Fostering Usability of Dynamic, Recursive, Heterogeneous Dictionaries in Statically-Typed Languages

Richard T. Saunders
Rincon Research Corporation
rts@rincon.com

Clint Jeffery
University of Idaho
jeffery@uidaho.edu

Abstract

Today’s modern software projects are written in many different languages: Python, C++, Perl, Java, Javascript, Lua, Unicon, C, all co-exist within a complex environment implementing different components of a system. This proliferation of languages causes dynamically-typed language concepts to infiltrate statically-typed languages. One important idea that seems ubiquitous across dynamic languages is the dynamic, recursive, heterogeneous dictionary: the Python dict, Perl hash, Javascript object, Lua table, and Icon/Unicon table are all realizations of this abstraction—JSON and, to a lesser extent, XML are language-agnostic versions of the dynamic dictionary. Statically typed languages (like C++ and Java) historically have trouble expressing and dealing with dynamic dictionaries without clumsy libraries. This paper explores how to represent dictionaries in C++ by introducing a simple, novel solution to dealing with dynamic constructs by embracing unique static-typing features of the C++ language: type-inference, user-defined conversions, type selection, and overloading. Taken all together, these static features paradoxically make dynamic dictionaries much easier to manipulate from C++, approaching the ease of dynamic dictionary manipulation in dynamic languages. Note that these techniques do not require special reflection facilities/libraries.

1. Introduction

Large software projects today are made up of many languages[21]. Performance critical code is typically written in statically-typed languages like C/C++ and possibly Java. Scripting and front-end code is typically written in dynamically-typed languages like Python, Ruby, Lua, Icon/Unicon and Javascript. This proliferation of languages in real software projects has made it necessary for information to propagate from one language to another: front-end dynamic languages need to communicate with back-end static languages and vice-versa[4][9].

PicklingTools is a library meant to bridge the gap between static and dynamic languages. The original goal of the PicklingTools library was to make it easier to use Python and C++ together: see http://www.picklingtools.com. At first, this involved getting serialization libraries working: Python has a built-in serialization format, called the pickle format[22], which is the de-facto format for Python serialization. To talk to Python, the PicklingTools library implemented the pickling serialization in C++. Once this was done, serialized data could then be transported between Python and C++ via files or sockets. The information that flowed between C++ and Python was the Python dictionary; it was the currency of the system.

Over time, it became obvious that manipulating Python dictionaries in C++ is non-trivial. C++ is a statically-typed language and Python is a dynamically-typed language; their approach to handling hash tables and arrays is very different. If Python dictionaries are the currency of choice, the C++ side has to be able to manipulate these dictionaries as easily as Python for the library to be successful. Originally a secondary goal, making dynamic dictionaries easy to manipulate from C++ became a primary goal of the PicklingTools library. A practical metric of success is the number of systems currently using the library; as of this date, there are at least 30 different projects across multiple companies using the library[21].

In Python, the dictionary (a.k.a., dict) is one of the fundamental building blocks of the language: namespaces, modules, classes, are all implemented using the dict; this in turn enables dynamic introspection. Because of the ubiquity of these dictionaries in Python, or perhaps because they are so important, dictionaries are fundamentally easy to manipulate, create and read.

The Python dictionary has three main properties of interest.

1. Heterogeneous: values in dictionaries can be any type, and keys can be almost any type (as long as they are hashable).

2. Recursive: dictionaries can contain dictionaries of dictionaries, ad infinitum. This is closely related to heterogeneity, as the dictionary is just another type, but the recursive nature is well-supported in Python. For example, cascading lookups and insertions are easily expressible (see below) so that nested dicts are accessible.

3. Dynamic: the values can change at run-time. This is also closely related to heterogeneity: values can be any type, but that type can also vary at run-time. A C struct, in contrast, can contain heterogeneous types, but can’t change after compile time.

Manipulating dictionaries is natural in the Python language.

# Python
>>> v = "abc"
>>> v = 1   # dynamic values

# heterogeneous types in dict
>>> d = { 'a': v, 'nest': { 'b': 3.14 }}

# recursive, supports cascading lookup
>>> print d['nest']['b']

# recursive insertions
>>> d['nest']['new'] = 17.6
Other dynamic languages support similar constructs to the Python dict; the Lua table, the Perl hash, the Unicon dict, and the JavaScript object. The Javascript object is so popular, it has led to a successful language agnostic standard: JavaScript Object Notation or JSON[8] (It is interesting to note that the JSON dictionary is almost an exact match of the Python dict). Why is the dictionary in dynamically-typed languages so popular? There is evidence[10][19][23] that dynamically-typed languages make certain kinds of tasks significantly faster to solve. The Prechelt[19] study in particular describes how the built-in dictionaries of dynamic languages gave users an advantage.

Expressing a dict in C++ can be done, but to do it simply so that it feels like Python inside of C++ is much more difficult. There has been some direct work towards expressing dynamic types in static languages[1][13][14][25], but either the work is not mainstream or doesn’t address the dynamic dictionary well (see Related Work for more discussion). The PicklingTools collection brings dynamic values into mainstream C++ with a simplicity approaching that of Python and other dynamic languages.

```cpp
// C++;
Val v = "abc";
  // dynamic values

// heterogenous types in Tab
Tab d="{'a':1,'nest': {'b': 3.14 }}";

// recursive, supports cascading lookup
cout << d["nest"]["b"] << endl;

// recursive insertions
d["nest"]["new"] = 17.6;
```

The PicklingTools library is an open-source toolkit and freely available. Note that the PicklingTools is simply a C++ library; any modern C++ compiler should be able to compile the library.

