Virtual Namespace Functions: An Alternative to Virtual Member Functions in C++ and Advice in AspectC++

Christian Heinlein
Dept. of Computer Structures, University of Ulm, Germany
heinlein@informatik.uni-ulm.de

ABSTRACT

Virtual namespace functions (VNFs) are introduced as C++ functions defined at global or namespace scope which can be redefined similar to virtual member functions. Even though this is a relatively simple concept, hardly more complex than ordinary C functions, it is shown that VNFs subsume object-oriented single, multiple, and predicate-based method dispatch as well as aspect-oriented before, after, and around advice. Their implementation by means of a "lazy" precompiler for C++ is briefly described.

1. INTRODUCTION

C++ provides both ordinary functions (defined at global or namespace scope, i. e., outside any class) and member functions belonging to a particular class [13].[1] The latter might be defined `virtual` to support dynamic binding and object-oriented programming, while the former are always bound statically.

A severe limitation of (virtual) member functions (that is also present in other object-oriented languages where member functions are usually called methods) is the fact that they must be declared in the class they belong to, and it is impossible to declare additional ones "later," i. e., in a different place of a program. This leads to the well-known "expression problem" [14], i. e., the fact that it is easy to add new (sub)classes to an existing system, but very hard to add new operations to these classes in a *modular* way, i. e., without touching or recompiling existing source code. The Visitor Pattern [5] has been developed as a workaround to this problem, but its application is rather complicated and must be planned in advance, i. e., it does not help to cope with *unanticipated software evolution*.

On the other hand, ordinary functions, which will be called *namespace functions* in the sequel since they are defined at (global or) namespace scope, can be added to a system in a completely modular way at any time without any problems. However, they suffer from the fact that they are always bound statically (i. e., they cannot be overridden or redefined), which makes it hard to add new subclasses to a system whose operations are implemented as such functions.

Given these complementary advantages and disadvantages of namespace and virtual member functions, it seems very promising to provide *virtual namespace functions* (VNFs, cf. Sec. 2) which combine the advantages of namespace and virtual functions:

- Because they will be defined at (global or) namespace scope, it will always be possible to add new VNFs to an existing system without needing to touch or recompile existing source code.
- Because they will be bound dynamically, it will always be possible to redefine them later if new subclasses have been added to a system, again without needing to touch or recompile existing source code.

In particular, the expression problem can be solved in a very simple and straightforward way (much simpler as suggested, for instance, by [14], where generics are used as a technical trick to solve a problem that is not inherently generic), and advanced dispatch strategies such as multiple or predicate dispatch [3] happen to become special cases of VNFs (cf. Sec. 3).

Furthermore, the particular kind of dynamic binding that will be employed will also allow the flexible extension and/or modification of existing VNF definitions in a manner similar to *advice* in AspectC++ [12]. To capture the other essential dimension of aspect-oriented programming, i. e., *quantification*, VNF *patterns* are provided as a means to conveniently define families of similarly structured VNFs (cf. Sec. 4).

Virtual namespace functions are implemented by a "lazy" precompiler for C++, i. e., a precompiler that does not perform a complete parse or semantic analysis of its input, but only recognizes particular keywords of interest, analyzes their context, and performs appropriate local source code transformations (cf. Sec. 5).

Since the concept of VNFs is similar to several other approaches found in the literature, a discussion of related work is given at the end of the paper (cf. Sec. 6).

Even though VNFs are presented in this paper as an extension to the C++ programming language (and C++ terminology is used), the underlying concept is actually language-independent and might be incorporated into any imperative (i. e., procedural or object-oriented) language. In fact, the same idea has also been implemented as *dynamic class methods* in Java [7] and as *dynamic procedures* in Oberon(-2) [6].

[1] Friend functions defined in a class are considered ordinary functions for the purpose of this distinction.

