Local Virtual Functions

Christian Heinlein

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

Abstract. Local functions provided by classical procedural programming languages are a useful concept for structuring and decomposing large functions into smaller and more comprehensible pieces. This is even more useful if local functions can be passed as parameters to other functions. On the other hand, this creates the problem of *dangling function references*, i. e., references to local functions whose enclosing function has terminated. To solve this problem, *local virtual functions* are presented as a restricted form of local functions which do not constitute functions in their own right, but are merely temporary redefinitions of already existing global functions. As such, local virtual functions turn out to be a straightforward generalization of *global virtual functions* proposed earlier by the author. After introducing the basic concepts, different applications are described which demonstrate its usefulness. In particular, exception handling with the possibility of resumption becomes possible, even in a language without an exception handling mechanism. Furthermore, the basic ideas to implement local virtual functions by means of a precompiler for C++ are explained.

1 Introduction

Local functions, i. e., functions (or procedures, methods, etc.) defined inside other functions, are provided by classical procedural programming languages such as Algol, Ada, and Pascal (including its successors Modula-2, Oberon, and Oberon-2), as well as many functional programming languages. On the other hand, languages influenced directly or indirectly by C, e. g., C++, C#, and Java, do not provide this possibility, and so many programmers today have got accustomed to live without it. Even though methods of *local classes*, i. e., classes defined inside functions, are similar to local functions at first sight, they suffer from severe restrictions, in particular that local variables of enclosing functions must not be used at all (e. g., in C++ [St00]) or only if they have been declared `final` (e. g., in Java [GJS96]). However, exactly the possibility to directly use local variables and parameters of enclosing functions without restrictions is one of the key advantages of local functions that distinguishes them from global functions (or methods of global classes). As has been shown in [He03a], this is even more advantageous if local functions can be passed as parameters to other functions which will use them as *callback functions*, e. g., to compare objects in a sorting algorithm or to perform some arbitrary action for each element of a tree during a traversal.

Unfortunately, passing around references to local functions is potentially dangerous for the same reason that passing around references to local variables is dangerous: One might easily end up with *dangling references*, i. e., references to objects or functions

which do not exist anymore since their enclosing function has terminated. To avoid this, the author has proposed *syntactic rules* to exclude the possibility of dangling function references by definition [He03a]. Here, the basic rule says that global and local functions might be freely passed as parameters to other functions, but must not be stored in *more global* variables, i. e., variables whose lifetime might exceed the lifetime of the referenced function. Even though this basic rule is rather simple and straightforward, several additional – and less straightforward – rules are needed to cover all practically relevant cases.

Therefore, a completely different approach to avoid the problem of dangling function references, called *local virtual functions*, is presented in the current paper. Here, the basic idea is that local functions are not functions in their own right, but only *temporary redefinitions* (or overridings) of already existing – but possibly empty – global functions. By that means, a reference to a local function `f` is actually a reference to the global function `f` which is temporarily redefined during the execution of some other function `g` (or a part of it). Therefore, calling `f` via a reference while `g` is being executed, actually executes the "local function" `f`, i. e., the temporary redefinition of the global function `f`. On the other hand, if `f` is called (either directly or via a reference) after the enclosing function `g` has terminated, the original global function `f` will be executed; in particular, the behaviour of such a call is always well-defined, in contrast to calling a truly local function via a dangling reference.

Since the same global function might be redefined multiple times by nestedly called other functions, a *stack* of temporary redefinitions might evolve, and a call to the function always executes the latest redefinition, i. e., the one that is currently on top of the stack. This behaviour fits very well with the concept of *global virtual functions* proposed earlier by the author [He03b, He05], where global functions can be redefined *permanently* by building up a *list* of redefinitions (which are also called *branches* of the function). Again, a call to the function always executes the latest redefinition, i. e., the last branch of this list, which is able to call the previous branch on demand, etc.