## 2. Basics

Every variable in C++ must have a static type; PicklingTools introduces the new type `Val` as a dynamic container. It can contain any primitive type (where primitive is defined as any size int, float, complex, string, or None) and some very specific composite types (Tab (Python dict), Arr (Python array), and a few others).

The `Val` is the centerpiece of the C++ library: it is a necessary part of constructing dynamic dictionaries, as it adds the dynamic component.

## 3. Overloading

The use of static overloading helps the compiler choose types at compile-time. C++ can create a `Val` by examining the static type of a given argument—from that type information, C++ can choose an appropriate constructor:

```cpp
Val a = 100; // int
Val b = 3.1415; // double
Val p = 3.1415f; // float
Val c = "hello"; // const char *
Val n = None; // empty
Val t = Tab("{'a': 1 }"); // Tab
```

The constructor for the `Val` is statically overloaded on all primitive types (and a few composite types) of the language; this means the choice of constructor is done at compile-time. This is actually an advantage over dynamic languages because it bypasses a lot of runtime checking. In essence, the proper object is chosen at compile-time with no extra runtime work needed. The constructor fills in the proper type tag and builds the right type inside a union (composite types are constructed using C++’s placement new inside the union).

### 3.1 Overloading All Primitive Types

One important implementation note: it is of crucial importance that every primitive type be constructible into a `Val`. If just one is left out, the compiler generates massive amounts of error output to show the static type inference engine can’t choose between the types. The C++ engine for choosing between overloaded types is quite complicated: it first looks for an exact match to the type; if it can’t find an exact type match, it will try several other type conversions to see if it can convert the type into an exact match. If there is more than one conversion, the overloading engine will simply error out with a compiler error, showing all different possibilities, forcing the user to choose one explicitly.

The simple example below demonstrates the folly in forgetting to create a constructor for one of the different floats:

```cpp
class Val {
public:
  // Constructors for Val overloaded
  // on all different types
  Val (int_1 a) : ...
  Val (int_1 a) : ...
  Val (int_2 a) : ...
  Val (int_2 a) : ...
  Val (int_4 a) : ...
  Val (int_4 a) : ...
  Val (int_8 a) : ...
  Val (int_8 a) : ...
  Val (real_4 f) : ...
  // whoops: forget double i.e., real_8

  Val f = 1.0; // 1? 1.0f? int(1)? int_4(1)?
  // COMPILER ERROR: What conversion to use?
  // Massive compiler output as it show ALL choices
}
```

For usability, it is important to handle all primitive type constructions. Errors can always be rectified by forcing the user to introduce more code to disambiguate types, but that undermines the simplicity of letting the static type determine the overload.

### 3.2 Overloading On Platform-Dependent Types

One subtle problem that comes up when using the `size_t` type: it is the type of the sizeof operation and is the return type of many STL container operations (usually for an index or length type). Any non-trivial usage of STL will involve integers of type `size_t`, so it is important that the `Val` works well with it. Sometimes `size_t` is typedef-ed to a long, sometimes a long long, sometimes an entirely different type, depending on the platform and operating system. Because the type of `size_t` varies, on some platforms the `Val` must have to have a special `Val(size_t)` constructor, and on some platforms, it can’t (because `size_t` is an int_u8 or int_u4 and two arms of the constructor would have the exact same type). A long-standing C technique is to #ifdef out code for `size_t`:

```cpp
class Val {
public:
  Val (int_u8 a) : ...

  #ifdef(SIZE_T_IS_NOT_TYPEDEFED_TO_INT_U8)
    Val (size_t a) : ...
  #endif
  Val (real_4 a) : ...
};
```
Of course, maintaining this code across platforms is a nightmare. There is a better way to handle this using the static type features of C++.

3.2.1 Type Selection
One of the newer features of C++ is the ability to use the generic template engine to type choose at compile-time. In essence, the compiler can detect if size_t is a different type than one of the primitives and only allow the size_t Val constructor then. Using the type selection technique from Modern C++ Design[3], a new auxiliary type is introduced:

```
// Have some default type for size_t in case
// size_t is an alias for another type
struct OC_UNUSED_SIZE_T {
  OC_UNUSED_SIZE_T (int_u8 init=0) : xx(init) { } 
  operator int_u8 () { return xx; } 
  int_u8 xx;
}; // Strictly speaking, this can be empty

// Use specialization, if size_t is same
// as int_u4 and int_u8, no problem, we make
// ALLOW_SIZE_T some unused type. Otherwise,
// if we can distinguish between int_u4, int_u8
// and size_t, we make ALLOW_SIZE_T a size_t

template <class T>
struct FindSizeT {
  typedef size_t Result;
};

template <>
struct FindSizeT<int_u8> {
  typedef OC_UNUSED_SIZE_T Result;
};

typedef FindSizeT<int_u4> {
  typedef OC_UNUSED_SIZE_T Result;
};

typedef FindSizeT<

Inside of Val, an arm for Val is always added (with no #ifdefs around it):

