

# Multi-Language Library Development

## Introduction to Generic Programming in C++

**Marcin Zalewski**<sup>1</sup>, Andreas Priesnitz<sup>1</sup>, Cezar Ionescu<sup>2</sup>, Nicola Botta<sup>2</sup>, and Sibylle Schupp<sup>1</sup>

<sup>1</sup>Department of Computer Science  
Chalmers University of Technology  
Sweden

<sup>2</sup>Potsdam Institute for Climate Impact Research,  
Potsdam, Germany

July 31, 2007

# An Overview

What, why, where. . . Why am I here?

## The Situation

1. Model software in Haskell
2. Implement software in C++
3. Real project

## The means

- ▶ Manual mapping of implementations

## An idea

C++



Haskell

# An Overview

What, why, where. . . Why am I here?

## The Situation

1. Model software in Haskell
2. Implement software in C++
3. Real project

## The means

- ▶ Manual mapping of implementations

## An idea

C++



Haskell

# An Overview

What, why, where. . . Why am I here?

## The Situation

1. Model software in Haskell
2. Implement software in C++
3. Real project

## The means

- ▶ Manual mapping of implementations

## An idea

C++



Haskell

# An Overview

What, why, where. . . Why am I here?

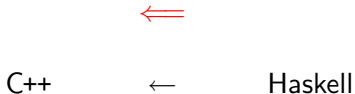
## The Situation

1. Model software in Haskell
2. Implement software in C++
3. Real project

## The means

- ▶ Manual mapping of implementations

## An idea



# An Overview

What, why, where. . . Why am I here?

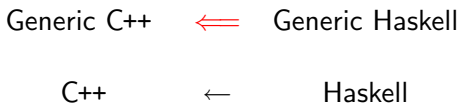
## The Situation

1. Model software in Haskell
2. Implement software in C++
3. Real project

## The means

- ▶ Manual mapping of implementations

## An idea



# An Overview

What, why, where. . . Why am I here?

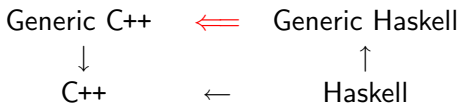
## The Situation

1. Model software in Haskell
2. Implement software in C++
3. Real project

## The means

- ▶ Manual mapping of implementations

## An idea



# Two Paradigms—Two different mindsets

Haskell and C++

## Differences

C++:

- ▶ Mostly imperative
- ▶ Mostly strict
- ▶ Not pure

Haskell:

- ▶ Functional
- ▶ Lazy
- ▶ Pure

## Generic programming—a common denominator

C++:

- ▶ Constrained templates

Haskell:

- ▶ Ad-hoc polymorphism with type classes

# Two Paradigms—Two different mindsets

Haskell and C++

## Differences

C++:

- ▶ Mostly imperative
- ▶ Mostly strict
- ▶ Not pure

Haskell:

- ▶ Functional
- ▶ Lazy
- ▶ Pure

## Generic programming—a common denominator

C++:

- ▶ Constrained templates

Haskell:

- ▶ Ad-hoc polymorphism with type classes

# Generic Code in Haskell and C++

## The (very) basic idea

```
concept DiscreteFunction<F> {  
  typename a_t;  
  typename b_t;  
  b_t value(F, a_t);  
}
```

```
class DiscreteFunction f a b  
  | f -> a b where  
  value :: f -> a -> b
```

# Generic Code in Haskell and C++

## The (very) basic idea

```
concept DiscreteFunction<F> {  
    typename a_t;  
    typename b_t;  
    b_t value(F, a_t);  
}
```

```
template<typename T>  
concept_map DiscreteFunction<vector<T>> {  
    typedef int a_t;  
    typedef vector<T>::value_type b_t;  
    b_t value(vector<T> v, a_t a) {  
        return v[a];  
    }  
}
```

```
class DiscreteFunction f a b  
    | f -> a b where  
    value :: f -> a -> b
```

```
instance DiscreteFunction  
    [a] Int a where  
    value fs i = fs !! i
```

