

A Static Framework for Scalable Emulation of Evaluation Semantics

Andreas P. Priesnitz

Chalmers Univ. of Technology
Dept. of Computer Science and Engineering
Gothenburg, Sweden
`priesnit@cs.chalmers.se`

Abstract. The power of a programming language depends to a significant extent on its semantics of expression evaluation. It is therefore rewarding and popular to emulate nonexistent evaluation features by library constructs. For instance, one can emulate the functional programming idiom of a (partially) unbound function in an imperative language by providing special functor types. Their instances are created anonymously and represent late bindings if used as function arguments. This approach is of limited scalability to further emulations in this style, because each function implementation has to account for any possible combination of such special argument types. We propose a library-based framework that systematically supports the emulation of evaluation semantics without increasing the complexity order of function implementations. C++ as implementation language allows applying these constructs statically and therefore to avoid performance penalties at run time.

1 Introduction

Expression evaluation semantics influence both code expressiveness and the generality and efficiency of software, as many examples illustrate:

- *sequential* (ordered) or *parallel* (unordered) evaluation of the expression operands,
- *immediate* or *delayed* evaluation of the expression operands in *applicative* or *normal* order, i.e., before or after binding them,
- *partial* evaluation of partially bound expressions,
- *static* evaluation of constant value expressions,
- *short-circuit* evaluation of an expression, i.e., just to the extent necessary to obtain its value, or
- *side effects* of evaluation, e.g., logging or profiling.

Evaluation semantics that are not provided by the chosen programming language can possibly be emulated by constructs combining existing language features. The benefit of implementing evaluation semantics by language means is the flexibility of providing, adapting, specializing, and combining them in libraries, in contrast to the fixed and universal specification of evaluation semantics at the

language level. This ideal has been pursued systematically by SmallTalk, which allows one to dynamically define, modify, and apply constructs that determine evaluation semantics.

Under the demand for high performance, though, a purely dynamic solution is inadequate. Alternative approaches emulate evaluation semantics statically by introducing special types that trigger particular evaluation of a function call if used as its argument types. This idiom relies on a corresponding parameter type dispatch in the implementation of each function whose evaluation should obey the extended semantics. The usual overload dispatch mechanisms cause programming efforts for these function implementations to grow at least linearly in terms of the number of different cases of emulated semantics. This behavior inhibits scalability and prohibits extensions that affect existing code.

We present a general framework for emulating evaluation semantics that keeps the costs of each function implementation at constant order. All function calls are delegated to a single generic binding construct. This construct both detects and performs any extension of the native evaluation semantics for corresponding argument types.

Sect. 2 motivates our choice of C++ as the underlying language, due to its static meta-programming facilities, which allows introducing such constructs without causing run-time performance overhead.

Sect. 3 illustrates by examples how conventional approaches of emulating evaluation semantics share the disadvantage to increase function implementation complexity. This complexity is not critical in the case of emulating a single case of evaluation semantics, whereas the combination of the given examples demonstrates that these approaches are practically not scalable to a simultaneous emulation of different cases.

Sect. 4 presents our solution that bases on an alternative representation of function definitions which allows providing meta-information for the function bindings. Their invocations are delegated to a central, generic construct, which determines the evaluation process, possibly respecting that binding meta-information. The effort of implementing a function turns out to be independent of the number of cases of emulated evaluation semantics.

Sect. 5 summarizes results and observations.

2 Underlying Language Requirements

In this section, we reason about the dependence of emulations of evaluation semantics on features of the underlying language and postulate according prerequisites on the latter. When developing computationally intensive software, e.g., a general purpose graph or matrix library, it is a major demand that its run-time performance be comparable to alternative lower level implementations that were adapted manually to particular applications. Under the requirement of utmost efficiency, emulations of evaluation semantics have to be carried out statically wherever possible in order to shift efforts from run to compile time.

Consequently, the underlying language is required to be statically typed to allow to detect statically whether emulated semantics apply.