```
class Val {
  public:
    // special size_t constructor
    Val (ALLOW_SIZE_T) : ... 
  };
```

If size_t is a unique type (distinct from int_u8 and int_u4), then ALLOW_SIZE_T is simply a typedef for size_t. If size_t is an alias for int_u4 or int_u8, then ALLOW_SIZE_T is the unused type (OC_UNUSED_SIZE_T) which will never be used in real code.

This technique portably allows size_t as part of the suite and can also be applied if there are other conflicts between basic types.

These statically-typed techniques allow construction of a dynamic object from static type information. Manipulating these objects is the next concern.

4. User-Defined Conversions, a.k.a., Outcasts
C++ has a unique feature called user-defined conversions which allows a user-defined type to export itself as a different type. There are two major contexts in which this facility is invoked: (1) type matching in function overload resolution (compile-time resolution), (2) user-requested casting (compile-time cast)

Consider the following example for function resolution:

```
class MyVal {
  operator int () { ... }
};

int f (int i); // prototype for f:
// f takes an int type
MyVal m;

f(m); // ERROR? No! MyVal is
// allowed to convert to an int
```

During C++ static function resolution for f, the only match found for f is f(int). However, since MyVal can convert itself to an int (using the operator int() method on MyVal), C++ will let MyVal do the conversion and allow f(m) to compile and run.

The long form, given below, illustrates what is happening. This is legal, working C++; it illustrates some of the syntactic sugar provided by C++:

```
MyVal m;
int __outcasted_temp__ = m.operator int();
f(__outcasted_temp__);
```

This C++ feature is pivotal in bridging dynamic and static types. Using this mechanism for Val, the Val class can be augmented with user-defined conversion operators for all the primitive types and a few composite types. For example:

```
Val d = 1.0f;
float f = d.operator float();
// Get value from d as a float
```

This long form illustrates above very simply what is happening: a special method on the Val class is chosen to convert the Val to a float. It is unfortunately verbose. C++, however, will automatically do an appropriate conversion (as it is performing overload resolution) when it tries to construct a float from a Val directly. The short form below is equivalent to the long form above:

```
float f = d;
// Can a float construct from a Val?
// Yes, Val supports operator float()
```

By using the user-defined conversion operators with overload resolution, the static primitive type is created from the dynamic type with intuitive constructs; just ask for a float and get a float. Note that the static type of the variable dictates which conversion is invoked. Said another way, the static type of the C++ variable informs the dynamic conversion out of Val. This approach embraces the static type system of C++ while still allowing dynamic types.

Of course, there can be problems if there is a type mismatch. What if the value stored in the dynamic Val does not match what is asked for? The Principle Of Least Surprise is the guiding philosophy here. For primitive types, if C++ allows the conversion, the cast will be allowed and performed exactly as C++ would. Otherwise, like most dynamic languages, a malformed conversion will cause an exception:

```
Val v = 3.14; // v stores a double
float f = v; // WORKS AS C++ would:
            // double in v copied out and
            // converted to float, losing precision
int i = v;  // WORKS AS C++ would:
            // double in v copied out and
            // converted to int, dropping fractional
```

What if the value stored in the dynamic Val does not match what is asked for?
Tab t = v; // AS A DYNAMIC LANGUAGE:
   // throws exception, as this
   // conversion doesn’t make any sense.

Calling the user-defined conversions never changes the type stored in the Val, simply how they are copied out. In general, primitive types convert just as C++ would: a float drops its fractional part when converting to an int, a conversion from a larger type to a smaller type simply drops precision, a malformed conversion simply flags an error (with Val, it behaves more like a dynamic type where the error is flagged at runtime):

```cpp
double d = 3.14;
float f = (float)d;
   // loses some precision, still converts
int i=(int)f;
   // drop fractional part, still converts
Tab t = i;
   // throws exception, as this conversion
   // doesn’t make sense
```

This matches what Python (or other dynamic languages) typically do:

```python
d = 3.14
i = int(d) # drop fractional part
t = dict(i) # malformed conversion
```

4.1 Implementation

The implementation is ugly, but the final user interface and usage is the important part; the implementation described below is not what the user sees. Essentially, it is implemented as a macro with C++ conversions for each type. In other words, for every user-defined conversion, there is a giant switch which chooses the runtime type to the static requested type. It is ugly, but it does have the one redeeming feature that it is all encapsulated in one place in the code, so only Val itself contains the excessive amount of code. In essence, each user-defined conversion operator has code that looks like:

```cpp
operator int () {  
   switch (this->type()) {  
   case DOUBLE: {  
      real_8* rp = (real_8*)&v.u.d;  
      return int(*rp);  
   }  
   case FLOAT: {  
      real_4* rp = (real_8*)&v.u.f;  
      return int(*rp);  
   }  
   // ... for all types the val can contain ...
   }  
}
```

To ease multiple maintenance, it is implemented using macros. The nice thing about the above solution is that it does allow control over exactly what every single conversion is. This was especially important to the engineers who work with the PicklingTools library. A complex number converting to scalar number is potentially confusing; is it phase or magnitude or magnitude-squared? It is better to let the engineer choose the proper conversion (by forcing them to call phase, mag, mag2). By default, converting from a complex to a scalar will be flagged as a conversion error (which is what Python does as well).

There is another solution which bypasses the code bloat of this implementation: the C++ type_info. Its benefits and flaws are discussed in the User-Defined Types section.

4.1.1 Overloading All Primitive Types

Just as the overloaded constructor for Val needs all primitive types represented to avoid confusing the overload engine, the user-defined conversion needs every primitive type for the same reasons (as well as a few select composite types like the Tab and Arr).

