JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 4, No. 10, 2005

Global and Local Virtual Functions in C++

Christian Heinlein, Dept. of Computer Science, University of Ulm, Germany

Global virtual functions (GVFs) are introduced as C++ functions defined at global or namespace scope which can be redefined later similar to virtual member functions. Even though GVFs are a relatively simple concept, hardly more complex than ordinary C functions, it is shown that they subsume object-oriented single, multiple, and predicate-based method dispatch as well as aspect-oriented before, after, and around advice. Furthermore, the well-known "expression problem" can be solved in a simple and natural way. *Local virtual functions* are a straightforward extension of GVFs allowing temporary redefinitions during the execution of some other function or part of it. Amongst others, this is quite useful to simulate "cflow join points" of aspect-oriented languages. The implementation of global and local virtual functions by means of a precompiler for C++ is briefly described.

1 INTRODUCTION

C++ provides both *global functions* (defined at global or namespace scope, i. e., outside any class) and *member functions* belonging to a particular class [18]. 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" [19], 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 [7] 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, global functions 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 global functions and virtual member functions, it seems very promising to provide *global virtual* *functions* (GVFs, cf. Sec. 2) which combine their advantages:

- Because they will be defined at global (or namespace) scope, it will always be possible to add new GVFs 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 [19], 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 [6] happen to become special cases of GVFs (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 GVF definitions in a manner similar to *advice* in AspectC++ [17] and comparable languages (cf. Sec. 4).

As a straightforward extension of global virtual functions, *local virtual functions* (LVFs) can be used to *temporarily* override an existing GVF during the execution of some other function (or part of it) by providing a local redefinition in the corresponding statement block. Amongst others, this concept is quite useful to simulate so-called *cflow join points* of aspect-oriented languages (cf. Sec. 5).

Even though the basic concept of global and local virtual functions is language-independent and might be incorporated into any imperative (i. e., procedural or object-oriented) programming language, it has been implemented as a precompiler-based language extension to C++ (cf. Sec. 6).[1]

Since the concept is similar to several other approaches found in the literature, a discussion of related work is given (cf. Sec. 7) before the paper is concluded in Sec. 8.

2 GLOBAL VIRTUAL FUNCTIONS

Basic Concept

A *global virtual function* (GVF) is an ordinary C++ 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* [18], i. e., a program must not contain more than one definition of the same function (except if it is an inline or

[1]Furthermore, GVFs have also been implemented earlier as so-called *dynamic class methods* in Java [9] and as *dynamic procedures* in Oberon(-2) [8].

template function, in which case the definitions in all translation units must be identical), a GVF 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 GVF) 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 below for a precise definition of *activation*). However, every branch of a GVF is able to call the previous branch of the same GVF by using the keyword `virtual` as the name of a parameter-less pseudo function that calls the previous branch with the same arguments as the current branch (even if the formal parameters have been modified before calling `virtual`).

To give a simple example, consider the following GVF `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 GVF 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 GVF, 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 GVF is `void`) or returns a default value of its result type by calling its parameter-less constructor.

Modules

A branch of a GVF is *activated* at the same time the initialization of a global variable defined immediately before or after the branch would be executed during the program's *initialization phase* [18]. This implies that the branches of a GVF which are defined in the same translation unit will be activated in textual order, which is reasonable. If branches of the same GVF 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 [20] and Oberon [21] 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 GVFs.

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 implicitly `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 nominate *base modules* (similar to the base classes of a class) on which it depends, i. e., whose public definitions it wants to use, e. g.:

```
// Module A depending on base modules B and C.
namespace A : B, C { /* definitions of module A */ }
```

By specifying such base modules (`B` and `C` in the example), their public definitions will be available in the dependent module (`A`) as if they have been inserted before its own definition (similar to `#include` files):[2]

```
namespace B { /* public definitions of module B */ }
namespace C { /* public definitions of module C */ }
namespace A { /* definitions of module A */ }
```

Furthermore, it will be guaranteed that modules `B` and `C` will be *initialized* (in this order) at run time (and consequently, the branches of GVFs defined in these base modules will be activated) before the dependent module `A` 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 inter-module dependencies (edges) in a depth-first, left-to-right manner (where left-to-right corresponds to the textual order of the base module specifications of 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` depends on `B` and `C` (in this order) and `C` depends on `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 specification as base modules of `C` is different, because `B`'s specification in `A` precedes that of `C`.)