2. VIRTUAL NAMESPACE FUNCTIONS

2.1 Concept

A virtual namespace function (VNF) is an ordinary function defined at (global or) namespace scope whose definition is preceded by the keyword `virtual` (whose application is restricted to member functions in original C++). In contrast to normal functions which adhere to the *one definition rule* (ODR) [13], i. e., a program must not contain more than one definition of the same function (except if it is an inline or template function, in which case the definitions in all translation units must be identical), a VNF might be defined multiple times with different function bodies in the same or different translation units. In such a case, every new definition (also called a *branch* of the VNF) completely overrides the previous definition, so that calls to the function will be redirected to the new definition as soon as it has been *activated* during the program's initialization phase (see Sec. 2.2 for a precise definition of *activation*). However, every branch of a VNF is able to call the previous branch of the same VNF by using the keyword `virtual` as the name of a parameter-less pseudo function (i. e., an object providing a definition of `operator()()`) that calls the previous branch with the same arguments as the current branch (even if the formal parameters have been modified before calling `virtual`). The type of the function object `virtual` is a subtype of the predefined type `virtual<R>` (i. e., `virtual` is treated as a *template name*) if `R` is the result type of the VNF in question. Knowing that, it is possible, for example, to pass `virtual` as an argument to another function and call it there. Similar to a pointer to a local variable, however, the value of `virtual` is valid only while the corresponding branch of the VNF is being executed. A VNF might also be an operator function or a function template.

To give a simple example, consider the following VNF `f` computing the partially defined mathematical function $f(x) = x \sin(1/x)$:

```
virtual double f (double x) {
  return x * sin(1/x);
}
```

Since this function is undefined for $x = 0$, mathematicians might extend its definition by declaring $f(0) = \lim_{x \to 0} f(x) = 0$. This can be reflected by an additional branch of the VNF that completely overrides the original definition, but calls it via the pseudo-function `virtual` if `x` is different from zero:

```
virtual double f (double x) {
  if (x == 0) return 0;
  else return virtual();
}
```

For every VNF, there is an initial *branch zero*, playing the role of the previous branch of the first branch, that is either empty (if the result type of the VNF is `void`) or returns the special value `null` whose type `struct null` must be convertible to the result type `R` of the VNF, either by defining a constructor of `R` that accepts a single argument of type `struct null` or by implementing the conversion operator `null::operator R` which is pre-declared as a member function template

```
template <typename R> operator R () const;
```

in `struct null`.

2.2 Modules

A branch of a VNF is *activated* at the same time the initialization of a variable defined immediately before or after the branch would be executed during the program's initialization phase. This implies that the branches of a VNF which are defined in the same translation unit will be activated in textual order, which is reasonable. If branches of the same VNF are distributed over multiple translation units, however, their activation order will be partially undefined since the C++ standard does not prescribe a particular initialization order for variables defined in different translation units. Because this is unacceptable for many practical applications, a module concept similar to Modula-2 [15] and Oberon [16] has been incorporated into C++, that (amongst other useful things) defines a precise initialization order of the modules making up a program and consequently also a precise activation order of the branches of VNFs.

A *module* in this extension of C++ is a top-level namespace that might be structured into `public`, `protected`, and `private` sections, just like a class or struct (with the first section being `public`). In contrast to normal namespaces, whose definition might be distributed over multiple translation units, a module must be completely defined in a single translation unit (and typically, a translation unit contains exactly one module). Furthermore, a module might contain *import declarations* which are extended `using` declarations of the form

```
using A { a1, a2 };
```

where `A` is the name of a module whose public part contains declarations of `a1` and `a2`. The effect of such an import declaration is identical to a sequence of ordinary `using` declarations

```
using ::A::a1; using ::A::a2;
```

with the additional effect that the public part of the imported module `A` will be inserted into the current translation unit as a global namespace definition of `A` exactly once before the first such `using` declaration. (This is a more convenient way to `#include` a header file containing the public part of the module `A`. It implies that other names declared in the public part of `A`, such as `a3`, will be accessible as qualified names `A::a3` or `::A::a3`.) Furthermore, it will be guaranteed that module `A` will be initialized at run time (and consequently, the branches of VNFs defined in `A` will be activated) before the importing module will be initialized (i. e., before branches defined there will be activated).

Expressed differently, the overall initialization order of the modules making up a program is determined by traversing the directed acyclic graph consisting of modules (nodes) and import relationships (edges) in a depth-first, left-to-right manner (where left-to-right corresponds to the textual order of import declarations in a module), starting at a designated main module and visiting each module exactly once (i. e., ignoring already visited ones). If, for example, the main module `A` imports `B` and `C` (in this order) and `C` imports `D` and `B` (in this order), the overall initialization order will be `B`, `D`, `C`, `A`. (In particular, `B` will be initialized before `D`, even though the textual order of their import declarations in `C` is different, because its import declaration in `A` precedes that of `C`.)

