Datatype-Generic Programming

Jeremy Gibbons
University of Oxford
March 2003

0. Outline

1. generic programming
2. C++ template meta-programming
3. algebra of programming
4. datatype-generic programming
5. two cultures
6. looking forward

1. Generic programming

Generic programming is a matter of *making programs more adaptable by making them more general*.

The intention is to allow the programmer to *capture more recurring patterns as abstractions*.

This is achieved by providing a wide variety of kinds of parameter: *non-traditional polymorphism*.

Parameters may have a rich structure: eg types, type constructors, other programs, class hierarchies, programming paradigms...

Mostly popular realization today is through the *C++ Standard Template Library*.

2. C++ template meta-programming

Methods and classes parametrized by *types*:

```
template<class T>
void swap (T& a, T& b) { T c = a; a = b; b = c; }
main() {
  int i1 = 3, i2 = 4;          swap<int>(i1,i2);
  double d1 = 3.5, d2 = 4.5; swap<double>(d1,d2);
}
```

... and by (integral, enumerated or pointer) values:

```
template<class T, int size>
class Vector { private: T values[size]; ... };
main() {
  Vector<int, 3> v;
  Vector<Vector<double,100>, 100> matrix;
}
```

2.1. Bluffer's guide to the STL

A library of collection types and algorithms using C++ templates.

Container types: parametrically-polymorphic collection types — arrays, sequences, etc

Iterators: abstraction of pointer; increment, decrement, arithmetic, dereference as l-value or r-value

Algorithms: functions defined over container types: `count`, `sort`, `reverse`, etc

Concepts: abstract requirements of a template parameter; iterator concepts form interface between algorithms and containers

Function objects: objects providing `operator()` method; other parameters to algorithms (eg ordering for sorting)

2.2. Compile-time meta-programming

In fact, there is a lot more to templates than appears in the STL.

The template language forms a simple, purely-functional sublanguage that is *executed at compile time*.

The language is Turing complete: Unruh showed how to compute primes in the compiler, and Czarnecki and Eisenecker implement a rudimentary LISP interpreter via templates. (Compilers typically provide only bounded recursion, though.)

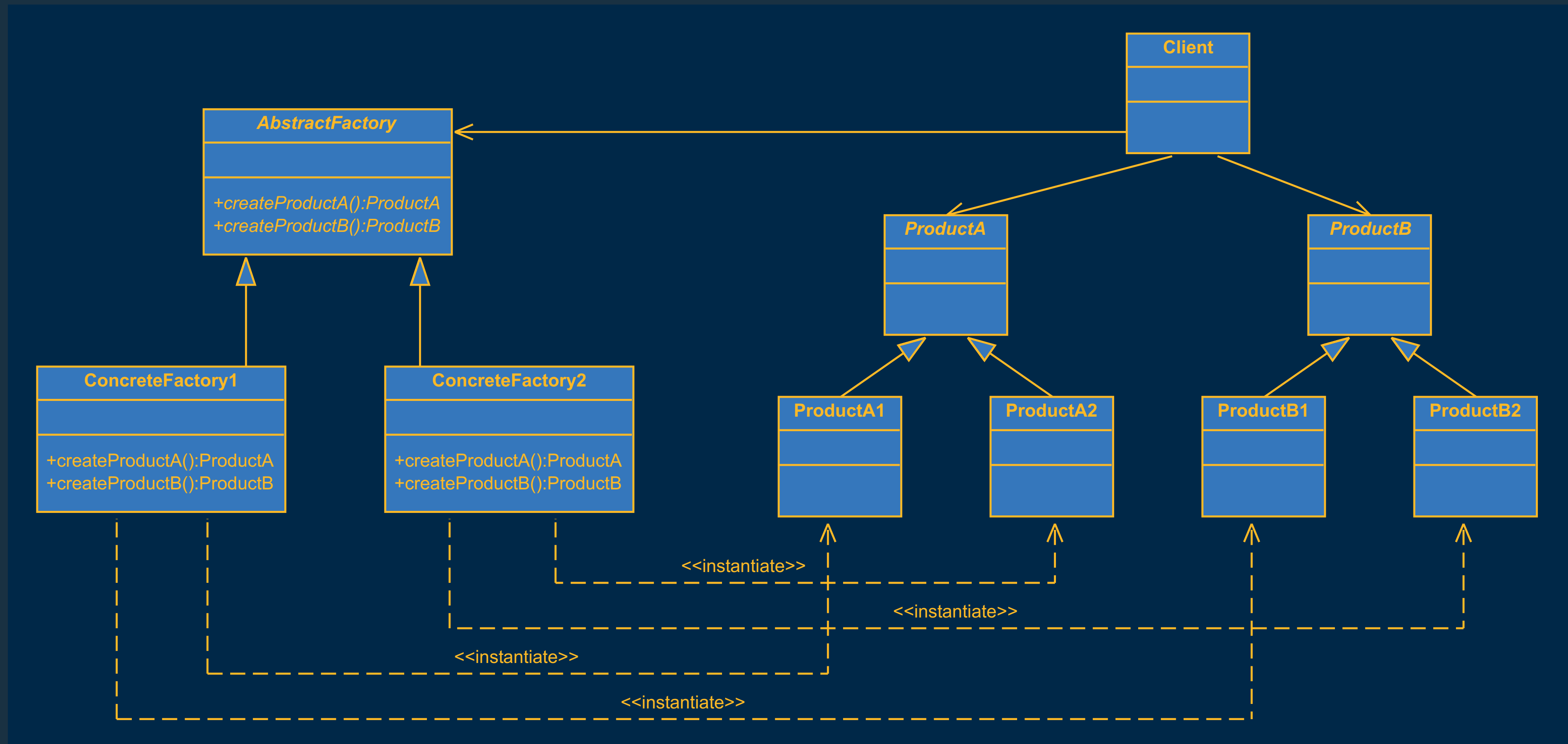
Not just a cute trick: Alexandrescu shows some startling meta-programming paradigms (such as the *generic abstract factory*).