GVF (and other) *definitions* appearing in a `public` section of a module will be reduced to bare *declarations* when the public part of the module gets inserted before 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 dependent

[2]To actually use a name such as `b` provided by a base module such as `B`, the normal C++ rules for name lookup apply, i. e., `b` must either be qualified as `B::b` or "imported" by a `using` declaration `using B::b;` To simplify the "import" of multiple names from the same module, the syntax of `using` declarations has been generalized to, e. g., `using B { b1, b2, b3 };`

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 GVF defined in a base module, its qualified name has to be used, even if its name has been explicitly imported by a `using` declaration.

3 OBJECT-ORIENTED APPLICATION

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 GVF `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.

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 `<<`.[3] In the normal object-oriented paradigm, adding this operation in a modular way (i. e., without 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

[3]C++ standard include files such as `iostream` can be used as base modules in a module definition.

```
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 {};

  // 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

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 global virtual functions, however, the extension can be done in a very simple and natural way.

```
namespace output : iostream, expr {
  using iostream { ostream };
  using expr { Expr, Const, Add, Sub, Mul, Div };

  // 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

Finally, Fig. 3 shows a subsequent *hierarchical* extension of the existing system that adds a new derived class Rem representing remainder expressions, together with appropriate redefinitions of the GVFs 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 [7], because in that case it would be necessary to add new member functions to all existing visitor classes. Again, by using global virtual functions, the extension can be done in a simple and natural way.

Multiple and Predicate Dispatch

Figure 4 shows that it is equally easy to write GVFs 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.

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

  // 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

Finally, Fig. 5 demonstrates that a GVF 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` [18]) 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.

4 ASPECT-ORIENTED APPLICATION

It is rather obvious that GVFs 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 GVF is guarded by an appropriate condition and the previous branch is 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 ter-

```
namespace file_output : iostream, fstream, expr, output, rem {
  ...... // using-declarations

  // 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 : iostream, expr, output, rem {
  ...... // using-declarations

  // 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 expression.
  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

minology [17], without the need to introduce any new language constructs nor to employ some additional "aspect weaving" mechanism for that purpose.

The "join points" directly supported that way are *call* resp. *execution* of global virtual functions. However, if the information hiding principle [16] is applied strictly by encapsulating all *set* and *get* operations on data fields in GVFs, these kinds of join points can be covered, too. Finally, Sec. 5 will demonstrate how *control flow* join points can be simulated by employing local virtual functions.

The "weaving" of "aspects" defined by global (or local) redefinitions of GVFs is implicitly performed by the general rule stated in Sec. 2 that each new definition of a GVF completely overrides the previous definition. Calling the pseudo-function `virtual` in a redefinition corresponds to executing `proceed` in AspectC++ and comparable languages.

```
namespace logging : iostream, expr, output {
  ...... // using-declarations

  // 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

5 LOCAL VIRTUAL FUNCTIONS

Basic Concept

Local virtual functions are a straightforward extension of global virtual functions, allowing a GVF to be *temporarily* redefined by a *local branch*, i. e., a branch defined locally in another function.[4] According to the normal scoping rules, such a local function can access the local variables of its enclosing function(s).

A local branch of a GVF is *activated*, i. e., pushed on a stack of local redefinitions of its function, when the control flow of its lexically enclosing function reaches its definition. It is *deactivated*, i. e., popped from the stack of redefinitions, when the *block* (or compound statement) containing its definition is left in any way, either normally by reaching its end or abruptly due to the execution of a `throw` expression or a `return`, `break`, `continue`, or `goto` statement (including non-local jump statements described below). Thus, the time of activation resp. deactivation corresponds exactly to the time where a constructor resp. destructor of a local variable defined instead of the local branch would be executed. Expressed differently, this means that a local redefinition of a GVF is *in effect* from its point of definition until the end of the enclosing block.