Section 2 briefly reviews the basic concept of global virtual functions and its generalization to local virtual functions. Furthermore, it introduces the idea of *non-local jump statements*, i. e., the possibility to use `return` (or `break`/`continue`) statements in a local function to immediately terminate an enclosing function (or a loop/an iteration of it). Section 3 describes four different application scenarios where local virtual functions turn out to be useful: temporary redefinitions of global functions, semi-global variables, iterator functions, and exception handling with the possibility of resumption. Section 4 sketches the basic ideas to implement local virtual functions as a language extension for C++, even though the concept as such is actually language-independent. Finally, Sec. 5 concludes the paper with a discussion of related work.

2 Concept

2.1 Global Virtual Functions

The concept of global virtual functions has been introduced earlier in [He03b], where they have been called *dynamic class methods* in Java, since class methods (i. e., `static` methods) in Java are comparable to global functions in other languages, as well as in [He05], where they have been called *virtual namespace functions*, since "global" functions in C++ might also be defined in namespaces.

Using C++ as the base language in the sequel, the basic idea is that a global function declared with the keyword `virtual` – which must only be applied to *member functions* of classes in original C++ – might be *redefined* or overridden with a new implementation later by simply providing a new definition of the function (which would lead to a compile time error for ordinary global functions). This kind of redefinition might be performed any number of times, and each new implementation (also called a *branch* of the function) is able to call the previous implementation (*previous branch*) on demand. By that means, a *linear list* of branches is built up, and a call to the function always calls the latest redefinition, i. e., the *last branch* of this list.

If all branches of a function are defined in the same translation unit, the notions of "later," "previous," and "last" simply refer to the textual order of the definitions. Otherwise, i. e., if branches are distributed over multiple translation units, a simple *module concept* similar to Modula-2 and Oberon [Wi88] is used to define a unique *initialization order* of modules which in turn defines a unique *activation order* of branches. This is described in more detail in [He05], but is actually irrelevant for the present paper.

2.2 Local Virtual Functions

Local virtual functions are a straightforward extension of global virtual functions, allowing a global virtual function to be *temporarily* redefined by a *local branch*, i. e., a branch defined locally in another function. This other function might be any kind of C++ function, i. e., an ordinary global function, a global branch of a global virtual function, or even a local branch of a global virtual function, leading to *nested local functions*. Furthermore, member functions of classes, including static member functions, constructors, destructors, and conversion functions, might also contain local virtual function definitions. Finally, overloaded operators are treated just like other functions, i. e., they might be both defined as global virtual functions and contain local virtual function definitions.

A local branch of a global virtual function is *activated*, i. e., pushed on the stack of local redefinitions of its function, when the control flow of its statically 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, cf. Sec. 2.3). 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 global virtual function 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 global virtual function, 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. 1).

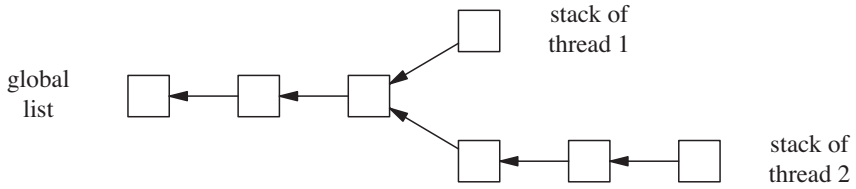

Figure 1: Global and thread-local branches of a global virtual function

2.3 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 might be 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.) 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[1] with one or more `extern` keywords to indicate that the statement shall be executed as if it were part of the *n*th statically enclosing function where *n* is the number of `extern` keywords given (cf. Fig. 2).

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. Therefore, the effect of a non-local jump

[1] `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.

```
// Global virtual function f.
virtual int f () { return 0; }

// Ordinary global function g.
bool g () {
  // Loop statement in g.
  while (......) {
    // Local redefinition of f.
    virtual int f () {
      // Loop statement in local f.
      while (......) {
        // Terminate loop statement in local f.
        if (......) break;

        // Terminate loop statement in g.
        if (......) extern break;

        // Terminate local f with result 1.
        if (......) return 1;

        // Terminate g with result true.
        if (......) extern return true;
      }
    }

    // Code calling f directly or indirectly.
    ......
  }
}
```

Figure 2: Example of non-local jump statements

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 that place. 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.

