Science of Computer Programming 75 (2010) 773–786

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Dynamic algorithm selection for runtime concepts

Peter Pirkelbauer [a,*], Sean Parent [b], Mat Marcus, Bjarne Stroustrup [a]

[a] Texas A&M University, College Station, TAMU 3112, TX 77843, USA

[b] Adobe Systems Inc., San Jose, CA, USA

ARTICLE INFO

Article history:
Received 8 August 2008
Received in revised form 3 February 2009
Accepted 30 April 2009
Available online 12 May 2009

Keywords:
Generic programming
Runtime polymorphism
C++
Standard template library

ABSTRACT

A key benefit of generic programming is its support for producing modules with clean separation. In particular, generic algorithms are written to work with a wide variety of types without requiring modifications to them. The *Runtime concept* idiom extends this support by allowing unmodified concrete types to behave in a runtime polymorphic manner. In this paper, we describe one implementation of the runtime concept idiom, in the domain of the C++ standard template library (STL). We complement the runtime concept idiom with an algorithm library that considers both type and concept information to maximize performance when selecting algorithm implementations. We present two implementations, one in ISO C++ and one using an experimental language extension. We use our implementations to describe and measure the performance of runtime-polymorphic analogs of several STL algorithms. The tests demonstrate the effects of different compile-time vs. run-time algorithm selection choices.

1. Introduction

ISO C++ [16] supports multiple programming paradigms [29], notably object-oriented programming and generic programming. Object-oriented techniques are used when runtime polymorphic behavior is desired. When runtime polymorphism is not required, generic programming is used, as it offers non-intrusive, high performance compile-time polymorphism; examples include the C++ Standard Template Library (STL) [6], the Boost Libraries [1], Blitz++ [31], and STAPL [5].

Recent research has explored the possibility of a programming model that retains the advantages of generic programming, while borrowing elements from object-oriented programming, in order to support types to be used in a runtime-polymorphic manner. In [20], Parent introduces the notion of non-intrusive value-based runtime-polymorphism, which we will refer to as the *runtime concept* idiom. Marcus et al. [19,3], and Parent [21] extend this idea, presenting a library that encapsulates common tasks involved in the creation of efficient runtime concepts. Järvi et al. discuss generic polymorphism in the context of library adaptation [18].

A key idea in generic programming is the notion of a *concept*. A concept [15] is a set of syntactic and semantic requirements on types. Syntactic requirements stipulate the presence of operations and associated types. In the runtime concept idiom, a class C is used to model these syntactic requirements as operations. The binding from C to a particular concrete type T is delayed until runtime. Any type T that syntactically satisfies a concept's requirements can be used with code that is written in terms of the runtime concept.

* Corresponding author. Tel.: +1 979 845 2938.
E-mail addresses: peter.pirkelbauer@tamu.edu (P. Pirkelbauer), sparent@adobe.com (S. Parent), mmarcus@emarcus.org (M. Marcus), bs@cs.tamu.edu (B. Stroustrup).
URLs: http://parasol.tamu.edu/~peterp/ (P. Pirkelbauer), http://parasol.tamu.edu/~bs/ (B. Stroustrup).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.04.002

In this paper, we apply these principles to develop a runtime-polymorphic version of STL sequence containers and their associated iterators. Runtime concepts allow the definition of functions that operate on a variety of container types.

Consider a traditional generic function expressed using C++ templates:

```
// conventional template code
template <class Iterator>
Iterator
random_elem(Iterator begin, Iterator end) {
  typename Iterator::difference_type dist = distance(begin, end);
  return advance(begin, rand() % dist);
}
// ...
int elem = *random_elem(v.begin(), v.end()); // v is a vector of int
```

Objects of any type that meet the iterator requirement can be used as arguments to random_elem. However, those requirements cannot be naturally expressed in C++03, (though they can in C++0x [7]), and the complete function definition is needed for type checking and code generation. The resulting code is very efficient, but this style of generic programming does not lend itself to certain styles of software development (e.g. those relying on dynamic libraries).

We can write essentially the same code using the runtime concept idiom (the classes of the runtime concept idiom and their implementation are discussed in Section 3):

```
// with runtime concept idiom
wrapper_forward<int>
random_elem(wrapper_forward<int> begin, wrapper_forward<int> end) {
  wrapper_forward<int>::difference_type dist = distance(begin, end);
  return advance(begin, rand() % dist);
}
// ...
int elem = *random_elem(v.begin(), v.end()); // v is a vector of int
```

Here the binding between the iterator type and the function is handled at runtime and we can compile a use of random_elem with only the declarations of random_elem available:

```
// with runtime concept idiom:
wrapper_forward<int>
random_elem(wrapper_forward<int> begin, wrapper_forward<int> end);
// ...
int elem = *random_elem(v.begin(), v.end()); // v is a vector of int
```

By using runtime concepts, function implementations (e.g., random_elem) are isolated from client code. The parameter type wrapper_forward subsumes all types that model the concept forward-iterator. The implementation can be explicitly instantiated elsewhere for known element types, and need not be available to callers.

This reduced code exposure in header files makes runtime concepts suitable for (dynamically linked) libraries and when source code cannot be shared. However, the use of runtime concepts comes at a cost. The function random_elem is written in terms of the concept forward-iterator. The runtime complexity of distance and advance is $O(n)$ for forward-iterators, while it is constant time for random access iterators. Passing iterators of vector<int> as arguments would incur unnecessary runtime overhead.

When using STL iterators directly, the compiler checks that two iterators have the same type (though it does not validate that two iterators belong to the same container instance). As a consequence of subsuming iterators from various data structures under a single runtime concept, type checking that two iterators have the same concrete type is postponed until run time.

This paper makes the following contributions:

- We apply the runtime concept idiom to part of a core C++ library (STL) and analyze the runtime overhead.
- We enhance the runtime concept idiom with a prototype of an open, extensible, and loosely coupled algorithm library– a runtime counterpart of the STL algorithms. Its dispatch mechanism selects the best matching algorithm instance according to the runtime concept and type information of the actual arguments. This eliminates the need for dynamic dispatch during the execution of an STL algorithm when a matching algorithm instance is present. Runtime analogs of four STL algorithms are presented and their performance is analyzed.
- We present a design of the algorithm library in ISO C++ and one with open multi-methods, an experimental language extension that simplifies the implementation and makes dispatch more efficient.