On the other hand, a library of constructs that emulate evaluation semantics that do not exist in the underlying language may be considered an embedded language. Programs in this language are translated to code in the underlying language, virtually extending the latter by the emulated evaluation semantics. The power of the embedded language is necessarily restricted by the kind of translations that can be expressed by the underlying language. As this language is expected to be statically typed, and as the evaluation of expressions involves both reasoning upon the involved types and upon providing resulting types, the underlying language has to provide static meta-programming facilities itself to allow to translate corresponding constructs. This requirement is met by C++, whose feature of *template specialization* provides a Turing-complete static meta-programming language [4].

3 Conventional Static Emulation of Evaluation Semantics

Two established examples of static, thus efficient emulations of evaluation semantics by compile-time meta-programming in C++ will be outlined in the following. The implementation of each example relies on a statically polymorphic definition of each function for whose calls the provided evaluation semantics are intended to hold. The implementation of a combination of both cases of emulating evaluation semantics, though, is shown to lead to a problem of in general prohibitively large, possibly exponential complexity.

3.1 Constant Evaluation

Expressions that involve constants should be (partially) evaluated at compile time to eliminate unnecessary run-time overhead. This effect can be achieved for integral constant expressions in C++—which offers very limited support of constant computations itself—by static meta-programming [19].

For instance, the binomial coefficient of two constant natural numbers can be computed statically by recursive template instantiation initiated by an expression like

```
Binomial< 4, 3 >::RESULT
```

instead of the call

```
binomial( 4, 3 )
```

of a corresponding function, which is bound and evaluated dynamically, thus causing run-time costs.

Although the appropriate alternative should be selected automatically, just based on the fact whether the arguments are given statically or dynamically, the syntactical difference forces the programmer to explicitly choose which version

of the function to call and therefore to invariantly decide whether to apply depending code on static or on dynamic data, which unnecessarily restricts its applicability. Instead, the binding time should be transparent to cover either case by equal syntax.

To achieve such transparency, it was suggested to map the problem settings to a meta level by representing each constant value as an object of unique type [1]. This type is given as the corresponding instance of a constant natural number parameter template class called, e.g., `Unsigned`. Hence, the former expression is replaced by calling an alternative (overloaded) version

```
binomial( Unsigned<4>(), Unsigned<3>() )
```

of `binomial`, which is implemented by means of the recursive template class `Binomial` to return an object of type `Unsigned<64>`, as 64 is the binomial of 4 over 3. Note that the semantics of objects of `Unsigned` instance types in the dynamic context is limited to their trivial construction, copy and destruction, which are empty instructions that are erased given sufficiently proper optimization by the compiler. `Unsigned` serves as a marker to indicate that the value is static rather than dynamic, which fact shall be called the argument's *technical category* regarding this particular emulation of evaluation semantics.

To cover mixed cases of both static and dynamic arguments as well, which may be evaluated partially, `binomial` has to accept each argument's type to be either the built-in type `unsigned` or an instance of the template `Unsigned` (in C++, this kind of polymorphism is realized by *overloading*). Therefore, there have to be 4 different definitions of `binomial`, or, generally, 2^a definitions for an a -ary function. Considering that a usually is a rather small number, this amount of variation is neither fully convenient nor prohibitively complex to be handled by the developer. Consequently, the given approach is not ideal, but fairly appropriate to implement a particular emulation of evaluation semantics.

3.2 Delayed Evaluation

Lambda expressions representing anonymous functions were emulated in C++ in a static manner—therefore not causing unnecessary run-time overhead—in libraries like `BLL` [9, 8], `FACT!` [16, 17], `FC++` [11, 14], and `Fusion` [6]. Besides various technical differences and implications, their basic construct is a template class called, e.g., `Lambda`, which marks objects of its instances as representing partially bound expressions, opposed to unmarked objects resulting from bound expressions. This binary alternative forms another technical category an argument may have within this emulation of evaluation semantics. A special placeholder instance of `Lambda` is used to denote free variables. If any argument of an expression is of `Lambda` instance type, a new `Lambda` instance type object representing the partially bound expression is created. In other words, the expression property of being partially bound is supposed to be *transitive*, implicitly carrying over to enclosing expressions.