3 Applications

3.1 Temporary Redefinitions of Functions

As will be described in more detail in Sec. 4.1, global virtual function definitions appearing in a C++ program are transformed to ordinary C++ functions (possessing some unique, system-generated name), plus some auxiliary data structures and functions, by means of a precompiler. This precompiler has been implemented by extending the EDG

C++ Front End[2] (simply called *parser* in the sequel) using global and local virtual functions, too, i. e., the final version of the precompiler is implemented by using its "own" language extensions.

To actually transform a global virtual function definition such as

```
virtual R f (P1 x1, ..., Pn xn) { ..... }
```

to an ordinary C++ function definition such as

```
R f__1234 (P1 x1, ..., Pn xn) { ..... }
```

the function `parse_decl`[3], which is called by the original parser to parse all kinds of declarations, has been permanently overridden by a branch that checks whether a global declaration starts with the keyword `virtual` (which is not allowed in normal C++ code). If this is the case, the keyword is removed, and the original implementation of `parse_decl` is called (using the keyword `virtual` as the name of a parameterless pseudo-function which calls the previous branch with the same arguments as the current branch) to parse the subsequent declaration as a normal function declaration (cf. Fig. 3).

However, to replace the original name of the function (`f` in the example) with some other name (`f__1234`), the function `parse_decl_id`, which is called indirectly by the original implementation of `parse_decl` to parse a "declarator id" (i. e., the name declared by a declaration), is temporarily overridden in this redefinition of `parse_decl`, before calling its original implementation, by a local virtual function that replaces the current token (`f`) with a different identifier (`f__1234`) before calling its original implementation.

By employing this strategy, language extensions such as global and local virtual functions can be implemented in a strictly modular way, i. e., without changing a single line of existing parser source code, and without needing a deep understanding of the parser's work.

3.2 Semi-Global Variables

The EDG parser mentioned in the previous section also contains two back ends to reproduce the parsed program, either as C or as C++ code. (The latter is actually used to implement the mentioned precompiler.) These back ends make heavy use of a large suite of output functions, such as `print_type`, `print_const`, `print_var`, etc. Since the precise output format depends on a large number of parameters (first of all, whether C or C++ code shall be produced, but also whether names must be qualified in particular contexts or not, etc.), each such function receives a pointer to a corresponding parameter record as an argument, which is inspected on demand and passed down to nested invocations of other output functions. (For example, `print_type` calls `print_const` if the type to print is a template id such as `X<1>` possessing an integer constant as a template argument.)

[2] See `http://www.edg.com`

[3] Function names and signatures have been changed resp. omitted, both to abstract from unimportant details and to respect the confidential disclosure agreement made with EDG.

```
// Global redefinition of parse_decl.
virtual void parse_decl (......) {
  // If the current token is the keyword "virtual":
  if (......) {
    // Remove the current token.
    ......
    // Local redefinition of parse_decl_id.
    virtual void parse_decl_id (......) {
      // Replace current identifier.
      ......
      // Call original implementation of parse_decl_id.
      virtual();
    }
    // Call original implementation of parse_decl
    // with redefinition of parse_decl_id in effect.
    virtual();
    // Perform additional transformations.
    ......
  }
  else  {
    // Delegate call to original implementation of parse_decl.
    virtual();
  }
}
```

Figure 3: Transformation of global virtual function definitions

Since every output function needs this parameter record and usually passes it down unchanged, it is tempting to provide it as a global data structure instead of declaring it as a parameter for every function. Alternatively, one could employ local virtual functions as follows to make this information *semi-global* and to pass it *implicitly* from one function call to another.

First, a parameterless global virtual function (call it `info`) is defined which returns a pointer to a default parameter record. Whenever an output function needs a particular parameter value, it calls this function to obtain the required information. If a client wants to call an output function with different parameters, it temporarily redefines the function `info` with a local virtual function returning a pointer to a different parameter record, before calling the output function (cf. Fig. 4).