Compared to our prior work [22], this article places emphasis on the algorithm library and makes the following additions: (1) detailed discussion of the implementation and its premises; (2) an extended performance analysis; (3) description of an

alternative implementation based on an experimental language extension that makes the design of the algorithm library more concise and dynamic algorithm selection more efficient; (4) comparison of both implementations in terms of dispatch performance under two different scenarios.

The structure of this paper is: Section 2 revisits fundamental ideas of generic programming and STL. Section 3 describes the application of the runtime concept idiom towards STL's iterators. Section 4 discusses our implementations of the loosely coupled algorithm library. Section 5 evaluates our prototype implementations and their runtime performance; Section 6 compares our model to alternatives; Section 7 points to possible extensions and summarizes our contribution.

2. Generic programming

The ideal for generic programming is to represent code at the highest level of abstraction without loss of efficiency in both actual execution speed and resource usage compared to the best code written through any other means. The general process to achieve this is known as *lifting*, a process of abstraction where the types used within a concrete algorithm are replaced by the semantic requirements of those types necessary for the algorithm to perform.

Semantic requirement: Types must satisfy these requirements in order to work properly with a generic algorithm. Semantic requirements are stated in tables, in documentation, and may at times be asserted within the code. Checking types against arbitrary semantic requirements is in general undecidable. Instead, compilers for current C++ check for the presence of syntactic constructs, which are assumed to meet the semantic requirements. For example, an implementation of **operator=** is expected make a copy.

Concept: Dealing with individual semantic requirements would be unmanageable for real code. However, sets of requirements can often be clustered into natural groups, known as *concepts*. Although any collection of requirements may define a concept, only concepts which enable new classes of algorithms are interesting.

Model: Any type that satisfies all specified requirements of a concept is said to be a model of that concept.

Generic algorithm: A generic algorithm is a derivative of a family of efficient algorithms, whose implementation is independent from concrete underlying data structures. The requirements that an algorithm imposes on a data structure can be grouped into concepts. An example that is part of the STL is find and the requirement on the template argument is to model a forward-iterator.

Concept refinement: A concept C_r that adds requirements to another concept C_0 is a concept refinement. The number of types that satisfy C_r is less or equal to the number of types that satisfy C_0. The number of algorithms that can be directly expressed with C_r is greater or equal than the number of algorithms expressed with C_0. For example, constant time random access, a requirement added by the concept randomaccess-iterator, enables the algorithm sort.

Algorithm refinement: Parallel with concept refinements, an algorithm can be refined to exploit the stronger concept requirements and achieve better space- and/or runtime-efficiency. For example, the complexity of reverse for a bidirectional-iterator is $O(n)$, while it is $O(n \lg n)$ for a forward-iterator (assuming less than $O(n)$ memory usage).

Regularity: Dehnert and Stepanov [12] define regularity based on the semantics of built-in types, their operators, the complexity requirements on the operators, and consistency conditions that a sequence of operations has to meet. Regularity is based on value-semantics and requires operations to construct, destruct, assign, swap, and equality-compare two instances of the same type. This is sufficient for a number of STL data structures and algorithms including vector, queue, reverse, find. A stronger definition adds operations to determine a total order, which enables the use of STL's map, set, sort. Programmers' likely familiarity with built-in types makes the notion of regularity important, because code written with built-in types in mind (e.g., STL data structures) will work equally well for regular user defined types. Consequently, conformance with regular semantics reduces the need for specialized and customized implementations.

2.1. Standard template library

The C++ STL provides implementations of efficient algorithms and data structures. The links between them are called iterators, which resemble pointers to elements plus operations that move this pointer to other elements in the container. Each data structure provides its own iterator implementation. The iterators can be grouped according to the access capabilities of their respective data structure. STL defines five iterator concepts: input iterator (for data sources), output iterator (for data sinks), forward iterator (for sequential and repeated access), bidirectional iterator, and random access iterator.

Algorithms are defined in terms of iterator concepts. At compile time the compiler statically selects the most suitable algorithm according to its category. The loose coupling of data structures and algorithms in terms of iterators solves the problem of providing separate algorithm implementations for each data structure. Instead of having about 720 (60*12) implementations, STL consists of about 60 algorithms plus 12 data structures [29]. Most algorithms operate on ranges. A range consists of a pair of iterators that delimits a sequence of elements. The first iterator points to the first element in the sequence, the second to the position one element past the end of the sequence. A common access pattern is to iterate through the range until the first and second iterator become equal.

3. Runtime concepts for STL iterators

This section presents our implementation of the runtime concept idiom for iterators in C++03. For a general treatment of the runtime concept idiom, along with its library support and optimizations we refer the reader to Marcus et al. [19] and [3]. We essentially follow Marcus et al.'s design, which extends the external polymorphism pattern [11] with a regular wrapper. The result is a three layered architecture: the concept layer, the model layer, and the wrapper layer. We illustrate the implementation of these classes and their interaction based on the concept forward iterator and its refinement bidirectional iterator. Each of the layers is parameterizable with a value-type and a reference-type. We omit the latter in our discussion.

3.1. Concept layer

The concept layer defines the runtime concept interfaces — abstract base classes without any data. Each of these interfaces defines a set of function signatures that reflect the concept requirements. The following code snippet shows parts of the forward-iterator class.

```
template <class ValueType>
struct concept_forward {
  virtual void next() = 0; // for operator++
  virtual bool equals(const concept_forward& rhs) const = 0; // for operator==
  virtual concept_forward& clone() const = 0; // for copy ctor, operator=
  virtual ValueType& deref() const = 0; // deref operator*
  // ...
};
```

For operators that have a prefix and a postfix version (e.g., **operator**++), the concept layer contains only one member function declaration (e.g., next), which returns **void**. The wrapper layer (Section 3.3) implements the prefix and postfix operator in terms of this function and returns the appropriate result.

