P2125R0: The Ecosystem Expense of Vocabulary Types

Author: Titus Winters (titus@google.com)
Date: 2020-02-20
Informational

Abstract

The types that are most commonly passed through interfaces in a given codebase are what we call “vocabulary types” - these are the most common generic forms of data for any project. These tend to be basic forms of data: strings, integers, arrays/vectors, timestamps and durations, perhaps a hashmap. It is important to recognize that perceived code health is tied to having a clear and consistent vocabulary to rely upon: programmers are going to complain about technical debt and code health if we use `int` or `time_t` in some libraries, `double` in others, and `std::chrono::time_point` in yet others. For non-scalar types (strings, vectors, hashmaps), having multiple common forms also leads to an ambient performance cost as chains of function calls convert back and forth between different forms.

This paper will attempt to clarify the costs involved in changing the set of vocabulary types for a codebase, in terms of developer frustration, API design problems, and maintenance concerns. We will also introduce some of the difficulties involved in rolling out new vocabulary types, and some new techniques that are being used to help transition between vocabulary types.

Video Form

Many of the themes of this paper are expressed in “Maintainability and Refactoring Impact of Higher-Level Design Features” at CppCon 2019 (although personally I like the slightly longer form of the same talk from ACCU, here).

Defining Vocabulary, And Why It Matters

Most large-scale changes (see Chapter 22 of the SWE Book, or Hyrum’s 2018 CppCon talk) and multi-step refactorings work by preserving semantics based on local reasoning. We can change

```
vector<string> tokens;
StringSplit(line, ',', &tokens);
```

into

```
vector<string> tokens = absl::StrSplit(line, ',');
```

because we can locally reason that the semantics of the two split invocations are the same in the case that the output parameter is empty, and then locally proving that `tokens` has been default constructed and hasn't been passed to any other mutating operation. Most changes that are generated by our refactoring tools are relying on some similar form of local reasoning, whether it is semantics of dataflow or graph properties of the #include or dependency graphs.

Types are inherently a little harder compared to functions and simple local renames, because types provide two salient features that are more complex to reason about.

First, types provide logical invariants on the data they hold. Many theoretical computer scientists can readily quote that it is easy to weaken the precondition invariants on a function, or strengthen the postcondition invariants of a function. This is true (other than Hyrum's Law surprises). Types make things more complex in that they add lifetimes to data: the data that is input for one call to a method on a type may well be the output of another method. Weakening outputs is hard, just as strengthening inputs is hard.

Especially in a large project or codebase, types form (complex, widespread) networks of dependencies across a project. When refactoring a function, one can often introduce a new version of the function and refactor calls one at a time in a semantic-preserving fashion. There is no inherent tie or connection in the order of updates to those calls. This is not so with types. If we wish to change `hash_map` to `node_hash_map` or `string` to a hypothetical `string2`, we cannot simply change individual definitions and instantiations. Types regularly pass through interfaces, thus requiring not only update of an instantiation, but all of the functions that instantiated object is passed through. And since functions are reused, we have to also find every other object that is passed to the same function. And since objects are passed to functions, we have to update all of the other functions that those other objects are passed to, and so on, transitively.

Mix in the recognition that our most fundamental types (integers, strings, hash tables, times and durations) are passed through interfaces hundreds of thousands of times and it becomes clear why changing between two different vocabulary types that serve the same purpose is expensive.

There is no clear delineation for what is and is not a "vocabulary" type: clearly this is shades of gray. Anecdotal analysis suggests that the distribution of type references within a codebase follows a Zipfian distribution, akin to the distribution of word occurrence in language. Some types don't pass through interfaces much at all. Some types aren't used much. Other types are used everywhere and pass through interfaces constantly. Modifying those types directly (while obeying existing invariants) is highly valuable. Changing the set of those types must be handled with care.

Cost of Conceptually-Overlapping Vocabulary Types

In addition to the raw difficulty of “how do we change from type A to type B”, there are a number of costs that arise during the transition that act as a drain on productivity and compute efficiency. Some of these are general and apply in any language, stemming from basic conceptual features of having strongly typed languages. Others are manifestations of language-specific issues. These various costs are particularly common when there are two overlapping vocabulary types (`hash_map` vs. `node_hash_map` or `string` vs `string2`). In cases where we are working to migrate fully, these costs will only manifest during the transition period. In cases where we expect to leave both types in place indefinitely, these are an ongoing tax on the project.

