Type-Soundness and Optimization in the Concepts Proposal

Author: Douglas Gregor, Indiana University
Document number: N2576=08-0086
Date: 2008-03-17
Project: Programming Language C++, Core Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

1 Introduction

There is a fundamental tension between the type-checking guarantees of constrained templates and their ability to provide the greatest level of optimization. Concepts work by specifying an interface of a constrained template, stating exactly what behavior the template arguments must exhibit to work with that template. By treating all of the different template arguments in the same way through this interface, concepts can provide *type soundness*, meaning that the instantiation of constrained templates will not fail. On the other hand, many of the optimizations we've come to expect in C++ rely on doing different things in templates depending on what data types we get at instantiation time: it is the reason that we can optimize templates extremely well, and also the reason that instantiation-time failures are so frequent and so verbose.

Recent examples by Howard Hinnant [1] have brought this tension to the forefront, because the type-checking of constrained templates inhibits some optimizations made possible by rvalue references. The issue is not limited to rvalue references, and further study uncovered some examples where concepts can disable some copy-elision—based optimizations that worked within C++03 but would be disabled by concepts [3].

This paper describes the tension between type-checking and optimization in the context of Hinnant's string example [1], and illustrates why the example behaves as it does inside a constrained template. We'll then explore potential solutions to the problem, and how they relate to this fundamental tension.

2 Hinnant's Example

Hinnant's example involves a string class that mimics the optimizations available using rvalue references, but prints which operation is invoked at each step to provide a log detailing which optimizations would be performed. In general, this string class mimics optimizations that permit the reuse of storage allocated by temporary strings. The string class is defined as:

```
class string {
public:
  string() {}
  string(const string&) {std::cout << "string(const string&)\n";}
  string& operator=(const string&) {std::cout << "string& operator=(const string&)\n";}
  string(string&&) {std::cout << "string(string&&)\n";}
  string& operator=(string&&) {std::cout << "string& operator=(string&&)\n";}
};

string operator+(const string&, const string&) {
  std::cout << "lv string + lv string\n";
  return string();
}

string operator+(string&&, const string&) {
  std::cout << "rv string += lv string\n";
  return string();
}

string operator+(const string&, string&&) {
  std::cout << "rv string insert at front lv string\n";
  return string();
}

string operator+(string&&, string&&) {
  std::cout << "rv string += rv string\n";
  return string();
}
```

When one exercises this string like so:

```
string s3 = s1 + s2 + string() + string() + string();
```

we receive the following output:

```
lv string + lv string
rv string += rv string
rv string += rv string
rv string += rv string
```

Here, we see the effects of the rvalue-reference optimizations. The first concatenation of s1 and s2 creates a new temporary from two lvalues. The next three concatenations pick the **operator+** that concatenates two rvalues, which will be more efficient for this example because storage can be reused.

Next, we put this code into an unconstrained template, where the string class is abstracted by a template type parameter T, and instantiate that unconstrained template with T=string:

```
template <class T>
void test() {
  T s1;
  T s2;
  T s3 = s1 + s2 + T() + T() + T();
}
```

```
int main() {
  test<string>();
}
```

As we expect with unconstrained templates, we receive the same output:

```
lv string + lv string
rv string += rv string
rv string += rv string
rv string += rv string
```

Now, using the standard definitions of the basic library concepts, we can constrain this template as follows:

```
template <class T>
requires std::DefaultConstructible<T> && std::Addable<T>
      && std::Convertible<T::result_type, T> && std::CopyConstructible<T>
void test() {
  T s1;
  T s2;
  T s3 = s1 + s2 + T() + T() + T();
}
```

For reference, the Addable concept is defined as[1]

```
auto concept Addable<typename T, typename U = T> {
  typename result_type;
  result_type operator+(T const&, U const&);
}
```

Now, when executing the constrained template, we receive the following output:

```
lv string + lv string
lv string + lv string
lv string + lv string
lv string + lv string
```

Thus, the constrained template is not calling the optimized, rvalue-reference—based concatenation operators.