Refinements of a runtime concept inherit from another runtime concept interface and add new operations or refine inherited signatures with covariant return types.

```
template <class ValueType>
struct concept_bidirectional : concept_forward<ValueType> {
  virtual void prev() = 0; // for operator--
  virtual concept_bidirectional& clone() const = 0; // override with covariant return type
  // ...
};
```

3.2. Model layer

The model layer defines model classes that inherit from the runtime concept interfaces. Models are parametrized classes, where the first template argument determines the concrete iterator type and the second template argument corresponds to the concept interface that this model will implement. For an iterator of list<**int**>, these would be list<**int**>::iterator and concept_bidirectional<**int**> respectively. At the root of the model-hierarchy is the class model_base that holds a copy of the concrete iterator (member variable it).

```
template <class Iterator, class IterConcept>
struct model_base : IterConcept { // connect implementation type to interface type
  Iterator it;
};
```

The class model_base together with its two template arguments Iterator and IterConcept already determine the model refinement.

The implementation of the pure virtual functions defined in the concept interfaces is accomplished by mixin techniques [26] and class hierarchy linearization [30]. For each runtime concept interface there exists a model class that implements the corresponding pure virtual functions. For example, model_bidirectional implements the pure virtual functions defined in concept_bidirectional. The model classes (directly or indirectly) derive from model_base and linearly mixin their function implementations, which forward the calls to the concrete iterator it.

```
template <class Iterator, class IterConcept>
struct model_forward : model_base<Iterator,IterConcept> {
  IterConcept& clone();
  bool equals(const concept_forward& rhs) const {
```

```
    assert(typeid(*this) == typeid(rhs));
    return this->it == static_cast<const model_forward&>(rhs).it;
  }
  /* ... */
};
template <class Iterator, class IterConcept>
struct model_bidirectional : model_forward<Iterator,IterConcept> {
  void prev();
  /* ... */
};
```

The model classes require the same template arguments as model_base. This is necessary because, for example, model_forward could be mixed in with a model_bidirectional to implement the functions defined by concept_forward. In this case, model_forward would indirectly derive from concept_bidirectional.

Operations with two polymorphic arguments (e.g., equals) require both arguments to have the same dynamic type. Since C++ does not allow covariant arguments, this restrictions cannot be modeled statically (i.e., equals takes a concept_forward), but the same type requirement allows the argument be cast to the dynamic type of the receiver object. This limits the use of the equality operator to comparisons with the same underlying iterator type; comparisons of an iterator with its const-counterpart are currently not supported, but could be encoded with true multi parametric functions (see. Section 4.3, double dispatch [9], or multi-method libraries [4,25]).

The factory-function generate_model selects a model refinement based on the concept of a concrete iterator. Its use guarantees the existence of exactly one runtime model type for each concrete iterator type. This allows operations with two runtime-concepts as arguments to rely on the same type requirement.

```
template<class Iterator>
typename map_iteratortag_to_model<Iterator>::type*
generate_model(const Iterator& it) {
  typedef typename map_iteratortag_to_model<Iterator>::type iterator_wrapper_type;
  return new iterator_wrapper_type(it);
}
```

The template meta function map_iteratortag_to_model uses the iterator category to generate the proper model type. For a list<int>::iterator, the generated class would be model_bidirectional<list<int>::iterator, concept_bidirectional<int>>. Then generate_model constructs an object of this type on the heap, and returns its address.

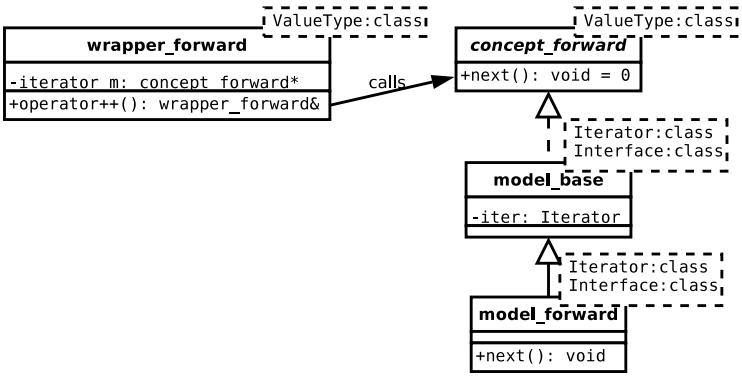
3.3. Wrapper layer

The wrapper layer defines regular classes that wrap a concept layer object and manage its lifetime.

```
template <class ValueType>
struct wrapper_forward {
  concept_forward<ValueType>* iterator_m;
  // Constructor that takes any iterator
  template <class ConcreteIterator>
  wrapper_forward<ValueType>(const ConcreteIterator& iter)
      : iterator_m(generate_model(iter)) {}
  wrapper_forward<ValueType>(const wrapper_forward<ValueType>& iter)
      : iterator_m(iter.iterator_m->clone()) {}
  // Postfix operator implemented in terms of concept_forward::next
  wrapper_forward<ValueType> operator++(int) {
    wrapper_forward<ValueType> tmp(*this);
    this->iterator_m->next();
    return tmp;
  }
  /* ... */
};
```

It implements operations that guarantee regular semantics (constructor, destructor, etc.) and other operations required by the STL concept. Here, the implementations of the copy constructor and the assignment operator make use of the function clone.

The following graphic summarizes the interaction of the different layers of the runtime concept idiom:



The wrapper class is the placeholder for a concrete iterator type and holds a pointer to an interface. The model base inherits from this interface and holds a copy of a concrete iterator (e.g., std::vector<**int**>::iterator), while the model classes implement the pure virtual functions. A member function call, for example **operator**++, invokes a virtual function of the concept interface (next), implemented by model_forward, which forwards the call to the **operator**++ of the concrete iterator. The functions in the wrapper class, and the call to the concrete iterator can be inlined. Each invocation of a wrapper function incurs the overhead of at most one virtual function call. Swap and move can be implemented by manipulating the concept interface pointer of the wrapper class [19].