In practice, partially bound arguments in a function are detected by matching overloaded definitions of the corresponding monomorphic function. For instance,

```
template <class Exp_, typename Arg_>
Lambda<...> binomial(Lambda<Exp_> lambda, Arg_ arg);
```

merely creates an instance of the return type, whose argument type is not of interest in this context. Any non-Lambda argument is bound, i.e., captured by the result.

Again, we need 4 different definitions to cover properly the cases of either argument being partially bound or not. For an a -ary function, 2^a definitions have to be provided to implement this behavior, which is feasible only for moderate values of a .

3.3 Complexity Problem

Each of the presented examples of emulating evaluation semantics requires to statically dispatch the parameter types for each function implementation, which is realized in C++ by corresponding function definition overloads. A more complex, but analogous example are implementations *expression templates* [18, 3], which are mandatory for state-of-the-art linear algebra libraries. The programmer has to have the necessary understanding of those implementations to provide appropriate function definitions. This burden is bearable if just a single case of emulated evaluation semantics is implemented, but increases when different cases of emulating evaluation semantics are combined, forcing to weave those alternatives to respect any combination of arguments appropriately.

The complexity increment can not be generally quantified. In the worst case, i.e., if each combination of argument types has to be dealt with by a separate function definition, it is of exponential order. In the best case, it is of linear order, which is only achieved if:

- the different cases of evaluation semantics are strictly ordered by priority, i.e., of any two cases one is more fundamental than the other in the sense that the former applies exclusively, ignoring the latter, whenever the arguments indicate that both cases have to be considered, and if
- the language offers a dispatching mechanism of matching a given expression with the corresponding definition of the polymorphic function that is capable of reflecting that priority and therefore allows to reduce the number of definitions to a minimum. (This is not the case in C++, which follows a best-match rule!)

For example, to support both constant and delayed evaluation of `binomial`, each argument may be either a dynamic value, a static value, a partially bound expression involving dynamic values or a partially bound expression involving just static values. One might simply provide definitions for all 16 possible combinations of those 4 technical categories for the two parameters. A minimal solution would be to provide 7 alternatives: one default implementation plus three alternatives for each case of emulating evaluation semantics, triggered by either one or both of the arguments. These 7 alternatives have to be considered in the following order:

1. If both arguments are partially bound expressions, create a new partially bound `binomial` expression object bound with those.
2. If only the first argument is partially bound, create a new partially bound `binomial` expression object bound with the first argument and with a new, completely bound expression object that provides the second argument as its result.
3. Same as before, the arguments' roles being swapped.
4. If both arguments are constant representations, evaluate the expression at compile time and provide the result as a constant representation.
5. If only the first argument is a constant representation (from which follows that the second is an `unsigned` value), partially evaluate the expression for the first argument at compile time, convert resulting constant representations back to their run-time counterparts and evaluate the resulting expression at run-time.
6. Same as before, the arguments' roles being swapped.
7. Both arguments are `unsigned` values, evaluate the expression at run time.

If those two cases of evaluation semantics were not strictly ordered by priority, though, this dispatch would be more difficult and possibly not of linear complexity.

Therefore, if given c cases of evaluation semantics, one has to replicate for each a -ary function between $(2^a - 1)c + 1$ and 2^{ac} function definitions realizing a dispatch like the one sketched above, of which only steps 4 and 7 actually respect the functional semantics of `binomial`. This amount of code replication is of no practical sense even for a moderate value of c and explains why combinations of such emulations of evaluation semantics are hardly found in any library, which seriously inhibits their systematic application.

4 Generalized Function Specification

The coding complexity problem explained in Sect. 3.3 has to be solved by avoiding the repeated provision of technical issues of evaluation semantics emulation within each function implementation. Instead, parts of the implementations that deal with those technical issues have to be extracted and generically encoded in a single place, to which the evaluation of each function call is directed. The particular functional semantics of the function are considered only if neither of those generic cases applies.