By employing this strategy, both the potential dangers of truly global variables (especially in multi-threaded programs) and the burden of declaring additional parameters for many functions and passing down the same information from one function call to another can be avoided.

```
// Parameter record.
struct ParmRec { ..... };

// Return pointer to default parameter record.
virtual ParmRec* info () { ..... }

// Print constant c.
void print_const (Const* c) {
  // Call info to obtain parameter values.
  ..... info()->flag .....
}

// Print type t.
void print_type (Type* t) {
  // Call info to obtain parameter values.
  ..... info()->another_flag .....

  // Call print_const to print constants.
  print_const(.....);

  .....
}

// Client using output functions.
void client () {
  // Local redefinition of info.
  virtual ParmRec* info () {
    // Return pointer to different parameter record.
    .....
  }

  // Call output functions.
  print_type(.....);
  print_const(.....);
  .....
}
```

Figure 4: Example of semi-global variables

3.3 Iterator Functions

The GNU C library provides a function `twalk` that traverses a binary tree and executes a callback function passed as an argument for every node of the tree. Apart from the fact that C lacks language support for type-safe and convenient generic functions (i. e., templates as provided by C++), the usage of this function is complicated by the fact that the callback function containing the "loop body," i. e., the code that shall be executed for every element of the tree, must be a separate global function that cannot access the local variables of the function containing the actual "loop," i. e., the call to `twalk`. This is one of the reasons why modern container libraries such as the C++ Standard Template Li-

brary or the Java Collection Framework provide the concept of *iterators* acting as logical pointers to container elements, instead of *iterator functions* such as `twalk`.

If, however, local functions are available and can be passed as arguments to other functions, iterator functions similar to `twalk` can be used rather conveniently. For example, in C++ one might define a (template) function `loop` that uses an iterator `i` to iterate through an arbitrary container `c` of type `C` and calls the global virtual function `visit` for each element `*i` (cf. Fig. 5). The global definition of the latter is empty since it is expected to be temporarily redefined by client functions such as `sum` before calling `loop`. In the example of Fig. 5, `visit<int>`, i. e., the particular instance of the template function `visit` for `T` equal to `int`, is temporarily redefined to accumulate all values `x` in the local variable `s` of its enclosing function `sum`. With this redefinition in effect, `loop` is called with the `int` vector `v`, which will call `visit` for every element of the vector, i. e., actually the redefinition just provided.

```
// Callback function for loop.
template <typename T>
virtual void visit (const T& x) {}

// Loop through container c and call visit for each element.
template <typename C>
void loop (const C& c) {
  typename C::iterator i = c.begin();
  while (i != c.end()) visit(*i++);
}

// Return sum of elements contained in v.
int sum (const vector<int>& v) {
  int s = 0;
  virtual void visit<int> (const int& x) { s += x; }
  loop(v);
  return s;
}
```

Figure 5: Definition and application of an iteration function

3.4 Exception Handling with Resumption

The exception handling mechanisms provided by today's mainstream programming languages allow programs to *throw* an exception at some point and to *catch* it at some other point in a *dynamically enclosing scope*, causing all intermediate scopes to be terminated abruptly. In particular, it is impossible to *resume* execution at the point where the exception has been thrown, even if its cause could have been removed.

To achieve such a behaviour, one must not throw an exception when encountering a problem such as lack of dynamic memory, but rather call a handler *function* that tries to fix the problem (e. g., by freeing some data structures which merely cache intermediate results which can be recomputed on demand) and either returns normally (if it was successful) or actually throws an exception (otherwise). This technique is used, for example,

in C++, where a handler function registered with `set_new_handler` is called when the allocation function implementing the `new` operator has not been able to allocate sufficient memory. If this handler function returns normally, `new` repeats its attempt to allocate memory; otherwise the handler function is expected to throw a `bad_alloc` exception [St00].

Even though this is a viable way from a pragmatic point of view, it actually requires the combination of two different mechanisms – function calls and exception handling – to appropriately deal with exceptional situations, even though exception handling has been explicitly introduced into programming languages to provide a single coherent mechanism to deal with exceptions. To make do with a single mechanism, one either needs an exception handling mechanism supporting resumption (cf. [Gr05]) – or a function mechanism supporting abrupt termination. Non-local jumps in local virtual functions exactly support the latter, and using this mechanism actually makes exceptions dispensable, as will be explained in the sequel.

