Generic Programming in Context

Johan Jeuring

Utrecht University

May 21, 2015

Introduction

Generic Programming

- Making programming languages more flexible without compromising safety
- Means different things to different people, because they have different ideas about combining flexibility and safety
- Possible interpretations: parametric polymorphism, libraries of algorithms and data structures, reflection and meta-programming, etc.

“Generic”

Question

When is something generic?

- “Generic” is an over-used adjective in computer science.
- Ada has generic packages, Java has generics, Eiffel has generic classes, etc.
- Usually, the adjective “generic” is used to describe a concept that allows abstraction over a larger class of entities than previously possible.

Outline

Genericity by Value

Genericity by Value (1)

Draw ASCII pictures:

```
* * * *
* * *
* *
*
```

First attempt:

```
triangle₁ = do
  putStrLn " * * * *"
  putStrLn " * * *"
  putStrLn " * *"
  putStrLn " *"
```

Genericity by Value (2)

Better attempt:

```
triangle₂ 0 = return ()
triangle₂ n = do
   line₂ n
   triangle₂ (pred n)
line₂ 0 = putStrLn ""
line₂ n = do
   putStr " *"
   line₂ (pred n)
```

Genericity by value is implemented by means of a function (procedure, method, subroutine, etc.).

Genericity by Type

Example (1)

$\mathbf{data}\ \mathsf{List_I} = \mathsf{Nil_I} \mid \mathsf{Cons_I}\ \mathsf{Int}\ \mathsf{List_I}$

$\mathsf{append_I} :: \mathsf{List_I} \to \mathsf{List_I} \to \mathsf{List_I}$
$\mathsf{append_I}\ \mathsf{Nil_I}\ \mathsf{ys} = \mathsf{ys}$
$\mathsf{append_I}\ (\mathsf{Cons_I}\ \mathsf{x}\ \mathsf{xs})\ \mathsf{ys} = \mathsf{Cons_I}\ \mathsf{x}\ (\mathsf{append_I}\ \mathsf{xs}\ \mathsf{ys})$

Example (2)

$$\textbf{data}\ \mathsf{List}_C = \mathsf{Nil}_C \mid \mathsf{Cons}_C\ \mathsf{Char}\ \mathsf{List}_C$$

$$\begin{aligned}
&\mathsf{append}_C :: \mathsf{List}_C \to \mathsf{List}_C \to \mathsf{List}_C \\
&\mathsf{append}_C\ \mathsf{Nil}_C\ ys && = ys \\
&\mathsf{append}_C\ (\mathsf{Cons}_C\ x\ xs)\ ys && = \mathsf{Cons}_C\ x\ (\mathsf{append}_C\ xs\ ys)
\end{aligned}$$

Parametric Polymorphism

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

```
append :: List a → List a → List a
append Nil ys            = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

This is a parametrically polymorphic function. It **cannot** depend on the (parameterized) type of its parameters. Why?

Free Theorems

The fact that **a parametric polymorphic function cannot depend on the type of its parameters** has a nice consequence: it satisfies a free theorem.

For example, given $\mathsf{map} :: (a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b$, we know that

$$\mathsf{append}\ (\mathsf{map}\ f\ xs)\ (\mathsf{map}\ f\ ys) \equiv \mathsf{map}\ f\ (\mathsf{append}\ xs\ ys)$$

Inclusion Polymorphism

Another form of type genericity is inclusion polymorphism.

```
class Shape { ... void draw(); ... }
class Circle extends Shape { ... }
class Rect extends Shape { ... }