The static examination and manipulation of the abstract syntax tree of an expression requires that static meta-information relating to the tree nodes—i.e., to its subexpressions—is provided, for instance, information on their types. The transformation from language level dynamic function bindings to a representation that is enriched by such binding information—which will be called a *specification* of the function, in contrast to the language element of a function definition—is performed in several steps outlined here.

Note, though, that this presentation is just intended to illustrate the topics of this text and therefore simplifies several technical details and subtleties of C++, which are discussed thoroughly elsewhere [13].

4.1 Generic Definition

Function definitions may in general not be restrictive on the evaluation semantics that influence their evaluation, to allow the independent provision of emulations of semantics without having to modify existing function definitions. On the other hand, the invocation of such emulations is triggered by the corresponding technical categories of the argument types. To support any emulation of evaluation semantics—even such which have not been implemented yet—and therefore to accept arguments of arbitrary technical category, the parameter types of these function definitions are generally chosen to be free. Consequently, no formal specification of parameter semantics is given at this point, leaving it to a later step to deal with these correspondingly, which means that typing is applied lazily.

```
template < typename P1, typename P2 >
... binomial( P1 p1, P2 p2 )           // Ellipsis 1
{
    ...                               // Ellipsis 2
}
```

In general, the return type is not known beforehand and might be provided by *traits* (like in the STL [15]) or an individual meta-function (like in the BLL [9]). Neither alternative is outlined here, as a different solution will be suggested in Sect. 4.2, where also appropriate replacements for the above ellipses will be given.

4.2 Hybrid Specification

Meta-information like the return type of a generic function has to be provided statically to not cause run-time overhead on its retrieval. The usage of separate constructs like the ones mentioned in Sect. 4.1 would split up the implementation of a function to a statically and a dynamically evaluated part, each providing its own syntax and therefore to be handled separately. To avoid the complications of such dual representation, the *hybrid function specification*

```
template < typename P1, typename P2 >
struct Binomial
{
    typedef ... Result;                // Ellipsis 3
    static Result result( P1 p1, P2 p2 )
    {
        ...                            // Ellipsis 4
    }
};
```

is proposed, that nests the static information, which is the function to apply and its return type, and the dynamic, which is the function body describing how to execute the function at run-time. This kind of specification allows to easily supply further meta-information on function bindings besides the return type if that is required to emulate some evaluation semantics.

The generic function `binomial`, as introduced in Sect. 4.1, merely serves as formal front-end to the given function specification, by forwarding each of its invocations to the `result` member function of the corresponding statically bound `Binomial` instance. Its return type is therefore given by

```
typename Binomial< P1, P2 >::Result
```

and its return value by

```
return Binomial< P1, P2 >::result( p1, p2 );
```

replacing the Ellipses 1 and 2 in the previous generic function definition, which itself does not introduce further functional semantics regarding the computation of the binomial. Such generic function definitions can therefore be generated automatically by means of a (the C++) preprocessor performing the corresponding literal replacements if invoked by a call like

```
Define(binomial,Binomial,2)
```

providing the function name, the specification name, and the arity as arguments.

In particular, this automation allows to hide the C++ peculiarities of generic function definition syntax and therefore significantly eases function implementation for the programmer.

4.3 Anonymous Specification

To allow the generic emulation of evaluation semantics, a function specification like `Binomial` introduced in Sect. 4.2 just delegates its evaluation to a higher-order formulation called `Bind`:

```
template < typename P0, typename P1, typename P2 >
struct Bind
{
    typedef ... Result; // Ellipsis 5
    static Result result( P0 p0, P1 p1, P2 p2 )
    {
        ... // Ellipsis 6
    }
};
```

Information on the functional semantics of calculating the binomial is supplied by a (stateless) “0th” argument, whose type is expected to be defined like

```

struct BinomialStrategy
{
    typedef unsigned P1;
    typedef unsigned P2;
    typedef unsigned Result;
    static Result result( P1 p1, P2 p2 )
    {
        ... // Computation
    }
};

```

and which models the strategy pattern, encapsulating the function implementation to apply in absence of emulated evaluation semantics. We leave the topic of how to provide implementations of the binomial for different “plain” argument types by side, as it is a significant yet separate issue. Here, we are only interested in modeling polymorphism with regard to types that differ by technical category, not to different models of the same concept.