```cpp
class Val {
   public:
      // Constructors for every primitive type
      // User-defined conversion for every primitive type
      operator int_1() ...
      operator int_u1() ...
      operator int_2() ...
      operator int_u2() ...
      operator int_4() ...
      operator int_u4() ...
      operator int_8() ...
      operator int_u8() ...
      operator real_4() ...
      operator real_8() ...
      operator complex_8() ...
      operator complex_16() ...
      operator ALLOW_SIZE_T () ...
      operator string () ...
      operator Tab& () ...
      operator Arr& () ...
};
```

Without this, the type matching engine for C++ can easily become confused when outcasting types. If there are multiple possible matches, C++ will fail with an error message at compile-time.

4.1.2 Limits

With the constructors and user-defined conversions, the usage of Vals and multiple types starts to feel much more like a dynamic language:

```python
# Python // C++
a = 3  Val a = 3;
   // constructs proper value
f = float(a)  float f = a;
   // outcasts to proper type
s = str(f)  string s = a;
   // stringizes
```

There are some limits to this approach:

```cpp
Val v = 1;
string s = 1;
   // Works, calls proper user-defined conv.
s = v;
   // FAILS TO COMPILE: string operator= and
   // outcast both match
```

The STL string, i.e., the standard string that comes with C++ has three competing definitions for the operator= defined:

```cpp
string& operator= ( const string& str );
string& operator= ( const char* s );
string& operator= ( char c );
```
The addition of the Val to the mix prevents C++ from choosing which Val user-defined conversion to use. There is a way around this by explicitly calling the user-defined conversion:

\[
s = \text{string}(v); // \text{Forces } v \text{ to outcast to a string}
\]

This solution works in general, but it does break the simplicity of letting the static type of the variable choose the outcast. The string (as defined by the STL) and the complex types complex\_8, complex\_16 (as defined by the OpenContainers library[20][21], which is included with the PicklingTools distribution) are the only types that suffer from this problem. Since other libraries control the interfaces of these classes, those libraries cannot be changed solely for the convenience of the PicklingTools. Fortunately, the simple workaround of forcing a specific outcast is very readable.

5. Composite Types

Once C++ can handle primitive types well, the composite types are paramount. The dict (a.k.a., Tab) is a type of hash table called the OCAVLHashT (which comes from the OpenContainers C++ library) where the key is a Val and the value is a Val.

```cpp
class Tab : public OCAVLHashT<Val, Val, 8> {
...
}
```

Tab is a cross between a hash table and a balanced-tree structure; the intent is to implement an extendible hashing scheme which keeps lookups quick (via integer keys) while avoiding rehashing. The hash table is implemented by an AVL Tree where the keys on the nodes are integers containing the full hash value of the key; this decreases the likelihood of hash collisions by keeping all the bits of information in the hash (no information is ever lost via a modulus operation which throws away upper bits). For rare collisions, the entire value must be stored as well. Keeping the node comparisons as integers also promotes quick lookups. There is never a full-table rehash when the table becomes too big, as the tree is always balanced.

The other primary composite type, the ordered heterogeneous sequence analogous to Python’s list, is implemented using a C++ array of Val\_s called an Arr; both the Python list and Arr are both contiguous arrays of heterogeneous objects that dynamically resize as they grow.

5.1 The Dictionary Literal

One goal of the PicklingTools is to be able to cut-and-paste dictionary literals between C++ and Python so that the same dictionary in Python would work with C++, and vice-versa. Python, of course, has no trouble with the literal:

```python
>>> d = { 'a':1, 'b':2.2, 'c':[1,2.2,'three'] }
```

But how can C++ handle this? Unfortunately, the syntax of C++ and Python is too disparate to hope that C++ might parse the above dictionary in some form (even with the new C++11 literals). However, by overloading the constructor of the Tab to take a string, Python literals can be supported inside the string:

```cpp
Tab d="{'a':1, 'b':2.2, 'c':[1,2.2,'three']}";
```

This will work and is legal C++, but the single-quoted version is much more readable. Further, the single-quoted version can be cut-and-paste directly between Python and C++ without change.

This may seem an overly simple method for supporting literals, as it just parses a C++ string to get a dictionary literal, but it has been surprisingly useful in real code to set up a basic structure from a dictionary literal—It is a minor feature that contributes to making C++ feel more like a dynamic language.

The parsing of these literals is relatively inexpensive. Section Putting It All Together: Speed discusses the speed of parsing literals (“text”) in both Python and C++.

5.2 Lookups and Insertions

In order to support lookups and insertions, the composite types (Tab and Arr) need to support lookup via the [] operator. C++, unfortunately, doesn’t distinguish between lookup and insertion when overloading the [] operator.

Operation [] returns a Val reference and can be used in both insertion and lookup contexts. As a lookup, a user would write code something like this:

```cpp
Tab d="{'a':1, 'b':2.2, 'c':[1,2.2,'three']}";
```

There are many extra steps and conversions happening here. Below is the equivalent code: it is the explicit (fully legal C++) long form showing all the interfaces and conversions:

```cpp
Tab d="{'a':1, 'b':2.2, 'c':[1,2.2,'three']}";
```

Val\_\_key\_ = "a"; // Create a Val for the key
Val& \_\_valref\_ = d.operator[\_\_key\_];
operator<<(cerr, \_\_valref\_);
```

For insertion, the style the user would use:

```cpp
d["key"] = 2.2;
```

The C++ long-form (which is equivalent code, but demonstrates all the conversions) looks like:

```cpp
Val\_\_key\_ = "key";
Val& \_\_valref\_ = d.operator[\_\_key\_];
// Not there, creates!
Val \_\_newthing\_ = 2.2;
\_\_valref\_ = &\_\_newthing\_;
```

Since C++ does not give the user a hook to distinguish between a lookup and insertion, the call to operator[] must always return a Val\&. If the key is there, it returns a reference to the value already there. If the key isn’t there, it must create a new Val of type None. This allows an insert to simply write over the new value, i.e.,

```cpp
Val\_\_valref\_ = &\_\_newthing\_;
```

Unfortunately, for lookup, this is not desirable; if the lookup a value that is not there, it has the side-effect of changing the table.

To get around this problem, the library encourages use of operator() for lookup and operator[] for insertion:

```cpp
cerr << t(\"not there\") << endl;
// throws exception as expected
```

Since C++ has a nice notion of const, where an insert and a lookup can be distinguished by whether or not the object is a constant object, but that approach is problematic for many reasons, mostly that depending on the const nature of a value may be limiting. See Meyers[16][17]. For this reason, the PicklingTools library opted
6.1 Syntax

6.1.1 Subscripting

Part of supporting nested dictionaries and lists well is supporting
simple interfaces to get and lookup items nested deep down in a
structure. In Python, this is directly supported by the language:

```python
Tab d = {'a':1, 'b':2.2, 'c':{'1,2,2, 'three'}}
print d['c'][1] # lookup
```

The PicklingTools libraries enable user C++ code to be similar:

```cpp
Tab d = {'a':1, 'b':2.2, 'c':{'1,2,2, 'three'}};
cout << d("c") (1); // lookup
```

Consider the user code for the cascading lookup:

```cpp
cout << d("c") (1); // lookup
```

The first portion of the cascading lookup works because the
operator() on Tab/Arr returns a const Val&. The second portion
(and further cascades) work because the Val itself has overloaded
the operator() and operator() to return a Val and a const Val
respectively. Rewriting this in long-form as (legal) C++ demonstrates
all the work C++ does for the user:

```cpp
Val _key1_ = "c";
const Val& _subc_ = d.operator()(._key1_);
Val _key2_ = 1;
const Val& _subc1_ = _subc_.operator()(._key2_);
operator<<(cout, _subc1_);
```

Consider the user code for the cascading insert:

```cpp
Val _key1_ = "c";
const Val& _subc_ = d.operator()(._key1_);
Val _key2_ = 1;
Val& _subc1_ = _subc_.operator()(._key2_);
_subc1_ = "one";
```

The corresponding long-form for the insert would look like:

```cpp
Val& _subc_ = _subc_.operator()(._key1_);
Val& _subc0_ = _subc_.operator()(._key2_);
_subc1_ = "one";
```

6. Putting It All Together

6.2 Speed

The speed of the C++ Val class provided in the PicklingTools li-
brary competes well with the internal speed of Python objects as
manipulated from within the traditional C Python runtime system.
The PicklingTools library provides a test suite for comparing the
speed of pickling and unpickling large dictionaries. While the pick-
ling tests are not a general benchmark of overall dictionary perfor-
ance, they represent an important common use case.

The suite, **concentrating solely on dynamic dictionary performance**,
demonstrates that the Val and the Python object are roughly
equivalent in speed. See Figure 1.

This test serializes a large dictionary, including many nested
dictionaries and lists; this approximates many common use cases
in one dictionary. The Python code creates a wide dictionary of
around 20,000 keys that are an assortment of primitive values and
shallow sublists and subdictionaries.

The Python object model is faster than the Python object
pickling protocol 2, as C++ could typically iterate over the
keys of the table faster than Python. For unpickling, the Python object
pickling is faster because Python object creation has been highly
optimized inside the single-threaded interpreter (Python uses a
Global Interpreter Lock) so that it quickly re-uses recently allo-
ated objects.

The Val class is thread-safe, and that limits the memory opti-
mizations available, but allows for the prospect of a true concur-
rent speedup for the C++ version of this benchmark in future.
See [20] for more discussion of thread-safety assumptions and their
limitations. It is somewhat surprising, but the speed of the creation
of the dynamic objects seems to be the limiting factor. For both
C++ and Python, the cost of dynamic object creation dominates
unpickling; the cost of dictionary iteration dominates pickling.