Even though C++ does not provide a notion of *threads* as part of the language, a separate stack of local redefinitions of a function is maintained for every thread of a program, if multi-threading is provided by some library. Thus, for every GVF, there is a global list of its global branches plus a thread-local stack of its currently active local branches per thread. A call to the function from within a particular thread executes the local branch on top of its stack (if any), whose previous branch is either the next lower branch on the stack (if any) or else the last branch of the global list, etc. (cf. Fig. 7).

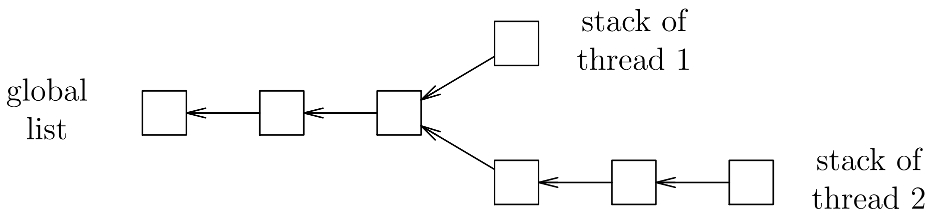

Figure 7: Global and thread-local branches of a global virtual function

Non-Local Jump Statements

When a *jump statement*, i. e., a `return`, `break`, `continue`, or `goto` statement, is executed within a local function, its effect is, of course, local to this function, i. e., a `return` statement terminates the local function, while a `break`, `continue`, or `goto` statement transfers control to the appropriate place within this function.

Occasionally, however, it is useful to perform a *non-local jump*, i. e., to execute a statement that transfers control out of the local function to a place within (one of) its *enclosing* function(s). (Since a local function is in effect only while its enclosing function is executing, transferring control to the latter is basically possible.)

[4]This other function might be any kind of function, including normal global and member functions as well as global and local branches of GVFs (allowing even arbitrarily *nested* local functions).

For example, one might want to execute a return statement that immediately terminates the enclosing function or a break/continue statement that immediately terminates/continues a loop within the enclosing function. For that purpose, it is possible to prefix a return, break, or continue statement[5] with one or more extern keywords to indicate that the statement shall be executed as if it were part of the nth statically enclosing function where n is the number of extern keywords given.

Executing such a *non-local jump statement* obviously causes abrupt termination of the function executing it as well as all intermediate functions that have been called directly and indirectly from the respective enclosing function. To stay compatible with common C++ semantics, terminating these functions implies the process of *stack unwinding* where destructors for all local variables declared in these functions are executed in reverse order of their constructor invocations [18]. Therefore, the effect of a non-local jump statement is equivalent to throwing an exception that is caught at the appropriate place in the designated enclosing function and executing the corresponding ordinary jump statement from there. The "appropriate place" in the enclosing function would be a catch block associated with a try block replacing the innermost block containing the local function definition.

Example

To give an example of local virtual functions and non-local jump statements, Fig. 8 shows a simple module users providing a structure type User with data fields id, name, and passwd as well as global virtual functions read_char (reading a single input character), read_field (reading an input *field*, i. e., a sequence of characters up to the next colon or newline character), and read_user (reading a complete user record consisting of id, name, and password fields). If user records are kept in a text file where each line contains three colon-separated fields, such a file can be read by repeatedly calling the function read_user. The only problem with this function is that it does not perform any error checking: If a line contains less than three fields, read_user will read the missing fields from the next line; if a line contains more than three fields, the remaining ones will be treated as part of the next record.

Figure 9 shows how this problem can be fixed in a completely modular way, i. e., without changing the existing module users, but by solely providing an additional module check containing a global redefinition of read_user with appropriate local redefinitions of read_char and read_field.