3 Type-Checking Templates

The primary change that concepts introduce into the language is that they provide type-checking for constrained template definitions. The intent is for constrained templates to provide a sound type system, where templates can be type-checked separately from their uses, much like a non-template function can be type-checked separately from any calls to that function. If the type system provided by constrained templates is sound, then template instantiation cannot produce an error if the constrained template type-checks and if the

[1] We have explicitly added the **const**& to the arguments of **operator**+. If not provided explicitly, it would have been added implicitly according to the concepts wording.

template arguments meet the requirements stated by the constrained template. It is this type-soundness that enables many of the other benefits of concepts, including improved error messages, syntax adaptation via concept maps, and the elimination of the need for **typename** and **template** when naming dependent types.

To enable separate type checking of an implementation from uses of that implementation, one must describe the interface precisely, so that both implementer and user can refer to the terms of the interface. Separate type-checking, then, is determining whether the two parties have met the requirements of the interface, and how. In the function-call example, the interface is defined by the function prototype, providing the types of the arguments and the result of the function. With constrained templates, the interface is defined by the requires clause and the concepts it refers to. The nature of that interface mitigates the interaction between the two parties, who are otherwise blind to each other's existence.

The interface provided by concepts is that of a forwarding function. The constrained template implementation can make a call into that forwarding function, because it knows the types of the arguments and the result. Thus, a requirement Addable<T> states that there exists a function **operator**+(T **const**&, T **const**&) for every T that the constrained template can be instantiated with. So long as the constrained template only uses functions that are available within its template requirements, the constrained template has upheld its side of the contract by using this interface.

On the other side of the interface, types like string provide implementations of those forwarding functions within concept maps. In Hinnant's example, the concept map Addable<string> will be implicitly defined as:

```
concept_map Addable<string> {
  typedef string result_type;
  string operator+(const string& x, const string& y) { return x + y; }
}
```

Note that the **operator**+ defined in Addable<string> is *not* found when looking for a suitable + operator in its own body, so this definition is not recursive. Rather, the + operator selected for the expression x + y is the first **operator**+ defined in Hinnant's example, which adds two lvalue strings together and returns a new, temporary string. With this definition, the user's data type—string—has satisfied the terms of the interface, so instantiation of the constrained template test<string> proceeds without error.

Figure 1 illustrates graphically how calls to the + operator within the constrained templates are mapped into the forwarding function and through to the string's concatenation operator. The arrows represent the results of overload resolution, illustrating which calls will go to which functions. Note that the concept is the interface layer between these two pieces, and that there is (separate) type-checking on both sides of this interface which establishes the arrows. From here, we see why the constrained template always uses the string-concatenation operator that operates on lvalues, missing the rvalue-based optimizations: every call to + within the constrained templates maps to a single **operator**+ in the concept (which accepts lvalues and rvalues), and that **operator**+ requirement is satisfied by the user-defined **operator**+(string **const**&, string **const**&).

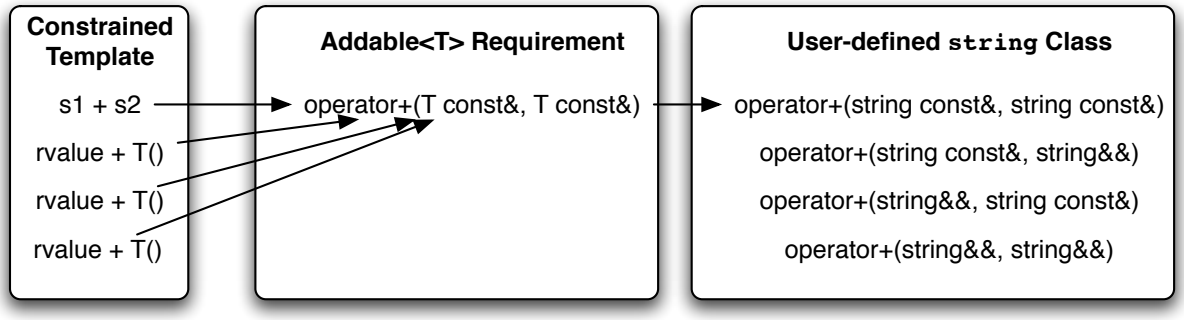

Figure 1: Illustration of the mapping of constrained templates to the concept interface and the mapping of the concept interface to a user-defined type. Follow the arrows to determine which function each expression in the constrained template ends up calling.