4. The algorithms library

In this section, we address performance problems that arise in the presence of type erasure [2]. First, virtual functions incur runtime overhead and impede inlining. Second, the functions distance and advance in the body of random_elem are selected at compile time according to the static type of the wrapper class which can lead to less than optimal runtime complexity. A dispatch solution (algorithm library) that postpones the function selection until runtime, when dynamic information about the wrapped iterator type and its concept is available, can do better (as the C++98 STL does through static resolution). We outline the design ideals and discuss two implementations: one in ISO C++ and one using an experimental language feature, open multi-methods [23].

4.1. Design goals

Practical solutions to the dispatch problem should address the following issues:

Extensibility: The runtime concept idiom allows for separate compilation of library code and user code. Therefore, the library implementation cannot make any assumption about specific container and iterator types in the user code. The algorithm library enables user code to optimize the runtime of the library functions for the data structures it uses.

Applicable to algorithms operating on multiple ranges: For STL algorithms that operate on a single range (e.g., find, reverse, sort, etc.) the best function can be selected according to the dynamic type of a single argument (the begin of a range); we can assume that the end of the range has the same dynamic type. Some algorithms operate on two ranges (e.g., merge, set_union, transform, etc.) and require the dispatch mechanism to be sensitive in respect to more than one argument type to achieve optimal performance. Any implementation that supports dispatch according to more than one dynamic argument, requires some policy to handle ambiguities.

Conservative in executable size: The executable should not contain code for algorithms that are not needed. This means that our example library for random_elem contains algorithm instances only for advance and distance but not for any other STL algorithm.

Independence from runtime concept classes: Providing new algorithms and better algorithm instances has to be achievable without repeated modification of the runtime concept classes.

Usability: From the user perspective, programming with the algorithm library should be similar to programming with the STL and providing algorithms and algorithm instances should be straightforward.

Implementable in ISO C++: A practical solution should be implementable in current C++ and not require any language or compiler specific extension. From a research perspective, however, we are interested in finding generally usable abstraction mechanisms that improve the design and implementation.

Performance: To be usable independently from the size of the data set, the runtime overhead for making a dispatch decision should be as small as possible.

Particularly, the support for multiple arguments and extensibility is so important for generality and performance that the designers of compile time concepts for C++0x decided to postpone checking of specialized algorithms for some potential ambiguities to instantiation time [17].

4.2. Implementation in ISO C++

Our implementation prototypes an algorithms library for several STL algorithms. Each function in the library originates from an algorithm instantiation with one of the iterator wrappers or a concrete iterator. By default, the library contains an instance for the weakest concept an algorithm supports. For example, the default entry for lower_bound would be instantiated with wrapper_forward. These default instances are meant to serve as fallback-implementations. To improve performance, the system integrator or even a (dynamically loaded) library can add more specialized algorithm instances. This is essential to preserve the algorithmic efficiency of key STL algorithms. The default implementation and the added instances form an algorithm family. Each algorithm family is defined in terms of an existing STL algorithm and the iterator-value_types (e.g., algolib::advance<**int**>).

To provide for modular extensibility, we decouple the algorithm library from static dispatch structures (i.e., enumerations, switch statements, visitor classes, type-lists, variant types). This is, what visitor based implementations and many C++ multimethod libraries depend on (e.g., Loki library [4], Shopyrin's deferred recursive mechanism [25], doublecpp [9], GIL [10]). Other open method implementations depend on language extensions and require preprocessor/compiler support (e.g., Cmm [27], Omm [23]) to extract the information that is necessary for an efficient implementation. Instead, our implementation relies on runtime type information (RTTI) and data embedded in the concept and model classes.

The mechanics of our implementation rest on three core components:

- A *class hierarchy traversal* that returns the typename of the base class for a given typename (or class).
- An *associative data structure* that stores functions and the RTTI information of their runtime polymorphic argument.
- A *lookup mechanism* that uses the class hierarchy traversal to find the best matching algorithm instance from the associative data structure.

To remain independent from specific compilers and their application binary interface (ABI), we encode the refinement (inheritance) relationship by mapping a typeid of a class to the typeid of its base class. In the context of the modeled iterator hierarchy, the use of single inheritance suffices.

```
struct inheritance {
  static std::map<const std::type_info*, const std::type_info*> bases;

  inheritance(const std::type_info& base, const std::type_info& derived) {
    bases[&derived] = &base;
  }
};
```

To automatically keep track of the refinement relationships of all instantiated runtime concept classes, we augment the class model_base and all classes of the concept layer (e.g., concept_bidirectional), except the base (concept_forward), with a static data member inh. By making the constructor refer to inh we force inh's instantiation without incurring any runtime overhead because referencing inh has no direct effect and can be optimized away.

```
template <class ValueType>
struct concept_bidirectional : concept_forward<ValueType> {
  static inheritance inh;

  concept_bidirectional() { inh; } // triggers instantiation of inh
  // . . .
};
```

inh's constructor records the refinement relationship in the global mapping inheritance::bases that we later use to query a class for its base.

```
// static member initialization
// inh's constructor records the refinement relationship
template <class ValueType>
inheritance
concept_bidirectional<ValueType>::inh(typeid(concept_forward<ValueType>),
typeid(concept_bidirectional<ValueType>));
```

Since a library developer is unaware of the specific data structures in user code, it is at the discretion of the user to enhance the runtime by providing better algorithm instances. This is supported by two functions that take the algorithm family and an iterator type as argument. The first (add_generic) instantiates a function that rewraps the model into a more powerful

wrapper class specified by the iterator type and invokes the STL algorithm with it. The second, add_specific generates a function that peels off all runtime concept layers and invokes the STL algorithm with the concrete iterator. The following sample code shows how to register a generic implementation for wrapper_randomaccess and a specific for list<**int**>::iterator. Note that, the distinction by name of add_generic and add_specific is not necessary, but choosing to do so makes the use of virtual functions inside of an algorithm instance explicit.

```
// add generic implementation suitable for all random access iterators.
algolib::add_generic<
    algolib::advance<int>, // library name
    wrapper_randomaccess<int> // iterator—type
  >();
// add specific implementation for std::list<int>.
algolib::add_specific<
    algolib::advance<int>, // library name
    std::list<int>::iterator // iterator—type
  >();
```