VNF definitions appearing in a `public` section of a module will be reduced to bare declarations when the public part of the module gets inserted into another module in order to avoid multiple definitions (and activations) of the same branches. Those appearing in a `protected` section can be redefined in the importing module, but cannot be called from there. This allows a module to provide "hooks" to internal functions where other modules can “hang on” extensions, without allowing them to directly call these functions.

To redefine a VNF of an imported module, its qualified name has to be used, even if its name has been explicitly imported as shown above.

3. OBJECT-ORIENTED APPLICATION

3.1 The Expression Problem

Figure 1 shows a simple class hierarchy for the representation of arithmetic expressions (root class `Expr`) consisting of constant expressions (derived class `Const`) and the four basic arithmetic operations (derived classes `Add`, `Sub`, `Mul`, and `Div` with common intermediate base class `Binary`). Furthermore, a VNF `eval` is defined which evaluates an expression `x`, i. e., computes its value.

The first branch of this function uses an explicit `dynamic_cast` operator to test whether the argument `x` is actually a constant expression `c` and, if this is the case, returns its value `val`. Otherwise, the previous branch of the function (i. e., branch zero) would be called via `virtual()`, which should never happen in this example, however, since all other kinds of expressions will be handled by the subsequent branches of the function.

The second branch shows a more convenient way to express this frequently occurring programming idiom: “If the function arguments satisfy some condition, execute some code, otherwise delegate the call to the previous branch.” By moving the condition from the body to the head of the function, where it acts as a kind of *guard*, the stereotyped `else` clause can be omitted.

The third branch shows an even more convenient way to perform a dynamic type test in such a condition by using the colon operator which is very similar to Java’s `instanceof` operator, but does not exist in standard C++. In addition to performing the respective `dynamic_cast`, this operator causes the static type of the parameter `x` (which is `Expr*` from a caller’s point of view) to become `Sub*` in the function’s body (and any guards that might appear in its head), thus eliminating the need for an extra variable of that type.

The conversion operator from `null` to `int` must be defined prior to the first branch of the VNF, because its implicitly generated branch zero contains a `return null` statement. The intent of this conversion operator is to return an otherwise meaningless value that shall indicate an actually missing value. Alternatively, this operator might throw an exception such as `incomplete_vnf`, because the fact that it gets called at run time indicates that the branches of some VNF returning `int` actually do not handle all possible cases. Finally, a programmer might decide to always define the first branch of a VNF to throw such an exception (which might carry useful information such as the function’s name and its current arguments), before defining the “real” branches of the function.

```
namespace expr {
  // General expression.
  struct Expr {
    // Virtual desctructor
    // to make the type "polymorphic",
    // i. e., allow dynamic_cast.
    virtual ˜Expr () {}
  };

  // Constant expression.
  struct Const : Expr {
    int val; // Value of expression.
  };

  // Binary expressions.
  struct Binary : Expr {
    Expr* left;  // Left and right
    Expr* right; // subexpression.
  };
  struct Add : Binary {};
  struct Sub : Binary {};
  struct Mul : Binary {};
  struct Div : Binary {};

  // Conversion of null to int.
  template <> null::operator int () const {
    return INT_MIN;
  }

  // Evaluate constant expression.
  virtual int eval (Expr* x) {
    if (Const* c = dynamic_cast<Const*>(x)) {
      return c−>val;
    }
    else return virtual();
  }

  // Evaluate addition.
  virtual int eval (Expr* x)
  if (Add* a = dynamic_cast<Add*>(x)) {
    return eval(a−>left) + eval(a−>right);
  }

  // Evaluate subtraction.
  virtual int eval (Expr* x : Sub*) {
    return eval(x−>left) − eval(x−>right);
  }

  // Likewise for Mul and Div.
  ......
}
```

Figure 1: Basic implementation of arithmetic expressions

Figure 2 shows a typical *operational* extension of the system defined so far that adds a new operation to the existing class hierarchy, namely the output operator <<.[2] In the normal object-oriented paradigm, adding this operation in a modular way (i. e., without

```
namespace output {
  using expr { Expr, Const, Add, ...... };
  using iostream { ostream };

  // Print constant expression.
  virtual ostream& operator<<
  (ostream& os, Expr* x : Const*) {
    return os << x−>val;
  }

  // Print addition.
  virtual ostream& operator<<
  (ostream& os, Expr* x : Add*) {
    return os << ’(’ << x−>left
      << ’+’ << x−>right << ’)’;
  }

  // Likewise for Sub, Mul, and Div.
  ......
}
```

