Incremental Type Migration Using Type Algebra

Hyrum K. Wright
Google
Pittsburgh, Pennsylvania 15237
hwright@google.com

***Abstract*—When a new set of types is introduced into a large software system, the task of retrofitting that system to take advantage of those types is generally cost-prohibitive. Existing automated type migration systems assume a one-to-one correspondence between old types and new types, making them unable to do partial migrations to more complex type sets. By using compiler-based migration tooling and an algebraically modeled type set, it is possible to use a small amount of manually seeded type information to automatically propagate the new types through a large monolithic C++ codebase. We demonstrate this technique by automatically generating 20k+ individual changes across a large industrial codebase of millions of lines of C++.**

***Index Terms*—type inference, refactoring, software maintenance**

I. Introduction

Strong types help software engineers by constraining the actions a well-formed program can perform [1]. When new sets of types are introduced, engineers can use them to write more clearly specified programs, avoiding bugs at compile-time instead of depending on testing or runtime discovery [2]. Existing legacy systems do not immediately benefit from the improved constraints of new types. For example, existing software may use an integer type to represent both position and velocity, but a new set of types may represent the two concepts as separate types with a constrained set of operations between them to improve program correctness.

Migrating existing systems to new types is a process known as *type migration*. Many tools exist which allow for automated single type migration across limited sets of code. This model of type migration assumes that new types are directly mappable to old ones, i.e., that the relationship between them is one-to-one. With more complex sets of types, that assumption no longer holds. In the above example, position and velocity would be modeled as two different types, and any type migration would need to infer the context in which a given integer is semantically meaningful as a velocity or a distance, or neither. The tool would also need to account for other numeric types of inputs such as floating point values.

Recently Google adopted types to more correctly model the concepts of time instants and intervals within our C++ codebase. These types are based on the mathematical model of time instants as points in a one-dimensional affine space, and time intervals as vectors in that space. Using this algebraic model of Time types, we implemented automated tools which can infer semantics of types from existing program syntax. This technique has helped engineers use the types correctly in new code, but also allows us to update existing legacy code across our entire 250M lines of C++.

In this paper, we present a case study of our experiences in doing this iterative large-scale type migration from numeric types to this newly introduced set of Time types, using the relationships between different variables to infer and propagate type information. From a small amount of manually placed initial information, known as a *seed*, we show how type information spreads to additional variables, *not* just linearly from caller to callee through a function call hierarchy. While this technique relies upon the assumption that the existing code is semantically correct, it also broadens the scope of traditional type migrations, and scales to large codebases.

We implemented our approach as a collection of open-source tools using the `clang-tidy` analysis and migration framework on top of the clang project's `LibTooling` infrastructure [3]. Using these tools, we have successfully applied over 20k separate changes to Google's corpus of 250M lines of production C++ code, and found dozens of latent bugs in the process.

Our goals when beginning this work were to explore the feasibility of applying partial type migrations to our large C++ codebase, and to demonstrate that iterative gradual type inferences are reasonable in a statically typed language. Concretely, we also wanted to make our existing C++ codebase more type safe when using time-related constructs, find existing defects in our codebase and prevent future ones by making code easier to write. We hope sharing our experience doing these transformations at scale in an industrial setting, including the limitations we encountered, is broadly useful.

II. Motivating Example

Throughout this paper, we use the example of the Abseil Time library, which uses the `absl::Time` and `absl::Duration` types to represent a time instant and interval, respectively. Section III-A discusses this library in detail. To illustate why this library is useful, consider the legacy code example in Listing 1.

Listing 1 shows interfaces and a function which use both time instants and time intervals, as well as several operations between them. The code in Listing 1a uses floating point numbers to represent both time instants and time intervals. In the case of the time instant, this value represents units of time since some zero point, measured in seconds. In the case of the time interval, the value represents some amount of time in seconds. In both of the cases shown in Listing 1a, both the

2576-3148/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSME46990.2020.00085

```
// Set the deadline to be `time` seconds from now.
void set_deadline(double time);

// Set the timeout at the timestamp `time`.
void set_timeout(double time);

double deadline();

double adjust(double offset_seconds) {
  double d =
    std::min(5, deadline()) - offset_seconds;

  set_deadline(d);
  return time(nullptr) + d;
}
```

(a) Example using numeric time types

```
void set_deadline(absl::Duration time);
void set_timeout(absl::Time time);
absl::Duration deadline();

absl::Time adjust(absl::Duration offset)) {
  absl::Duration d =
    std::min(absl::Seconds(5), deadline()) - offset;

  set_deadline(d);
  return absl::Now() + d;
}
```

(b) Example using strong time types

Listing 1: Comparison of numeric-typed time information versus strongly-typed time information

kind of the variable, as well as the scale of its values (i.e., whether they are measured in seconds, milliseconds or hours) is not part of the underlying type, and is communicated out-of-band, either through documentation or variable naming.

Generic numeric types like `double` or `int` have fewer constraints than the strongly-typed `absl::Time` and `absl::Duration`. Because the meaning of these numeric types is encoded by convention, and their constraints are not enforced by the type system, programmers can easily provide the wrong scale, or in some cases the wrong type (e.g., an instant vs. an interval), when assigning values to variables or function parameters. This is particularly easy when working with ambiguous interfaces which lack a clear convention within a large software system, or when values are propagated through several interfaces before being used. Thorough testing helps reduce, but not eliminate, these kinds of bugs.

