P0131R0
September 27, 2015
Bjarne Stroustrup

Unified call syntax concerns

Abstract

This note is a follow-up to the unified call syntax proposals (N4145, N4174, N4474) and discussions in EWG in Urbana and Lenexa.

This note discusses some problem examples and concerns raised about the unified call syntax proposal. It suggests remedies for some concerns. The key questions addressed here are:

- Given that every lookup rule can cause problems, does unified calls add new problems or make old problems significantly worse?
- Does the benefits of unified call outweigh the problems?

My answers are “no” (note “significant”) and “yes!” Current conventional use of namespaces addresses the most strongly voiced concerns.

Section 7 present a serious name lookup problem presented by David Vandevoorde in Lenexa and discusses design alternatives to deal with that. This reflects a discussion at Cppcon. A companion paper by Richard Smith and Chandler Carruth (P???) should have been integrated with the discussion here, but there was no time for a merge.

A draft of this note was posted to the –ext reflector in May. This version represent significant updates (especially section 7).

An experimental implementation by Faisal Vali now exists (section 10).

1. Unified call syntax

When Herb Sutter and I proposed a unified call syntax we relied on ideas going back at least to D&E in the context of C++, and on experience with other languages. C# was mentioned. The debate that resulted (erupted) on the web mentions ML, Java, Lisp, D, Swift, and other languages with similar facilities. Generally, uniform call syntax (by any name) is a highly valued feature in those languages.

At the Urbana meeting, concerns about backwards compatibility were raised. The compatibility concerns were solved in a revision for the Lenexa meeting, but there worries about the stability of interfaces, “surprises”, and compilation costs were expressed. These concerns are the ones I address here.

2. Benefits

We discuss potential problems a lot. Sometimes, the benefits of a proposal get forgotten in all the worry and due diligence, so here I will briefly re-state the purpose and presumed benefits of unified call syntax. I think that the benefits obtained far outweigh the potential problems caused.

The basic proposal is

- A call **x.f(y)** will first look for members of **x**'s class (as ever) and if no valid call is found will look for functions as if **f(x,y)** had been written
- A call, **f(x,y)** fist looks for functions (as ever) and if no valid call is found looks for functions as if **x.f(y)** had been written

This breaks no existing code.

The perceived benefits are

- *Ease of use*: a caller need not know whether a function is implemented as a member or a free-standing function or a lambda
- *Decreased coupling / better encapsulation*: a class implementer does not have to place every function manipulating and object into the class. Only the minimal set of functions needing access to the object representation needs to be members. This simplifies comprehension and maintenance.
- *Less replication*: Eliminate the need to provide two functions (a member and a free-standing function) to simplify operation on an object (e.g., **begin(x)** and **x.begin()**).
- *Non-intrusive extension of functionality*: We can add functionality to a class without modifying the class definition
- *Simpler refactoring*: We can change a function from a member to a non-member (or vice versa) without modifying user code.
- *More general templates*: a template can take arguments of types supporting **f(x)** as well as arguments of types requiring **x.f()**.
- *Simpler concepts*: a concept can be written without duplicating requirements for **f(x)** and **x.f()**.
- *Chaining*: we can write **h(z,g(y,f(x)))** or **f(x).g(y).h(z)** according to our preferences.
- *Functional notation for virtual functions*: if the logical or preferred syntax is functional, we can still implement an operation as a virtual function without adding a helper function.
- *Allow member function notation for types without members*: for example adding **size(A)** to an array type **A** and call it using **a.size()**. This decreases the temptation to write adaptor functions as unconstrained templates.

These benefits are not orthogonal. In fact, some can be seen as mere re-statements of others.

Note that operators and range-**for** already offer this benefit and that replication of functions (e.g., **swap(x,y)** and **x.swap(y)**) is common in the standard library and elsewhere. Users of other languages generally consider these benefits (as they apply to the various languages) significant.

3. The fundamental problem

You can be surprised by what is invoked by a call if you have

- nested scopes (as with derived/base classes, nested namespaces, or library include paths)
- hiding (or access control)
- overloading

We have lived reasonably happily with those potential problems in C++ for 30+ years. Uniform syntax can be defined in terms of nested lookups (as the current proposal and range-**for lookup**) or overloading (like operator lookup). It follows that for every possible design, we can construct examples that could surprise someone. Several were presented in the proposals.

