Chad Austin

Sum Types Are Coming: What You Should Know

Just as all mainstream languages now have lambda functions, I predict sum types are the next construct to spread outward from the typed functional programming community to the mainstream. Sum types are very useful, and after living with them in Haskell for a while, I miss them deeply when using languages without them.

Fortunately, sum types seem to be catching on: both Rust and Swift have them, and it sounds like TypeScript's developers are at least open to the idea.

I am writing this article because, while sum types are conceptually simple, most programmers I know don't have hands-on experience with them don't have a good sense of their usefulness.

In this article, I'll explain what sum types are, how they're typically represented, and why they're useful. I will also dispel some common misconceptions that cause people to argue sum types aren't necessary.

What is a Sum Type?

Sum types can be explained a couple ways. First, I'll compare them to product types, which are extremely familiar to all programmers. Then I'll show how sum types look (unsafely) implemented in C.

Every language has product types – tuples and structs or records. They are called product types because they're analogous to the cartesian products of sets. That is, int * float is the set of pairs of values (int, float). Each pair

contains an int AND a float.

If product types correspond to AND, sum types correspond to OR. A sum type indicates a value that is either X or Y or Z or ...

Let's take a look at enumerations and unions and show how sum types are a safe generalization of the two. Sum types are a more general form of enumerations. Consider C enums:

```
enum Quality {
  LOW,
  MEDIUM,
  HIGH
};
```

A value of type Quality must be one of LOW, MEDIUM, or HIGH (excepting uninitialized data or unsafe casts — we are talking about C of course). C also has a union language feature:

```
union Event {
  struct ClickEvent ce;
  struct PaintEvent pe;
};
```

ClickEvent and PaintEvent share the same storage location in the union, so if you write to one, you ought not read from the other. Depending on the version of the C or C++ specification, the memory will either alias or your program will have undefined behavior. Either way, at any point in time, it's only legal to read from one of the components of a union.

A sum type, sometimes called a discriminated union or tagged variant, is a combination of a tag (like an enum) and a payload per possibility (like a union).

In C, to implement a kind of sum type, you could write something like:

```
enum EventType {
  CLICK,
  PAINT
};

struct ClickEvent {
  int x, y;
};
struct PaintEvent {
  Color color;
};

struct Event {
  enum EventType type;
  union {
    struct ClickEvent click;
    struct PaintEvent paint;
  };
};
```

The `type` field indicates which event struct is legal to access. Usage looks as follows:

```
// given some Event event
switch (event.type) {
  case CLICK:
    handleClickEvent(&event.click);
    break;
  case PAINT:
    handlePaintEvent(&event.paint);
    break;
  // most compilers will let us know if we didn't handle every event
type
}
```

However, there is some risk here. Nothing prevents code from accessing `.paint` in the case that `type` is `CLICK`. At all times, every possible field in `event` is visible to the programmer.

A sum type is a safe formalization of this idea.

Sum Types are Safe

Languages like ML, Haskell, F#, Scala, Rust, Swift, and Ada provide direct support for sum types. I'm going to give examples in Haskell because the Haskell syntax is simple and clear. In Haskell, our Event type would look like this:

```
data Event = ClickEvent Int Int
           | PaintEvent Color
```

That syntax can be read as follows: there is a data type Event that contains two cases: it is either a ClickEvent containing two Ints or a PaintEvent containing a Color.

A value of type `Event` contains two parts: a small tag describing whether it's a ClickEvent or PaintEvent followed by a payload that depends on the specific case. If it's a ClickEvent, the payload is two integers. If it's a PaintEvent, the payload is one color. The physical representation in memory would look something like `[CLICK_EVENT_TAG][Int][Int]` or `[PAINT_EVENT_TAG][Color]`, much like our C code above. Some languages can even store the tag in the bottom bits of the pointer, which is even more efficient.

