The Function Concept

An empirical study

Daniel Lincke (Potsdam Institute for Climate Impact Research)
and
Sibylle Schupp (Hamburg University of Technology)

Nov. 5, 2009
Function Types in Programming

- Functional programmers are used to function *types*
- For instance in Haskell: A → 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
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!

<table>
<thead>
<tr>
<th></th>
<th>no optimization</th>
<th>optimization level 3 (-O3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPt</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>OO</td>
<td>59.41</td>
<td>43.05</td>
</tr>
<tr>
<td>Boost</td>
<td>284.25</td>
<td>123.74</td>
</tr>
<tr>
<td>FC++</td>
<td>88.63</td>
<td>72.86</td>
</tr>
<tr>
<td>Concept</td>
<td>11.60</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Function Pointers

- Small template class, directly supported
- Internally, a function pointer is stored as a member
- Application operator resolves the function pointer at runtime (overhead!)
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
The concept models a static interface
One class for every concrete function, declared as a model of the function class with a \texttt{concept\_map}; static binding
Application operator resolved in the \texttt{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;
}

Daniel Lincke and Sibylle Schupp  
The Function Concept
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:

<table>
<thead>
<tr>
<th></th>
<th>no optimization</th>
<th>optimization level 3 (-O3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fptr</td>
<td>5.37</td>
<td>2.56</td>
</tr>
<tr>
<td>OO</td>
<td>3.84</td>
<td>1.06</td>
</tr>
<tr>
<td>Boost</td>
<td>8.91</td>
<td>8.59</td>
</tr>
<tr>
<td>FC++</td>
<td>12.42</td>
<td>5.24</td>
</tr>
<tr>
<td>Concept</td>
<td>3.73</td>
<td>1.00</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th></th>
<th>no optimization</th>
<th>optimization level 3 (-O3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fptr</td>
<td>17.18</td>
<td>5.45</td>
</tr>
<tr>
<td>OO</td>
<td>13.82</td>
<td>6.18</td>
</tr>
<tr>
<td>Boost</td>
<td>25.19</td>
<td>16.76</td>
</tr>
<tr>
<td>FC++</td>
<td>34.50</td>
<td>10.55</td>
</tr>
<tr>
<td>Concept</td>
<td>12.92</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Daniel Lincke and Sibylle Schupp
PIK, TU Hamburg
The Function Concept
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:

<table>
<thead>
<tr>
<th></th>
<th>no optimization</th>
<th>optimization level 3 (-O3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fptr</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>OO</td>
<td>59.41</td>
<td>43.05</td>
</tr>
<tr>
<td>Boost</td>
<td>284.25</td>
<td>123.74</td>
</tr>
<tr>
<td>FC++</td>
<td>88.63</td>
<td>72.86</td>
</tr>
<tr>
<td>Concept</td>
<td>11.60</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- Once again, the function concept performs best.
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:

```cpp
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

<table>
<thead>
<tr>
<th>Feature</th>
<th>Fptr</th>
<th>OO</th>
<th>FC++</th>
<th>Boost</th>
<th>Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application operator</strong></td>
<td>+</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td><strong>Higher-order functions</strong></td>
<td>+</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td><strong>Function composition</strong></td>
<td>−</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Function comp., efficiency</strong></td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td><strong>Partial application</strong></td>
<td>−</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Partial appl., efficiency</strong></td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td><strong>Pretty function types</strong></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>−</td>
</tr>
</tbody>
</table>