In contrast, using stronger types such as `absl::Time` and `absl::Duration` to represent time instants and intervals encodes some of this information in the type which makes writing correct programs easier, as in Listing 1b. Using these types, programmers can clearly communicate intent by constraining the meaning of a program's data and operations. This also eliminates the need to specify the type and scale of a variable or parameter out-of-band.

Stronger types also improve implementations as well as interfaces. The implementation of `adjust` in Listing 1a, lacks strong type information compared to the updated implementation in 1b.These stronger types make updating an unfamiliar implementation easier, as the type system constrains the kinds of edits that can be made.

While these properties are useful for new code, existing code in our codebase still uses the old types, limiting the benefits of the new types by requiring conversions whenever new code interacts with old systems. Additionally, engineers often reference old code when writing new code [4], thus perpetuating obsolete techniques. By using tools which take advantage of the underlying type algebra, automatically migrating code to stronger types is feasible.

Changing a single function's implementation to use the improved types may be reasonable for an individual programmer, but for codebases with millions of lines of existing code, manual upgrades are too costly to be feasible. Instead, if we want to update legacy systems, we must use automation to make the same kinds of transformations which a human programmer would do. As we show in Section III, the `absl::Time` and `absl::Duration` types are modeled on a one-dimensional affine space. We can use this model and the strong relationship between these types, along with compiler-based tooling, to automatically perform transformations of the kind shown in Listing 1b across a very large code base.

III. Time Type Set

A *type set* is simply a collection of related types. While any collection of types satisfies this broad definition, to be useful for migration purposes, a type set should embody a specific mathematical model which defines the valid operations between types in the set. A simple example is C++ pointers and integers. The C++ standard defines arithmetic operations between pointer types and integers, modeled on the underlying numeric operations, but some operations, such as addition of two pointers, are undefined.

A. Abseil Time Library

The Abseil Time library defines the set of types for time instants and intervals as `absl::Time` and `absl::Duration`, respectively. The library also defines operators which implement the defined operations in a type-safe way, and omits those for which the operations are not defined by the underlying algebra. The result of this increased type safety is a natural way to do computations on time intervals and instants, and fewer runtime bugs due to compile-time enforcement. A complete description of the Abseil Time Library can be found in its public repository [5].

To assist with migrations from existing codebases, and to provide interoperability with legacy systems, Abseil Time also provides conversion functions to and from the `absl::Time` and `absl::Duration` types. A selection of these functions is listed in Table I. These functions allow a system to incrementally migrate to the new types, which is important when doing non-atomic refactoring over a large system [6].

These functions operate over six different fixed-length scales: hours, minutes, seconds, milliseconds, microseconds and nanoseconds. For brevity, we will omit hours, microseconds and nanoseconds from most of our examples, but migration techniques for them are analogous to the other scales.

```
absl::Duration absl::Minutes(double);
absl::Duration absl::Seconds(double);
absl::Duration absl::Milliseconds(double);
```

(a) `absl::Duration` factory functions: convert numeric values to `absl::Duration` value at the given scale.

```
double absl::ToDoubleMinutes(absl::Duration);
double absl::ToDoubleSeconds(absl::Duration);
double absl::ToDoubleMilliseconds(absl::Duration);
```

(b) `absl::Duration` conversion functions: convert an `absl::Duration` to the indicated scale.

```
absl::Time absl::FromUnixMinutes(int64_t);
absl::Time absl::FromUnixSeconds(int64_t);
absl::Time absl::FromUnixMillis(int64_t);
```

(c) `absl::Time` factory functions: convert a numeric value at the given scale since the Unix epoch to an `absl::Time` representing that time instant.

```
int64_t absl::ToUnixMinutes(absl::Time);
int64_t absl::ToUnixSeconds(absl::Time);
int64_t absl::ToUnixMillis(absl::Time);
```

(d) `absl::Time` conversion functions: take an `absl::Time` and return a numeric value at the given scale since the Unix epoch.

TABLE I: Selections from the Abseil Time API

Expression	Result
`Time + Duration`	`Time`
`Time - Duration`	`Time`
`Time + Time`	*Undefined*
`Time - Time`	`Duration`
`Duration + Duration`	`Duration`
`Duration - Duration`	`Duration`
`Duration + Time`	`Time`
`Duration - Time`	*Undefined*

TABLE II: Addition and Subtraction operations for Abseil Time types

B. Time Algebra

Throughout this paper, we have discussed two types that represent two distinct, yet related concepts: that of a time *instant* and a time *interval*. Mathematically, these are part of a one-dimensional affine space, with time instants as points in that space and time intervals as vectors in that space. The Abseil Time library defines the various operations which are valid with and between and within this type set, summarized in Table II. It is these algebraic relationships which we later use to do type inference as part of our type migration.

This mathematical relationship implies operations beyond addition and subtraction. For example, multiplication and division by scalars are defined for `absl::Duration`, but undefined for `absl::Time`. Because time intervals and time instants are ordered, relational comparisons between like types are also defined. `absl::Duration` factory functions also distribute over mathematical operations and other C++ language constructs. Examples of some additional operations are summarized in Table III.

With enough existing type information, we can use this type algebra to deduce additional type information about the surrounding code. For example, if we know the result of an addition expression is an `absl::Time`, and that its first operand is also an `absl::Time`, we can deduce that the second operand is semantically a time interval, regardless of its declared type. We will explore the implications of these kinds of deductions in Section IV.