4 Potential Solutions

There are several possible solutions to the problem illustrated by Hinnant's example. Some of these solutions trade some type-soundness for improved optimization opportunities, by allowing more variation at the time that a constrained template is instantiated. This is not a new idea for the concepts proposal, because there are already two places in concepts where we sacrify some type-soundness to permit additional optimizations: allowing the use of specializations in the instantiation of constrained templates, and instantiation-time partial ordering of function templates called from constrained templates.

4.1 Manual Introduction of Overloads

From the diagram in Figure 1, we can see that the initial loss of rvalue information in the use of + comes from the mapping from the constrained template into the concept interface. Therefore, we can expand the interface described by the concept to differentiate between lvalues and rvalues. For example, rather than use the standard Addable concept, we will use the following RVAddable variant:

```
auto concept RVAddable<typename T, typename U> {
  typename result_type;
  result_type operator+(T const&, U const&);
  result_type operator+(T const&, U&&);
  result_type operator+(T&&, U const&);
  result_type operator+(T&&, U&&);
}
```

With this change, Hinnant's example produces the same output for both the constrained and the unconstrained templates, because we've made the rvalue-based versions part of the interface. Figure 2 illustrates how this concept interface permits the optimizations.

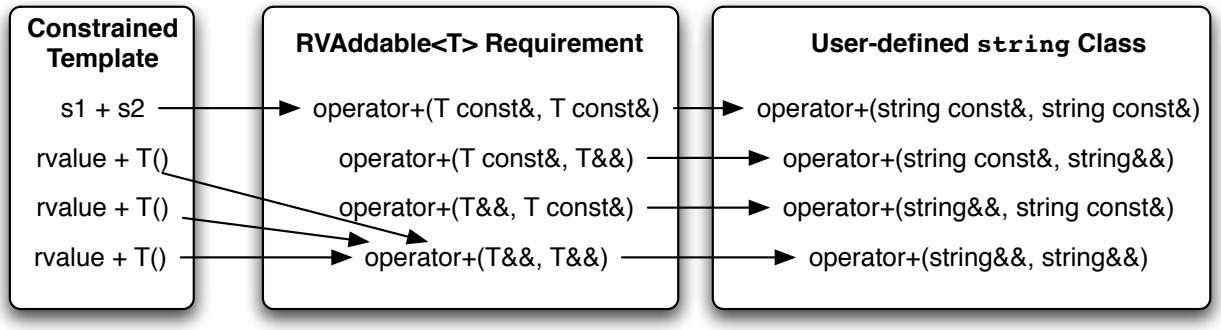

Figure 2: Illustration of the mapping of constrained templates to the RVAddable concept interface and the mapping of the concept interface to a user-defined type.

4.2 "Eliminating" Forwarding Functions

This approach attempts to make associated functions a bit more abstract, allowing a single associated function to resolve to different user functions depending on how it is called from the constrained template. Figure 3 attempts to illustrate this process graphically, using the LateAddable concept:

```
auto concept LateAddable<typename T, typename U = T> {
  typename result_type;
  result_type operator+(T const&, U const&);
}
```

In Figure 3, the LateAddable concept only provides a single associated function **operator**+ that accepts two arguments of types T **const&** and U **const&**, respectively. When filling in the details of a concept map LateAddable<string>, a forwarding function like the following will be implicitly defined:

```
concept_map LateAddable<string> {
  typedef string result_type;
  string operator+(T const& x, T const& y) { return x + y; }
}
```

With this new scheme, we intend to eliminate the forwarding function. Instead, we look only at the body of the forwarding function, and attempt to compile the expression x + y, where x and y are lvalues of type **const** string, just as in the forwarding function. When successful, the + expression will bind to some candidate in overload resolution, whether it is a function or a built-in. We call this candidate the *seed*. The return type of the seed is used for deduction of associated types (e.g., result_type), and the seed itself will be used to evaluate which other functions will be considered to match the associated function requirements.