In addition, we provide library functions with names that match their STL counterparts (e.g., distance). They look up and forward the call to the best matching algorithm instance. These functions are defined in the same namespace as the wrapper classes. Together with argument dependent lookup (ADL) in C++, this allows source code resemble code written with STL iterators. The following code snippet shows a function that takes two iterator wrappers as arguments and calls the library functions (i.e., distance, advance).

```
wrapper_forward<int>
random_elem(wrapper_forward<int> begin, wrapper_forward<int> end) {
  wrapper_forward<int>::difference_type dist = distance(begin, end);
  return advance(begin, rand() % dist);
}
```

At runtime, a library call selects the best applicable function present based on the dynamic type of the model. Starting with the **typeid** of the actual iterator model, it walks the **typeid**s of the inheritance chain until an algorithm or the fallback implementation is found. The following code snippet shows the lookup mechanism:

```
template <class A>
typename A::instance_map::data_type
lookup(typename A::dyn_iterator_type_pack pack) {
  typename A::instance_map::iterator end = A::instances.end();
  do {
    // lookup algorithm instance for type tuple described by pack
    typename A::instance_map::iterator pos = A::instances.find(pack);
    if (pos != end) return pos—>second;
    // try with next type tuple in the poset
    pack = next(pack);
  } while (pack != A::fallback_iterator_pack);
  return A::fallback_instance; // return fallback, if no better instance is present
}
```

The template argument A is a data structure that describes the algorithm and has the following dependent types defined:

- dyn_iterator_type_pack is a type tuple that initially contains type_ids of the iterator models that are involved in the dynamic dispatch.
- instance_map is the type of the associative data structure, which maps a dyn_iterator_type_packs to an algorithm instance.

In addition, A defines the following static data members:

- fallback_iterator_pack is a type tuple (of type dyn_iterator_type_pack) that identifies the concept classes of the fallback implementation.
- fallback_instance is a pointer to the fallback instance.

The function next uses the stored inheritance relationships to generate a poset of type tuples with the following ordering relation: $a \succeq b$ if for each component i of the tuples the subclassing relationship $a_i <: b_i$, or $a_i = b_i$ holds.

This ordering relation guarantees the lookup to find a most specific algorithm instance, if one exists. Should multiple equally specific instances exist, one of them is chosen depending on the implementation of next. When, like in the presented algorithms, the selection depends on the dynamic type of a single parameter, next's order becomes total.

If random_elem's begin and end wrap the concrete type std::list<**int**>::iterator, the lookup mechanism will find an algorithm instance that peels off all runtime concept layers and calls std::advance with a std::list<**int**> iterator. In case begin and end belong to a std::vector, the runtime model is re-wrapped by a wrapper_randomaccess iterator and std::advance<wrapper_randomaccess> is invoked.

Virtual function based design: Although the dispatch mechanism is semantically equal to virtual function calls, we rejected alternative library designs that would model algorithms as pure virtual functions declared in concept interfaces. This would break the separation between concept requirements and algorithms. Providing a new algorithm would require adding a new function signature to the concept interface, thereby breaking binary compatibility with existing applications. In addition, such a design would create a number of unused instantiated functions. For example, the class concept_forward would need virtual function declarations for all STL algorithms that are defined for forward iterators (e.g., adjacent_find, fill, equal_range, etc.). Consequently, the model classes would need to implement those functions regardless whether a specific program uses them or not. Finally, extending a virtual function based design towards multiple virtual arguments is complex when not all classes are known beforehand.

Evaluation: The described library eliminates virtual function calls inside an algorithm, when a matching algorithm instance is present. Its dispatch mechanism meets the outlined design criteria except for performance and the capability to consider more than one dynamic iterator type for dispatch decisions. The performance tests presented in Section 5.2 (i.e., for lower_bound) indicate that the dispatch cost can be significant. In particular, for small data sets the overhead of algorithm lookup can outweigh the more efficient execution of the STL algorithm. Extending the dispatch towards multiple arguments aggravates the situation. Even for optimized implementations, that flatten inheritance data and record uniquely best matching functions for a given type tuple (as implemented in Smith's Cmm [27]), Pirkelbauer et al. [23] report that a significant overhead remains.

Incorporating parts of the implementation outlined in [23] into a ISO C++ based design of the algorithm library with the goal to improve performance is possible, but raises other problems. For example, dispatch tables, which map a combination of types to a function, need to be computed beforehand. Moreover, a library implementation cannot modify the object-model, thereby requiring an additional indirection to map RTTI to indexes into the dispatch table.

Consequently, we argue that a practical solution to the multi-parametric dynamic dispatch and performance problems would benefit from proper language support. A design principle of C++ is to aim for language features that are general and address a number of specific concerns in several domains [28]. Following these guidelines we postulate a design based on an experimental language extension — open multi-methods. Open multi-methods, allow for programming styles that mix elements from object-oriented and functional programming [24] and object-oriented and generic programming [23].

Open multi-methods address two fundamental problems in object-oriented software design. First, they overcome dispatch problems where the choice of method depends on the dynamic type of more than one argument. This is paralleled by generic programming, where all arguments have equal weight during overload resolution. Second, extending classes, possibly defined in third party libraries, with dynamically dispatched functions (virtual functions) is intrusive or requires provision for the visitor pattern [14]. This often requires more foresight than class designers are given, complicating maintenance and limiting the extensibility of libraries. Open multi-methods separate the definition of dynamically dispatched functions from class definitions. Knowing that classes can be efficiently extended later allows class designers focus on the core functionality and eliminate unnecessary code. This is another parallel with generic programming, where algorithms build on core requirements that concepts define.

4.3. Implementation using open multi-method extension

In this section, we describe our algorithm library implementation based on open multi-methods. To begin, we show an open multi-method declaration advance for forward iterators with a more specific implementation for random access iterators. In open multi-method terminology, the most general declaration is called a *base-method*, and a more specific declaration is called an *overrider*. A base-method declaration has to precede an overrider declaration.

```
void advance(virtual concept_forward<int>& it, int dist); // base—method
void advance(virtual concept_randomaccess<int>& it, int dist); // overrider
```

With these open-method declarations, we can provide an implementation for an iterator wrapper:

```
void advance(wrapper_forward<int>& it, int dist) {
  advance(∗it.iterator_m, dist); // invokes open—method advance
}
```

Depending on the dynamic type of iterator_m the call to advance either invokes the base-method (for forward and bidirectional iterators) or the overrider (for random access iterators).