Expression	Result
`Duration * Scalar`	`Duration`
`Duration / Scalar`	`Duration`
`Duration * Duration`	`Scalar`
`Time <=> Time`	`bool`
`Duration <=> Scalar`	`bool`
`absl::Seconds(b ? x : y)`	`b ? absl::Seconds(x) : absl::Seconds(y)`

TABLE III: Additional operations for Abseil Time types

C. Compiler-Based Transformations

Our tooling uses compiler-based techniques to scalably match patterns found in a program's abstract syntax tree (AST) and generate proposed changes. This process can be efficiently parallelized to run at scale across many machines, resulting in full-system analyses which take only tens of minutes across millions of lines of code. For C++, we use a collection of tools built on the `clang-tidy` static analysis infrastructure, which can be parallelized using the ClangMR architecture [7].

Using Clang's AST matcher library allows us to efficiently match specific patterns of nodes in the AST. After a node is matched, we transform its text, using semantic information about that node and its surroundings. This technique becomes powerful when doing type transformations, since we can look for specific expressions and then examine their context to determine what the appropriate transformation should be. For example, if a variable's type is being changed, its references must also be changed, but how to do so will depend on whether it is being used as an lvalue or an rvalue—information which is available in AST-based tools, but not in text-based ones.

These tools are implemented as part of the `clang-tidy` infrastructure and can be run manually by engineers over specific changes, as well as part of a broader static analysis pipeline, such as the Tricorder framework during code review [8]. Surfacing these fixes at review time helps prevent new usage of old patterns from creeping back into our codebase, and that ensures good patterns are applied moving forward. Our set of `clang-tidy` checks for the Abseil Time library is open source and available as part of the large `clang-tidy` tooling suite [9].

IV. Transformation Process

We now describe the process we use to transform numeric types, such as `int` and `double`, to the Abseil Time types `absl::Time` and `absl::Duration`. Because this is *not* a one-to-one mapping, we use the algebra described in Section III-B to infer semantics from the existing syntax and perform *partial* type migration. As these tools are applied iteratively, they spread an initial small amount of information about the time type set across millions of line of code.

Transformation	Purpose
Expression-based Transformations	
DurationComparison	Convert comparisons to the `absl::Duration` domain.
TimeComparison	Convert comparisons to the `absl::Time` domain.
Subtraction	Deduce type information from subtraction expressions
Addition	Deduce type information from addition expressions
Variable Transformations	
DurationLocalVariable	Convert variables to `absl::Duration`
TimeLocalVariable	Convert variables to `absl::Time`
DurationClassVariable	Convert class members to `absl::Duration`
TimeClassVariable	Convert class members to `absl::Time`
Interfunction Transformations	
DurationReturn	Convert return type to `absl::Duration`
TimeReturn	Convert return type to `absl::Time`
DurationParameter	Convert parameters to `absl::Duration`
TimeParameter	Convert parameters to `absl::Time`
DurationOverload	Convert arguments at call sites to `absl::Duration`
DurationOverload	Convert arguments at call sites to `absl::Time`

TABLE IV: Abseil Time transformations

A. Symbol Names

First, we note a tempting, but futile avenue for deducing time type information: symbol names. Many functions and variables hint at their time type semantics through their names.[1] Function names such as `DeadlineSeconds` and `GetSecondsSinceYesterday`, as well as variable names such as `deadline_sec` may seem to indicate that the object in question could be unconditionally converted to a `absl::Duration`, regardless of its surrounding context.

We have discovered that such transformations, however tempting, are not advisable, because a function or variable name is not sufficient to unambiguously determine whether the value in question is a time interval or time instant. Making the wrong transformation is often worse than making no transformation at all, since the iterative nature of the transformation process means that such a mistake would propagate through the entire software system. In these cases, we wait until more context is available to determine the correct transformation.

B. Transformation Categories

The time algebra in Section III-B, allows us to enumerate the different possible transformations we can perform using only partial type information. We show why they are equivalent to expressions using integers, and then show the limitations of these transforms. A list of various transformations and their purposes is shown in Table IV, and each one is described in detail within the remainder of this section.

1) Expression-based Transformations: We refer to the first class of transformation functions as *expression-based transformations* because they use information available in a single expression to propagate type information. Since expressions often contain references to multiple variables, these transformations help deduce information about related variables in an expression, providing a mechanism to spread type information beyond the variables originally declared with a strong type.

[1]Google's style guide constrains these names to a particular form, giving some degree of parsability.

```
double deadline_seconds;
if (absl::ToDoubleSeconds(duration) >
  deadline_seconds) ...
```

(a) Before transformation

```
double deadline_seconds;
if (duration > absl::Seconds(deadline_seconds)) ...
```

(b) After *DurationComparison* has been applied

Listing 2: *DurationComparison* transformation example

a) Comparison: When a time interval is compared against some other value, we infer that that value must also be a time interval, and then deduce the scale of the interval from the scale of the conversion function used. We call this transformation *DurationComparison*, and apply it in any boolean context. An example is shown in Listing 2.

It is more semantically correct to perform comparisons in the `absl::Duration` domain both because it is more precise—having defined behavior for overflow and saturation—and better matches the semantic intent of the code. We can later use the result in Listing 2 to infer information about the `deadline_seconds` variable, a fact we use in Section IV-B2.