Figure 2: Operational extension

[2] C++ standard headers such as `iostream` can be used like modules in an import declaration.

touching or recompiling the existing source code) would be impossible, because in order to be dynamically dispatchable it would be necessary to *add* it as virtual member functions to the *existing* classes `Expr`, `Const`, etc. Furthermore, the operation is problematic from an object-oriented point of view because it shall not dispatch according to its first argument (which is the output stream), but according to the second. (This problem could be solved by defining `operator<<` as a normal function that calls an auxiliary member function on its second argument.) With virtual namespace functions, however, the extension can be done in a very simple and natural way.

Finally, Fig. 3 shows a subsequent *hierarchical* extension of the existing system that adds a new derived class `Rem` representing remainder expressions, together with matching redefinitions of the virtual functions `eval` imported from `expr` and `operator<<` imported from `output`. Even though adding new subclasses to an existing class hierarchy is basically simple in the object-oriented paradigm, this extension would be difficult, too, if the operational extension mentioned above would have been performed by employing the Visitor Pattern [5], because in that case it would be necessary to add new member functions to all existing visitor classes. Again, by using virtual namespace functions, the extension can be done in a simple and natural way.

```
namespace rem {
  using expr { Expr, Binary, eval };
  using output { operator<< };
  using iostream { ostream };

  // Remainder expression.
  struct Rem : Binary {};

  // Evaluate remainder expression.
  virtual int expr::eval (Expr* x : Rem*) {
    return eval(x->left) % eval(x->right);
  }

  // Print remainder expression.
  virtual ostream& output::operator<<
  (ostream& os, Expr* x : Rem*) {
    return os << '(' << x->left
      << '%' << x->right << ')';
  }
}
```

Figure 3: Hierarchical extension

3.2 Multiple and Predicate Dispatch

Figure 4 shows that it is equally easy to write VNFs that dispatch on the dynamic type of more than one argument, i. e., perform multiple dispatch. In this (somewhat artificial) example it is assumed that output to a file shall be more verbose than output to other kinds of streams.

Finally, Fig. 5 demonstrates that a VNF might actually dispatch on any predicate over its arguments (or even other information such as values of global or environment variables, user preferences read from a configuration file, etc.). The module shown maintains an RPN flag for every output stream (e. g., by employing `xalloc` [13]) that indicates whether output of expressions to that stream shall be performed in reverse polish notation. If this flag is set for a particular stream, output of binary expressions is changed accordingly. To keep the implementation hierarchically extensible, the internal helping function `opchar` that returns the operator character corresponding to a binary expression is declared `protected` so that other modules can add additional branches on demand.

```
namespace file_output {
  using expr { Expr, Const, ...... };
  using rem { Rem };
  using output { operator<< };
  using iostream { ostream };
  using fstream { ofstream };

  // Print constant expression to a file.
  virtual ostream& output::operator<<
  (ostream& os : ofstream&, Expr* x : Const*) {
    return os << "constant expression"
      << " with value " << x->val;
  }

  ......

  // Print remainder expression to a file.
  virtual ostream& output::operator<<
  (ostream& os : ofstream&, Expr* x : Rem*) {
    return os << "remainder expression"
      << " with left operand (" << x->left
      << ") and right operand (" << x->right
      << ")";
  }
}
```

Figure 4: Multiple dispatch

```
namespace rpn {
  using expr { Expr, Binary, Add, ...... };
  using rem { Rem };
  using output { operator<< };
  using iostream { ostream };

  // Set and get RPN flag of output stream.
  virtual void setrpn (ostream& os, bool f);
  virtual bool getrpn (const ostream& os);
protected:
  // Get operator character of binary expr.
  virtual char opchar (Expr* x : Add*)
  { return '+'; }
  virtual char opchar (Expr* x : Sub*)
  { return '-'; }
  ......
public:
  // RPN output of binary expression.
  virtual ostream& output::operator<<
  (ostream& os, Expr* x : Binary*)
  if (getrpn(os)) {
    return os << x->left << ' ' << x->right
      << ' ' << opchar(x);
  }
}
```