For instance, when used inconsistently, conversion between the two forms may be costly. For simple scalar types (`time_t` vs. `std::chrono::time_point`) this is a relatively minor and bounded compute cost. Even if we switch between vocabulary types for time every function in a long chain of APIs, it is hard to imagine the math involved in converting between these two representations actually dominating any performance graph. For more complex types (`string` or `hash_map`), we may well be accruing a meaningful runtime cost. These inter-type conversion costs are pure overhead: we aren’t performing any meaningful calculation, we’re just changing the representation of the data that is being passed around.

In the same scenario we also see a hard-to-measure but very present cost: programmer frustration and drop in developer productivity. When regularly faced with “I have a `Foo` but I need a `Bar` to pass to this interface (or vice-versa)”, programmers naturally become frustrated when those two types are different technical representations of the same underlying data. Constantly having compilation failures for seemingly-trivial reasons and constantly littering your code with conversion functions detracts from a good flow state and contributes to feelings of technical debt. Historically this was particularly acute in Java where Guava’s `Optional` became redundant with the newer Java `Optional`. Keeping track of which form of optional value you had and which you needed was (and continues to be) a significant and unnecessary cognitive friction. C++ benefits by allowing implicit type conversions … but only a single conversion per expression, meaning this design can mitigate but not remove these annoyances. However, for complex types we often want to make such conversions explicit to avoid computationally-costly conversions from happening with no textual trace left in the code to signal the conversion.

When we look at per-language issues that stem from language rules, transitions and co-existing vocabulary types can be seen to be even more complex. For instance, in C++ we may be required to answer some or all of the following questions:

- What about existing template specializations for the old type? Do we duplicate those for the new type? Do we have proof that is correct at scale?

- How do we manage overload sets? Imagine a generic interface “I work on any Object” with special handling for `Widget`. When we have a `Widget2` the special handling doesn’t kick in. Should we identify all overload sets that refer to `Widget` and copy that handling for `Widget2`? Do we have proof that is correct at scale?
- Implicit conversion from existing types: what do we do when an existing type implicitly converts to `Widget`? Do we add a conversion to `Widget2`? That becomes ambiguous if we ever pass this to an overload set, which leads to a compilation failure.

All of which is to say: although it is easy to say “We’ll just add a new `string`”, the ecosystem costs can be very significant. Even with the most up-to-date tools and refactoring technologies, many questions here require consideration from code owners. In many cases we must assume these costs will be ongoing if there isn’t a coherent plan for removing the old type - something C++ tends to avoid in all but the most extreme cases.

Approaches to Deploying New Vocabulary

These are what we’ve found to work in places where we have perfect visibility and are constrained by massive scale. In scenarios where we have excellent automation, some of these can be made to work in smaller scales without visibility (across project boundaries). As discussed in other essays on refactoring, the best scenario is when there is visibility and small scale: refactoring away from one type and moving to another in a single commit is always the easiest approach.

The “Hit it with a hammer” approach

To be fair, the problem of connected-components of declaration/definition when migrating between two vocabulary types may be mitigated by a couple of practical concerns:

- A language like C++ can add an overload set for utility code to break the most-common nodes in the graph. (For example, changing `absl::StrCat` to work on both `string` and `string2`.)
- A fair number of instantiations are used purely locally.

Thus although we’ve recognized that these dependency networks can make the general problem difficult or intractable (especially in cases where we are aiming for 100% conversion) we can get a significant amount of conversion with relatively little manual effort with this One Simple Trick: generate all of the changes in the normal fashion (as if there weren’t connected-API constraints), and see which shards of that change actually apply cleanly (the build succeeds, tests pass).

