Misusing the Type System for Fun and Profit

Ian Dees

me@ian.dees.name

Abstract

It's amazing what a type system can do at compile time. This paper offers a peek at the C++ type system and standard libraries, which can make certain types of bugs difficult or even impossible to express in code. By brewing more robust code interfaces, engineers can save their QA teams from chasing down rare crashes, and let them focus instead on usability and performance.

Writing good software is difficult; doubly so in an unsafe language like C++. A simple off-by-one indexing error can cause code execution vulnerabilities. A couple of lines of cut-and-paste code can lead to memory leaks that slowly rob users of good performance.

Whenever an error strikes, one's reaction should be, "How can we make it harder for this bug to happen again?" Typical solutions involve more tests, or code review. These are both good things. But what if engineers could make errors disappear from a code base forever—that is, prevent them at compile time? With its support for generic and functional programming, C++ is finally becoming an interesting enough programming language to explore this style of bug prevention.

Programmers may be familiar with using smart pointers to reduce memory leaks, or with using constructs like `boost::optional` to prevent data corruption. This paper will take these familiar ideas to the next logical step, using a technique called *denotational design*.

With denotational design, as espoused by Conal Elliott, a developer can first sketch a design in a type-safe language like Idris—even mathematically proving parts of a design correct—before translating to C++. One can dramatically change a C++ API by looking at it through the lens of denotational design.

Biography

Ian Dees was first bitten by the programming bug in 1986 on a Timex Sinclair 1000, and has been having a blast in his software apprenticeship ever since. By day, Ian slings code, tests, and puns in Portland, Oregon. By night, he dons a cape and keeps watch over the city as Sidekick Man. In his (heh) "spare time," he converts espresso into programming books, including (with others) Seven More Languages in Seven Weeks*.*

1 Background

Writing correct software is hard—doubly so in a systems programming language like C++. The same language features that offer low-level control over the machine are hard to code and debug.

With a slip of the keyboard, a developer types <= instead of <, and tomorrow the application is on the front pages with a zero-day security vulnerability—buffer overflows are among the oldest classes of exploits (Litterio). An engineer forgets to check just one memory access among thousands, and the program crashes to a rude halt—or worse, silently corrupts data that will only reveal itself days later.

1.1 Traditional Advice

Pundits are eager to explain how to solve these problems. Their solutions typically fall into one of the following bits of advice:

- *Just be extra careful*. If everyone could simply never commit boundary errors, never forget to check a return value, and so on, these errors would vanish. But developers already are being careful! These errors are creeping in despite everyone's efforts.
- *Just use another language*. C++ may unsafe, the argument goes, but switching to Haskell, Rust, Go, or the next language *du jour* could eliminate these problems. Changing languages may indeed be a workable approach for a new project. But teams rarely have the luxury of starting from scratch. A typical project has legacy code, portability constraints, or performance requirements that restrict the available language choices.
- *Just test more*. What the programming language can't do, the development team pawns off on testing. Coders write unit tests to verify what they couldn't check at compile time. The QA team hopes to catch the rest of the problems with exploratory testing. These test techniques are valuable. But they do not prevent writing the errors in the first place. "You can't test quality in," as the saying goes (Mack).
- *Just review your code*. This is the one piece of advice that stands the greatest chance of preventing bugs. Even the most conservative estimate is that peer code review can catch 60% of errors (Glass). Code review is fully compatible with the techniques discussed in this paper, and can even help identify opportunities to use them.

All of these recommendations—careful coding, safe languages, testing, and peer review—are useful. But they do not all apply to legacy systems, nor do they all help prevent bugs in the first place. Prevention requires a new approach to code.

1.2 A New Approach: Lean on the Type System

With languages like C and older dialects of C++, engineers are used to thinking of types as expressing what the code can do. For example, one may say, "This function multiplies two floating-point numbers." "This data structure holds an employee ID and a name."

There is, however, another way to look at what type systems do. Not only do they describe what operations *are* allowed on a piece of data, they also express what's *not* allowed. One may not take the square root of a string, nor concatenate characters onto an integer.