Figure 5: Predicate dispatch

4. ASPECT-ORIENTED APPLICATION

4.1 Crosscutting Concerns

It is rather obvious that VNFs might also be used to implement typical crosscutting concerns such as logging by grouping appropriate redefinitions together in a single module (cf. Fig. 6). In contrast to the examples seen so far, where every branch of a VNF is guarded by an appropriate condition and the previous branch is

```
namespace logging {
  using expr { Expr };
  using output { operator<< };
  using iostream { cout, endl, ostream };

  // Log executions of eval.
  virtual int expr::eval (Expr* x) {
    int val = virtual();
    cout << "value of " << x
      << " is " << val << endl;
    return val;
  }

  // Log executions of operator<<.
  virtual ostream& output::operator<<
  (ostream& os, Expr* x) {
    cout << "output of " << x << endl;
    return virtual();
  }
}
```

Figure 6: A crosscutting concern

called implicitly if this condition is violated, the redefinitions shown here are unconditional and call the previous branch explicitly in their body. By that means, it is easily possible to implement before, after, and around *advice*, to use aspect-oriented terminology [12], without the need to employ any additional "aspect weaving" mechanism.

4.2 VNF Patterns

To capture the other essential dimension of aspect-oriented programming, i. e., *quantification* [4], it is possible to define families of similarly structured VNFs with a single definition by using *ellipses* for parameters, the function name, and/or the result type: an ellipsis used in the parameter list of a VNF is a placeholder for any number of parameters of any types, while an ellipsis used as the function name or result type is a placeholder for any name or type, respectively. In contrast to normal VNFs, however, such *VNF patterns* do not introduce any new functions, but only define additional branches for all VNFs defined so far (including those defined in imported modules) whose signature *matches* the pattern. (A function signature matches a pattern if and only if it is possible to replace the ellipses occurring in the pattern with function or type names [or sequences of the latter in the case of a parameter ellipsis] to make the resulting signature identical to the given function signature.) To allow the programmer to employ more selective matching strategies, the names of the matched function and its parameter and result types are available in the body (and head) of a VNF pattern as `const char*` values `&virtual` (function name), `virtual[0]` (name of result type), `virtual[1]` (name of first parameter type), etc., while the number of parameters is available as an `int` value `*virtual` (i. e., the type of `virtual` provides corresponding definitions of the operators `&`, `[]`, and `*`).

To give a standard AOP example, the following VNF pattern defines additional branches for all VNFs whose name starts with `A::get`:

```
virtual ... ... (...)
if (strncmp(&virtual, "A::get", 6) == 0) {
  cout << virtual[0] << " " << &virtual << "(";
  for (int i = 1; i <= *virtual; i++) {
    if (i > 1) cout << ", ";
    cout << virtual[i];
  }
  cout << ")" << endl;
  return virtual();
}
```

Similarly, the following pattern matches all VNFs accepting exactly two parameters whose first parameter type is `Expr` and whose result type is `int`:

```
virtual int ... (Expr, ...) if (*virtual == 2)
{ ...... }
```

5. IMPLEMENTATION

5.1 Lazy Precompiler

The extensions to the C++ programming language described in this paper have been implemented by a "lazy" precompiler, i. e., a precompiler that does not perform a complete parse or even a semantic analysis of its input. Instead, it only recognizes a few significant keywords (such as `namespace`, `public`, or `virtual`), determines their context (e. g., whether `public` or `virtual` is used in a namespace or a class), and then performs appropriate "local" source code transformations. Because a complete description of these transformations would be far beyond the scope of this paper, only a few basic ideas will be sketched in the sequel.

Basically, each branch of a VNF is transformed to a normal C++ function possessing the same parameter list and result type as the VNF and a uniquely generated internal name.[3] Its body is augmented with the definition of a local class defining operators `()` (cf. Sec. 2.1), `[]`, `*`, and `&` (cf. Sec. 4.2), together with a single instance of this class storing the values of all function arguments. Each appearance of the keyword `virtual` inside the body (and not inside a local class) will be replaced by the name of this object. Furthermore, a declaration and initialization of a function pointer variable is generated which will perform the activation of the branch at run time by appending it to the end of a linked list.

When the first branch of a particular VNF is encountered, a declaration of another function pointer variable which will always point to the last branch of that list as well as an additional *dispatch function* is generated whose signature (i. e., name, parameters, and result type) is identical to the VNF and whose body simply calls the "current" branch via this variable. This is the function that will actually be called when the VNF is called anywhere in the program.

