berlin, Heidelberg, 97–136.

ACKNOWLEDGMENTS

Acknowledgement redacted to avoid identifying information