4. The end of stable interfaces?

In Lenexa, Chandler Carruth presented this example of a problem we'd face if uniform-call syntax was adopted:

```
// my_library.h:
struct FancyAPI {
   void member1(int);
   void member2(int);
};

 // user_code.cpp:
#include "my_library.h"
void my_code(FancyAPI &widget) {  // The FancyAPI doesn't have a super useful member3
     auto member3 = [](FancyAPI &widget, int i) { if (i<0) { /* yikes error! */ } ... };
     …
     widget.member3(-1);         // calls local member3
}
```

The programmer's idea (assuming that I interpret Chandler's intent correctly) is that the user thinks that the **FancyAPI** lacks an operation **member3()** and (relying on uniform call) adds it locally using a lambda. In the next release, the provider of **FancyAPI** improves the API by adding a **member3()**:

```
// my_library.h:
struct FancyAPI {
   void member1(int);
   void member2(int);
   void member3(size_t);         // added later
};

// user_code.cpp:
#include "my_library.h"
void my_code(FancyAPI &widget) {  // Now, the FancyAPI has a super useful member3
   auto member3 = [](FancyAPI &widget, int i) { if (i<0) { /* yikes error! */ } ... };
   …
   widget.member3(-1);           // calls widget's member3
}
```

Obviously, the behavior of the user code changes (now **FancyAPI::member3()** is called) – to what anyone looking at **FancyAPI** would expect. This will give the wrong result if **FancyAP**'s **member3()** is less appropriate than the local one. The argument types **int** and **unsigned** were chosen to get the well-known language problems with implicit conversions (as described in many places, including N4477). For extra credit, remember that **sizeof(size_t)** is larger than **sizeof(int)** on some popular systems and not on other popular systems.

I don't dispute that this will happen in some form or another or that even a better result from **FancyAPI::member3()** than from the local **member3()** could be a serious problem. What I do claim is that

- such problems will be rare
- we have lived happily with equivalent problems for decades (see section 6)
- implementations can warn
- we will learn not to patch interfaces in an ad hoc and non-obvious manner
- the problem is mitigated by conventional use of namespaces
- other languages have had this exact problem for years, without considering it significant

Consequently, I consider the risk of breaking such future code (we don't yet have uniform call syntax, so no current code will be broken) worth the benefits we would get from uniform call syntax.

In the example, the introduction of **member3** was a local fix to a deficiency of **FancyAPI**. Defining a local lambda and then in the same function calling it with the functional syntax could be seen as a bit odd and ad hoc. The programmer would have to do that for every call of **member3** in the program to get consistent answers. I doubt that code would pass code review. Also, if you are locally patching an API, I think it would be reasonable to expect you to be alert to changes (supposedly improvements) to that API. However, that's just one variant of the problem, possibly chosen primarily to fit on a slide.

A more common variant of the problem would use a non-local function. For example:

```
// user_code.cpp:
#include "my_library.h"

void member3(FancyAPI &widget, int i) { if (i<0) { /* yikes error! */ } ... };

void my_code(FancyAPI &widget) {
   …
   widget.member3(-1);
}
```

Now, which function is invoked depends on which version of **FancyAPI** we included. This differs from the lambda version in that now every call where the global **member3** is in scope will invoke the same function.

Had we decided to go with a solution based on overloading (based on a union of overload sets), many variants of this problem would have been caught by the compiler. However, for backwards compatibility and other reasons, we decided to give a member function priority when we use the **x.f(y)** syntax, so the problem (where it is a problem) must be caught by other means (e.g., a lint or a compiler warning).

I say “where it is a problem” because I consider a member function primary and almost by definition the right choice when we have an object of a class as the first argument. It’s the privilege of the class writer to get “first dibs” on defining such an operation and the class writer’s obligation to provide the most appropriate semantics. In my original proposal, the member function was always considered first, independently of which syntax was used. My ideal language would have every call syntax (member, functional, operator) lead to the same function being called. The proposal is an approximation to that ideal. See also section 13.

So far, we have assumed that the writer of the call **widget.member3(-1)** wanted the “local” **member3()**. In real code we would not be able to assume that. A lot of code might separate the definition of the “local” **member3()** and the call (even in the lambda variant). The “local” **member3()** might even be in a separate header (just like **FancyABI**’s **member3()**). If we are writing, maintaining, or debugging such code we must be relatively neutral about which resolution is the right one. It is non-trivial to determine a programmer’s intent, and the programmer might even be misguided. We are not all “super coders”, so explicit specification can be wrong.