To give an example of these transformations, figures 7 and 8 show the (simplified and beautified) output of the precompiler produced for the first and second branch of the VNF `eval` shown in Fig. 1.[4]

VNF pattern definitions are simply transformed to sets of normal VNF definitions by replacing the ellipses of the pattern with

[3] Even though it is impossible in principle to guarantee the uniqueness of a generated name, it can be easily achieved in practice by employing unnamed namespaces (to guarantee uniqueness across multiple translation units, including dynamically loaded modules) and by using names containing double underscore characters (which are reserved for implementations and standard libraries and thus must not be used by programmers).

[4] In practice, the code generated by the precompiler is much more complicated, since a lazy precompiler is basically unable to distinguish the branches of overloaded VNFs if they possess the same number of parameters. (This is due to the fact, that type names might have multiple "aliases" introduced, e. g., by means of `typedef` or `using` declarations.) To overcome this problem, the precompiler actually generates very tricky template code that delegates this work to the underlying C++ compiler.

```
// Function pointer type.
typedef int (*eval__type) (Expr*);

// Branch zero.
int eval__0 (Expr* x)
{ return (Expr*)null; }

// Variable pointing to current branch.
eval__type eval__current = eval__0;

// Dispatch function.
int eval (Expr* x)
{ return eval__current(x); }

// Variable pointing to previous branch.
eval__type eval__prev__1 = eval__current;

// This branch.
int eval__1 (Expr* x) {
  // Instance of local class
  // replacing keyword virtual.
  struct virtual__class : virtual__base<int> {
    // Copy of function argument
    // and constructor initializing it.
    Expr* x;
    virtual__class (Expr* x) : x(x) {}

    // Function call operator
    // calling previous branch.
    int operator() () const
    { return eval__prev__1(x); }

    // Introspection operators.
    const char* operator[] (int i) const {
      switch (i) {
      case 0: return "int";
      case 1: return "expr::Expr*";
      default: return 0;
      }
    }
    int operator* () const
    { return 1; }
    const char* operator& () const
    { return "expr::eval"; }
  } virtual__inst(x);

  // Original function body.
  if (Const* c = dynamic_cast<Const*>(x)) {
    return c->val;
  }
  else return virtual__inst();
}

// Adjust variable pointing to current branch
// by initializing a dummy variable.
eval__type eval__current__1 =
 eval__current = eval__1;
```

Figure 7: Transformation of first branch of `eval` (cf. Fig. 1)

matching type and function names from all normal VNF definitions encountered so far.

A module is transformed to a C++ source file containing the complete code of the module plus an additional header file containing only the public part. Import declarations are transformed as described in Sec. 2.2.

```
// Variable pointing to previous branch.
eval__type eval__prev__2 = eval__current;

// This branch.
int eval__2 (Expr* x) {
  // Instance of local class
  // replacing keyword virtual.
  struct virtual__class : virtual__base<int> {
    ......
  } virtual__inst(x);

  // Original function head and body.
  if (Add* a = dynamic_cast<Add*>(x)) {
    return eval(a->left) + eval(a->right);
  }
  // Implicit call of previous branch.
  else return virtual__inst();
}

// Adjust variable pointing to current branch
// by initializing a dummy variable.
eval__type eval__current__2 =
 eval__current = eval__2;
```

Figure 8: Transformation of second branch of `eval` (cf. Fig. 1)

5.2 Optimization

Because the dispatch function of a VNF described above always calls the current branch indirectly via a function pointer variable, every VNF call incurs an additional function call, even if there is only a single branch of the VNF (which happens frequently in practice, if every public function of a module is defined `virtual` in order to allow for later modular extensions). To avoid this performance penalty (and thus to encourage programmers to actually pursue this strategy), the dispatch function can test a flag to check whether the VNF possesses exactly one branch (this flag is set when the first branch is activated and reset when additional branches are activated) and directly execute a copy of the code of the first branch in place if this is the case; otherwise, the current branch will be called as before. By that means, VNFs possessing a single branch can be executed almost as efficient as normal functions, and an optimizing compiler will even be able to inline their code into calling functions.