While we use `absl::Duration` in this example, the same principle applies for automated transformations to `absl::Time` comparisons, which we call *TimeComparison*. And while we use the seconds scale in our example, the tooling for this and other examples works for all of the time scales Abseil Time supports.

b) Subtraction and Addition: From Table II, we note that subtraction is defined for several combinations of `absl::Duration` and `absl::Time` types. Using the partial type information available in a subtraction expression, we infer information about the other members of the expression. For example, the subtraction expression in Listing 3a has an `absl::Duration` result, and the first operand is converted from an `absl::Time`. Using the type information from Table II, we infer that the second operand must also represent a time instant. From this inference, we apply a transformation which performs the subtraction operation using Abseil Time types natively and avoids the final conversion of the entire expression. We call this transformation *Subtraction*, and the result is shown in Listing 3b.

It is important to note that just knowing the type of the first operand in the above example is not sufficient to do a proper transformation: we cannot infer from Table II what the second operand's type is without knowing the result (or the result's type without knowing the second operand's). However, in the case where we know the second operand is an `absl::Time`, we have sufficient information to infer the types of both

```
int x;
absl::Time t;
absl::Duration d = absl::Seconds(
  absl::ToUnixSeconds(t) - x);
```

(a) Before transformation

```
int x;
absl::Time t;
absl::Duration d = t - absl::FromUnixSeconds(x);
```

(b) After *Subtraction* has been applied

Listing 3: *Subtraction* transformation example

```
absl::Duration dur;
int number_of_milliseconds =
  absl::ToInt64Milliseconds(dur);
int number_of_minutes = absl::ToInt64Minutes(dur);

absl::Seconds(dur / absl::Seconds(1));
absl::Seconds(absl::ToInt64Seconds(dur));
absl::Seconds(absl::ToDoubleSeconds(dur));
absl::Seconds(number_of_milliseconds / 1000.0);
absl::Seconds(number_of_minutes * 60);
```

Listing 4: Complex `absl::Duration` expressions

the first operand and the result, because subtraction of an `absl::Time` from an `absl::Duration` is undefined.

The *Addition* transformation infers information in a similar manner to the *Subtraction* transformation. Much like *DurationComparison* and *TimeComparison*, these transformations allow type information to be propagated between separate variables or subexpressions, which is crucial to the overall goal of broad type propagation through a software system.

c) Local Canonicalizations: The final set of expression-based transformations performs local canonicalizations. While these do not have a direct role in propagating type information, they simplify existing `absl::Time` and `absl::Duration` expressions to enable the AST-based pattern matchers to be simpler and to match a more comprehensive set of candidates. Each of the expressions shown in Listing 4 can be simplified, so rather than handle all of these cases in each transformation tool, we instead use separate tools to homogenize the codebase to the simplest possible expression.

2) Local Variable Transformations: After applying the above expression-based transformations, we can then expand our scope to look at changes which span expressions within the same function. These transformations fall into a single category: local variable type changes.

Consider the function in Listing 5a—which may have been the result of a previous application of *DurationComparison*. We infer from the use of `absl::Seconds` in line 3 that the variable `x` is interpreted as a number of seconds. Using this information, we change the type of `x`, and update all its references with appropriate conversion functions to maintain existing semantics of the function, as shown in Listing 5b.

We call this transformation *DurationLocalVariable*, with an analog of *TimeDurationVariable* for the `absl::Time` domain. In general, we identify candidate variables in two ways.

```
void func(absl::Duration d) {
  int x = get_deadline();
  if (d < absl::Seconds(x)) ...

  x += 60;
  x *= 2;
  int y = x;
}
```

(a) Before transformation

```
void func(absl::Duration d) {
  absl::Duration x = absl::Seconds(get_deadline());
  if (d < x) ...

  x += absl::Seconds(60);
  x *= 2;
  int y = absl::ToInt64Seconds(x);
}
```

(b) After *DurationLocalVariable* has been applied

Listing 5: *DurationLocalVariable* transformation example

First, we look for variables which are initialized or assigned to by calls to `absl::Duration` conversion functions, such as `absl::ToDoubleSeconds`. Second, we look for variables which are used as arguments to `absl::Duration` factory functions, as is the case with `x` in Listing 5a. Finally, we prune the candidate list by eliminating variables for which the address is taken, because the tooling is not yet robust enough to handle transformations for pointers, only values.

This identification method is conservative, because it only finds variables which are currently being used in an Abseil Time context. We prefer this conservative approach because it avoids making the wrong inference, and possibly propagating the wrong time information further through the program.

DurationLocalVariable is an example of how AST-based tooling makes type set migration possible. The context in the AST gives information about how each reference to a variable should be transformed. For example, variable references which are the result of an assignment should apply a conversion function to the expression they are being assigned to, whereas references used as an rvalue should themselves be wrapped in a conversion function. Certain parts of the language, such as lambda capture lists, remain unchanged.

We discovered early in our pilot process that human reviewers preferred certain variable name changes in conjunction with the type change. Accordingly, the *DurationLocalVariable* tool updates variable names in cases where it can infer that the name was being used to denote a certain scale. Thus `deadline_seconds` becomes `deadline`, but `sleep_interval` remains unchanged. The result is code which looks more natural to a human reader.