class ... { void drawShape(Shape s){ s.draw(); } }
```

- `drawShape` takes a parameter of possibly different types.
- The parameter's type must be a subtype of `Shape`.
- `drawShape` only knows about the parameter's fields in `Shape` (modulo casting).
- Inclusion polymorphism is typically found with subtyping in object-oriented programming languages.

Polymorphism in Programming Languages

- Haskell and ML: parametric
- Java and C#: inclusion
- Generics add parametric polymorphism to Java and C#
- Other languages combine these in different ways: see Ada, Scala, Timber

Genericity by Function

Example

Write two very similar functions using `toUpper` `toLower` from `Data.Char`

$$
\begin{aligned}
&\mathsf{upper}_1\ \mathsf{Nil} &&= \mathsf{Nil}\\
&\mathsf{upper}_1\ (\mathsf{Cons}\ x\ xs) &&= \mathsf{Cons}\ (\mathsf{toUpper}\ x)\ (\mathsf{upper}_1\ xs)\\
&\mathsf{lower}_1\ \mathsf{Nil} &&= \mathsf{Nil}\\
&\mathsf{lower}_1\ (\mathsf{Cons}\ x\ xs) &&= \mathsf{Cons}\ (\mathsf{toLower}\ x)\ (\mathsf{lower}_1\ xs)
\end{aligned}
$$

Or use a higher-order function

$$
\begin{aligned}
&\mathsf{map} :: (a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b\\
&\mathsf{map}\ f\ \mathsf{Nil} = \mathsf{Nil}\\
&\mathsf{map}\ f\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ (f\ x)\ (\mathsf{map}\ f\ xs)\\
&\mathsf{upper}_2 = \mathsf{map}\ \mathsf{toUpper}\\
&\mathsf{lower}_2 = \mathsf{map}\ \mathsf{toLower}
\end{aligned}
$$

Note that parametric polymorphism fits well with higher-order functions.

Another Example (1)

These functions are also very similar

$$
\begin{aligned}
&\mathsf{sum}_1 && \mathsf{Nil} &&= 0\\
&\mathsf{sum}_1 && (\mathsf{Cons}\ \mathsf{x}\ \mathsf{xs}) &&= \mathsf{x} + \mathsf{sum}_1\ \mathsf{xs}\\
&\mathsf{concat}_1 && \mathsf{Nil} &&= \mathsf{Nil}\\
&\mathsf{concat}_1 && (\mathsf{Cons}\ \mathsf{x}\ \mathsf{xs}) &&= \mathsf{append}\ \mathsf{x}\ (\mathsf{concat}_1\ \mathsf{xs})
\end{aligned}
$$

We can use $\mathsf{fold}_{\mathsf{List}}$ (a.k.a. foldr) to define both

$$
\begin{aligned}
&\mathsf{fold}_{\mathsf{List}} :: (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{List}\ \mathsf{a} \to \mathsf{b}\\
&\mathsf{fold}_{\mathsf{List}}\ \mathsf{c}\ \mathsf{n}\ \mathsf{Nil} = \mathsf{n}\\
&\mathsf{fold}_{\mathsf{List}}\ \mathsf{c}\ \mathsf{n}\ (\mathsf{Cons}\ \mathsf{x}\ \mathsf{xs}) = \mathsf{c}\ \mathsf{x}\ (\mathsf{fold}_{\mathsf{List}}\ \mathsf{c}\ \mathsf{n}\ \mathsf{xs})\\
&\mathsf{sum}_2 = \mathsf{fold}_{\mathsf{List}}\ (+)\ 0\\
&\mathsf{concat}_2 = \mathsf{fold}_{\mathsf{List}}\ \mathsf{append}\ \mathsf{Nil}
\end{aligned}
$$

Instances of $\mathsf{fold}_{\mathsf{List}}$ replace the list constructors Nil and Cons with supplied arguments.

Another Example (2)

In fact, the following are also instances of $\mathsf{fold}_{\mathsf{List}}$:

$$
\begin{aligned}
&\mathsf{append}\ \mathsf{xs}\ \mathsf{ys} &&= \mathsf{fold}_{\mathsf{List}}\ \mathsf{Cons}\ \mathsf{ys}\ \mathsf{xs} \\
&\mathsf{map}\ \mathsf{f} &&= \mathsf{fold}_{\mathsf{List}}\ (\mathsf{Cons} \circ \mathsf{f})\ \mathsf{Nil}
\end{aligned}
$$

Genericity by Structure

C++ Templates

- Perhaps the most popular use of the term "generic programming" is with C++ templates.
- Class/function templates are parametrized by type/value parameters.

```
template<class T> void swap(T& a, T& b) {
  T c(a); a = b; b = c;
}
```

- Instantiating a template results in the C++ compiler generating specialized code for the given parameters.
  - Aside: As C++ grew from C, the community continued to require the highest performance from its code. C++ developers put a strong emphasis on templates imposing no performance penalty.

C++ Standard Template Library

- The C++ Standard Template Library (STL) uses templates to provide "generic" containers and algorithms.
- The containers provided in the STL are parametrically polymorphic datatypes.
  - sequence containers: e.g. `vector`, `list`, and `deque`
  - associative containers: e.g. `set` and `map`

STL Iterators (1)

- Containers support a common mechanism for accessing their elements: iterators.
- The iterator is a generalization of the pointer.