Now, to see what type of Event a value contains, and to read the event's contents, you must pattern match on it.

```
-- given some event :: Event
case event of
  ClickEvent x y -> handleClickEvent x y
  PaintEvent color -> handlePaintEvent color
```

Sum types, paired with pattern matching, provide nice safety guarantees. You cannot read `x` out of an event without first verifying that it's a ClickEvent. You cannot read `color` without verifying it's a PaintEvent. Moreover, the `color`

value is only in scope when the event is known to be a PaintEvent.

Sum Types are General

We've already discussed how sum types are more general than simple C-style enumerations. In fact, in a simple enumeration, since none of the options have payloads, the sum type can be represented as a single integer in memory. The following DayOfWeek type, for example, can be represented as efficiently as the corresponding C enum would.

```
data DayOfWeek = Sunday | Monday | Tuesday | Wednesday | Thursday |
Friday | Saturday
```

Sum types can also be used to create nullable data types like C pointers or Java references. Consider F#'s option, or Rust's Option, or Haskell's Maybe:

```
data Maybe a = Nothing | Just a
```

(`a` is a generic type variable – that is, you can have a `Maybe Int` or `Maybe Customer` or `Maybe (Maybe String)`).

Appropriate use of Maybe comes naturally to programmers coming from Java or Python or Objective C — it's just like using `NULL` or `None` or `nil` instead of an object reference except that the type signature of a data type or function indicates whether a value is optional or not.

When nullable references are replaced by explicit Maybe or Option, you no longer have to worry about NullPointerExceptions, NullReferenceExceptions, and the like. The type system enforces that required values exist and that optional values are safely pattern-matched before they can be dereferenced.

> > @davetchepak *"What can C# do that F# cannot?"*

> *NullReferenceException :-)*
>
> *— Tomas Petricek (@tomaspetricek) March 21, 2013*

Imagine writing some code that closes whatever window has focus. The C++ would look something like this:

```
Window* window = get_focused_window();
window->close_window();
```

Oops! What if there is no focused window? Boom. The fix:

```
Window* window = get_focused_window();
if (window) {
  window->close_window();
}
```

But then someone could accidentally call a different method on window outside of the if statement... reintroducing the problem. To help mitigate this possibility, C++ does allow introducing names inside of a conditional:

```
if (Window* window = get_focused_window()) {
  window->close_window();
```

Pattern matching on Maybe avoids this problem entirely. There's no way to even call `close_window` unless an actual window is returned, and the variable `w` is never bound unless there is an actual focused window:

```
window <- get_focused_window
case window of p
  Nothing -> return ()
```

```
    Just w -> close_window w
```

This is a big win for correctness and clarity.

Thinking in Sum Types

Once you live with sum types for a while, they change the way you think. People coming from languages like Python or Java (myself included) to Haskell immediately gravitate towards tuples and Maybe since they're familiar. But once you become accustomed to sum types, they subtly shift how you think about the shape of your data.

I'll share a specific memorable example. At IMVU we built a Haskell URL library and we wanted to represent the "head" of a URL, which includes the optional scheme, host, username, password, and port. Everything before the path, really. This data structure has at least one important invariant: it is illegal to have a scheme with no host. But it is possible to have a host with no scheme in the case of protocol-relative URLs.

At first I structured `UrlHead` roughly like this:

```
type SchemeAndHost = (Maybe Scheme, Host)
type UrlHead = Maybe (Maybe SchemeAndHost)
-- to provide some context, the compete URL type follows
type Url = (UrlHead, [PathSegment], QueryString)
```

In hindsight, the structure is pretty ridiculous and hard to follow. But the idea is that, if the URL head is `Nothing`, then the URL is relative. If it's `Just Nothing`, then the path is treated as absolute. If it's `Just (Just (Nothing, host))`, then it's a protocol-relative URL. Otherwise it's a fully-qualified URL, and the head contains both a scheme and a host.

However, after I started to grok sum types, a new structure emerged:

```
data UrlHead
  = FullyQualified ByteString UrlHost
  | ProtocolRelative UrlHost
  | Absolute
  | Relative
```

Now the cases are much clearer. And they have explicit names and appropriate payloads!