5. Use of namespaces to improve interfaces

My favorite style of class definition directly addresses the **FancyABI** example. I minimize the number of member functions by using “helper functions” defined in the namespace of the class (see TC++PL3 and TC++PL4):

```
// my_library.h:
namespace FancyAPI {
   struct Fancy {
      void member1(int);
      void member2(int);
      void member3(size_t);        // added later (maybe)
  };

   void helper1(Fancy&, int);
   void helper2(Fancy&, int);
   void helper3(Fancy&, size_t);            // added later (maybe)
}
```

Uniform call helps here by making it easy to move functions in and out of the class as needed. Using a namespace enables ADL and minimizes namespace pollution.

```
// user_code.cpp:
#include "my_library.h"

void member3(FancyAPI::Fancy &widget, int i) { if (i<0) { /* yikes error! */ } ... };

void my_code(FancyAPI::Fancy &widget) {
   ...
   widget.member3(-1);            // call widget’ member3 if it exists; otherwise “local” member3
   member3(widget,-1);            // call “local” member3
```

```
    widget.helper3(-1);           // do overload resolution for Widget::helper3
                                  // and "local" helper3
    helper3(widget,-1);           // do overload resolution for Widget:: helper3
                                  // and "local" helper3
}
```

Now, it is obvious that new functions that do not touch the representation of **Fancy**, should be in **FancyABI** and that only functions that do touch the representation should be members of **Fancy**. Adding a helper function outside **FancyABI** is asking for surprises and potential maintenance problems. We could use the code above, add **member3** to the **FancyABI** in **my_library.h**, or reopen the namespace:

```
Namespace FancyABI {              // reopen namespace
        void member3(Fancy &widget, int i) { if (i<0) { /* yikes error! */ } ... };
}
```

Yes, we can still get the problem with a lambda not being found (hidden by a member function) and yes, we can still get a problem with functions being added to **FancyABI** and **FancyABI::Fancy**. However, there is an obvious place to look and if you don't want to pick up "local stuff" (lambdas and functions not in **FancyABI**) you can use explicit qualification:

```
void my_code(FancyAPI::Fancy &widget) {
    …
    Fancy_API::member3(widget,-1);      // will not find a local or global member3
}
```

In **widget.member3(-1)**, what lookup should happen if there is no class member **member3**? Status quo has that as an error. The current proposal tries **member3(widget,-1)** which would find a local lambda. If there is no local lambda, **member3(widget,-1)** would do overload resolution among global functions and members of **FancyABI** found using ADL. Thus an added function to **FancyABI** would only "hijack" the "local" function if it was a better match. In the most likely scenarios where the added **FancyABI** function and the "local" function are of identical types, an ambiguity error would result. I assume this to be the most common case because given the same name the two functions are likely to perform very similar operations and are likely to follow local conventions for argument types.

I am satisfied that this reasonable and recommended style of namespace use addresses the concerns raised by the **FancyABI** example. It decreases the likelihood of an uncaught problem below the level of similar problems that we currently deal with successfully.

6. Similar current problems

Variants of the **FancyAPI** problem exist today and we successfully cope with them. The examples in this section do not assume unified call.

6.1. C API

Let's see what the equivalent problem using a C-style interface from a C++ in a C++ program would be:

```
// my_C_library.h:
struct FancyAPI {
   // …
};

void member1(FancyAPI*, int);
void member2(FancyAPI*, int);
void member3(FancyAPI*, size_t);         // added later (maybe)


// user_code.cpp:
#include "my_C_library.h"

void member3(FancyAPI* widget, int i) { if (i<0) { /* yikes error! */ } ... };

void my_code(FancyAPI& widget) {
   …
   member3(&widget,-1);                  // do overload resolution for all member3()s
}
```

Depending on the argument types we may get an ambiguity error or select one of the alternative **member3()**s. If **member3()** is called from other parts of the program, we may get different **member3()**s called. If the two **member3()**s have the same type, we may and may not get a linker error (depending on build procedures). I hope for a double-definition link-time error for **member3()**. However, we just might get every part of a program using **user_code.cpp**'s **member3()** instead of **my_C_library.h**'s. Where overloading is not used, we de facto have a variant of the problem in C – since 1972.