Given the suggested convention, Ellipsis 3 is replaced by

```

typename Bind< BinomialStrategy, P1, P2 >::Result

```

and Ellipsis 4 by

```

return Bind< BinomialStrategy, P1, P2 >
    ::result( BinomialStrategy(), p1, p2 );

```

within the specification of `Binomial` in Sect. 4.2. Therefore, the preprocessor macro `Define`, which was proposed in Sect. 4.1 to automatically generate generic function definitions like that of `binomial` by lexical replacement, can be extended to also generate generic function specifications like that of `Binomial`.

In the places indicated by the Ellipses 5 and 6, an instance of `Bind` can analyze the argument types by static (possibly as well by dynamic) meta-functions to dispatch any relevant combination of their technical categories and to evoke proper treatment according to the indicated evaluation semantics. We leave open how analysis and reaction are encoded, as different established approaches fit into this framework and may be chosen upon separate considerations. We just mention some common alternatives, like using

- attribute as tags and switches or
- *inheritance* and *virtual functions*

to express technical categories and selection dynamically, and using

- attribute types as tags, captured by *traits* [12],
- *policies* [2],
- *adaptors*, mentioned in Sect. 3.2, or
- *associated types* as tags, captured by *concepts* [7],

and (partial) specialization of class templates to model static categories and selection.

To emulate further cases of evaluation semantics, e.g., parallel evaluation of subexpressions, merely this central dispatch mechanism has to be adapted, automatically affecting all functions that are implemented as referring to `Bind`.

Only if none of the cases of emulated evaluation semantics applies, the functional semantics of the invoked function matter. The code to be evaluated and its return type are then obtained as `P0::result` and `P0::Result`, as provided by the actual implementation of the function.

A trivial version of `Bind` could just provide

```
typename P0::Result
for Ellipsis5 and
return P0::result( p1, p2 );
```

for Ellipsis6, applying common C++ evaluation semantics to function calls.

But there is a fundamental and notorious difference to the evaluation of a plain C++ function: Due to the genericity of the constructs involved, types are checked late, where the actual operations to perform are specified. Calls by invalid argument types are only detected at that point, remote from where the function was called. Error messages issued during the type check of the underlying language are therefore only vaguely related to the code in the embedded language. From the point of view of our intents, though, this effect is desired, as the behavior upon erroneous calls forms part of the evaluation semantics.

Thus, separate efforts may and must be undertaken to emulate evaluation semantics for the case of erroneous calls by invalid arguments. Again, different issues can be covered, e.g., the detailedness of error messages. We leave this topic out from this discussion, as various solutions, e.g., the `enable_if` construct [10], have been proposed and established, e.g., in `boost` libraries [5].

Similarly, our approach suffers from the usual problems of template meta-programming applications: Compilation efforts may increase dramatically, debugging is highly problematic—at least for the developer—and separate compilation is impossible. Some of these disadvantages might be diminished by the ongoing evolution of C++ compilers and development tools. Other apparent disadvantages change shade if they are judged from a different angle of looking at the roles of compilers and libraries, considering the overall costs of language, embedded language, and application development.

5 Summary

The emulation of evaluation semantics that are not provided by a programming language by constructs expressed in the same language—and most likely provided in a library—allows overcoming restrictions of the language without switching to another language and without preprocessing the code by separate

means. To avoid run-time overhead, such emulating constructs must be evaluated statically, which requires the language to offer both static typing and, correspondingly, static meta-programming facilities to operate on types. We chose C++, which fulfills those requirements, as underlying language for emulation.

Existing approaches focus on emulating single cases of evaluation semantics and prove to be not scalable in practice to emulating different cases of evaluation semantics simultaneously. The reason is that , as they add significant complexity to each function implementation.