The work of Prechtl[19], although slightly dated, shows dy-
namic languages solutions of their study relying heavily upon dic-
tionaries: "...the script group’s programmers found the hash tables
built into the language to be an obvious choice of implementation". The empirical results of [19] demonstrate that runtimes
and memory usage among several dynamic languages (Tcl, Rexx,
Python, and Perl) clustered together during the two main phases of
the study program (with Rexx performing poorly during the second
phase). Given that the study indicates that hash tables are the “ob-
vious choice” of the study participants using scripting languages,
this offers a loose comparison of speed and memory usage of dy-
namic dictionaries in different languages. Although there is vari-
ability among the scripting language speeds (due to programmer
participant quality, language libraries and implementation details),
the scripting language speeds tended to cluster together. Unfortu-
nately, this is a weak correlation and suggests further benchmarking
work of dynamic dictionaries across languages. It is notable,
however, that Python and Perl were the fastest of the scripting lan-

```cpp
Tab d = {'a':1, 'b':2.2, 'c':{'1,2,2, 'three'}};
int v = d("c")[0];
v += 3;
d["c"][2] = v;
```

Other current libraries implement this simplicity to a lesser
extent. See Related Work (below) on JSON, XML, or the any,
packages below.
7. User-Defined Types

An apparent drawback of the Val type is that it only supports primitive types and a few composite types (Tab, Arr, tuples, Array<POD> and OTab); it doesn't directly support user-defined types. It can, but in an orthogonal way.

Philosophically, supporting only lists, dicts and primitive types isn’t limiting; this is the basic argument of XML (as all data constructs can all be expressed through some decomposition into lists, dictionaries, strings and primitive types).

A user-defined class can be expressed by how it adapts into a Val or back. By having a class write a user-defined conversion to a Val and a constructor from a Val, it can be used within the Val framework like any other type; the key is that the underlying Val is always just a mix of the standard primitives and composites. For example:

class MyType {
public:
  // Construct a MyType from a Val. This allows:
  //   Val v = ... data from socket ...  
  //     MyType m(v);
  MyType (const Val& v) { ... }
     //... convert table in v to this data structure...

  // Construct a Val from a MyType. This allows:
  //   MyType m;
  //     m.doAnything();
  //   Val v = m;  // storing a MyType in a
  operator Val () { ... }
     //... convert this to a Tab or string ...
};

Any class that wants to be a “dynamic” type in this framework has to know how to convert itself to and from a Val; this usually involves constructing a dictionary which holds the state of an object (from complex composite types) or bitblitting to a string (for POD types). If a class can’t be changed (because it is in a untouchable library), global adapters can always be written:

```cpp
MyType ConvertValToMyType (const Val& in);
    // Val -> MyType
Val ConvertMyTypeToVal (const MyType& in);
    // MyType-> Val
```

The major disadvantage (currently) of this approach is that the Val currently doesn’t carry the type of the user-defined classes; it is just a standard Val. In practice, this hasn’t been a problem; if a class gets a Val it doesn’t understand, it simply throws an exception, like any good dynamic language would when trying to convert between disparate types.

7.1 An Alternative Approach

One approach to making user-defined types more cohesive is to use the built-in C++ type_info. The type_info is a built-in C++ STL library mechanism that allows capturing the compile-time type of an object; it is a very limited introspection capability that only allows two operations: comparison of types and a very implementation-dependant printing of said types.

```cpp
MyType m;
type_info type1 = typeid(MyType);
type_info type2 = typeid(m);
cout << ''Not same type'' << endl;  // Human readable name
if (type1 != type2)
  cerr << ''Not same type'' << endl;
```

If a type_info field is added to the Val, this would allow the Val to carry user-defined classes much better.

```cpp
class Val {
  type_info dynamicType_;

  // Use a template for ALL constructors
  template <class T>
  Val (T& a) : dynamicType_(type_info(a)) ... 
};
```

There are two problems with this approach. First, the user doesn’t have as much control over how the types combine. As mentioned earlier, without a strict control of the overloaded types, the C++ overload resolution engine quickly degenerates into voluminous compiler error output as the C++ overload matching system cannot figure how to overload. Limiting type interaction during overloading is crucial to making it easier for a user.

The second problem of the type_info is that it has a virtual method (virtual destructor), which makes it difficult to use with cross-process shared memory (a minor goal embraced later the PicklingTools).
8. Related Work

8.1 Other Languages

One earlier attempt to add dynamic typing to statically typed languages was Abadi, Cardelli, Pierce, and Plotkin[1]; this work introduced a new language called Dynamic which added the dynamic and typecase constructs to a simply-typed lambda calculus[7][11]. The typecase keyword essentially allowed the programmer a C-like switch statement, where switching was on the type tag of dynamic values. The dynamic keyword bypassed the simple static typing in the language, allowing type checking to defer to runtime. The Dynamic programming language never became a mainstream programming language, but the dynamic keyword (and its ilk) survived to become part of at least one mainstream modern language (see below).