Overloading

The variant that has worried me most (long before I saw Chandler's; you can see versions in N4477) involves overloading:

```
// my_library.h:
struct FancyAPI {
   void f(double);
   void f(size_t);          // added later (maybe)
};


// user_code.cpp:
#include "my_library.h"

void my_code(FancyAPI &widget) {
   …
   widget.f(-1);
}
```

Today, a function added in **FancyAP**I can lead to a different function being chosen (quietly).

Everyone who calls a function (member or non-member) is vulnerable to “hijacking” by a better-match function. We have lived with this since 1983.

There are organizations that have banned overloading to avoid such problems, but overloading (often in the guise of templates) is the backbone of modern C++.

6.2. Non-member functions

At least that **FancyAPI struct** offers a closed set of alternatives. With free functions, we get an open set:

```
// my_library.h:
struct FancyAPI {
        // …
};

void f(FancyAPI*, double);
void f(FancyAPI*, size_t);        // added later (maybe)


// user_code.cpp:
#include "my_library.h"

void f(FancyAPI* widget, int i) { if (i<0) { /* yikes error! */ } ... };

void my_code(FancyAPI &widget) {
   …
   f(&widget,-1);
}
```

If the “local” function is a perfect match, the local function will be chosen, but a slight change brings back the problem:

```
// my_library.h:
struct FancyAPI {
        // …
};

void f(FancyAPI*, double);
void f(FancyAPI*, size_t);        // added later (maybe)


// user_code.cpp:
#include "my_library.h"

void f(FancyAPI* widget, int i) { if (i<0) { /* yikes error! */ } ... };

void my_code(FancyAPI &widget, size_t i) {
   …
   f(&widget,i);
}
```

Now the **FancyAPI** intercepts/hijacks, the call.

In 1983 this was considered so scary that I experimented with ways of marking all overloaded functions as special. That proved unmanageable. Today, we do regularly see problems with "the wrong function" being picked, but I don't consider that one of the worst problems with C++. My guess is that "overloading selecting an unexpected function" will remain a more significant problem than "finding the wrong function when looking in nested scopes."

6.3. Inheritance

It has been pointed out that the problem with uniform syntax is that a remote change to a program, such as adding a member to a class or a nonlocal function can change the meaning of a program.

It should be noted that such a change sometimes is exactly what was intended by a change and that changing the behavior in many places by a single change is an ideal for some uses. For example, it is often considered good that you can change the behavior of a set of derived classes by changing a base class. In such cases, the "hijacking" is a feature, not a bug. However, consider an inheritance variant of the **FancyAPI** example:

```
// my_library.h:
struct FancyAPI {
  void member1(int);
  void member2(int);
  void member3(size_t);          // (maybe)
};

// user_code.cpp:
#include "my_library.h"
class My_class : public FancyABI {

        void member3 (int i) { if (i<0) { /* yikes error! */ } ... };

        void my_code() {
                ...
                member3(-1);   // calls My_class::member3; hides FancyABI::member3 (maybe)
        }
}

void my_code(My_class &widget) {  // use derived class
  widget.member3(-1);
}

void my_code(FancyABI& widget) {  // use base class
  widget.member3(-1);
}
```

By using

```
        using FancyABI::member3;        // alleviate hiding problems
```

I could turn the hiding problem into an overload resolution problem.

Virtual functions do a lot to alleviate the potential problems, as do systematic use of **override**. We did, however, survive without **override** for 25+ years, and with the equivalent problem for data members (for which we don't have, and probably don't need a remedy).

A trickier variant arises from using a template argument as the base (often to circumvent the need for virtual functions):

```
// user_code.cpp:
#include "my_library.h"
template<class Base>
class My_class : public Base {

        void member3 (int i) { if (i<0) { /* yikes error! */ } ... };

        void my_code() {
                ...
                member3(-1);   // calls My_class::member3; hides FancyABI::member3 (maybe)
        }
}

void my_code(My_class<FancyABI> &widget) // use derived class
{
   widget.member3(-1);
   widget.my_code(-1);
}

void my_code(My_class<OtherABI>& widget)  // use base class
{
   widget.member3(-1);
   widget.my_code(-1);
}
```

Again, this is a delight to language lawyers and a challenge to language designers, but we have lived happily with it since 1990 or so.