By combining this "can't do" mentality with a sophisticated enough type system, one can make certain classes of errors *inexpressible*. In other words, one can move errors from runtime, where tests might catch them, to compile time. A good type system can prevent the following kinds of errors:

- *Null pointer accesses*, where a program attempts to access a memory location that doesn't exist
- *Unclear allocation policies*, which end up skipping cleanup code and leaking memory
- *Indexing errors*, where a routine may corrupt data outside a container

- *Unchecked return values*, where a coder assumes a computation succeeded without actually looking

As it turns out, mainstream systems programming now has a language and type system sophisticated enough to advance this goal.

The C++11 language and its libraries offer a number of ways to reduce or prevent certain kinds of errors. Reference types have all the memory efficiency of pointers, but can never be null. Smart pointers make memory leaks much more difficult to write. The `optional` class from the add-on Boost library can nudge a forgetful developer to check a return value.

This paper will explore similar ideas through a technique called *denotational design*. It will offer a peek at how even more advanced type systems work, and will show how to apply a few of these lessons in C++.

2 Denotational Design

In his 2009 paper "Denotational design with type class morphisms," Conal Elliott explains a technique for designing mathematically sound APIs. His core recommendation is, "When designing software, in addition to innovating in your implementations, relate them to precise and familiar semantic models" (Elliott).

In other words, he is urging architects to design their programs in two steps:

1. Sketch the design in clear, expressive notation
2. Translate this idea into the implementation language

In Elliott's paper, these two notations—the sketching and implementation languages—can be the same programming language. Indeed, he gives examples of a design that begins in a mathematically rigorous Haskell style, and ends in more idiomatic, performant Haskell.

However, one can also use two different languages for these two steps. This paper will offer a glimpse of API design improvement techniques using the following two languages:

1. A dependently typed language called Idris for the sketching language
2. C++11 for the implementation language

Much of the expressive power of Idris's type system will carry over into the C++11 translation.

2.1 Dependent Types in Idris

Idris is a Haskell-like programming language with an even more powerful type system based on the idea of *dependent types* (Brady). In Idris, functions can depend on types—a bit like C++'s template functions, but more expressive. The type system is so deep that one can even encode correctness proofs directly into source code.

A simple example may help illustrate the concept of dependent types. Here's an Idris function that takes a type as an input parameter and returns an ordinary Boolean value:

```
isNumeric : Type -> Bool
isNumeric Int    = True
isNumeric Float  = True
isNumeric String = False
--
-- and so on...
```

To use this function, the caller does not pass in an individual integer such as 42, or a string such as “hello world.” Instead, the caller passes in the `Int` or `String` types themselves. The function returns `True` if the passed-in type is a numeric type, and `False` otherwise.

This ability to “do math on types” lets developers express constraints *at compile time* on what functions are or aren’t allowed to do. One encodes these constraints directly into the function declaration. If the implementation ever violates the restrictions, the program will not compile.

For example, here’s the declaration for a function that adds together the elements from two vectors of length *k*, resulting in a third vector—also of length *k*:

```
add : Vect k a -> Vect k a -> Vect k a
```

The compiler will check that every caller correctly passes in two lists of the same length, *and* it will verify that the function implementation (not shown here) returns a list of the same length.

Idris’s type system can express much more sophisticated properties than just collection size. The following snippet from Ben Sherman declares a quicksort function that is guaranteed by the compiler to result in a list that’s a sorted permutation of its input (Sherman):

```
quicksort : TotalOrder a lte
          -> (xs : List a)
          -> (ys : List a ** (IsSorted lte ys, Permutation xs ys))
```

In a mainstream language, one would need to write several test cases to be confident in the implementation of the sort routine. In a design-by-contract language, one would have to add runtime preconditions and post-conditions all over the code. But with Idris, the *compiler* can prove that the `IsSorted` property will hold for all possible inputs.

Not all of these concepts will translate directly to C++. But stepping into a more rigorous language has other benefits, even if one can’t bring all of Idris into day-to-day work. In particular:

- Shifting notations helps clarify the essential parts of the API and avoid class bloat
- Sketching in a math-like language encourages more rigorous models than pseudo-code does
- Some of the type-checking features still apply in C++

The next section will walk through a couple of examples of API improvement where Idris can inform the design of the C++ code.