3) Interfunction Transformations: The combination of expression-based transformations and local variable transformations can push type information to the boundaries of a function: its inputs as parameters and its outputs as return values. From here, we can continue pushing type information through our system by identifying and changing function parameter

```
void f1(double deadline_seconds) {
  double my_deadline = deadline_seconds + 5.0;
}

void f2() {
  absl::Duration dur = absl::Seconds(3);
  f1(absl::ToDoubleSeconds(dur));
}
```

(a) Before transformation

```
void f1(absl::Duration deadline) {
  double my_deadline = absl::ToDoubleSeconds(
      deadline) + 5.0;
}

void f2() {
  absl::Duration dur = absl::Seconds(3);
  f1(dur);
}
```

(b) After *DurationParameter* has been applied

Listing 6: *DurationParameter* transformation example

types and return types, along with the associated callers, using traditional data flow and type inference techniques.

a) Function Parameters: We can identify function parameter candidates for transformation much the same way that we can identify local variables: looking for parameters which are initialized or assigned to by calls to `absl::Duration` conversion functions within a function, or finding parameters which are used as arguments to `absl::Duration` factory functions. It is also possible to use information *external* to a function to identify parameter migration candidates: if a function is called with arguments which are themselves conversions from an `absl::Duration`, we know that the argument can be migrated. An example is given in Listing 6.

This type of transformation is possible for both `absl::Duration` and `absl::Time` types, and we call the respective transformations *DurationParameter* and *TimeParameter*. Changing the type of a function parameter spreads type information across function boundaries, and more importantly to other callers of the same function which might be in completely unrelated contexts.

The size of our codebase prohibits changing all callers to all functions atomically. In cases where we cannot show that all callers can be changed at the same time as the function being updated, we add an appropriate overload in one change, and then apply a separate transformation, *DurationOverload* (or *TimeOverload* for `absl::Time` parameters) to update callers to use the new overload in subsequent changes.

Adding separate transformations to only migrate callers to a new overload has an additional advantage: we can seed high-caller functions by manually adding new overloads and then run the standard transformation to migrate their callers. In our case study, we seeded just 10 functions, which collectively had tens of thousands of callers, and then allowed that information to inform the remaining transformations.

b) Return Types: As with function parameters, we can identify return type migration candidates using information

```
int get_future() {
  return absl::FromUnixSeconds(absl::Now() + absl::
      Seconds(5));
}

void func() {
  int future = get_future();
}
```

(a) Before transformation

```
absl::Time get_future() {
  return absl::Now() + absl::Seconds(5);
}

void func() {
  int future = absl::ToUnixSeconds(get_future());
}
```

(b) After *TimeReturn* has been applied

Listing 7: *TimeReturn* transformation example

```
class Fuzz {
 private:
  int delay_ms;

  void func() { set_deadline(absl::Milliseconds(
      delay_ms)); }
};
```

(a) Before transformation

```
class Fuzz {
 private:
  absl::Duration delay;

  void func() { set_deadline(delay); }
};
```

(b) After *DurationClassVariable* has been applied

Listing 8: *DurationClassVariable* transformation example

from both inside and outside a function. If a function's return value is converted from an `absl::Duration` inside the function's return statement, or if the value being returned from a function is converted to an `absl::Duration` at the callsite, then the function is a reasonable candidate for the *DurationReturn* transformation (or *TimeReturn* if the value is converted from or to an `absl::Time` value). As with *DurationParameter*, changes to both the function and its callers must be made simultaneously. Listing 7 shows an example of the *TimeReturn* transformation.

c) Class Variables: The final class of supported intra-function transformations touches class variables. We limit these transformations, *DurationClassVariable* and *TimeClassVariable*, to private variables, so that we can see all of their references in a single translation unit.

As with other intrafunction transformations, changing the type of private variables allows other references to those variables to infer further information about the type characteristics of their enclosing expressions, as in Listing 8.

C. Iteration

One final note about the transformations presented in this section: we have observed that because of their constructive nature, they are best run iteratively across our codebase. For example, *DurationComparison* might find a variable which *LocalDurationVariable* can then migrate, which might then generate an expression which *Subtraction* can change. Using a regularly running static analysis framework enables the expression-based changes to be flagged during code review and allows engineers to group several transformations together into a single committed change.

The result is a suite of tools which can be run constantly across our large C++ codebase and, over time, cause this codebase to converge to a more type-safe state using Abseil Time types. This kind of iterative type transformation results in a fixed-point iteration for a given set of transformations.

V. Case Study

Google's codebase is a massive collection of code representing many different types of development patterns and usages [10]. Our codebase uses time concepts extensively to express deadlines and timeouts. In this section, we present the practical experience deploying our tools across this corpus of 250M lines of C++ code. We also explore the limitations of our techniques as discovered through this deployment process.

A. Methodology

Using the distributed analysis infrastructure mentioned in Section III-C, we ran each of the transformations described in Section IV-B across our entire corpus of C++ code. This process generated one large set of changes for each transformation spanning the entire corpus. This large change was then split along individual project boundaries for testing and code review purposes. These boundaries generally correspond to individual directories within the monolithic corpus. Each of these subchanges was then tested and reviewed like any other change to code going into our production systems.

For practical reasons in our evaluation, we limited ourselves to 50 pending simultaneous outstanding changes[2]. We also focused our efforts on specific transformations, rather than attempting to run all of them over the codebase simultaneously. Over the course of many months, this process generated thousands of individual changes which have been committed to our production C++ codebase. Table V shows a summary of the number of committed changes for each transformation.