# Generic Code in Haskell and C++

## The (very) basic idea

```
concept DiscreteFunction<F> {
  typename a_t;
  typename b_t;
  b_t value(F, a_t);
}
```

```
template<typename T>
concept_map DiscreteFunction<vector<T>> {
  typedef int a_t;
  typedef vector<T>::value_type b_t;
  b_t value(vector<T> v, a_t a) {
    return v[a];
  }
}
```

```
template<typename F>
requires DiscreteFunction<F>
b_t evaluate(F f, a_t a) {
  return value(f, a);
}
```

```
class DiscreteFunction f a b
  | f -> a b where
  value :: f -> a -> b
```

```
instance DiscreteFunction
  [a] Int a where
  value fs i = fs !! i
```

```
evaluate f a = value f a
```

# Structuring

## Taxonomies of abstractions

- ▶ One idea builds on another
  - ▶ **context** for type classes
  - ▶ **refinement** and **requirements** for concepts

### Context

```
class Functor f => LambdaRelation r a b f ...
```

### Refinement and requirements

```
concept LambdaRelation<R, F> : Functor<F> ...
```

```
concept LambdaRelation<R> {  
  typename f_t;  
  require Functor<f_t>;  
}
```

# Structuring

## Taxonomies of abstractions

- ▶ One idea builds on another
  - ▶ **context** for type classes
  - ▶ **refinement** and **requirements** for concepts

## Context

```
class Functor f => LambdaRelation r a b f ...
```

## Refinement and requirements

```
concept LambdaRelation<R, F> : Functor<F> ...
```

```
concept LambdaRelation<R> {  
  typename f_t;  
  require Functor<f_t>;  
}
```

# Structuring

## Taxonomies of abstractions

- ▶ One idea builds on another
  - ▶ **context** for type classes
  - ▶ **refinement** and **requirements** for concepts

## Context

```
class Functor f => LambdaRelation r a b f ...
```

## Refinement and requirements

```
concept LambdaRelation<R, F> : Functor<F> ...
```

```
concept LambdaRelation<R> {  
  typename f_t;  
  require Functor<f_t>;  
}
```

# Outline

## Translation

Haskell

C++

## Further Translation

Promising Directions

Constructor Classes

## Conclusions and Questions

# An Algorithm

## The non-generic version

- ▶ A real software pattern: relation based algorithms
- ▶ Specified in Haskell
- ▶ As often done, express things conveniently as lists

```
rba :: [[Int]] -> [a] -> ([a] -> b) -> [Int] -> [b]
rba rss fs h js = [ h [ fs!!i | i <- rss!!j ] | j <- js ]
```

- ▶ How to translate the prototype to C++?
  - ▶ The problem is too specific to answer in general

# An Algorithm

## The non-generic version

- ▶ A real software pattern: relation based algorithms
- ▶ Specified in Haskell
- ▶ As often done, express things conveniently as lists

```
rba :: [[Int]] -> [a] -> ([a] -> b) -> [Int] -> [b]
rba rss fs h js = [ h [ fs!!i | i <- rss!!j ] | j <- js ]
```

- ▶ How to translate the prototype to C++?
  - ▶ The problem is too specific to answer in general

# Lifting the Algorithm

The devil is in the details...

## Question

- ▶ What is it that we represent with each of the lists?
- ▶ What do we actually need to know?

## Abstractions: the Answer

- ▶ A set of values to map over
- ▶ A relation
- ▶ A discrete function

```
class Functor ...
  map ...
class Functor ... => LambdaRelation ...
  lambda ...
class DiscreteFunction f a b | f -> a b where
  value :: f -> a -> b
```

# Lifting the Algorithm

The devil is in the details...

## Question

- ▶ What is it that we represent with each of the lists?
- ▶ What do we actually need to know?

## Abstractions: the Answer

- ▶ A set of values to map over
- ▶ A relation
- ▶ A discrete function

```
class Functor ...
  map ...
class Functor ... => LambdaRelation ...
  lambda ...
class DiscreteFunction f a b | f -> a b where
  value :: f -> a -> b
```

# Lifting the Algorithm

The devil is in the details...

## Question

- ▶ What is it that we represent with each of the lists?
- ▶ What do we actually need to know?