2.3. Primes at compile-time (after Unruh)

```
template<int p, int d>
  struct noDiv {
  enum {
    ans = (p==2) ||
      ((p%d) &&
      noDiv<(d>2?p:0),d-1>::ans )
  };
};
template<> struct noDiv<0,1> {
  enum { ans = 1 };
};
template<> struct noDiv<0,0> {
  enum { ans = 1 };
};

template<int n> struct isPrime {
  enum { ans = noDiv<n,n-1>::ans };
};
```

```
template <int i> struct D {
  D(void*);
};

template<int n> struct primes {
  primes<n-1> rest;
  void f() {
    D<n> current =
      isPrime<n>::ans ? 1 : 0;
    rest.f();
  }
};
template<> struct primes<1> {
  void f() { D<1> current = 0; }
};

main() {
  primes<10> junk; junk.f();
}
```

2.4. Generic abstract factory (Alexandrescu)



2.5. The problem with templates

A class template is *not a class*: it cannot be manipulated until it is instantiated.

There is *no static checking* (other than some syntax checking) of templates. The instances are statically checked, but not the template itself.

A class template is not a formal construct with its own semantics: it is more like a macro. There is *no hope of a theory* of generic programming with templates.

Template meta-programming is extremely awkward. True meta-programming would *support 'programs as data'* as first-class citizens, with perspicuous rather than obscure techniques for manipulation.

3. The algebra of programming

The STL succeeds in providing a generic library of datatypes and algorithms. However, the template mechanism prohibits reasoning about those datatypes and algorithms.

Moreover, the library is rather impoverished: the datatypes are all kinds of sequence. An iterator is a very small window through which to view a data structure. ‘The moving finger writes, and having writ, moves on.’

It is impossible using STL to write generic algorithms that exploit the shape of data (parsers, pretty-printers, encoders, marshallers, etc.) One can only write algorithms that ignore the shape or discard it.

We know how to do better.