The dynamic keyword of C#(14) was a concept introduced as part of the Microsoft .NET suite in C# version 4.0 in Visual Studio 2010; it was for supporting dynamic languages better. The .NET suite was originally designed for statically-typed languages and Microsoft realized there was a need for more dynamic language support. The dynamic keyword on an object causes the compiler to bypass the static checking; all errors are deferred to runtime (which may cause exceptions to be thrown); the approach C# has taken, at least initially, is very similar to the Dynamic programming language approach[1].

The support for dynamic in C# is excellent; The implementation uses a complex runtime system, as the dynamic presents a single interface for three different types of dynamic dispatch: COM IDispatch for COM objects, IDynamicMetaObjectProvider for DLR interface, and reflection run-time libraries for everything else. The use of dynamic does allow any object to be carried around; this is more powerful than the limited Val, but C# requires the much more complex runtime to support.

As a language, C# supports all of the static type features mentioned in this paper: static overlapping, user-defined conversions, overloading and type-inference. Although C# has operator overloading, [] is not overloadable explicitly; instead the same effect is achievable through the use of indexers. The support for dictionary literals is much better than C++ (with similarities to Python). Putting the static type features together with the dynamic keyword, C# has all the features necessary to manipulate the simple dynamic dictionary of this paper. The methods presented in this paper for C++ don’t require the complex runtime system of C#, but using C#’s sophisticated introspection facilities (which C++ doesn’t have), the dict may be even easier to express. What’s surprising is that there doesn’t currently seem to be any “normal” efforts to put all the pieces together for the simple, easily-expressed dict; other issues may arise (such as supporting multiple integers cleanly) when porting to C#. The 10+ years of experience with the PicklingTools library in C++ has shown that truly vetting the library and its idiosyncrasies requires usage by at least a few large projects. Without some motivating C# projects, a naïve attempt at converting the PicklingTools dict to C# would be subpar.

8.2 Quasi-Static Typing

The work of Thatte[25] introduces the notion of Quasi-Static Typing which explores mixing static and dynamic types within a static framework (some follow up work was done in [18]). There are some major similarities structurally: the positive coercions of [25] from static to dynamic types reflect what C++ does in constructor overloading. The negative coercions of [25] from static to dynamic types is similar to what C++ does with user-defined conversions. The paper is theoretical, based in λ-calculus and semantics proofs. Quasi-static typing provides some theoretical grounding to many of the ideas in the C++-based PicklingTools library.

8.3 Boost any

The Boost any[2][12] is the most similar work to the PicklingTools Val work. It is more general in being able to express multiple types, but it suffers from a more verbose interface because of its generality. In contrast, the Val interface has been strictly designed to be simple and look like dynamic languages. For instance:

```
// Val structure  // Boost any
Val v = 5;     any x(5);
cout << v;     cout << any_cast<int>(x);
int out = v;   int out = any_cast<int>(x);
```

The any can do one thing the Val currently can’t: capture a general type that meets its specifications for copy-construction and assignment requirements. This allows any class to be copied around with those. The Val has taken a different approach; it only allows primitive types and a few composite types (namely the dictionary and array, see previous section). All data should be able to be expressed in this manner (it is, after all, the basic argument of XML: all data can be expressed as a composition of primitive types into dictionaries and lists).

The most important point is that the Val/Tab/Arr all work in harmony and allow nested, recursive types. This is more difficult to do with the any type. Frequently, STL users manipulate the map<string, any> or map<any, any> when a dynamic dict is necessary. Using any as the key type fails because the map must be ordered (map requires a total ordering; it must support operator< and the like). Using the string as a key works, but then lists and dicts can’t be interwoven interchangeably. 1 Compare:

```
// Val/Tab/Arr
Tab t;
t[“one”] = Arr();
t[2] = 17;  // dict lookup as number ok
int r = t[2];  // list lookup okay

// any
map<string, any> t;
t[“one”] = vector<any>;
t[“2”] = 17;  // Has to be string!
int r = any_cast<int>(t[“2”]);
```

The version using any is futher removed from the simplicity of Python:

```
# Python:
t = { }
t[“one”] = []
t[2] = 17
r = int(t[2])
```

While the any approach above is verbose, it is still reasonable. Where the any syntax starts to become truly unwieldy is in the recursive lookup or insertion (aka., cascading lookup or insertion). The any approach can work with enough casts, but even with those, the syntax becomes cumbersome; three simple lines of Val code becomes twelve complex lines of any code full of casts. Consider the Val approach below:

```
// Val approach: supports cascading
// lookups and insertion like Python
Tab t="{‘a’:{‘nest’:{}}}”;
cout << t[“a”][“nest”];
t[“a”][“nest”] = 17;
```

1 There is an unordered map that is part of the recent C++ standard, but at this time, this standard is not ubiquitously implemented. Even with it, the any still suffers most of the same problems.
Consider the complexity of the any approach below. Note that there are no literals so the table structure has to be constructed manually:

```cpp
// BOOST any approach:
// no "literals": Create equiv 
map<string, any> t;
map<string, any> subtable;
subtable["nest"] = 1;
t["a"] = subtable;

// cascade lookup
any& inneri = t["a"];
map<string, any>& inner_table = any_cast<
map<string, any>>& > (inneri);
int r = any_cast<int>(inner_table["nest"]) ;
cout << r << endl;
```

Regardless of the unreadable cascade code, there are still two major functional problems with the any solution: First, there is no literal to create the dictionaries, so the any has to build the dictionary completely from scratch. Secondly, the contents of the full dictionary cannot be output, as the any and map don’t have enough information to print out anything.

