



POTSDAM INSTITUTE FOR CLIMATE IMPACT RESEARCH **PIK**

MEMBER OF THE LEIBNIZ ASSOCIATION

# The Function Concept

An empirical study

Daniel Lincke (Potsdam Institute for Climate Impact Research)

and

Sibylle Schupp (Hamburg University of Technology)

August 30th, 2009



Leibniz  
Gemeinschaft



# Function Types in Programming

- ▶ Functional programmers are used to function *types*
- ▶ For instance in Haskell:  $A \rightarrow B$
- ▶ Imperative languages lack native support for function types
- ▶ However, they are the base of generic programming
- ▶ Higher-order functions, for instance, require function types
- ▶ We investigate function types in C++, as this is the imperative language with the best support for generic programming

# Simulate Function Types in C++

- ▶ As function types are not present in C++, they have to be simulated
- ▶ The idea: represent  $A \rightarrow B$  as a class, since classes have types and their instances can be passed around
- ▶ As a multi-paradigm language, C++ offers many techniques for simulating functions
- ▶ For instance: pointers, inheritance, dynamic binding, templates, meta-programming
- ▶ Not yet, but in the future: concepts
- ▶ Concepts are different from the other techniques: concepts are no types, but a feature to specify properties of types

# Comparing Function Types in C++

- ▶ How do the existing implementations compare?
- ▶ Surprisingly, it was not clear at all how they compare
- ▶ In many situations an evaluation is useful
- ▶ What about a function concept? How does it compare to other approaches?
- ▶ Measure it! Evaluate it!

	no optimization	optimization level 3 (-O3)
FPtr	-	-
OO	59.41	<b>43.05</b>
Boost	284.25	<b>123.74</b>
FC++	88.63	<b>72.86</b>
Concept	11.60	1.00

# Outline

Introduction

Functions in C++

Evaluation

Outlook

# Function Types in C++: Methods

Different programming methods lead to different solutions:

- ▶ Function pointers: built-in technique in C++, but cannot be used directly as function *type*; we have to implement a wrapper class
- ▶ Object-oriented programming: implement a virtual base class for a function type
- ▶ Library: use an external library like FC++ or Boost
- ▶ Concepts: every function is implemented by a separate, a function concept is used to specify them as functions

Note:

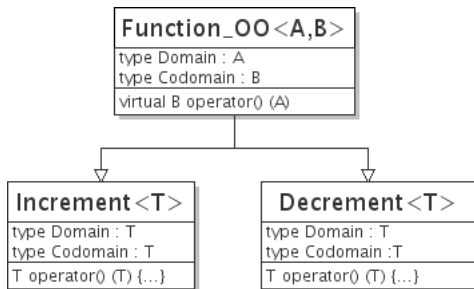
- ▶ All function datatypes will be template classes
- ▶ All function datatypes provide an application operator to enable function-like call syntax

# Function Pointers

Function_FP<A,B>
type Domain : A
type Codomain : B
type B (*fct_ptr)(A)
my_fun : fct_ptr
Function_FP( B (*fct_ptr)(A) )
B operator() (A)

- ▶ Small template class, directly supported
- ▶ Internally, a function pointer is stored as a member
- ▶ Application operator resolves the function pointer at runtime (overhead!)

# Object-Oriented Programming

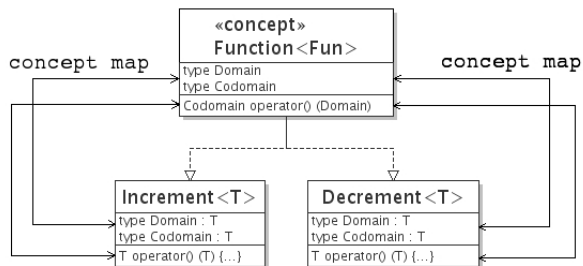


- ▶ Function is an abstract base class
- ▶ One class for every concrete function; inherits from the abstract base class, dynamic binding
- ▶ Application operator resolves virtual call at runtime (overhead!)

# Libraries

- ▶ Several libraries provide function datatypes
- ▶ Most prominent: Boost
- ▶ In addition, we picked the FC++ library
- ▶ How to use: create an instance of a library object and pass along

# Concepts



- ▶ The concept models a static interface
- ▶ One class for every concrete function, declared as a model of the function class with a `concept_map`; static binding
- ▶ Application operator resolved in the `concept_map` at compile time (no overhead!)