Because we grouped changes for the same transformation together when sending them to individual teams, the results in Table V are a lower bound on the number of discrete edits our tooling has produced. These numbers also omit changes made as part of pre-review testing or automated static analysis, due to tracking deficiencies in our source control system.

The number of *DurationOverload* changes dominates the total results primarily because it was the first change to start running. Recall that to start the process we manually add overloads to high-caller functions, and then use *DurationOverload* to spread that initial type information throughout the codebase. While still a very rough estimate, the remaining changes outnumber those of *DurationOverload* by a 2:1 ratio.

[2]Since we are doing this work on live systems, this limitation prevents our automated tooling from overwhelming engineer review capacity.

Transformation	Change count
Subtraction	1779
Addition	1979
DurationComparison	839
TimeComparison	904
DurationLocalVariable	3743
TimeLocalVariable	1583
DurationPrivateVariable	1473
TimePrivateVariable	458
DurationOverload	7266
Total	20024

TABLE V: Number of changes applied by transformation

1) Correctness: We used three methods to evaluate our tools' correctness: the compiler, existing unit tests, and human inspection. As part of the review process, each generated change was run through the compiler to ensure that the transformation tooling produced syntactically valid output. When we discovered cases where it did not, we used that case as an example to refine our tooling.

After compilation, the change was then run through our unit test system, which runs not only the tests directly affected by a given change, but all tests transitively impacted by the change [11]. The result is a robust assurance that the change generated is semantic-preserving.

Finally, each change was inspected by a human reviewer as part of Google's standard code review process. More than 97% of changes were approved by reviewers without comment. Of the remaining changes, most reviewers suggested additional improvements inspired by the transformation at hand, and some were simply complimentary of the change being made. In a few instances, reviewers were concerned about the proposed change, but this almost always stemmed from unfamiliarity with the Abseil Time library itself, and not the correctness of the underlying change.

2) Performance: Because our transformation tooling operates on the AST, and ASTs from different translation units are independent, we can parallelize our analysis across translation units. Using an analysis system such as MapReduce [12], and sharding our analysis across thousands of machines, it is possible to analyze our 250M line C++ corpus in less than an hour. The dominant factor in getting each independent change submitted to the codebase was not the analysis step, but the time spent in testing and human review.

We continue to run the automated tools across our codebase to catch newly-added instances of transformation candidates. Anecdotally, it is not uncommon for recipients of an automated change to make further manual improvements in the area around the automated transformation, rather than wait for future transformations to make those changes. These improvements often preempt cases in which the output of one transformation produces an input pattern to a subsequent one,

```
void func(double wait_seconds);

void gunc() {
  double wait_milliseconds = ...
  func(wait_milliseconds);
}
```

(a) Calling a function a numeric parameter with incorrect scale

```
void func(absl::Duration wait);

void gunc(double wait_milliseconds) {
  double wait_milliseconds = ...
  func(absl::Seconds(wait_milliseconds));
}
```

(b) After *DurationOverload* has been applied

Listing 9: Example of an incorrect time interval scale

so those are not counted in Table V. We expect the work to be complete when the transformations reach a fixed point.

B. Defects Found

One of the aims of this work is to find and fix software defects resulting from using numeric types to hold time values. We found a number of these defects, though the rate of occurrence was less than we expected, around twenty confirmed cases, which represents a defect rate of only 0.13%. We expect that due to a robust testing culture, semantic mismatch bugs tend to be resolved quickly, and not linger for this type of analysis to find. The low number of existing defects does not diminish the value of performing these changes: engineers within Google report that the more explicit type set makes future development less error prone, though we have not measured the numbers of prevented bugs. Here we highlight two common defect modes we found, and which improved types help prevent: incorrect scale, and incorrect type.

1) Incorrect Scale: One bug pattern which emerged from this work is that of passing a value with the incorrect scale as a parameter to a function (see Listing 9). The function `func` takes a numeric value which it interprets as some number of seconds, but the calling function passes a value which is scaled in milliseconds. This means the value provided to `func` is a thousand times larger than the engineer likely intended.

After performing the *DurationArgument* transformation, the argument to `func` is wrapped in an `absl::Duration` factory for an explicit scale. This does not fix the bug, because transformations are intended to be behavior preserving, but it does make the bug much more obvious, making it likely that reviewers will fix the bug in a subsequent change.

2) Incorrect Type: The other kind of defect we have encountered is the substitution of a time interval for a time instant, or vice versa. For example, the code in Listing 10 incorrectly passes an integer representing the number of seconds since the Unix epoch as a time interval. The caller is inadvertently providing an interval on the order of many years, rather than a few seconds as is probably intended.

After several transformations, the bug is much more obvious to a reader of the code, who can then independently fix

```
void func(int wait_seconds);

void gunc() {
  int wait_until = time(nullptr) + 5;
  func(wait_until);
}
```

(a) Calling a function with a interval parameter with an instant value

```
void func(absl::Duration wait_seconds);

void gunc() {
  absl::Duration wait_until =
    absl::Seconds(time(nullptr) + 5);
  func(wait_until);
}
```

(b) After *DurationOverload* has been applied

Listing 10: Example of an incorrect time type

it. The reason for maintaining semantic equivalence and not fixing these bugs when they are discovered comes from the realization that seemingly minor "fixes" like these can often lead to unexpected effects in broader systems, and those effects should be separated from the transformation itself.