To *declare* a new kind of exception, such as `out_of_memory`, one does not introduce this as a new type (e. g., as a subclass of `Throwable` in Java), but rather as a global virtual function whose initial implementation simply aborts program execution, possibly after printing some appropriate error message.

To *throw* such an exception, one simply calls this function. Depending on the particular kind of exception, the function might take parameters whose values serve to describe the exceptional situation in more detail, in the same way constructors of exception types might have parameters. As long as the function has not been redefined, calling it leads to program termination, in the same way as throwing an exception that is not caught anywhere leads to program termination.

To *catch* such an exception in some dynamically enclosing scope, one temporarily redefines the global virtual function by a local virtual function, whose body therefore corresponds to a `catch` block for the respective exception, while the code following the local function in the same scope corresponds to the respective `try` block: Whenever the function is called directly or indirectly from within this code, the local redefinition will be called instead of the initial global definition, in the same way as throwing an exception from within a `try` block causes execution of an accompanying `catch` block.

This local function has basically three possibilities to deal with the exceptional situation: First, it might cure the problem and return normally, causing execution to be continued at the point where the function has been called, i. e., to *resume* execution at the point where the exception has been "thrown." Second, it might use a non-local jump statement to abruptly terminate its own execution as well as all intermediate dynamic scopes up to one of its statically enclosing scopes (cf. Fig. 6[4]). The resulting control flow is very similar to that achieved with normal exception handling where the abrupt termination already happens during the search for a matching `catch` block, i. e., *before* this block is executed, while here it is performed only *after* the "handler" has decided to do so. Nevertheless, the point of continuation after execution of the "handler" is in the same scope in both cases.

[4] `do { ..... } while (false)` is an idiom describing a "loop" that is executed exactly once. It differs from a simple block since (`extern`) `break` can be used to terminate it prematurely.

```
// "Declare" out_of_memory exception.
virtual void out_of_memory () {
  cerr << "out of memory" << endl;
  abort();
}

// Install out_of_memory as new_handler.
set_new_handler(out_of_memory);

void f () {
  // "Try/catch statement".
  do {
    // "Catch" out_of_memory exception.
    virtual void out_of_memory () {
      // Try to free some dynamic data.
      ......

      // If successful, "resume",
      // otherwise terminate "try/catch statement".
      if (......) return;
      else extern break;
    }

    // Use operator new directly
    // or call functions using it.
    ......
  } while (false);

  // Other code.
  ......
}
```

Figure 6: Example of exception handling with local virtual functions

The third possibility for the local function to deal with the exceptional situation is to delegate the call to its previous implementation, i. e., to *rethrow* the exception. If no other redefinitions of the function are currently in effect, this will call the initial global definition, i. e., abort the program. Otherwise, however, the exception is propagated to the next responsible "handler," in the same way as a rethrown exception, and this handler has the same three choices to deal with it.

In addition to the possibility of temporary redefining an exception function, it is also possible to redefine it globally, i. e., to install a kind of global "catch block" for the exception.

4 Implementation

4.1 Global Virtual Functions

As already mentioned in Sec. 3.1, every definition of a global virtual function is transformed to an ordinary C++ function (possessing some unique, system-generated name) plus some auxiliary data structures and functions by means of a precompiler. The auxiliary data structures are actually global function pointer variables used to represent the list of global branches and to enable each branch to call its previous branch via such a variable. The auxiliary functions are on the one hand an empty *branch zero* representing the previous branch of the first branch and on the other hand a *dispatch function* possessing the same signature as the global virtual function itself and calling its last branch via one of the function pointer variables. This is the function that is actually called when the global virtual function is called from anywhere in the program. A more detailed description is given in [He03b] and [He05].

4.2 Local Virtual Functions

A local virtual function definition is basically transformed to a member function of a local auxiliary class. 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.