## Abstractions: the Answer

- ▶ A set of values to map over
- ▶ A relation
- ▶ A discrete function

```
class Functor ...
  map ...
class Functor ... => LambdaRelation ...
  lambda ...
class DiscreteFunction f a b | f -> a b where
  value :: f -> a -> b
```

# Generic RBA

Using the vocabulary of abstractions

- ▶ Reformulate the algorithm using the abstract vocabulary
  - ▶ lists  $\rightarrow$  functors
  - ▶ lists  $\rightarrow$  functions
  - ▶ lists  $\rightarrow$  relations

```
riba rss fs h js = fmap f js
  where f j = h (fmap g (lambda rss j))
         where g i = value fs i
```

- ▶ Now we can think how to translate the generic algorithm to C++
  - ▶ no need to worry about details

# Generic RBA

Using the vocabulary of abstractions

- ▶ Reformulate the algorithm using the abstract vocabulary
  - ▶ lists  $\rightarrow$  functors
  - ▶ lists  $\rightarrow$  functions
  - ▶ lists  $\rightarrow$  relations

```
rba rss fs h js = fmap f js
  where f j = h (fmap g (lambda rss j))
         where g i = value fs i
```

- ▶ Now we can think how to translate the generic algorithm to C++
  - ▶ no need to worry about details

# Generic RBA

Using the vocabulary of abstractions

- ▶ Reformulate the algorithm using the abstract vocabulary
  - ▶ lists  $\rightarrow$  functors
  - ▶ lists  $\rightarrow$  functions
  - ▶ lists  $\rightarrow$  relations

```
riba rss fs h js = fmap f js
  where f j = h (fmap g (lambda rss j))
         where g i = value fs i
```

- ▶ Now we can think how to translate the generic algorithm to C++
  - ▶ no need to worry about details

# Classes to Concepts

## Moving ideas

- ▶ Haskell type classes are translated to C++ concepts

```
class DiscreteFunction f a b | f -> a b where  
  value :: f -> a -> b
```



```
concept DiscreteFunction<typename F>  
{  
  typename a_t;  
  typename b_t;  
  b_t value(F, a_t);  
};
```

# Classes to Concepts

## Moving ideas

- ▶ Haskell type classes are translated to C++ concepts

```
class DiscreteFunction f a b | f -> a b where  
  value :: f -> a -> b
```



```
concept DiscreteFunction<typename F>  
{  
  typename a_t;  
  typename b_t;  
  b_t value(F, a_t);  
};
```

# Translation Rules

Should you read it all? **No!!!**

$\llbracket \text{class } \textit{superClasses} \Rightarrow C \textit{ typeVars} \mid \textit{funDeps} \rrbracket =$

```
concept  $\tilde{C}$  <reduce(paramTypeVars(typeVars, funDeps), typeParamDecl, ',')>
  : reduce(refinementClasses(paramTypeVars(typeVars, funDeps), superClasses),
           conceptInstance,
           ',')
{
  reduce(assocTypeVars(funDeps), assocTypeDecl,  $\epsilon$ )
  require reduce(requirementClasses(assocTypeVars(funDeps), superClasses),
                 conceptInstance,
                 ',');
}
```

# Translation Rules

Should you read it all? **No!!!**

$\llbracket \text{class } \textit{superClasses} \Rightarrow C \textit{ typeVars} \mid \textit{funDeps} \rrbracket =$

```

concept  $\tilde{C}$  <reduce(paramTypeVars(typeVars, funDeps), typeParamDecl, ',')>
  : reduce(refinementClasses(paramTypeVars(typeVars, funDeps), superClasses),
           conceptInstance,
           ',')
{
  reduce(assocTypeVars(funDeps), assocTypeDecl,  $\epsilon$ )
  require reduce(requirementClasses(assocTypeVars(funDeps), superClasses),
                 conceptInstance,
                 ',');
}

```

# Translation Rules

Should you read it all? **No!!!**

$\llbracket \text{class } \textit{superClasses} \Rightarrow C \textit{ typeVars} \mid \textit{funDeps} \rrbracket =$

```

concept  $\tilde{C}$  <reduce(paramTypeVars(typeVars, funDeps), typeParamDecl, ',')>
  : reduce(refinementClasses(paramTypeVars(typeVars, funDeps), superClasses),
           conceptInstance,
           ',')
{
  reduce(assocTypeVars(funDeps), assocTypeDecl,  $\epsilon$ )
  require reduce(requirementClasses(assocTypeVars(funDeps), superClasses),
                 conceptInstance,
                 ',');
}
  