The redefinition of read_char simply calls its previous branch, i. e., the original implementation of the function, and additionally stores its result value in a local variable last of the enclosing function read_user. The redefinition of read_field uses this variable to check whether a premature end of line has been reached; if this is the case, its enclosing function read_user is immediately terminated by executing

[5] goto statements are excluded, because their use is generally discouraged and could easily lead to very complicated control flows when variable declarations with (explicit or implicit) initializations are crossed by a jump.

```
namespace users : iostream, string {
  using iostream { cin };
  using string { string };

  // User with id, name, and password.
  struct User {
    string id, name, passwd;
    User (string i, string n, string p)
    : id(i), name(n), passwd(p) {}
  };

  // Read next input character.
  virtual char read_char () {
    char c;
    if (cin.get(c)) return c;
    else return 0;
  }

  // Read next input field.
  virtual string read_field () {
    char c; string s;
    while ((c = read_char()) && c != ':' && c != '\n') s += c;
    return s;
  }

  // Read next user.
  virtual User* read_user () {
    string u = read_field(), n = read_field(), p = read_field();
    return new User(u, n, p);
  }
}
```

Figure 8: Module `users`

a non-local `return` statement returning a null pointer instead of a real pointer to a `User` object; otherwise, its previous branch, i. e., the original implementation of `read_field` is executed.

The global redefinition of `read_user` also calls its original implementation, with these local redefinitions in effect. Therefore, calls to `read_char` and `read_field` performed by this implementation will be redirected to these redefinitions. If the local branch of `read_field` detects a premature end of line and therefore executes its `extern return` statement, all active functions up to and including the redefinition of `read_user` – i. e., the local branch of `read_field` itself, the original implementation

```
namespace check : users {
  using users { User, read_char };

  // Global redefinition of read_user.
  virtual User* users::read_user () {
    // Last character read by read_char.
    char last = 0;

    // Local redefinition of read_char.
    virtual char users::read_char () {
      // Call original implementation
      // and store result in local variable last.
      return last = virtual();
    }

    // Local redefinition of read_field.
    virtual string users::read_field () {
      // If end of line has been reached, terminate read_user.
      // Otherwise, call original implementation.
      if (last == '\n') extern return (User*)0;
      else return virtual();
    }

    // Call original implementation of read_user
    // with local redefinitions in effect.
    User* u = virtual();

    // If end of line has been reached, terminate normally.
    if (last == '\n') return u;

    // Otherwise, read until end of line and return null pointer.
    while (last != '\n') read_char();
    return (User*)0;
  }
}
```

Figure 9: Module check

of read_user that has called it, and the redefinition of read_user that has called this – will be terminated abruptly and the latter will return a null pointer to indicate the error.

If no such error is detected, the call to the previous branch of read_user returns normally. To catch the second kind of error, i. e., an input line with more than three fields, the local variable `last` is used once more to check whether end of line has been reached as expected. If not, the remaining characters of the current line are read and skipped, and a null pointer is returned again to indicate the error.

Control Flow Join Points

If a global redefinition of a GVF `f` is considered as aspect-oriented advice associated with call/execution join points of `f` (cf. Sec. 4), a local redefinition of `f` in another function `g` corresponds to advice associated with those call/execution join points of `f` that occur during executions of `g`, i. e., within the dynamic control flow (*cflow*) of `g`.

6 IMPLEMENTATION

Global Virtual Functions and Modules

The extensions to the C++ programming language described in this paper have been implemented by a precompiler based on the EDG C++ Front End (cf. `www.edg.com`). It recognizes 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 source code transformations to map the extensions to pure C++ code. 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 GVF is transformed to a normal C++ function possessing the same parameter list and result type as the GVF and a uniquely generated internal name. Its body is augmented with the definition of a single object of a local class storing the values of all function arguments and providing a definition of the function call operator `()` that calls the previous branch of the function with these arguments. Then, each appearance of the keyword `virtual` inside the body of the function (and not inside a local class) is replaced by the name of this object. Furthermore, a declaration and initialization of a global 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 GVF 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 GVF and whose body simply calls the "current" branch via this variable. This is the function that will actually be called when the GVF is called anywhere in the program.