3.1. Lists in Haskell

```
> data List a = Nil | Cons (a, List a)

> foldL :: b -> ((a,b)->b) -> List a -> b
> foldL e f Nil              = e
> foldL e f (Cons (a,x)) = f (a, foldL e f x)

> mapL :: (a->b) -> List a -> List b
> mapL f = foldL Nil (\ (a,y) -> Cons (f a,y))
```

3.2. Binary trees in Haskell

```
> data Tree a = Tip a | Bin (Tree a, Tree a)

> foldT :: (a->b) -> ((b,b)->b) -> Tree a -> b
> foldT f g (Tip a)      = f a
> foldT f g (Bin (t,u)) = g (foldT f g t, foldT f g u)

> mapT :: (a->b) -> Tree a -> Tree b
> mapT f = foldT (Tip . f) Bin
```

3.3. Generic definitions in Haskell

```
> class Bifunctor h where bimap :: ...
> data Bifunctor h => Fix h a = In (h a (Fix h a))

> foldF :: Bifunctor h => (h a b->b) -> Fix h a -> b
> foldF f (In x) = f (bimap (id, foldF f) x)

> mapF :: Bifunctor h => (a->b) -> Fix h a -> Fix h b
> mapF f = foldF (In . bimap (f,id))

> data ListF a b = NilF | ConsF (a,b)
> instance Bifunctor ListF where ...
> type List’ a = Fix ListF a

> data TreeF a b = TipF a | BinF (b,b)
> instance Bifunctor TreeF where ...
> type Tree’ a = Fix TreeF a
```

3.4. Generic Haskell

Standard Haskell suffices for *higher-order parametrically-polymorphic* definitions like `foldF` and `mapF`. These preserve shape, but cannot manipulate it.

More generally, we want to *compute with shape* (for example, to define a generic marshaller and unmarshaller between complex datatypes to bit-sequences).

That entails the ability to *inspect* datatype definitions, as provided in Generic Haskell:

```
map<Unit> ()              = ()
map<Const a> x            = x
map<:+:> f g (Inl u) = Inl (f u)
map<:+:> f g (Inr v) = Inr (g v)
map<:*:> f g (u,v)   = (f u,g v)
```

4. Datatype-generic programming

We already have a good understanding of the theory of *first-order parametric polymorphism* (such as `foldL` and `foldT`), and of *higher-order parametric polymorphism* (such as `foldF`).

We are making some steps forward in programming using *higher-order ad-hoc polymorphism* (for example, in Generic Haskell). But we know very little of the theory underlying this.

We call this kind of parametrization *datatype-generic programming*. (We considered talking about *type-parametrized–type—parametrized programs*.)

An understanding of this theory would allow us to treat datatype-generic programs (the analogue of template meta-programs) as formal constructs.

We would then have a sound basis for manipulating them:
static checking, reasoning, derivation, etc.

5. Two cultures

Proceedings of the IFIP TC2 Working Conference on Generic Programming, J. Gibbons and J. Jeuring (eds), Kluwer Academic Publishers, 2003.

1. *Generic Programming within Dependently-Typed Programming*
2. *Generic Haskell, Specifically*
3. *Generic Accumulations*
4. *A Generic Algorithm for Minimum Chain Partitioning*
5. *Concrete Generic Functionals*
6. *Making the Usage of STL Safe*
7. *Static Data Structures*
8. *Adaptive Extensions of Object-Oriented Systems*
9. *Complete Traversals as General Iteration Patterns*
10. *Efficient Implementation of Run-Time Generic Types for Java*

5.1. . . . and maybe the twain should meet?

I believe that the *template meta-programming* and the *algebra of programming* worlds could each teach the other a thing or two:

- template meta-programming sorely needs the sound theoretical foundations that the algebra of programming provides;
- the algebra of programming could do with a corpus of examples and case studies to motivate development.

5.2. Patterns generically

Perhaps certain *design patterns* form a suitable meeting point?

For example, the STL *input iterator* concept roughly corresponds to the ITERATOR design pattern. There ought to be a COITERATOR design pattern, corresponding to *output iterators*. These are analogues of functions that yield the *contents* of a collection, and that generate a collection (of some otherwise-determined shape) from such contents.

The VISITOR pattern is approximately the OO equivalent of the AoP `fold`. It provides a means to traverse a data structure, accumulating some result.

We might be able to bridge the gap by exploring the relationships between the OO and the AoP concepts.

6. Looking forward

The *Datatype-Generic Programming* project starts on 1st August 2003.

I have a DPhil studentship to offer, working on the ideas described here:

- modelling current practice with generic meta-programming
- developing a methodology for constructing datatype-generic programs
- feeding back into practice by providing tool support

Roland Backhouse at Nottingham has a postdoc position to offer, working on more theoretical aspects:

- modular specifications via parametrized signatures
- higher-order naturality properties
- generic theories of termination and well-founded datatypes