In the current, forwarding-function—based model, the seed function is the only function that can be called via the constrained template. With this new scheme, we will use the seed as a pattern to compute a *candidate set* containing other functions that could be called to satisfy that same requirement. The candidate set contains:

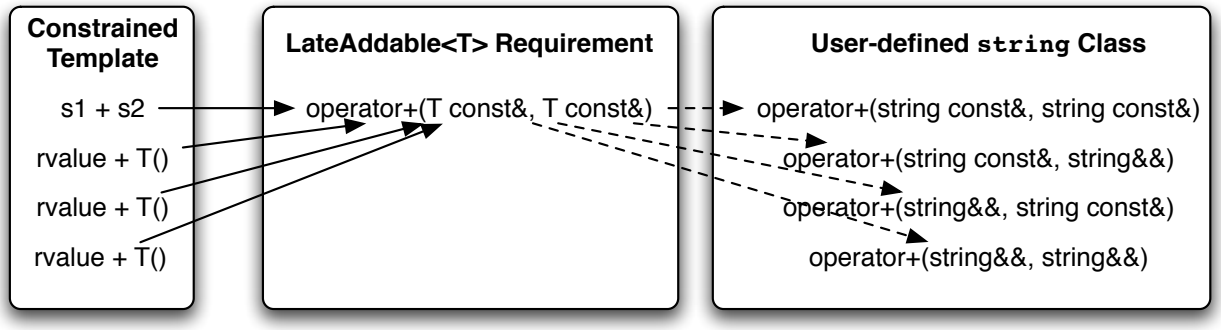

Figure 3: Illustration of the mapping of constrained templates to the LateAddable concept interface and the mapping of the concept interface to a user-defined type.

- the seed, if it is a non-template function or built-in operation,
- the function template from which the seed was instantiated, if the seed is a function template specialization,
- any function with the same name as the seed function that
  - is declared in the same namespace as the seed,
  - has the same return type of the seed, after references and then top-level *cv*-qualifiers have been removed, and
  - has the same parameter types as the seed, after references and then top-level *cv*-qualifiers have been removed and ignoring any parameters for which default arguments have been used; and
- any function template with the same name as the seed function that is declared in the same namespace as the seed.

This candidate set is stored for later use. When we instantiate a constrained template that uses that concept map, e.g., LateAddable<string>, we replace calls to LateAddable<string>::**operator**+ with calls to the functions in that candidate set. Function template specializations produced by template argument deduction on the templates in the candidate set are only permitted to enter the overload set if they:

- have the same return type as the seed, after references and then top-level *cv*-qualifiers have been removed, and
- have the same parameter types as the seed, after references and then top-level *cv*-qualifiers have been removed and ignoring any parameters for which default arguments have been used.

Since the candidate set permits variation in which functions are selected based on *cv*-qualifiers and whether the argument is an lvalue or rvalue, we get "perfect" forwarding of arguments, and Hinnant's example produces the same result as in the unconstrained template case:

```
lv string + lv string
rv string += rv string
rv string += rv string
rv string += rv string
```

This mechanism trades type-soundness for optimization. By eliminating the forwarding function call completely, and saving the candidate set for later use, we get the most specific function from each call. However, we have opened up the possiblity for more instantiation-time failures. For example, say the function **operator**+(string&&, string **const**&) was deleted: in this case, the constrained template would be a part of the candidate set, but in some cases—rvalue + lvalue—we would get an error when we end up calling the deleted function. Other failures can occur due to overloading ambiguities, function template partial ordering ambiguities, calls resolving to inaccessible functions, etc. These are some of the same problems that remain with the selection of more-specialized algorithms during the instantiation of constrained templates, as we do with the binary_search/advance example.

Syntax adaptation in concept maps : with this scheme, concept maps should also be able to provide overload sets to satisfy the requirements of the concept. To do this, we can just permit multiple overloads specified within concept maps for each requirement in the concept.

Implementation experience : the latest version of ConceptGCC, available from the ConceptGCC Subversion repository, implements this scheme as well as the existing concepts proposal. To enable this scheme, use the flag -fabstract-signatures. I have confirmed that Hinnant's example properly deals with l/r-valueness, and of course that there are easy-to-construct examples that do cause instantiation-time failures in constrained templates due to this change. Note that the use of this flag does *not* break any existing test cases within the GNU C++ compiler or its C++ Standard Library test suite, so this change does not seem to have a significant impact on backward compatibility.

References

[1] Howard Hinnant. Some concerns about concepts. C++ Library Reflector message c++std-lib-20050, January 2008.

[2] Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, and Jeremy 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*, pages 272–282, New York, NY, USA, 2006. ACM Press.

[3] Sean Parent. Re: Some concerns about concepts. C++ Library Reflector message c++std-lib-20052, January 2008.