The internal mechanism of open multi-methods renders the core components of the previous implementation (Section 4.2) obsolete. Our experimental compiler extracts the signature of each open multi-method and the class hierarchy from each translation unit. (This eliminates the need for recording the inheritance relationship of the concept- and

model-classes.) The information is passed to a pre-linker, which synthesizes this data across translation units, checks for ambiguities, and generates a dispatch table containing entries for each possible type combination. (This eliminates the need to maintain the data structure, that associates algorithm instances with RTTI of the dedicated argument.) The runtime lookup to find the best matching function occurs in constant time for a given number of virtual parameters. (This eliminates the hierarchy traversal during algorithm instance lookup.)

For the implementation of our algorithm library, we reject the simple design outlined in the previous code snippet. This would require library users to specify function bodies for overriders (e.g., for advance with concept_randomaccess<**int**>). Moreover, runtime concept based libraries could only contain declarations of the open multi-methods. They could not contain definitions for open multi-methods because other libraries that use the runtime concept idiom could define them as well, leading to violations of the one definition rule (ODR). Solutions to this problem would either require users to write more open multi-method bodies, or make the pre-linker runtime concept aware and have it generate the missing definitions. Neither of these approaches seems practical at this stage.

Instead, we base our implementation on a combination of open-methods and the template mechanism. Currently, open multi-methods cannot be defined as template, and for this work we refrain from extending our experimental compiler in that direction. C++ does not support templated virtual functions — as this feature would require the linker to support creating the v-tables [28]. Templated open multi-methods would take this idea one step further by allowing templated virtual parameters:

```
template <class T>
void advance(virtual T& it, int dist);
// or
template <class T>
void advance(virtual concept_forward<T>& it, int dist);
```

Such a language feature is unprecedented and the definition of its semantics is a topic for further research. However, the instantiation of classes and their functions is well defined. For our current implementation, we harness the friend mechanism and the standard template class instantiation to generate algorithm instances. Friends allow us to comply with the requirement that base-methods and overriders have to be free standing functions that are defined in the same scope. Class template instantiation allows the user to generate open multi-methods without the need to write function bodies. In addition, the C++ standard has relaxed ODR rules for functions that are generated from templates. The problem of linker generated v-tables is overcome by having a pre-linker create open-method tables (the open multi-method analogue to v-tables).

We show a somewhat simplified implementation of reverse to illustrate the idea. The actual code is a bit more involved and requires, for example, a superfluous extra parameter to enable overload resolution to include friend functions that are only declared inside of a class into the candidate set. The omitted details are transparent to the library user and therefore do not impact the usability.

The base-method reverse is declared as friend inside a template class and thus is not a class member per se. The template class rtc_reverse allows us to indirectly define reverse based on a template argument.

```
// defined in namespace algolib
template <class ValueType>
struct rtc_reverse {
  typedef concept_bidirectional<ValueType> base_concept;

  friend void reverse(virtual base_concept& begin, base_concept& end) {
    /* wrap begin,end into wrapper_bidirectional and call std::reverse */
  }
};
```

Similarly, we can define rtc_reverse_generic and rtc_reverse_specific to generate instances that use wrapper classes or concrete iterators respectively. Deriving these classes from rtc_reverse guarantees that the compiler will find a base-method for the overrider.

```
// defined in namespace algolib
template <class Wrapper>
struct rtc_reverse_generic : rtc_reverse<typename Wrapper::value_type> {
  using typename rtc_reverse<typename Iterator::value_type>::base_concept;
  typedef typename Wrapper::concept_type iter_concept;

  friend void reverse(virtual iter_concept& begin, base_concept& end) {
    /* wrap begin, end into the matching wrapper class and call std::reverse */
  }
};
```

```
template <class Iterator>
struct rtc_reverse_specific : rtc_reverse<typename Iterator::value_type> {
  using typename rtc_reverse<typename Iterator::value_type>::base_concept;
  typedef typename map_iteratortag_to_model<Iterator>::type iter_model;

  friend void reverse(virtual iter_model& begin, base_concept& end) {
    /* unwrap begin, end and call std::reverse with the concrete iterator */
  }
};
```

The analogues to the functions add_generic and add_specific in Section 4.2 are explicit template instantiation directives.

```
// add generic implementation suitable for all random access iterators.
template class rtc_reverse_generic<wrapper_randomaccess<int> >;

// add specific implementation for std::list<int>.
template class rtc_reverse_specific<std::list<int>::iterator>;
```

The described library functions (Section 4.2) can simply invoke the open multi-method:

```
// defined in the same namespace as the wrapper classes
template <class WrapperType>
void reverse(const WrapperType& begin, const WrapperType& end) {
  algolib::reverse(*begin->iterator_m, *end->iterator_m);
}
```

This call is resolved through the open-method table of the first dynamic argument and invokes the best matching algorithm instance. In contrast to Section 4.2, this involves a simple lookup, similar to a virtual function call, because the open-method tables have been precomputed by the pre-linker.

Ambiguity resolution policy: With the modeled concept hierarchy and the dynamic dispatch restricted to only a single parameter, the ISO C++ implementation has no ambiguities. An extension towards considering a second argument requires some ambiguity resolution policy. The options for a library implementation are to either signal the ambiguity at runtime or choose somehow a preferred candidate (e.g., higher preference of earlier parameters, arbitrary choice, or user defined). A compiler based implementation has one more possibility: it can flag ambiguities before runtime and require the user to provide a resolving overrider. This is what the open multi-method based implementation does. Only when it is too late for programmer intervention (e.g., in cases that involve dynamic linking), the runtime would make an unspecified non-random choice. This is sufficient to guarantee the correctness of STL's algorithms. An optimal choice between ambiguous overriders would minimize the number of virtual function calls. For an algorithm such as merge this would require runtime information of the length of its ranges before a call can be dispatched.