3 Taking It to (C++)11

Designing C++ APIs in dependently-typed languages is far from a theoretical exercise. In his talk “The Intellectual Ascent to Agda,” David Sankel provides practical advice for C++ implementers sketching in Agda, a language with similar power to Idris (Sankel). This paper will adapt some of Sankel’s techniques to Idris, which has more tutorial information for newcomers than Agda does.

Some examples of dependently typed code are directly expressible in C++. For example, recall the signature of the Idris function for adding two vectors of length *k*:

```
add : Vect k a -> Vect k a -> Vect k a
```

Simple constraints like collection size are fairly easy to translate. Here’s a C++ function similar to the Idris one, using the `array` template that comes with the C++11 standard library:

```
template <typename A, size_t K>
array<A, K> add(array<A, K>, array<A, K>);
```

More sophisticated concepts may not map so directly into C++. Even so, expressing an idea in Idris can still help a C++ design. This section will demonstrate this technique using a couple of fictional examples based loosely on real programming errors discovered in the wild.

3.1 Example 1: A Simple Matrix Data Type

Consider the following type representing a two-dimensional matrix of numbers, indexed by row and column:

```
template <typename T>
class Matrix
{
public:
    T operator()(size_t row, size_t col);

    // ...
};
```

This API looks straightforward enough: the caller passes in a row and column, and the method returns the value located there. But what happens when a maintenance programmer forgets that the parameter order is `(row, column)` and instead passes indices as `(x, y)`, the opposite order?

```
Matrix<double> m(3, 5);

size_t const x = 4;
size_t const y = 0;
cout << m(x, y) << endl; // wrong order!
```

Depending on how the matrix values are stored, the method could walk right off the end of an internal array and corrupt memory. The program will either crash right away, or—if the developer is unlucky—silently and mysteriously overwrite data belonging to a different part of the program. These kinds of data corruption issues are notoriously hard to track down.

3.1.1 Sketching in Idris

At this point, it's worth stepping back from the C++ code and building a mental model of what a matrix should look like. A simple Idris function declaration will help sketch the idea out.

A matrix is a two-dimensional collection of values, arranged in a specific number of rows and columns. In Idris terms, one would write a function taking two natural numbers for the dimensions, plus the type of data being kept in the matrix:

```
Matrix : Nat -> Nat -> Type -> Type
```

The matrix would provide access to individual elements through a function—call it `get`—that takes two natural numbers for the row and column and returns an element:

```
get : Nat -> Nat -> Matrix rows cols a -> a
```

This is a decent first cut. It's roughly the equivalent of the C++ version from earlier, with no range checking. But one can describe a matrix's properties more thoroughly.

For retrieving an element, the row index must be less than the total number of rows, and the column index must be less than the number of columns. One can model this behavior using a built-in Idris data type for bounded numbers, called `Fin` (for "finite").

```
get : (Fin rows) -> (Fin cols) -> Matrix rows cols a -> a
```

Now, the `get` function takes a row index bounded by `rows`, the number of rows in the matrix. A similar constraint holds for the column index.

If a caller tries to use `get` with an out-of-bounds index, their program won't pass the type checker. Rather than having to lean on code review or runtime tests (though both of these things are good!), one can ensure at compile time that we are using legal row and column indices.

3.1.2 Back to C++

How might this idea translate to C++? If C++ had a `Fin` type like Idris's, one could make the row and column sizes a part of the `Matrix` data type:

```
template <size_t R, size_t C, typename T>
class Matrix
{
public:
    T operator()(Fin<R> row, Fin<C> col);

    // ...
};
```

Then, the compiler would catch an accidental out-of-bounds use of the array:

```
Matrix<3, 5, double> m;

auto x = Fin<4>();
auto y = Fin<0>();

// error: no known conversion from 'Fin<4>' to 'Fin<3>' for 1st argument
cout << m(x, y) << endl;
```

This is a decent proof of concept. However, the C++ type system is not quite expressive enough to define `Fin` universally. (It is possible to define a limited version that works for this example.) It is time to step back a bit and rethink the data type.