C. Limitations

In spite of the power of our existing library of transformations, there are several instances where we cannot or do not automatically convert variables or expressions to further propagate type information, which we outline below.

1) Conservative Transformations: We intentionally limit the scope of our transformations to those which we can demonstrate maintain functional and semantic equivalence. This means that we miss some transformation opportunities, rather than speculate on the result of a transformation and potentially change the meaning of a program. Our experience with complex software systems indicates that it is hard to show that any change is perfectly safe, so we limit ourselves to changes for which we have a high degree of confidence.

In practice, this means rather than changing every variable in a transformation such as *DurationLocalVariable*, we only change variables with names which indicate they are likely to be time intervals, such as `waiting_seconds` or `update_interval`. This limitation means we may not fully propagate as much information as we would like.

The nature of the Clang AST matcher library imposes another constraint. Because the library matches on very specific constructs of the abstract syntax tree, it misses cases where programs use non-conventional syntax in specifying `absl::Duration` values, such as by casting. The local canonicalization transformations help by making these constructs more homogeneous, but ultimately some cases will be missed because of non-matching by the AST-based tooling.

Finally, while we support the Abseil Time library across the entire C++ codebase, we have discovered that, individual teams have adopted custom types for time instants and intervals. These relatively rare cases, rarely extend past a single team or subsystem, so we just ignore them.

2) Information Deficiencies: Lack of information about some expressions also limits our current process. In Section III-B, we mentioned that multiplying and dividing by scalar values is a supported operation for `absl::Duration` values, yet we do not automatically distribute multiplication or division across `absl::Duration` expressions. In theory this transformation should produce an equivalent result, but in practice, many scalar values have additional semantic meaning, much like pre-Abseil Time integers do.

Consider scalars which represent a rate: when multiplied by an `absl::Duration`, they do not yield a time interval, but different semantic value instead. Listing 11 shows such an example.

```
absl::Duration d = ...
int frames_per_seconds = ...
int frames = absl::ToInt64Seconds(d) *
  frames_per_second;
```

Listing 11: Nontransformable multiplication by a scalar

In this case, we do not want to move the multiplication inside the call to `absl::ToInt64Seconds`, because the resulting argument is not semantically a time interval, but a number of frames. Being overly aggressive when transforming multiplications may have the effect of making future type changes more difficult, so we currently omit them.

System boundaries also represent a type of information deficiency. Because the C++ software in our corpus eventually interacts with systems not written in C++, and those systems do not share a common representation of time information, they represent a hard limit on how far we are able to push our type propagation. These system inputs and outputs may be in the form of flags, configuration values, user input, or a remote procedure call serialization framework.

3) Overridden Functions: While *TimeParameter* and *DurationParameter* work for both free and class member functions and constructors, they do not work for virtual functions. Function overrides and their base class definitions need to be updated in concert, and because we are primarily focused on single-translation-unit transformations, we avoid virtual functions and their overrides. This is a practical limitation, not a theoretical one, and could be improved in the future.

4) Templates and Macros: C++ templates and macros provide a mechanism for writing generic code. Macros are not part of a program's AST, and patterns found inside a macro definition often confuse the `clang-tidy`-based tools. Templates also present additional tooling challenges. Fortunately, these kinds of tooling failures result in compile-time errors which can be caught early in the process. In reality, we found fewer than 10 such failures across our entire codebase, and opted to make these changes by hand, rather than implement workarounds for them in our tooling.

VI. Related Work

This case study builds upon two primary areas of work: type inference and type migration. The tools developed for our case study combine both of these techniques to migrate types in more complex ways than traditional type migration tools. Our work builds upon earlier theoretical efforts by producing changes which are tested, reviewed and submitted into a production corpus, rather than just identifying cases where type mismatches may occur, as in traditional static type analysis. Our work also differs because it allows many-to-many type inference and migration, rather than the one-to-one limitations of existing work.

A. Type Inference

The area of type inference is not new [13], and remains a subject of active research, particularly for dynamically typed languages. For example, Ruby [14], Javascript [15], [16], and Python [17], [18] have all seen a significant work in inferring types to help improve program correctness. There has been much less work for more strongly typed languages, such as C++. Previous work on dynamic languages is usually limited to identifying problems during static analysis, rather than making permanent edits to the program. They also follow the type system linearly, rather than using algebraic underpinnings of a type set to spread time information both vertically through a function call hierarchy and horizontally through expressions.

Our tools build upon earlier efforts in the area of gradual typing [19], [20]. While most theoretical efforts in this area, require complete soundness, our practical experience indicates that we can rely upon "defense in depth" to catch deficiencies in our tooling. Most gradual typing techniques have been applied to languages more amenable to the approach, such as functional languages [21], rather than a more common industrial language like C++. While completely sound gradual typing may face significant theoretical challenges [22], in practice complete soundness is not required for usefulness.

B. Type Migration

Researchers have extensively examined the process of migrating from one type to another, primarily in the context of library upgrades and migration [23], [24]. Other approaches include generating new adapters for new libraries [25] or migrating types based on existing examples [26]. Many of these approaches do not scale to large codebases, nor do they work in iteratively as described in our case study. Other work shows that type migration can scale to a larger and more complex call hierarchy, but still relies upon the ability to commit changes atomically [27].