5. Tests

To assess the performance cost of runtime concepts we tested the approaches described in Sections 3 and 4. The numbers presented in this section were obtained on an Intel Pentium-D (2.8 GHz clock speed; 512 MB of main memory at 533 MHz) running CentOS Linux 2.6.9-42. We compiled with gcc 4.1.2 using −O3 and −march=prescott.

5.1. Algorithm performance

Initially, the vector contained 8 million numbers in ascending order starting from zero. Then we invoke four algorithms: reverse, find of zero, sort, and lower_bound of zero.

vector<T>: As a reference point for our performance tests we use the vector instantiated with a concrete type. The table shows the number of cycles each operation needs to complete for a container of **int** and **double** respectively. The column to the right of the number of cycles shows the slowdown factor compared to vector<**int**>. The algorithms with O(n) runtime complexity (i.e., reverse and find) run approximately twice as long when used with type **double**. This discrepancy can be explained by the size of the stored data-type; **double** is twice as big as **int**.

	int	x vector<**int**>	**double**	x vector<**int**>
reverse	50 230 726	1	101 301 256	2.0
find	24 801 042	1	54 668 768	2.2
sort	554 503 572	1	1 157 274 566	2.1
lower_bound	5838	1	13 544	2.3

Operations on a Sequence<T>: The following table shows the results, when the algorithm library contains instantiations for concrete iterators. The time needed to select the best match is the only overhead that occurs. Note, that compared to [22]

we have enhanced the dispatch (by avoiding 3–4 heap allocations), which improved the runtime of lower_bound noticeably.

	int	x vector<**int**>	**double**	x vector<**int**>
reverse	50 635 816	1.0	101 897 418	2.0
find	25 428 424	1.0	53 027 609	2.1
sort	551 084 688	1.0	1 173 371 150	2.1
lower_bound	6244	1.1	14 616	2.5

Only when instances for the concrete iterators are missing, does our system resort to fallback implementations. The following table reports the runtime of these operations. Note, that the runtime of lower_bound is based on an implementation for wrapper_randomaccess iterators (cmp. the call to add_generic in Section 4.2).

	int	x vector<**int**>	**double**	x vector<**int**>
reverse	4 724 088 964	94.0	4 770 263 204	95.0
find	474 239 598	19.1	529 099 774	21.1
sort	16 956 331 602	30.6	17 419 730 692	31.4
lower_bound	17 066	2.9	18 522	3.1

The 94x slower performance for reverse is unacceptable, even for a fallback implementation. The analysis of these tests reveals three responsible factors: fallback algorithms can be significantly slower (though the slowdown is not always driven by a worse runtime complexity), virtual iterator functions, and model allocation on the heap.

To quantify the contribution of each of these factors we performed additional experiments: adding a reverse instance for wrapper_randomaccess improved performance marginally, 92x for int and 94x for doubles. Each iteration of reverse calls iter_swap once. Gcc's implementation of iter_swap calls another function that swaps the two elements to which the iterators point. Each function invocation creates copies of the iterators, which results in 16 million *unnecessary* heap allocations (and deallocations). By providing our own reverse implementation, we eliminated those copies. Then, reverse is only 7.4x slower for **int** (7.6x for **double**). However, passing arguments by value, which is the source of the heap allocations, is common in STL. For example, the fallback implementation of sort involves more than 36 million heap allocations. Instead of rewriting the STL algorithms, we could adopt Adobe's small object optimization [19] where the wrapper classes reserve a buffer to embed small objects (Adobe's open source library [3]).

The analysis of find indicates the following causes degrade performance. The fallback implementation is based on the forward iterator concept, while the optimal implementation takes advantage of random access iterators by calculating the trip-count beforehand and manually unrolling the main loop four times. Adding a generic instance for wrapper_randomaccess eliminates $\approx \frac{1}{3}$ of the virtual functions calls and improves performance to a factor of 11.7x for ints (12.4 for **double**).

5.2. Library function selection

With open multi-methods: Since our experimental compiler extends the EDG frontend [13] and generates C files as output, we modified the test setup. Instead of directly compiling the C++ source, we translate all test cases first to C and compile those with gcc 4.1.2 and the same optimization flags (−O3 and −march=prescott). We measure the dispatch overhead by comparing the performance of static calls to STL functions, with the performance when specific algorithm instances have been added to the library. To underscore the time spent for dispatching, we reduced the data size to 8 elements of type **int**. For such a small data set the status of processor cache and predictors make a difference. Therefore we split the tests into two, where the first measures an initial algorithm invocation — the dispatch structures are not yet in cache, and the second measures a subsequent invocation of the same four algorithms.

The following table shows the number of cycles each call takes. The first call to an algorithm from the runtime library (reverse) incurs higher cost as it requires loads of the open method tables/indexes in the case of open multi-methods and RTTI in the case of the ISO C++ implementation. The calls to algorithms that return an iterator (find and lower_bound) incur additional overhead for allocating the result of the operation (an iterator model) on the heap. The open multi-method based implementation is significantly slower than a static call but outperforms the ISO C++ implementation by 500 to 2000 cycles.

Cold-cache	Static STL call	Open multi-methods	ISO C++
reverse	585	1731	3786
find	927	3065	3905
sort	7776	7912	8634
lower_bound	377	1994	2512

The second run tests an ideal scenario, where all the data (i.e., open method tables/indexes, RTTI, code) is in cache and the processor's predictors can draw from statistical data of previous runs. The overhead of the open multi-method invocation

shrinks to less than 100 cycles when compared to a static call (i.e., reverse and sort. find and lower_bound require a heap allocation). The ISO C++ implementation is about 200 cycles slower.

Hot-cache	Static STL call	Open multi-methods	ISO C++
reverse	157	220	445
find	158	1008	1181
sort	5631	5725	5934
lower_bound	224	1159	1209

6. Related work

The ASL [3] introduced the runtime concept idiom, employing type erasure [2] to provide the *any regular* library (similar to the boost any library), and its generalization, the *poly* library. The *poly* library generalizes the idiom to support refinement and polymorphic down-casting, encapsulates the common tasks required to create non-intrusive runtime-polymorphic value-based wrappers. The *poly* library design goals and implementation are elaborated in [19].