To give an example of these transformations, figures 10 and 11 show the (simplified and beautified) output of the precompiler produced for the first and second branch of the GVF eval shown in Fig. 1.

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

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

// 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 {
    // 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); }
  } virtual__inst(x);

  // Original function body
  // with "virtual" replaced by "virtual__inst".
  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 10: Transformation of first branch of eval (cf. Fig. 1)

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. If base

```
// 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__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 11: Transformation of second branch of `eval` (cf. Fig. 1)

modules are specified, `#include` directives for the corresponding header files are inserted.

Local Virtual Functions and Non-local Jump Statements

A local virtual function definition is basically transformed to a member function of a local auxiliary class. While this allows a completely "local" source code transformation (in contrast to the alternative possibility of moving the local function out of its enclosing function(s) and transforming it to an ordinary global function), it suffers from the restriction that member functions of local classes must not use local variables of the enclosing function [18]. To circumvent this problem, references to all local variables of the enclosing function are provided as data members of the auxiliary class, which are initialized by a constructor receiving the actual references as arguments. Therefore, local variables of the enclosing function(s) can be used in the local function without any restrictions.

The constructor and destructor of the auxiliary class are responsible for pushing resp. popping the local branch on resp. from the stack of local redefinitions of the function. This stack is implemented as a linked list using the auxiliary class instances as elements. A pointer to the topmost element of the stack is either provided in a global variable (if multi-threading is not an issue) or in a thread-local variable. The dispatch function of the GVF mentioned earlier actually uses this variable to locate the topmost element of the stack (belonging to the current thread) and call its member function representing the local branch or – if the stack is currently empty – call the last global branch as described above.

A non-local jump statement is transformed to a `throw` expression where the thrown object encodes the kind of statement (`return`, `break`, or `continue`), the destination function, and, if appropriate, the value of a `return` expression. To catch such exceptions at the appropriate place, blocks containing LVF definitions are embedded into `try` statements with corresponding `catch` clauses which decode the information in the thrown object and execute the corresponding normal jump statement.

7 RELATED WORK

It has already been shown in Sec. 3 that global virtual functions are a generalization of object-oriented single, multiple [11, 14, 3, 1], and predicate-based [6] 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 GVF 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.

GVFs also capture aspect-oriented advice for simple call and execution join points [17]. If the information hiding principle is applied strictly, i. e., all set and get operations on data fields are encapsulated in GVFs, they also capture set and get join points. Finally, control flow (cflow) join points can be simulated quite easily by globally overriding the "top level" function of a particular cflow with a branch containing local redefinitions of all "subordinate" functions before calling its previous implementation. Thus, a broad range of pointcut expressions provided by AspectC++ and comparable languages is covered. By defining the predicates used in the guards of GVFs and LVFs as GVFs themselves, it is even possible to redefine them later and by that means achieve effects similar to *virtual pointcuts*.

In contrast to mainstream aspect-oriented languages such as AspectJ [13] and AspectC++ [17], no distinction is drawn between a base language (such as Java or C++) providing, e. g., "normal" methods or functions and an additional aspect language providing, e. g., advice. Instead, GVFs constitute a single, coherent concept covering both “normal” functions (represented by original definitions of GVFs) and advice (represented by appropriate redefinitions).

Similarly, the model of composition filters “unifies traditional object behaviour with crosscutting behaviour” [2] by providing powerful facilities for intercepting, controlling, and manipulating method invocations. Nevertheless, it introduces numerous concepts in addition to bare methods, such as filter interfaces, filter expressions, selectors, and superimpositions, while GVFs do not really introduce something new, but only extend the traditional concept of procedures/functions in a straightforward manner.