```

# Translation Rules

## Type classes to concepts

- ▶ Type variables
  - ▶ concept parameters
  - ▶ associated types

```
a -> b --straightforward  
a b -> c, a c -> b --difficult
```

- ▶ Context
  - ▶ refinements
  - ▶ requirements

# Outline

## Translation

Haskell

C++

## Further Translation

Promising Directions

Constructor Classes

## Conclusions and Questions

# Open Issues

## Functor classes and other issues

I want to discuss:

- ▶ Type constructor classes
  - ▶ e.g.,  $(f\ a)$  where  $f :: * \rightarrow *$

Other important issues:

- ▶ How to translate generic algorithms?

# Constructor Classes

What is it?

- ▶ A class of type constructors

```
f :: * -> *
```

- ▶ Allow a view into the structure of a type
- ▶ Allow to “compute” new type

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

# Constructor Classes

What is it?

- ▶ A class of type constructors

```
f :: * -> *
```

- ▶ Allow a view into the structure of a type
- ▶ Allow to “compute” new type

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

# Type Constructors in C++

## A possible solution

Express type constructors by:

- ▶ a concept to pattern match on a “type constructor”
- ▶ a concept for each operation
  - ▶ type parameters of the operation become type parameters of the concept
- ▶ type computations provided in concept maps

Issues:

- ▶ One concept per one type constructor class
- ▶ No enforcement of results in “the same family”
  - ▶ e.g., `fmap :: (a -> b) -> f a -> f b`
- ▶ Where to state the properties of the return type
  - ▶ e.g., is it a Functor or not

# Type Constructors in C++

## A possible solution

### Express type constructors by:

- ▶ a concept to pattern match on a “type constructor”
- ▶ a concept for each operation
  - ▶ type parameters of the operation become type parameters of the concept
- ▶ type computations provided in concept maps

### Issues:

- ▶ One concept per one type constructor class
- ▶ No enforcement of results in “the same family”
  - ▶ e.g., `fmap :: (a -> b) -> f a -> f b`
- ▶ Where to state the properties of the return type
  - ▶ e.g., is it a Functor or not

# Functor in C++

How to examine and construct types?

class Functor f where

fmap :: (a -> b) -> f a -> f b

⇓?

*// Functor constructor*

```
concept FunctorC<typename F> {  
    typename value_type;  
};
```

*// Functor class*

```
concept Functor<typename F, typename Fun>  
: FunctorC<F> {  
    requires Callable1<Fun, value_type>;  
    typename result_type;  
  
    result_type fmap(Fun&, const F&);  
};
```

## Functor in C++

How to examine and construct types?

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

⇓?

```
// Functor constructor
concept FunctorC<typename F> {
  typename value_type;
};

// Functor class
concept Functor<typename F, typename Fun>
: FunctorC<F> {
  requires Callable1<Fun, value_type>;
  typename result_type;

  result_type fmap(Fun&, const F&);
};
```

## Functor in C++

How to examine and construct types?

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

⇓?

```
// Functor constructor
concept FunctorC<typename F> {
  typename value_type;
};

// Functor class
concept Functor<typename F, typename Fun>
: FunctorC<F> {
  requires Callable1<Fun, value_type>;
  typename result_type;

  result_type fmap(Fun&, const F&);
};
```

# Constructor Class instances

Concept maps that provide type computations

```
template<typename Element>
concept_map FunctorC<vector<Element> > {
    typedef Element value_type;
}
```

# Constructor Class instances

Concept maps that provide type computations

```
template<typename Element>
concept_map FunctorC<vector<Element> > {
    typedef Element value_type;
}
```

```
template<typename Element, typename Fun>
requires Callable1<Fun, Element>
concept_map Functor<vector<Element>, Fun> {
    ...
};
```

# Outline

Translation

Haskell

C++

Further Translation

Promising Directions

Constructor Classes

Conclusions and Questions

# Conclusions

## Translating Haskell to C++

- ▶ Generic programming
  - ▶ makes translation easier
  - ▶ allows to leave implementation out
  - ▶ requires translation of abstractions
- ▶ Translation of multi-parameter type classes to concepts
  - ▶ parameters and associated types
  - ▶ refinements and requests
- ▶ Translation of type-constructor classes