The presented framework provides a generalized way of emulating evaluation semantics by mapping generic function definitions to hybrid function specifications, which provide both statically and dynamically relevant information, and by delegating the evaluation of these specifications to a higher-order formulation using the `Bind` construct. The instances of `Bind` represent nodes of the abstract syntax tree and provide related information. Within `Bind`, static meta-functions can be applied to decide whether to apply some emulation of evaluation semantics or to fall back to the default behavior using common evaluation semantics as provided by the language.

If function implementations generally adhere to that scheme, the emulation of evaluation semantics can be introduced locally by modifying the definition of `Bind`, without the needs to replicate any aspect of the emulation per function. The decision whether the emulation is applied is taken statically, not causing run-time overhead. The efficiency of the emulation itself depends on its implementation, which usually can be provided in the conventional manner.

To introduce a new function in compliance with the framework's expectations, a programmer has to provide a preprocessor macro call like

```
Define(binomial,Binomial,2)
```

generating the generic function definition and specification, and the definition of a strategy like `BinomialStrategy` given in Sect. 4.3 describing how to evaluate a call to this function in the default case that none of the cases of emulated evaluation semantics applies. Given this rather simple pattern, the demand for constant programming complexity of implementing functions that are subject to emulated evaluation semantics is fulfilled.

Acknowledgments

The author is grateful to Sibylle Schupp and Marcin Zalewski for extensive and intensive discussions, to the MPOOL'07 reviewers for helpful comments, and to the Swedish Research Council (Vetenskapsrådet) for financial support of his work.

References

1. A. Alexandrescu. Mappings between types and values. *C/C++ Users J.*, 18(10), Oct. 2000.
2. A. Alexandrescu. *Modern C++ Design*. C++ In-Depth Series. Addison-Wesley, 2001.
3. J. Crotinger, S. Haney, S. Smith, and S. Karmesin. PETE: The portable expression template engine. *Dr. Dobbs's J.*, Oct. 1999. available at <http://www.ddj.com/documents/s=898/ddj9910h/>.
4. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.
5. B. Dawes and D. Abrahams. Boost homepage. <http://www.boost.org>.
6. J. de Guzman and D. Marsden. Fusion library homepage. http://spirit.sourceforge.net/dl_more/fusion_v2/libs/fusion, 2006.
7. D. Gregor and B. Stroustrup. Concepts. Technical Report N2042=06-0112, ISO JTC1/SC22/WG21 – C++ working group, June 2006.
8. J. Järvi and G. Powell. Boost lambda library (BLL) homepage. <http://www.boost.org/libs/lambda>, 2004.
9. J. Järvi, G. Powell, and A. Lumsdaine. The lambda library: Unnamed functions in C++. *Software: Practice and Experience*, 33(3):259–291, Mar. 2003.
10. J. Järvi, J. Willcock, H. Hinnant, and A. Lumsdaine. Function overloading based on arbitrary properties of types. *C/C++ Users J.*, 21(6):25–32, June 2003.
11. B. McNamara and Y. Smaragdakis. Functional programming with the FC++ library. *J. of Functional Programming*, 14(4):429–472, July 2004.
12. N. Myers. A new and useful template technique: “traits”. *C++ Report*, 7(5):32–35, June 1995.
13. A. P. Priesnitz. *Multistage Algorithms in C++*. PhD thesis, Univ. of Göttingen, Nov. 2005. available at <http://webdoc.sub.gwdg.de/diss/2006/priesnitz/>.
14. Y. Smaragdakis. FC++ homepage. <http://www.cc.gatech.edu/~yannis/fc++>, 2003.
15. A. A. Stepanov and M. Lee. The standard template library. Technical Report HPL-95-11(R.1), Hewlett Packard Laboratories, Palo Alto, CA, USA, Nov. 1995.
16. J. Striegnitz. Making C++ ready for algorithmic skeletons. Technical Report IB-2000-08, Research Centre Jülich, 2000.
17. J. Striegnitz. FACT! homepage. <http://www.kfa-juelich.de/zam/FACT>, 2003.
18. T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.
19. T. L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.