```
int a[100];
int n = 100;
...
for (int* p = a; p != a+n; ++p)
  printf("%d", *p);
```

```
vector<int> v;
...
for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)
  printf("%d", *i);
```

STL Iterators (2)

Iterator Classifications

input	- one-way, read-only
output	- one-way, write-only
forward	- sequential access, one-way
bidirectional	- sequential access, two-way
random access	- pointer arithmetic

STL Iterators (3)

- Iterators form the interface between container types and algorithms over data structures.
- These include many general-purpose operations such as searching, sorting, and filtering.
- Rather than operating directly on a container, an algorithm operates on iterators.
- The algorithm is generic, in the sense that it applies to any container that supports the appropriate kind of iterator.

```
template<class T, class U> void sort(T first, T last, U comp);
```

Concepts

```
template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          Compare comp);
```

- The exact set of requirements on parameters is called a concept.
- A concept encapsulates the operations required of a formal type parameter and provided by an actual type parameter.
- For example, the STL's input iterator concept encompasses pointer-like types which support comparison for equality, copying, assignment, dereferencing as an r-value, and incrementing.
- The success of the STL lies in the careful choice of such concepts as an organizing principle for a large library.

Concepts cannot be defined in C++!

It is an informal artifact and not a formal construct.

Concepts in Haskell

- In Haskell, we can define a concept with a type class.

$$\text{sort} :: (\text{Ord a}) \Rightarrow \text{List a} \rightarrow \text{List a}$$

- $\text{Ord a} \Rightarrow$ is a type class context.
- sort is not parametrically polymorphic: it is not applicable to all list element types, only those in the type class Ord.
- Ord includes exactly those types that support $\leqslant$:

```
class Ord a where
  (⩽) :: a → a → Bool
```

Instantiating Concepts

- Numerous types are instances of the type class:

instance Ord Integer **where**
$m \leqslant n = \text{isNonNegative}\ (n - m)$

- Attempting to apply $\leqslant$ to two values of some type that is not in the type class Ord, or sort to a list of such values, is a type error, and is caught statically.
- In contrast, while the equivalent error using the C++ concept is still a statically-caught type error, it is caught at template instantiation time, since there is no way of declaring the template's dependence on the concept.

Polymorphism in Concepts

- Concepts in C++ and Haskell serve as a kind of polymorphism
- Not parametric polymorphism: demonstrated
- Not inclusion polymorphism: why?
- Ad-hoc polymorphism
  - "ad-hoc" - non-uniform, heterogeneous
  - There is no requirement by the type system on the implementation of a concept.
  - In Haskell, the implementation of $(\leqslant) :: (\mathrm{Ord}\ a) \Rightarrow a \rightarrow a \rightarrow \mathrm{Bool}$ only depends on the type. It can be implemented in different ways for different types.

Genericity by Property

Properties for Concepts

- Structural genericity is often not enough.
- For example, `Ord` should define a partial order (reflexivity, antisymmetry, transitivity)
- A property is a statement that (usually) cannot be specified directly in programs.
  - External tests can check properties (e.g. using QuickCheck), though these are usually not conclusive.
  - Some languages have (a) explicit support for properties or (b) interesting type systems that allow properties to be defined and verified.

Example: Functors in Haskell

- The generalized type class for `map`:

$$\textbf{class}\ \mathrm{Functor}\ f\ \textbf{where}$$
$$\quad \mathrm{fmap} :: (a \to b) \to f\ a \to f\ b$$

- We expect instances like this:

$$\textbf{instance}\ \mathrm{Functor}\ \mathrm{List}\ \textbf{where}$$
$$\quad \mathrm{fmap} = \mathrm{map}$$

- But, informally, we also ex-
pect the instances to obey the following properties (the "functor laws"):

$$\begin{aligned} \mathrm{fmap}\ (f \circ g) &\equiv \mathrm{fmap}\ f \circ \mathrm{fmap}\ g \\ \mathrm{fmap}\ \mathrm{id} &\equiv \mathrm{id} \end{aligned}$$

Example: Monads in Haskell

- Monad is yet another specification for a concept: computation with impure effects.

$$\textbf{class}\ (\mathrm{Functor}\ m) \Rightarrow \mathrm{Monad}\ m\ \textbf{where}$$
$$\quad \mathrm{return} :: a \rightarrow m\ a$$
$$\quad (\ggg\!=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$

- Example: the state monad, in which a "computation affecting a state of type s" amounts to a function of type $s \rightarrow (a, s)$:

$$\textbf{newtype}\ \mathrm{State}\ s\ a = \mathrm{St}\ \{\mathrm{runSt} :: s \rightarrow (a, s)\}$$
$$\textbf{instance}\ \mathrm{Functor}\ (\mathrm{State}\ s)\ \textbf{where}$$
$$\quad \mathrm{fmap}\ f\ mx = \mathrm{St}\ (\lambda s \rightarrow \textbf{let}\ (a, s') = \mathrm{runSt}\ mx\ s\ \textbf{in}\ (f\ a, s'))$$
$$\textbf{instance}\ \mathrm{Monad}\ (\mathrm{State}\ s)\ \textbf{where}$$
$$\quad \mathrm{return}\ a = \mathrm{St}\ (\lambda s \rightarrow (a, s))$$
$$\quad mx \ggg\!= k = \mathrm{St}\ (\lambda s \rightarrow \textbf{let}\ (a, s') = \mathrm{runSt}\ mx\ s\ \textbf{in}\ \mathrm{runSt}\ (k\ a)\ s')$$

Example: Monads Laws

- Since `Functor` is a superclass of `Monad`, we expect the properties of `Functor` to be inherited.
- Additionally, a `Monad` instance must satisfy the following laws:

$$
\begin{array}{lll}
\mathrm{return}\ a \ggg k & \equiv k\ a & \text{-- left unit} \\
m \ggg \mathrm{return} & \equiv m & \text{-- right unit} \\
m \ggg (\lambda x \rightarrow k\ x \ggg h) & \equiv (m \ggg k) \ggg h & \text{-- associative}
\end{array}
$$

Question

How do you verify the monad laws?

Genericity by Stage

Metaprogramming

- metaprogramming - constructing programs that write or manipulate other programs
- Examples
  - Program generation: generating source code
    - lex, yacc
  - Reflection: observing and modifing a program's structure and behaviour
    - Java, C#, JavaScript, Smalltalk
  - Multi-stage programming: partitioning computation into phases
    - MetaOCaml, Template Haskell
  - A compiler could also be considered a generative metaprogram, though this is not usually the case.

Example: C++ Templates

- The C++ template mechanism provides a metaprogramming facility.
- Template instantiation takes place at compile time, so one can think of a C++ program with templates as a two-stage computation.
- Some high-performance numerical libraries rely on these generative properties.
- The template instantiation mechanism is Turing complete: you can determine if a number is a prime at compile time!