3.1.3 Stepping Back from the Brink

If it's impossible to constrain the index bounds at compile time, perhaps there is another way to prevent using a row in place of a column. One approach might be to define `Row` and `Col` as two different types:

```
data Row a = aRow a
data Col a = aCol a
```

These declarations define two new types, plus constructors `aRow` and `aCol` to create values of those types. Here's how one might adapt the `get` function to use them:

```
get : Row (Fin rows) -> Col (Fin cols) -> Matrix rows cols a -> a
```

Now, the type checker will notice if a caller tries to pass a row index for a column. This design, as it turns out, is implementable in C++. First, a simple class, Row, wraps an integer in a type-safe way:

```
class Row
{
public:
    explicit Row(size_t value) : value_(value) {}
    operator size_t() { return value_; }

private:
    size_t value_;
};
```

Because the constructor is marked as `explicit`, it's impossible to use a raw integer by accident in place of a row. The `Col` class will be nearly identical; it makes sense to abstract these two types into a template:

```
enum class Dimension
{
    Row,
    Col
};

template<typename T, Dimension D>
class Wrapper
{
public:
    explicit Wrapper(size_t value) : value_(value) {}
    operator size_t() { return value_; }

private:
    size_t value_;
};

typedef Wrapper<size_t, Dimension::Row> Row;
typedef Wrapper<size_t, Dimension::Col> Col;
```

Now, the signature for the matrix indexer can take `Row` and `Col` types, just like the Idris version:

```
T operator()(Row row, Col col);
```

The compiler will catch accidental switching of the row and column:

```
// error: no known conversion from
// 'Wrapper<[...], 1>' to 'Wrapper<[...], 0>'
cout << m(x, y) << endl;
```

It will not, however, catch deliberate misuse, such as indexing a 3×5 matrix with `Row(1000000)`. Still, the new design uses the C++ type system more fully than before.

3.2 Example 2: Audio Clips

Sketching in a type-safe language is not just about preventing accidental API misuse. It can also help simplify and clarify a library.

Consider the following class that represents a clip of audio data:

```
class Audio
{
public:
    size_t count() const;

    int16_t  operator[](size_t index) const;
    int16_t& operator[](size_t index);

    double timeAtZero() const;
    double sampleRate() const;

    double verticalScale() const;

    std::string fileName() const;
    std::string comments() const;
    bool isStereo() const;

    void loadFromFile(const std::string& filename);
    void saveToFile(const std::string& filename);
};
```

This class uses a fairly common C++ style. The method names are clear, but the class is somewhat large for what it does. It mixes together unrelated concepts: element access, scaling parameters, metadata, and file I/O.

One can simplify this class slightly by moving the metadata and I/O routines elsewhere:

```
class Audio
{
public:
    size_t count() const;

    int16_t  operator[](size_t index) const;
    int16_t& operator[](size_t index);

    double timeAtZero() const;
    double sampleRate() const;

    double verticalScale() const;
};
```

The improved version still has issues, though. It exposes the internal representation of the data: 16-bit integers. The class also depends on the caller to handle conversion and scaling from integer values (indices and samples) to their real-world values (times and voltages). First, the caller would have to convert their time value into an index:

```
Audio audio;
double time  = 0.5;
size_t index = (time - audio.timeAtZero()) /
               audio.sampleRate();
```

Next, they would have to retrieve the *nth* integer sample and convert it to a real-word audio level:

```
int16_t sample = audio[index];
double  level  = static_cast<double>(sample) *
                 audio.verticalScale();
```

Doing this kind of conversion in every caller is tiresome and error-prone. Moreover, error checking (such as making sure the index value is within bounds) is left up to the caller.

One could provide these conversions as methods on the class, but doing so would make the class even larger and more difficult to understand. Instead, it's worth taking a step back and rethinking this API using Idris.

3.2.1 Sketching in Idris

To simplify this design, one can ask the question, "What is an audio clip?" It is tempting to leap straight to the representation and answer something like, "An array of evenly spaced 16-bit samples, scaled by a common factor." But those are implementation details. They don't describe what an audio clip really is from the user's perspective.

A better answer would be, "An audio level changing over time." This answer does not mention bit widths or sample rates. These are important details, but they are not relevant to the fundamental question.

If an audio clip is a changing audio level, what is an audio level? There doesn't need to be a single answer to this second question. It could be a floating-point number, a custom data type representing audio levels, or some other type.