6.4. Nested namespaces

Nested namespaces is probably the most direct equivalent to the unified call problem that we have today:

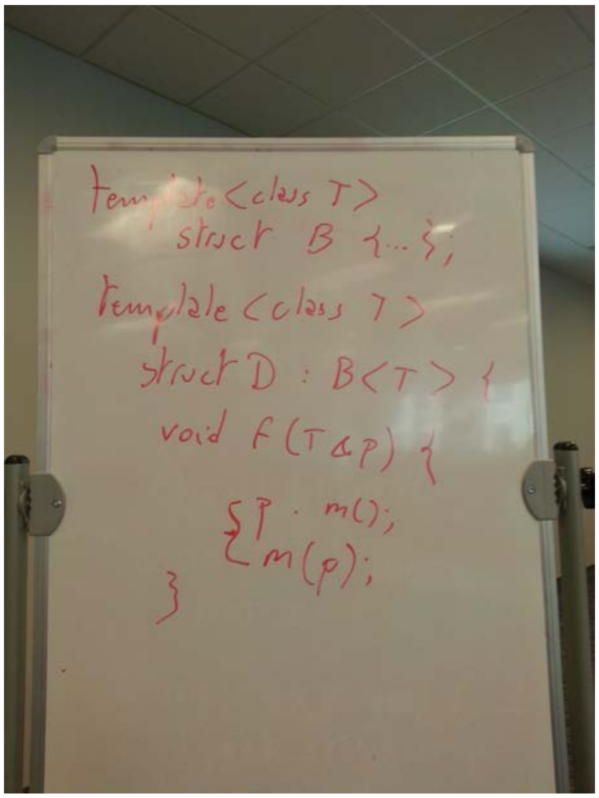
```
// my_library.h:
namespace FN {
   struct FancyAPI {
         // ...
  };

   void f(FancyAPI*, double);
```

```
    void f(FancyAPI*, size_t);       // added later (maybe)
    …

    Namespace MN {
    // user_code.cpp:
    #include "my_library.h"

    void f(FancyAPI* widget, int i) { if (i<0) { /* yikes error! */ } … };

    void my_code(FancyAPI &widget) {
      …
      f(&widget,-1);
    } // MN
  } // FN
```

Here **MN::f** hides **FN::f**.

7. Template lookup

In Lenexa, David Vandevoorde presented this example:



```
template<class T>
    struct B {...};
template<class T>
  struct D : B<T> {
    void f(T&p) {
        { p.m();
        { m(p);
    }
```

Daveed was wondering how lookup occurs:

> Currently, for **m(p)**, lookup is split in two phases: Ordinary lookup when the template is first parsed (phase 1), and argument-dependent lookup when it's instantiated (phase 2). For **p.m()**, there is only a lookup at instantiation time.

> So suppose we have a call **p.m()**. If at instantiation time, we realize it cannot be resolved the traditional way, we're past the point where we could make it equivalent to **m(p**), because the phase-1 lookup context is gone. We could always assume it's going to fail and pre-emptively perform a phase-1 lookup for every **p.m()** form, but that might be quite expensive.

This is a surprising (to me at least) consequence of the two-phase lookup that I had not taken into account earlier.

There are two obvious ways handling this:

- For both **x.f(y)** and **f(x,y)**, do the lookup for **f(x,y)** and later do the lookup for **x.f(y)** and only after that decide which resolution to use. This would incur the double lookup cost in all cases, but would allow use to choose semantics (member preference, preference based on notation used, or overload resolution) exclusively based on need. Waiting for the second lookup would delay error detection significantly for template functions (instantiation time for templates).
- Unify calls only for the **f(x,y)**, notation. That would mean that the only change from status quo would be a phase-2 lookup where we currently give an error.

I don't know how expensive would be to do the double lookup required by the first alternative.

The second alternative is exactly opposite to Herb Sutter's original proposal: Unify calls only for the **x.f(y)**, notation. Herb was at the Cppcon meaning and didn't respond with an instant "over my dead body."

Note that all STL-style code (and that's essentially all generic code) rely on the **f(x,y)** notation. Note also that the **f(x,y)** notation is used to define customization points (e.g., **std::swap()**).

So if we generalize only one notation, it must be the **f(x,y)** notation: STL, lambdas, and customization points.