The any suite is more general, at the cost of simplicity and some functionality. The Val<1> suite is simpler (more akin to Python), at the cost of some generality. This generality is compensated for by being able to express dynamic, recursive, heterogeneous dictionaries easily.

### 8.4 JSON

The Javascript Object Notation[8] is a language-agnostic way to specify dictionaries, much like XML is a language agnostic way of expressing documents. JSON captures the same essentials as a dict: a dynamic, recursive, heterogeneous dictionary. It is an interesting look at the state of the art to see what techniques other libraries have done to capture dynamic, recursive, heterogeneous JSON dictionaries in C++. At the time of this writing, there are about ten C++ JSON libraries at http://json.org. This repository holds the most current and relevant libraries for manipulating JSON in C++ (and other languages). See Figure 2 for a full comparison of features of the current JSON libraries.

The libraries seem to fall into two camps: returning nodes of a syntax tree (very similar to the DOM model of XML) or a structure which feels like a Python dictionary. Either way, the libraries use most of the techniques described in this paper, to one degree or another. None of the JSON libraries, except the PicklingTools, leverages all the techniques described herein to get a full-fledged, easy-to-use dynamic dictionary that works well with the C++ static type system.

### 8.5 XML and HTML

Although XML[6] (and HTML) are language-agnostic standards for specifying documents (placing textual content within nested structure rather than dictionaries), many people use XML to express dictionaries without the textual content. The existence of JSON (“the fat-free alternative to XML”)[8] and its popularity as an alternative to XML seems to bear this out[24].

XML has its place as a document standard; it has ordered keys and allows textual content to be annotated. Taking away those two specific features of XML, what is left is JSON objects, Python dictionaries, Perl hashes, Lua dictionaries and Javascript objects. It is surprising how often people use XML when they really just want dictionaries.

There are many different XML libraries (too many to enumerate) with different philosophies. Four popular approaches are:

1. SAX for a callback based parse: the user-supplies code when keys are parsed from the XML tree
2. DOM for a tree-based model: the user is given the entire element/tag/attribute/text tree
3. JAXB for building classes: the classes built are populated with data-members that represent the elements of the XML
4. IDE based parsing: the IDE (like Eclipse or Netbeans) uses built-in tools or plug-ins to construct, parse, and generate XML.

The SAX callback based model has not seemed to penetrate the JSON libraries; at the time of this writing, there is no known SAX-like JSON libraries or Python parsing libraries. This is because the SAX model is more difficult and less intuitive to use. The lack of SAX-like C++ JSON and Python parsing libraries seems to confirm this.

The DOM models are very popular, as this model is represented in all the JSON libraries. PicklingTools essentially uses this model, as it just gives the user a handle to some objects which represent nested structure. The XPath libraries are an XML approach to giving PicklingTools like simplicity.

The JAXB model is a very different approach; it requires knowing the expected structure of XML, via DTDs or schemas specifying that structure. The DTD/schema is analyzed and Java/C++ classes are statically generated capturing said structure. Once this code is generated, the user can start manipulating the classes from the generated code. When it works, It is an excellent approach, as the keys and values just becomes data members and classes in the language; they are easy to manipulate as they are first-class objects of the language. The problem is that it really isn’t dynamic—a schema or DTD must exist for this to work well. This approach also requires a separate “analyze schema, generate code” pass to generate the classes.

Finally, it is interesting to note how much XML processing the IDEs (NetBeans and Eclipse) do. Within the IDE, there are plugins and tools for manipulating XML: generating schema, generating code (like JAXB) to manipulate that XML, parsing XML, and so on. The Java programmers especially seem to lean especially hard on the environment and tools to handle XML for them. In dynamic languages like Unicon, Python, Lua and Perl, dictionaries are so easy to manipulate; there seems to be less desire for a complex IDE to manipulate XML, as the dynamic language users are much more comfortable with the dictionaries.

If dynamic, recursive, heterogeneous dictionaries had been easier to manipulate in static languages like Java or C++, perhaps not as many tools and different approaches would have been necessary to handle XML.

### 9. Conclusion

The Val<1> type leverages the static type system of C++ to make dynamic, recursive, heterogeneous types easier to use. Some libraries like JAXB can manipulate the heterogeneous, recursive dictionary well but lack the dynamic capability. Up until now, previous work with dynamic dictionaries (the Boost any type, the JSON and XML libraries), all struggle against the C++ static type system: the sheer number of libraries, the diversity of implementation techniques amongst these libraries, and the complex interfaces therein demonstrate the need for unifying ideas to manipulate dynamic types. Using static techniques of overloading constructors, overloading oper-
The techniques described in this paper are particular to C++ currently (and possibly C#), but all of these features (overloading, user-defined conversions, type-inference, type selection) can be added to any statically typed programming language to make dynamic types easier to use.

References


[22] R. T. Saunders. Everything you wanted to know about pickling, but were afraid to ask! In PyCon 2011, 2011.