In other words, an audio clip is a dependent type! This idea is easy to express in Idris. One could write an `Audio` function that takes an audio level type, `a`, and returns a new type:

```
Audio : (a : Type) -> Type
```

The resulting type represents the entire audio clip. It needs to provide a mapping from time values to audio levels. This mapping could simply be a function from `Float` to `a`.

```
Audio a = (Float -> a)
```

But this approach does not consider time values before and after the audio clip's contents. The API needs a way to return nothing if the time value is out of range. Fortunately, Idris provides just such a construct, called `Maybe`:

```
Audio a = (Float -> Maybe a)
```

If a caller passes in a valid time value, the result will be an audio level. Otherwise it will be a special value called `None`. Crucially, the Idris compiler will not allow callers to forget to check for `None`. Their code must explicitly consider this case.

3.2.2 Back to C++

It is possible to express the key ideas of this design in C++. The dependent `Audio` type becomes a class template, taking the audio level type as an argument:

```
template <typename T>
class Audio
{
public:
    // mapping goes here
};
```

All that's left is to provide the mapping from time to audio level. The Idris version does so with an ordinary function. A C++ class can provide a similar interface; it can act like a function call by defining the call operator.

What about the return type? The function needs to return an audio level only if the time is in range, and some kind of placeholder value otherwise. Fortunately, the add-on Boost library provides a type called `optional` that works much like Idris's `Maybe`:

```
boost::optional<T> operator()(double t);
```

Now, callers have a much simpler interface for getting the audio level for a given time:

```
double time = 0.5;
auto level = audio(time);
```

Moreover, the C++ compiler will remind callers to check the return value; if they try to use `level` directly, they will get a compilation error:

```
// error: invalid operands to binary expression
// ('boost::optional<double>' and 'double')
double result = level + 1.0;
```

Instead, callers will check that `level` has a value (by using it in a conditional statement), and then get that value by dereferencing `boost::optional`:

```
if (level) { double result = *level + 1.0; }
```

Although one could maliciously dereference the value without checking, the compiler will guard against accidentally forgetting to check.

4 Next Steps

This paper has shown how changing one's perspective helps focus on the essential parts of the problem being solved. The Idris sketches have hinted at what's possible with a notation that supports expressing thoughts in code—not just how many arguments a function has, but the relationships between those arguments.

The point is not to abandon day-to-day languages and switch to something new like Idris or Agda. The point is to bring a little rigor from strongly typed languages into one's mental models, and then return to C++ with a renewed understanding. Some of the power of dependent type systems will inevitably be left behind, but C++11 is expressive enough to preserve many of the concepts.

As a next step, readers should watch David Sankel's talk on Agda, and see how he was able to clean up several C++ APIs by modeling them in simple mathematical notation. The following questions can then guide future exploration:

1. Have you encountered a hard-to-use C++ library in your daily work?
2. Was it crash-prone if passed bad arguments?
3. Did the parameters have extra rules that weren't obvious in the function signature?

Readers could then try making these hidden constraints explicit, by describing them in a dependently typed language. What would a new and improved version of the C++ API look like?

References

Brady, Edwin, et al. “Idris: A Language with Dependent Types,” University of St. Andrews, http://www.idris-lang.org (accessed July 24, 2015).

Elliott, Conal. “Denotational design with type class morphisms.” *LambdaPix Technical Report 2009-01*.

Glass, Robert. *Facts and Fallacies of Software Engineering*. New Jersey: Addison-Wesley Professional, 2002.

Litterio, Francis. “The Internet Worm of 1998,” http://web.archive.org/web/20070520233435/http://world.std.com/~franl/worm.html (accessed July 30, 2015).

Mack, Wayne, et al. “Quality Assurance is Not Responsible for Quality,” Cunningham & Cunningham, http://c2.com/cgi/wiki?QualityAssuranceIsNotResponsibleForQuality (accessed July 24, 2015).

Sankel, David. “The Intellectual Ascent to Agda,” CppNow 2013, http://2013.cppnow.org/session/the-intellectual-ascent-to-agda (accessed July 24, 2015).

Sherman, Ben. “Quicksort in Idris,” https://github.com/bmsherman/blog/wiki/Quicksort-in-Idris (accessed July 24, 2015).