The only problem with this approach is the fact that member functions of local classes must not use local variables of the enclosing function [St00]. The reason for this restriction is the possibility that an instance of a local class might survive the execution of the enclosing function (e. g., if it is allocated dynamically), and then calling a member function of this instance would lead to undefined behaviour if this function tries to access variables of the no longer existing enclosing function.

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 (cf. Fig. 7). Using this trick, local variables of the enclosing function can be used in the local function without restrictions. On the other hand, the problem mentioned above might reappear in principle. However, since the only instance of the auxiliary class that will be created is a local one, it will disappear as soon as the enclosing function (actually the enclosing block) terminates.

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 mentioned in Sec. 4.1 uses this variable to locate the topmost element of the stack (belonging to the current thread) and call its member function `operator()` or – if the stack is currently empty – call the last global branch as described earlier.

```
// Source code:
void g () {
  // Local variables of g.
  int a; bool b;

  // Local redefinition of some global virtual function f
  // using a and b.
  virtual void f () { ..... a ..... b ..... }

  // Other code of g.
  ......
}

// Transformed code:
void g () {
  // Local variables of g.
  int a; bool b;

  // Auxiliary class.
  class f__1234 {
    // References to local variables of g.
    int& a; bool& b;

    // Constructor initializing these references
    // and pushing instance on stack of local redefinitions.
    f__1234 (int& a, bool& b) : a(a), b(b) { ..... }

    // Destructor popping instance from stack.
    ~f__1234 () { ..... }

    // Member function replacing local function.
    void operator() () { ..... a ..... b ..... }
  };

  // Instance of auxiliary class receiving references
  // to local variables of g as constructor arguments.
  f__1234 f__inst__1234(a, b);

  // Other code of g.
  ......
}
```

Figure 7: Implementation of local virtual functions

4.3 Non-Local Jump Statements

As already mentioned in Sec. 2.3, 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 that place. This is exactly the way a non-local jump statement is implemented by the precompiler: It is transformed to a statement throwing an exception which encodes the kind of statement

(return, break, or continue), the designated enclosing function (using unique numbers), and – in the case of a return statement with an expression – the value of this expression. Furthermore, every block containing local function definitions is replaced by a try block with accompanying catch blocks which catch these kinds of exceptions and execute the corresponding normal jump statement (cf. Fig. 8).

```
// Ordinary global function g.
bool g () {
  // Loop statement in g.
  while (......) {
    try {
      class f__1234 {
        ......

        // Local redefinition of f.
        int operator() () {
          // Loop statement in local f.
          while (......) {
            // Terminate loop statement in local f.
            if (......) break;

            // Terminate loop statement in g.
            if (......) throw Break__<5678>();

            // Terminate local f with result 1.
            if (......) return 1;

            // Terminate g with result true.
            if (......) throw Return__<5678, bool>(true);
          }
        }
      };
      f__1234 f__inst__1234;

      // Code calling f directly or indirectly.
      ......
    }
    catch (Break__<5678>) { break; }
    catch (Return__<5678, bool> r) { return r.value; }
  }
}
```

Figure 8: Implementation of non-local jump statements (transformed code for Fig. 2)

5 Related Work

The most obvious related work to local virtual functions are *dynamically scoped functions* as proposed by Costanza [Co03], and in fact, the idea to extend the already existing concept of global virtual functions with local virtual functions has been influenced by

this work. Non-local jump statements, however, are an important extension of this approach, allowing local virtual functions to be used for exception handling where a handler can choose to either terminate or resume execution at the point where the exception has been "thrown" (cf. Sec. 3.4). As already pointed out there, employing local virtual functions in that way actually makes a separate exception handling mechanism of a programming language dispensable (at least on the language level).

Costanza's implementation of dynamically scoped functions is based on Common Lisp's *special variables* [St90], i. e., dynamically scoped variables, which correspond closely to the idea of semi-global variables mentioned in Sec. 3.2. In fact, the standard print functions of Common Lisp make heavy use of such variables in the same manner as described there to avoid cluttering their signatures with lots of parameters determining the exact output format.