  - `http://homepage.mac.com/sigfpe/Computing/peano.html`

Genericity by Shape

Fold: Again

- Consider the polymorphic datatype of binary trees:

$$\textbf{data}\ \mathsf{Tree\ a} = \mathsf{Tip\ a} \mid \mathsf{Bin\ (Tree\ a)\ (Tree\ a)}$$

- A natural pattern of recursion on these trees (recall $\mathsf{fold}_{\mathsf{List}}$):

$$\begin{aligned}
&\mathsf{fold}_{\mathsf{Tree}} :: (\mathsf{a} \to \mathsf{b}) \to (\mathsf{b} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{Tree\ a} \to \mathsf{b}\\
&\mathsf{fold}_{\mathsf{Tree}}\ \mathsf{t\ b\ (Tip\ x)} \quad\quad\ = \mathsf{t\ x}\\
&\mathsf{fold}_{\mathsf{Tree}}\ \mathsf{t\ b\ (Bin\ xs\ ys)} = \mathsf{b}\ (\mathsf{fold}_{\mathsf{Tree}}\ \mathsf{t\ b\ xs})\ (\mathsf{fold}_{\mathsf{Tree}}\ \mathsf{t\ b\ ys})
\end{aligned}$$

Fold: Instances

- As with $\text{fold}_{\text{List}}$, instances of $\text{fold}_{\text{Tree}}$ replace the datatype's constructors Tip and Bin with supplied functions:

$$
\begin{aligned}
&\text{reverse}_{\text{Tree}} :: \text{Tree a} \to \text{Tree a}\\
&\text{reverse}_{\text{Tree}} = \text{fold}_{\text{Tree}}\ \text{Tip}\ (\text{flip Bin})\\
&\text{flatten}_{\text{Tree}} :: \text{Tree a} \to \text{List a}\\
&\text{flatten}_{\text{Tree}} = \text{fold}_{\text{Tree}}\ (\text{flip Cons Nil})\ \text{append}
\end{aligned}
$$

- Note: $\text{flip} :: (\text{a} \to \text{b} \to \text{c}) \to \text{b} \to \text{a} \to \text{c}$

Fold: Similarities

$$\mathsf{fold}_{\mathsf{List}} :: (a \to b \to b) \to b \to \mathsf{List}\ a \to b$$
$$\mathsf{fold}_{\mathsf{Tree}} :: (a \to b) \to (b \to b \to b) \to \mathsf{Tree}\ a \to b$$

- Parametric polymorphism unifies commonality of computation, abstracting over variability in irrelevant types.
- Higher-order functions unify commonality of program construction, abstracting over variability in some of the details.
- How can we unify the higher-order, polymorphic functions $\mathsf{fold}_{\mathsf{List}}$ and $\mathsf{fold}_{\mathsf{Tree}}$?

DGP to the Rescue

- What differs between $\mathrm{fold}_{\mathrm{List}}$ and $\mathrm{fold}_{\mathrm{Tree}}$ is the shape of the data on which they operate.
- We have come to call this approach to generic programming datatype-generic programming or DGP.

DGP Example (1)

One approach to abstracting over the shape of `List` and `Tree`:

$$\textbf{data}\ \mathsf{List}\ a = \mathsf{Nil} \mid \mathsf{Cons}\ a\ (\mathsf{List}\ a)$$
$$\textbf{data}\ \mathsf{List}_{\mathsf{F}}\ a\ r = \mathsf{Nil}_{\mathsf{F}} \mid \mathsf{Cons}_{\mathsf{F}}\ a\ r$$

$$\textbf{data}\ \mathsf{Tree}\ a = \mathsf{Tip}\ a \mid \mathsf{Bin}\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a)$$
$$\textbf{data}\ \mathsf{Tree}_{\mathsf{F}}\ a\ r = \mathsf{Tip}_{\mathsf{F}}\ a \mid \mathsf{Bin}_{\mathsf{F}}\ r\ r$$

Now that we have parameterized over the repeated types, we can fill them back in.

$$\textbf{data}\ \mathsf{Fix}\ f = \mathsf{In}\ \{\mathsf{out} :: f\ (\mathsf{Fix}\ f)\}$$

$$\textbf{type}\ \mathsf{List}'\ a = \mathsf{Fix}\ (\mathsf{List}_{\mathsf{F}}\ a)$$
$$\textbf{type}\ \mathsf{Tree}'\ a = \mathsf{Fix}\ (\mathsf{Tree}_{\mathsf{F}}\ a)$$

DGP Example (2)

- The usefulness of Fix ?
- We only need to define fold once:

$$\mathsf{fold} :: (\mathsf{Functor}\ f) \Rightarrow (f\ c \rightarrow c) \rightarrow \mathsf{Fix}\ f \rightarrow c$$
$$\mathsf{fold}\ f = f \circ \mathsf{fmap}\ (\mathsf{fold}\ f) \circ \mathsf{out}$$

- Though we do need an instance of Functor for each datatype.

$$\textbf{instance}\ \mathsf{Functor}\ (\mathsf{List}_{\mathsf{F}}\ a)\ \textbf{where} \ldots$$
$$\textbf{instance}\ \mathsf{Functor}\ (\mathsf{Tree}_{\mathsf{F}}\ a)\ \textbf{where} \ldots$$

DGP Example (3)

- The instances of fold are straightforward.

$$
\begin{aligned}
&\mathsf{sum}_{\mathsf{List}}\ \mathsf{xs} = \mathsf{fold}\ \mathsf{f} \\
&\quad \textbf{where}\ \mathsf{f}\ \mathsf{Nil}_{\mathsf{F}} \qquad\qquad = 0 \\
&\qquad\qquad\ \ \mathsf{f}\ (\mathsf{Cons}_{\mathsf{F}}\ \mathsf{n}\ \mathsf{r}_1) = \mathsf{n} + \mathsf{r}_1 \\
&\mathsf{sum}_{\mathsf{Tree}}\ \mathsf{xs} = \mathsf{fold}\ \mathsf{f} \\
&\quad \textbf{where}\ \mathsf{f}\ (\mathsf{Tip}_{\mathsf{F}}\ \mathsf{n}) \qquad = \mathsf{n} \\
&\qquad\qquad\ \ \mathsf{f}\ (\mathsf{Bin}_{\mathsf{F}}\ \mathsf{r}_1\ \mathsf{r}_2)\ \ = \mathsf{r}_1 + \mathsf{r}_2
\end{aligned}
$$

- We can do better! (And we will see how.)
- For example, with other approaches to DGP, we can define a single sum function instead of $\mathsf{sum}_{\mathsf{List}}$ and $\mathsf{sum}_{\mathsf{Tree}}$.

Conclusion

- There are many interpretations of genericity.
- Each kind of genericity is useful.
- We will focus on datatype-generic programming.