Our work also builds on constraint-based type migrations [28], only our constraints are informed by the underlying type algebra, rather than inference. Derivative constraint-based solutions often require an existing type-correct program, and while our solution requires that code be *syntactically* correct (so that the compiler-based tools can function), it works in a real-world environment where code is often not *semantically* correct (as shown by the types of defects we discovered).

Furthermore, most of the existing type migration tools target the Java language. With a different type system design, C++ presents a different set of practical considerations than Java

when doing type migration. Our work shows that it is also practical to do automatic type migration in C++ as well.

VII. Conclusion

Historically, large-scale automated migrations have focused on changing single function calls or types through dataflow analysis. Type inference has likewise been used to propagate information about existing types linearly through a codebase. In this paper we have shown that is is possible to do deeper type inference by using the algebra that models a set of types to do inference of related types, and then iteratively spread that information through a C++ codebase of 250M lines.

This work has demonstrated the feasibility of doing partial type migrations across a large codebase, and has inspired additional efforts within our codebase, such as inferring and migrating pointer ownership analysis. By being able to automatically migrate entire type sets, not just single types, we reduce the cost of adding further type safety to legacy code, and prevent bugs by making it easier for engineers to understand and use legacy systems.

Acknowledgment

The author wishes to thank Greg Miller and Bradley White for their work on designing and implementing the Abseil Time library, as well as the many reviewers of drafts of this paper.

References

[1] L. Cardelli, "Type systems," *ACM Computing Surveys*, vol. 28, no. 1, pp. 263–264, Mar. 1996. [Online]. Available: http://doi.acm.org/10.1145/234313.234418

[2] B. C. Pierce, *Types and Programming Languages*, 1st ed. The MIT Press, 2002.

[3] "Clang-tidy," https://clang.llvm.org/extra/clang-tidy/, 2019.

[4] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '09. New York, NY, USA: ACM, 2009, pp. 1589–1598. [Online]. Available: http://doi.acm.org/10.1145/1518701.1518944

[5] "Abseil time," https://github.com/abseil/abseil-cpp/tree/master/absl/time, 2019.

[6] T. Winters, "Non-atomic refactoring and software sustainability," in *Proceedings of the 2nd International Workshop on API Usage and Evolution*, ser. WAPI '18. New York, NY, USA: ACM, 2018, pp. 2–5. [Online]. Available: http://doi.acm.org/10.1145/3194793.3194794

[7] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan, "Large-scale automated refactoring using ClangMR," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 548–551. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2013.93

[8] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 598–608. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818828

[9] "Abseil time tools," https://github.com/llvm-mirror/clang-tools-extra/tree/master/clang-tidy/abseil, 2019.

[10] R. Potvin and J. Levenburg, "Why Google stores billions of lines of code in a single repository," *Communications of the ACM*, 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/2854146

[11] A. Memon, Zebao Gao, Bao Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 233–242. [Online]. Available: https://ieeexplore.ieee.org/document/7965447

[12] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[13] A. M. Tenenbaum, "Type determination for very high level languages." Ph.D. dissertation, USA, 1974, aAI7509706.

[14] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks, "Static type inference for Ruby," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 1859–1866. [Online]. Available: http://doi.acm.org/10.1145/1529282.1529700

[15] C. Anderson and P. Giannini, "Type checking for JavaScript," *Electron. Notes Theor. Comput. Sci.*, vol. 138, no. 2, pp. 37–58, Nov. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2005.09.010

[16] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi, "Type inference for static compilation of JavaScript," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: ACM, 2016, pp. 410–429. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984017

[17] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 607–618. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950343

[18] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker, "Design and evaluation of gradual typing for Python," in *Proceedings of the 10th ACM Symposium on Dynamic Languages*, ser. DLS '14. New York, NY, USA: ACM, 2014, pp. 45–56. [Online]. Available: http://doi.acm.org/10.1145/2661088.2661101

[19] J. Siek and W. Taha, "Gradual typing for objects," in *Proceedings of the 21st European Conference on Object-Oriented Programming*, ser. ECOOP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 2–27. [Online]. Available: http://dl.acm.org/citation.cfm?id=2394758.2394762

[20] A. Rastogi, A. Chaudhuri, and B. Hosmer, "The ins and outs of gradual type inference," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 481–494. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103714

[21] J. Siek and W. Taha, "Gradual typing for functional languages," 01 2006.

[22] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen, "Is sound gradual typing dead?" in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 456–468. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837630

[23] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 265–279. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094832

[24] P. Kapur, B. Cossette, and R. J. Walker, "Refactoring references for library migration," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 726–738. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869518

[25] V. L. Winter and A. Mametjanov, "Generative programming techniques for Java library migration," in *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, ser. GPCE '07. New York, NY, USA: ACM, 2007, pp. 185–196. [Online]. Available: http://doi.acm.org/10.1145/1289971.1290001

[26] Z. Xing and E. Stroulia, "API-evolution support with diff-catchup," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 818–836, Dec. 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70747

[27] A. Ketkar, A. Mesbah, D. Mazinanian, D. Dig, and E. Aftandilian, "Type migration in ultra-large-scale codebases," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 1142–1153. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00117

[28] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. De Sutter, "Refactoring using type constraints," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 3, pp. 9:1–9:47, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1961204.1961205