6. RELATED WORK

It has already been shown in Sec. 3 that VNFs are a generalization of object-oriented single, multiple, and predicate-based [3] method dispatch. In contrast to these approaches, however, no attempt is made to find the *best* matching branch of a function, but always the *first* matching branch (in reverse activation order) is executed. While this heavily simplifies both the semantics and the implementation of the approach, the resulting semantics is obviously somewhat different. For many practical applications, however, the two semantics (best matching vs. first matching branch) coincide. In particular, if VNF branches are defined in the same order as the classes they operate on, the total order of branches is compatible with the partial order between base classes and derived classes, since a derived class is necessarily defined after its base classes. Furthermore, if the guards of all branches test for mutually disjoint predicates, the order of the branches becomes totally irrelevant.

VNFs also capture aspect-oriented advice for simple call and execution join points [12]. If the information hiding principle is applied strictly, i. e., all set and get operations on data fields are encapsulated in VNFs, they also capture set and get join points.

Finally, control flow (cflow) join points can be simulated by overriding the "top level" function with a branch that sets a flag when entering the function and resets it when leaving it, and overriding all "subordinate" functions (typically by employing a VNF pattern) with branches that test this flag. Thus, a broad range of pointcut expressions provided by AspectC++ is covered.

The concept of VNF patterns is a rather simple and straightforward extension of plain VNFs which per se is of course much less expressive than the pattern matching facilities built into AspectC++. On the other hand, the ability to refer to the function's name as well as its parameter and result types (in demangled form!) in the function's guards and therefore to use this information in arbitrary predicates, allows programmers to implement even more expressive or convenient matching strategies by exploiting the full expressiveness of C++. In other words, instead of introducing a separate *pointcut language* with numerous additional language constructs, the base language C++ is reused to express "pointcuts." By defining the predicates used in the guards of VNFs as VNFs themselves, it is even possible to redefine them later and by that means achieve effects similar to virtual pointcuts in AspectC++.

VNFs have some obvious similarities with generic functions in CLOS [8] (and other languages based on comparable ideas, e. g., Dylan [2]) since both are defined outside any class (and thus can be freely distributed over a program) and both provide multiple dispatch. Furthermore, before, after, and around methods in CLOS provide a great deal of flexibility in retroactively extending existing functions, which can be enhanced even further by user-defined method combinations [9]. However, even with the latter, the specificity of methods remains a primary ordering principle, and it is impossible to get the list of all applicable methods simply in the order of their declaration. Furthermore, it is impossible to define two or more methods having the same specificity and the same method qualifiers (e. g., two generally applicable around methods). In contrast, the fact that VNFs do not care about the specificities of their branches, but simply use their linear activation order, does not only simplify their semantics, implementation, and use, but also allows complete redefinitions of a function without losing its previous definition.

The same is true for dynamically scoped functions [1], with one noteworthy difference: Instead of a linear list, which is only extended by new VNF branches, but never reduced, dynamically scoped functions are actually based on a stack of definitions that grows and shrinks dynamically. Furthermore, push and pop operations on this stack are not provided explicitly and independently, but only in implicit combination by means of a `dflet` macro that pushes a new definition, executes some code with this definition in effect, and finally pops it again. Therefore, it is impossible to install permanent redefinitions of functions whose effect exceeds the lifetime of their defining scope. Furthermore, installing a large number of extensive redefinitions via `dflet` (instead of using global definitions for that purpose) might significantly reduce the readability of the code. On the other hand, the possibility to redefine a function only temporarily, that is currently missing in the concept of VNFs, appears to be quite useful in certain circumstances.

Finally, the way `virtual` is used to call the previous branch of a VNF resembles the way `inner` is used in BETA [10]. However, the order of execution is exactly reversed: While `virtual` is used in a redefinition to call the previous definition, `inner` is used in the original definition to call a possible redefinition, which implies that the original definition cannot be changed, but only extended by a redefinition in BETA.

The module concept for C++ introduced in this paper is actually a mixture of Modula-2 modules [15], Oberon modules [16], and C++ classes: The basic idea has been taken from Modula-2 where it is possible to import both complete modules (and use qualified names to refer to their exported entities) and individual names from particular modules (which can then be used unqualified). In a language such as C++ that supports overloading of (function) names, it is possible to import the same name from different modules as long as their definitions do not conflict.