This approach was first popularized by Matt Kulukundis (kfm@google.com) during the migrations from `std::unordered_map` and `__gnu_cxx::hash_map` to `absl::node_hash_map`. It isn’t theoretically sound, and it burns a lot of compute resources on shards that don’t build, but

we achieved something like 70% conversion to the new type[1] with this blunt “hit it with a hammer and see if it goes through” approach. (Matt mentions this approach in his 2019 CppCon talk)

The “Compute connected components” approach

In cases where full-conversion between two types is a goal, our experience is that the “hit it with a hammer” approach will stall out at some point. The “proper” solution here is to identify (ideally minimal) connected components of definitions/declarations and generate source changes that match that connected component analysis. Since most of the modern refactoring tools (regardless of language) are compiler-based, there is the possibility of adding steps in the change generation to do this analysis directly and emit changes that are aware of / compatible with this structure.

As of this writing (Jan 2020), Google has some preliminary successful results of applying this sort of interprocedural analysis, and is using that tool to push the `absl::node_hash_map` migrations further than the previous approach allowed.

The “Infer semantics from weak types” approach

There is an important technique for separating out the usage of a semantically-distinct use case from a more general type, based on the presence of unit tests and the invariant assumption that *most code is correct*. Hyrum Wright pioneered this approach by inferring `absl::Time` and `absl::Duration` semantics from weakly typed interfaces trafficking in `int`, `double`, `time_t`, etc. If we can identify (manually) interfaces that are taking a weakly-typed value to mean a duration (or a time), we can add an overload for that interface that works in `absl::Duration`, and then identify calls to the weaker overload and convert to the stronger. Using the inherent algebra between primitives, `absl::Time`, and `absl::Duration`, this can then propagate outward through the codebase to convert various ints and doubles to the stronger-typed forms. (Clearly the same applies to `chrono` types.)

Preliminary investigations have suggested that similar approaches can help propagate stronger types and better semantic constraints in other forms, like separating owned and unowned `T*` parameters, moving to clear ownership via `std::unique_ptr` at scale. We believe we can remove 99%+ of owning `T*` from our codebase in 2020.

This does not make these migrations / vocabulary type separations cheap or free, but it does allow for certain forms of automation and propagation. For more on this approach, see Hyrum’s talk “Applying Gradual Typing to Time Types.” Manual effort is still required to identify API seeds and introduce appropriate overloads. We still have the same sorts of ecosystem friction for

[1] Measured by spelling count: the number of times these types are instantiated in the codebase. (As opposed to resource consumption in production or other tracking measurements.)

having multiple ways to express the same concept. But it does provide some prior art for reasoning about these sorts of conversions. Interestingly, this really only works for vocabulary types: the same interprocedural dependency network that makes it hard to change vocabulary types in an unstructured/unsequenced way gives us exactly the necessary knowledge to perform this sort of migration.

This approach does not help break the “connected component” issue above. It provides a means and mechanisms for splitting one vocabulary with multiple meanings into multiple vocabulary types each with a single meaning. Consider that sometimes an `int64` is a number of seconds, sometimes it is a number of seconds since the unix epoch, and sometimes it has nothing to do with time at all.

Conclusions

Types are harder to change than functions, because the temporal nature of an object adds a time-dimension to reasoning. This almost always makes reasoning about invariants more challenging. Vocabulary types are particularly difficult because of the pervasive interconnected nature of their declaration/definition dependency networks. Simple migrations between two vocabulary types can be quite challenging, even with our growing experience and toolkit.

The presence of multiple conceptually-overlapping vocabulary types in an ecosystem also tends to produce a significant productivity and computational friction. Converting back and forth between semantically-similar types has a cognitive and computational cost. Language-level issues also add significant costs, especially in the form of ambiguous conversions and mismatched special-handling for these types. Every type conversion or specialization for one vocabulary type will have to be manually managed when introducing a similar vocabulary type to the ecosystem. Any semantic gap between the two will be especially expensive and manual to manage.

These costs need to be explicitly addressed when we are dealing with proposals to add new vocabulary types. Duplicating the base conceptual functionality of `string` or `unordered_map` may allow for backward compatibility, but evolution-via-addition comes with (potentially significant) ecosystem costs. Any significant discussion of duplication, be that in vocabulary type or adding new versions to the standard library, needs to be balanced with an understanding of these costs.