Dynamic variables proposed by Hanson and Proebsting [HP01] basically implement the same concept in an imperative language such as C++, and the authors describe similar advantages with respect to parameter lists of functions. They also point out that there is a close relationship of dynamic variables to exceptions, both on a conceptual level – dynamic variables are a data construct with dynamic scope, while exception handling is a control construct with dynamic scope – and in terms of implementation techniques – techniques developed for efficient implementations of exception handling can also be used for efficient implementations of dynamic variables.

Implicit parameters described by Lewis et. al [Le00] basically pursue the same goal, but employ a different strategy: Instead of permitting functions to access variables of other dynamic scopes, i. e., to search for the most recent definition of an identifier in the call stack, functions might have implicit parameters whose values are inferred from the context and which are automatically passed on to nested function calls on demand.

Local virtual functions differ from true *closures* provided by many functional languages, since their "lifetime" is restricted by the "lifetime" of their statically enclosing function. Therefore, it is sufficient to keep their "environment" on the procedure stack instead of allocating (and garbage-collecting) it on the heap.

As Costanza points out, dynamically scoped functions can be seen "as the essence of AOP" (aspect-oriented programming) – or at least as one of two essential parts, with *quantification*, e. g., by means of *pointcut expressions*, being the other one [FF00]. In fact, dynamically scoped functions and likewise local virtual functions can be seen as *around advice* associated with a *call* or *execution join point* (designating the global virtual function being redefined) that is restricted by a *cflow join point* (designating the function or block containing the local virtual function definition). However, in contrast to full-fledged aspect-oriented programming languages such as AspectJ [Ki01] or AspectC++ [SGS02], which extend their base language Java resp. C++ with a large number of additional language constructs, global and local virtual functions are a rather small extension of the fundamental procedure concept of imperative languages, which nevertheless results in a significant gain of expressiveness and flexibility.

In fact, global and local virtual functions constitute one major building block of so-called *advanced procedural programming languages*, whose main goal is to provide the expressiveness and flexibility of object-oriented and aspect-oriented languages which a significantly smaller number of language constructs [He05].

References

[Co03] P. Costanza: "Dynamically Scoped Functions as the Essence of AOP." *ACM SIGPLAN Notices* 38 (8) August 2003, 29–36.

[FF00] 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).

[GJS96] J. Gosling, B. Joy, G. Steele: *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[Gr05] A. Gruler, C. Heinlein: "Exception Handling with Resumption: Design and Implementation in Java." In: H. R. Arabnia (ed.): *Proc. Int. Conf. on Programming Languages and Compilers (PLC'05)* (Las Vegas, NV, June 2005), 165–171.

[He03a] C. Heinlein: "Safely Extending Procedure Types to Allow Nested Procedures as Values." In: L. Böszörményi, P. Schojer (eds.): *Modular Programming Languages* (Joint Modular Languages Conference, JMLC 2003; Klagenfurt, Austria, August 2003; Proceedings). Lecture Notes in Computer Science 2789, Springer-Verlag, Berlin, 2003, 144–149.

[He03b] 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. (An extended version is available as a Technical Report at `http://www.informatik.uni-ulm.de/pw/berichte/`)

[He05] C. Heinlein: "Virtual Namespace Functions: An Alternative to Virtual Member Functions in C++ and Advice in AspectC++." In: *Proc. 2005 ACM Symposium on Applied Computing (SAC)* (Santa Fe, New Mexico, March 2005), 1274–1281.

[HP01] D. R. Hanson, T. A. Proebsting: "Dynamic Variables." In: *Proc. 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (Snowbird, UT, June 2001). ACM, 2001, 264–273.

[Ki01] 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.

[Le00] J. R. Lewis, M. B. Shields, E. Meijer, J. Launchbury: "Implicit Parameters: Dynamic Scoping with Static Types." In: *Proc. 27th ACM Symp. on Principles of Programming Languages* (Boston, MA, January 2000). ACM, 2000, 108–118.

[SGS02] 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.

[St00] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.

[St90] G. L. Steele Jr.: *Common Lisp: The Language* (Second Edition). Digital Press, Bedford, MA, 1990.

[Wi88] N. Wirth: "The Programming Language Oberon." *Software—Practice and Experience* 18 (7) July 1988, 671–690.