# The Function Concept

```
concept Concept_Function<class F> {
    typename Domain;
    typename Codomain;

    Codomain operator()(F&, Domain);
};

template<class T>
concept_map Concept_Function<Increment<T> > {
    typedef T Domain; ++
    typedef T Codomain;
}
```

# Function Application: A measurement

## Evaluation

- ▶ Function application is the basic operation for functions
- ▶ Important to have no overhead for the application operator
- ▶ A simple test: repeated function application. Results:

	no optimization	optimization level 3 (-O3)
Fptr	5.37	2.56
OO	3.84	1.06
Boost	8.91	8.59
FC++	12.42	5.24
Concept	3.73	1.00

# Function Application: A Measurement

Some interpretations:

- ▶ As expected: the concept has no overhead
- ▶ The function class for the object-oriented case is as fast as the function object
- ▶ Function pointer wrappers suffer from pointer indirection costs
- ▶ Library datatypes use a complicated machinery with pointer indirections and virtual calls

# Higher-order Functions: Performance

- ▶ Higher-order functions are a main reason for introducing function datatypes
- ▶ Function application for higher-order functions should be efficient
- ▶ A simple test: repeated higher-order function application. Results:

	no optimization	optimization level 3 (-O3)
Fptr	17.18	5.45
OO	13.82	6.18
Boost	25.19	16.76
FC++	34.50	10.55
Concept	12.92	1.00

# Higher-order Functions: Performance

Some interpretations:

- ▶ As expected: concept performs best
- ▶ The object-oriented solution resolves the virtual application operator call, which leads to an overhead
- ▶ Function pointer wrappers and library datatypes: as before

# Other Evaluation Criteria

- ▶ We are looking further: operations on function types like composition, currying are needed
- ▶ Questions: can these operations be implemented at all?
- ▶ And if so, can they be implemented efficiently?
- ▶ Interestingly, it turns out that not all of them are expressive enough
- ▶ A simple function pointer wrapper, for instance, cannot support composition

## Example: Partial Application

- ▶ We also tested the performance of function application for curried, partially applied functions
- ▶ Results show surprisingly big differences in terms of performance:

	no optimization	optimization level 3 (-O3)
Fptr	-	-
OO	59.41	43.05
Boost	284.25	123.74
FC++	88.63	72.86
Concept	11.60	1.00

- ▶ Once again, the function concept performs best

# Type Declarations

- ▶ The function concept performs very good in all performance evaluations
- ▶ So, where is the rub?
- ▶ The function concept leads to difficult type declarations, compared with other solutions:

```
boost::function<B, A> f1 = ...;  
boost::function<C, B> f2 = ...;  
boost::function<C, A> f1_f2 = boost_compose(f1,f2);
```

# Type Declarations

- ▶ With function objects and concepts:

```
Test_Function1 f1; // f1 : A->B
Test_Function2 f2; // f2 : B->C
Composed_Function<Test_Function1,
                  Test_Function2>
f1_f2 = concept_compose(f1,f2);
```

- ▶ Hard to understand for nested composition / currying

# Evaluation Summary

Feature	Fptr	OO	FC++	Boost	Concept
<b>Application operator</b>	+	-	-	-	+
<b>Higher-order functions</b>	+	-	-	-	+
Function composition	-	+	+	+	+
Function comp., efficiency	-	-	-	-	+
Partial application	-	+	+	+	+
<b>Partial appl., efficiency</b>	-	-	-	-	+
<b>Pretty function types</b>	+	+	+	+	-

# Conclusion

- ▶ We found big differences in terms of performance
- ▶ In particular, the function datatype from the **Boost** library is not suitable for use in scientific applications
- ▶ A function concept is the best solution in terms of performance
- ▶ A function concept is also very general: Every function datatype can be declared as a model of the concept
- ▶ Furthermore it is compatible with the STL

# Work already done or in progress

- ▶ We are aware of the fact that there is a concept `Callable` in the C++0x proposal
- ▶ A discussion of `Callable` vs. our concept design starts in our paper, but still goes on
- ▶ Application: we implemented a concept framework for functional structures like functors and monads using a function concept
- ▶ Currently, we investigate if the presented function types can be used for higher-rank polymorphic functions (`scrap++`)

# Thanks!

## Thank you for your attention.

For further information we refer to the paper and to  
[www.pik-potsdam.de/favaia](http://www.pik-potsdam.de/favaia)