ASL also provides the *any iterator* library offering runtime-polymorphic iterators for specific types as a proof of concept. Becker [8] presents a similar library. Bourdev and Järvi [10] discuss a mechanism for a closed set of types that falls back to static-dispatch when type erasure is present.

Our work extends the previous results to an open library of algorithms operating on runtime-polymorphic containers, achieving realistic performance levels by using static dispatch where possible.

7. Future work and conclusion

In the described systems, an algorithm library handles the dispatch to the most appropriate algorithm present in the system. Which algorithms a system contains depends on programmers and system integrators. This requires manual intervention, which is not always possible. In particular, programs that involve dynamic linking remain hard to optimize. Iterator types for which repeated algorithm invocations frequently resolve to fallback implementations would benefit from a runtime system that is capable of dynamic algorithm instantiation.

In this paper, we have discussed runtime polymorphic versions of several STL algorithms. Our implementation enables the use of STL's sequences where the binding to a concrete data structure is deferred until runtime. To improve runtime performance, if the data structures in use are known to the system integrator, our algorithms can leverage static dispatch. As a result, the runtime overhead becomes negligible for large data sets.

Acknowledgments

This project was supported by NSF grant C08-00028. We thank Jaakko Järvi, Damian Dechev, Yuriy Solodkyy, Luke Wagner and the anonymous referees for their helpful suggestions.

References

[1] The Boost C++ libraries, http://www.boost.org/, July 2008.
[2] D. Abrahams, A. Gurtovoy, C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series), Addison-Wesley Professional, 2004.
[3] Adobe System Inc., Adobe source library. http://opensource.adobe.com, 2005.
[4] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, in: AW C++ in Depth Series, Addison Wesley, 2001.
[5] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, L. Rauchwerger, Stapl: A standard template adaptive parallel C++ library, in: LCPC '01, Cumberland Falls, Kentucky, 2001.
[6] M.H. Austern, Generic Programming and the STL: Using and Extending the C++ Standard Template Library, Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 1998.
[7] P. Becker, Working draft, standard for programming language C++, Tech. Rep. N2521, February 2008.
[8] T. Becker, Type erasure in C++: The glue between object oriented and generic programming, in: K. Davis, J. Striegnitz (Eds.), Multiparadigm Programming 2007: Proceedings of the MPOOL Workshop at ECOOP'07, 2007.
[9] L. Bettini, S. Capecchi, B. Venneri, Double dispatch in C++, Software - Practice and Experience 36 (6) (2006) 581–613.
[10] L. Bourdev, J. Järvi, Efficient run-time dispatching in generic programming with minimal code bloat, in: Workshop of Library-Centric Software Design at OOPSLA'06, Portland Oregon, 2006.
[11] C. Cleeland, D. Schmidt, T. Harrison, External polymorphism — an object structural pattern for transparently extending concrete data types, in: R. Martin, F. Buschmann, D. Riehle (Eds.), Pattern Languages of Program Design, Addison-Wesley, Reading, MA, 1997.
[12] J.C. Dehnert, A.A. Stepanov, Fundamentals of generic programming, in: Selected Papers from the International Seminar on Generic Programming, Springer-Verlag, London, UK, 2000.
[13] Edison Design Group, C++ front end. http://www.edg.com/, July 2008.
[14] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 1995.
[15] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G.D. Reis, A. Lumsdaine, Concepts: Linguistic support for generic programming in C++, in: OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2006.
[16] ISO/IEC 14882 International Standard, Programming languages: C++, American National Standards Institute, 1998.

[17] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, J. Siek, Algorithm specialization in generic programming: Challenges of constrained generics in C++, in: PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, 2006.
[18] J. Järvi, M.A. Marcus, J.N. Smith, Library composition and adaptation using C++ concepts, in: GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, ACM Press, New York, NY, USA, 2007.
[19] M. Marcus, J. Järvi, S. Parent, Runtime polymorphic generic programming—mixing objects and concepts in Concept C++, in: K. Davis, J. Striegnitz (Eds.), Multiparadigm Programming 2007: Proceedings of the MPOOL Workshop at ECOOP'07, 2007.
[20] S. Parent, Beyond objects: Understanding the software we write, presentation at C++ connections, November 2005.
[21] S. Parent, Concept-based runtime polymorphism, presentation at BoostCon, May 2007.
[22] P. Pirkelbauer, S. Parent, M. Marcus, B. Stroustrup, Runtime concepts for the C++ standard template library, in: SAC '08: Proceedings of the 2008 ACM Symposium on Applied Computing, ACM, New York, NY, USA, 2008.
[23] P. Pirkelbauer, Y. Solodkyy, B. Stroustrup, Open multi-methods for C++, in: GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, ACM Press, New York, NY, USA, 2007.
[24] A. Shalit, The Dylan Reference Manual, 2nd edition, Apple Press, 1996.
[25] D. Shopyrin, Multimethods implementation in C++ using recursive deferred dispatching, IEEE Software 23 (3) (2006) 62–73.
[26] Y. Smaragdakis, D.S. Batory, Mixin-based programming in C++, in: GCSE '00: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers, Springer-Verlag, London, UK, 2001.
[27] J. Smith, Draft proposal for adding multimethods to C++., Tech. Rep. N1463, 2003.
[28] B. Stroustrup, The Design and Evolution of C++, ACM Press/Addison-Wesley Publishing Co, New York, NY, USA, 1994.
[29] B. Stroustrup, Evolving a language in and for the real world: C++ 1991–2006, in: HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, ACM, New York, NY, USA, 2007.
[30] B. Stroustrup, G. Dos Reis, Supporting SELL for high-performance computing, in: 18th International Workshop on Languages and Compilers for Parallel Computing, in: LNCS, vol. 4339, Springer-Verlag, 2005.
[31] T.L. Veldhuizen, Arrays in Blitz++, in: ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments, Springer-Verlag, London, UK, 1998.