If we generalize only one notation, we should expect every generic library and every concept to use it.

Potential disadvantage of generalizing only the **f(x,y)** notation include

- People who like the **x.f(y)** might get very upset on aesthetic grounds. After all, variants of that notation has been a key part of the OO culture.
- People who fear side effects from generalization might then prefer **x.f(y)** because it is more restrictive
- The benefits of auto completion of **x.f(** would not be achieved (see section 8).

It was observed (by several people) that the major objections to generalizing the **x.f(y)** notation to also handle the **f(x,y)** notation might be alleviated once we have modules. In that case, the search for **f(x,y)** would be limited to the modules of **x** and **y**. However, we don't yet have modules, so I won't try to make this idea concrete.

8. IDEs

Many (but not all) programmers rely on IDEs with helpful "auto completion" features (lists of possible functions to call). For example **x.f(** can yield a list of all member functions **f** in **x**'s class. Similarly, **f(x** can yield a list of all free standing functions called **f** that can be called with **x** as their first argument. The task for **f(x** is harder than the task for **x.f(** both because the set of non-member **f**s is open and because conversions of **x** are allowed. Obviously, uniform call would enlarge the set of function that could be called (to the union of what can be called for **x.f(** and **f(x**). I don't see that as a major problem and it would actually be useful in cases where the operation a programmer considered did not support the syntax used.

9. Warnings

It has been suggested that compilers can detect clashes between member and non-member functions. It has also been pointed out that this may slow down compilation.

I'd like warnings, but we should not base a language design on assumptions about better warnings. In this case, I worry less about compiler speed than about false positives from the many double definitions (e.g. **begin(x)** and **x.begin()**) that we have because of the lack of uniform calls. I suspect warnings are best provided optionally (e.g., -Wuniformcall) or as a separate static analysis tool (e.g., lint) with a way of suppressing known false positives.

10. Implementation experience

Faisal Vali implemented a couple of the alternatives. He also ran a small experiment based on the examples from Nicholai Josuttis' book and found no case where a "wrong" function was selected.

11. Explicit qualification of calls

There were suggestions that a way of qualifying a call to say "call a member only" and/or "call a non-member only" would be useful. Such a feature would do nothing but restrict the set of alternatives for an individual call. Systematic use would be a matter for a coding guideline. Programmers would be confused about what usage would be best and opinions would vary, leading to complaints and confusion.

We already have qualification with a namespace name to direct a call into a specific interface.

12. Explicit qualification of classes

There were suggestions that a way of qualifying a class to say "if not found, look for a non-member function" (the suggested uniform call semantics) and/or "if not found, do not look for a non-member function" (the status quo semantics) would be useful.

This would do far more harm than good. When calling a class member or trying to extend an interface with a helper function, the programmer would have to consider what kind of class was being used. Different libraries would require different use and different techniques. This is against the fundamental idea of uniform access. It is also against the idea that we should not have many distinct kinds of classes.

Also, we would need an equivalent way of annotating a set of overloaded functions with something saying "if not found, look for a member function" (the suggested uniform call semantics) and/or "if not found, do not look for a member function" (the status-quo semantics). I don't know how to do that effectively (where would you put such an annotation?).

13. What about ->?

There has been some discussion about the meaning of **p->f()**. This note is a clarification and a documentation of what we decided in Lenexa.

- For **p->f()**: try **p->f()** and if it is not valid, try **f(*p)**
- For **f(p)**: try **f(p)** and if that's not valid, try **p.f()**

These are not new rules, they are simply consequences of the ancient C rule that **p->f()** is equivalent to **(*p).f()**. This implies that when **p** is a pointer, **p->f()** can match a function taking a reference (e.g. **void f(X&)**), rather than a pointer (e.g. **void f(X*)**), so that dereference happens in both cases. Conversely, **f(p)** can match a member function for an object **p** rather than a pointer **p**, so that dereferencing happens in neither case.

These rules apply to smart pointers as well as ordinary pointers.

Summary

Uniform call provides significant benefits (simplifications) to the programmer and the problems with interface design conjectured are no worse than what we already manage quite well (section 6) and is addressed by conventional use of namespaces (Section 5). Users of other languages with similar features deem their versions of uniform call a valuable facility. Technical issues with lookup and their possible resolution are discussed in section 7.