In contrast to Modula-2, but in accordance with Oberon, the public and private parts of a module are not separated into different translation units, but rather integrated into a single unit. Finally, the idea to structure a module into sections introduced by the keywords `public` and `private` (and possibly `protected`) − instead of using special export marks to distinguish exported names as in Oberon −, has been adopted from C++ classes. (In every other respect, however, a module is quite different from a class. In particular, it cannot be instantiated explicitly, but rather constitutes a singleton global entity.) By following the convention to split a module into a single public section at the beginning that contains bare declarations of all exported entities and a subsequent private section containing the corresponding definitions (plus necessary internal entities), the Modula-2 approach of separating these parts can be simulated without actually needing two separate translation units.

In addition to the purpose mentioned in Sec. 2.2, i. e., establishing a unique initialization order among multiple translation units of a program which in turn defines a unique activation order for VNF branches, modules provide a simple yet effective way to enforce the well-known principle of information hiding [11]: By defining data structures (such as `struct Const : Expr { int val; }`) in the private part of a module and exporting only corresponding pointer types (e. g., `typedef Const* ConstPtr`) and (virtual) functions operating on them (e. g., `virtual int value (ConstPtr c) { return c->val; }`), it is possible to hide implementation details of a module from client modules without needing to employ classes for that purpose. If a single module contains definitions of multiple data types (e. g., a container type and an accompanying iterator type), its functions are naturally allowed to operate on all of their internals, without needing to employ sophisticated constructs such as nested or friend classes to achieve that aim.

7. REFERENCES

[1] P. Costanza: "Dynamically Scoped Functions as the Essence of AOP." *ACM SIGPLAN Notices* 38 (8) August 2003, 29−36.

[2] I. D. Craig: *Programming in Dylan*. Springer-Verlag, London, 1997.

[3] M. Ernst, C. Kaplan, C. Chambers: "Predicate Dispatching: A Unified Theory of Dispatch." In: E. Jul (ed.): *ECOOP'98 − Object-Oriented Programming* (12th European Conference; Brussels, Belgium, July 1998; Proceedings). Lecture Notes in Computer Science 1445, Springer-Verlag, Berlin, 1998, 186−211.

[4] R. E. Filman, D. P. Friedman: "Aspect-Oriented Programming is Quantification and Obliviousness." In: *Workshop on Advanced Separation of Concerns* (OOPSLA 2000, Minneapolis, MN, October 2000).

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[6] C. Heinlein: *Vertical, Horizontal, and Behavioural Extensibility of Software Systems*. Nr. 2003-06, Ulmer Informatik-Berichte, Fakultät für Informatik, Universität Ulm, July 2003. http://www.informatik.uni-ulm.de/pw/9239

[7] C. Heinlein: “Dynamic Class Methods in Java.” In: *Net.ObjectDays 2003. Tagungsband* (Erfurt, Germany, September 2003). tranSIT GmbH, Ilmenau, 2003, ISBN 3-9808628-2-8, 215−229. (See http://www.informatik.uni-ulm.de/pw/9238 for an extended version.)

[8] S. E. Keene: *Object-Oriented Programming in Common Lisp: A Programmer’s Guide to CLOS*. Addison-Wesley, Reading, MA, 1989.

[9] G. Kiczales, J. des Rivières, D. G. Bobrow: *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[10] O. Lehrmann Madsen, B. Møller-Pedersen, K. Nygaard: *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Wokingham, England, 1993.

[11] D. L. Parnas: “On the Criteria to Be Used in Decomposing Systems into Modules.” *Communications of the ACM* 15 (12) December 1972, 1053−1058.

[12] O. Spinczyk, A. Gal, W. Schröder-Preikschat: “AspectC++: An Aspect-Oriented Extension to the C++ Programming Language.” In: J. Noble, J. Potter (eds.): *Proc. 40th Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)* (Sydney, Australia, February 2002), 53−60.

[13] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.

[14] M. Torgersen: “The Expression Problem Revisited. Four New Solutions Using Generics.” In: M. Odersky (ed.): *ECOOP 2004 − Object-Oriented Programming* (18th European Conference; Oslo, Norway, June 2004; Proceedings). Lecture Notes in Computer Science 3086, Springer-Verlag, Berlin, 2004, 123−143.

[15] N. Wirth: *Programming in Modula-2*. Springer-Verlag, 1982.

[16] N. Wirth: “The Programming Language Oberon.” *Software—Practice and Experience* 18 (7) July 1988, 671−690.