GVFs have some obvious similarities with generic functions in CLOS [11] (and other languages based on comparable ideas, e. g., Dylan [5]) 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 [12]. 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 GVFs 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 [4], which embody exactly the same idea as local virtual functions (and in fact have triggered the idea to extend the already existing concept of GVFs with LVFs). However, dynamically scoped functions do not allow to install *permanent* redefinitions of functions whose effect exceeds the lifetime of their defining scope. Furthermore, installing a large number of extensive redefinitions locally (instead of using global definitions for that purpose) might significantly reduce the readability of code.

Finally, the way `virtual` is used to call the previous branch of a GVF resembles the way `inner` is used in BETA [15]. 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 [20], Oberon modules [21], 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, i. e., establishing a unique initialization order among multiple translation units of a program which in turn defines a unique activation order for GVF branches, modules provide a simple yet effective way to enforce the well-known principle of information hiding [16]: 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 [18] to achieve that aim.

8 CONCLUSION

Global virtual functions have been presented as a straightforward extension of the traditional notion of procedures. Even though the basic concept is rather simple, it leads to a significant gain in expressiveness, covering object-oriented single, multiple, and predicate-based method dispatch as well as aspect-oriented before, after, and around advice, without requiring any additional language constructs for that purpose.

As another straightforward extension, local virtual functions and non-local jump statements have been added in order to extend the expressiveness and flexibility of functions once more. By effectively combining these basic building blocks, it is possible, for instance, to perform exception handling with the possibility of resumption (i. e., continuing execution at the point where an exception has been raised), without needing a dedicated exception handling mechanism provided by the programming language [10].

Global and local virtual functions constitute one of two core concepts of so-called *advanced procedural programming languages*, i. e., languages which are not based on object-oriented principles, but rather on the traditional concepts of procedural programming, i. e., data structures and procedures. However, by specifically generalizing these concepts, advanced procedural programming languages provide a surprisingly high degree of expressiveness and flexibility with a comparatively small number of concepts. Their second core concept, *open types*, which generalizes the traditional notion of record types, shall be described elsewhere.

REFERENCES

[1] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein: "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java." In: *Proc. 2000 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '00)* (Minneapolis, MN, October 2000). *ACM SIGPLAN Notices* 35 (10) October 2000, 130–145.

[2] L. Bergmans, M. Aksit: "Composing Multiple Concerns Using Composition Filters." *Communications of the ACM* 44 (10) October 2001, 51–57.

[3] C. Chambers, W. Chen: "Efficient Multiple and Predicate Dispatching." In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)* (Denver, CO, November 1999). *ACM SIGPLAN Notices* 34 (10) October 1999, 238–255.

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

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

[6] 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.

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

[8] 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

[9] 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.)

[10] C. Heinlein: “Local Virtual Functions.” In: R. Hirschfeld, R. Kowalczyk, A. Polze, M. Weske (eds.): *NODe 2005, GSEM 2005* (Erfurt, Germany, September 2005). Gesellschaft für Informatik e. V., Lecture Notes in Informatics P-69, 2005, 129–144.

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

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

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: “An Overview of AspectJ.” In: J. Lindskov Knudsen (ed.): *ECOOP 2001 – Object-Oriented Programming* (15th European Conference; Budapest, Hungary, June 2001; Proceedings). Lecture Notes in Computer Science 2072, Springer-Verlag, Berlin, 2001, 327–353.

[14] G. T. Leavens, T. D. Millstein: “Multiple Dispatch as Dispatch on Tuples.” In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ’98)* (Vancouver, BC, October 1998). *ACM SIGPLAN Notices* 33 (10) October 1998, 374–387.

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

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

[17] 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.

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

[19] 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.

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

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

ABOUT THE AUTHORS

Christian Heinlein received a Ph.D. in Computer Science from the University of Ulm in 2000. Currently, he works as a scientific assistant in the Department of Computer Structures at the University of Ulm. His research interests include programming language design in general, especially support for modular extensibility and unanticipated evolution of software systems. His email address is heinlein@informatik.uni-ulm.de.