Sum Types You Already Know

There are several sum types that every programmer has already deeply internalized. They've internalized them so deeply that they no longer think about the general concept. For example, "This variable either contains a valid reference to class T or it is null." We've already discussed optional types in depth.

Another common example is when functions can return failure conditions. A fallible function either returns a value or it returns some kind of error. In Haskell, this is usually represented with Either, where the error is on the `Left`. Similarly, Rust uses the Result type. It's relatively rare, but I've seen Python functions that either return a value or an object that derives from Exception. In C++, functions that need to return more error information will usually return the error status by value and, in the case of success, copy the result into an out parameter.

Obviously languages with exceptions can throw an exception, but exceptions aren't a general error-handling solution for the following reasons:

1. What if you temporarily want to store the result or error? Perhaps in a cache or promise or async job queue. Using sum types allows sidestepping the complicated issue of exception transferability.
2. Some languages either don't have exceptions or limit where they can be thrown or caught.

3. If used frequently, exceptions generally have worse performance than simple return values.

I'm not saying exceptions are good or bad – just that they shouldn't be used as an argument for why sum types aren't important. :)

Another sum type many people are familiar with is values in dynamic languages like JavaScript. A JavaScript value is one of many things: either undefined or null or a boolean or a number or an object or... In Haskell, the JavaScript value type would be defined approximately as such:

```
data JSValue = Undefined
             | Null
             | JSBool Bool
             | JSNumber Double
             | JSString Text
             | JSObject (HashMap Text JSValue)
```

I say approximately because, for the sake of clarity, I left out all the other junk that goes on JSObject too. ;) Like whether it's an array, its prototype, and so on.

JSON is a little simpler to define:

```
data JSONValue = Null
               | True
               | False
               | String Text
               | Number Double
               | Array [JSONValue]
               | Object (HashMap Text JSONValue)
```

Notice this type is recursive — Arrays and Objects can refer to other JSONValues.

Protocols, especially network protocols, are another situation where sum types

frequently come up. Network packets will often contain a bitfield of some sort describing the type of packet, followed by a payload, just like discriminated unions. This same structure is also used to communicate over channels or queues between concurrent processes. Some example protocol definitions modeled as sum types:

```
data JobQueueCommand = Quit | LogStatus | RunJob Job
data AuthMethod = Cookie Text | Secret ByteString | Header ByteString
```

Sum types also come up when whenever sentry values are needed in API design.

Approximating Sum Types

If you've ever tried to implement a JSON AST in a language like C++ or Java or Go you will see that the lack of sum types makes safely and efficiently expressing the possibilities challenging. There are a couple ways this is typically handled. The first is with a record containing many optional values.

```
struct JSONValue {
  JSONBoolean* b;
  JSONString* s;
  JSONNumber* n;
  JSONArray* a;
  JSONObject* o;
}
```

The implied invariant is that only one value is defined at a time. (And perhaps, in this example, JSON null is represented by all pointers in JSONValue being null.) This limitation here is that nothing stops someone from making a JSONValue where, say, both `a` and `o` are set. Is it an array? Or an object? The invariant is broken, so it's ambiguous. This costs us some type safety. This approximation, by the way, is equivalent to Go's errors-as-multiple-return-values idiom. Go functions return a result and an error, and it's assumed (but

not enforced) that only one is valid at a time.

Another approach to approximating sum types is using an interface and classes like the following Java:

```
interface JSONValue {}
class JSONBoolean implements JSONValue { bool value; }
class JSONString implements JSONValue { String value; }
class JSONArray implements JSONValue { ArrayList<JSONValue> elements; }
// and so on
```

To check the specific type of a JSONValue, you need a runtime type lookup, something like C++'s `dynamic_cast` or Go's type switch.

This is how Go, and many C++ and Java JSON libraries, represent the AST. The reason this approach isn't ideal is because there's nothing stopping anyone from deriving new JSONValue classes and inserting them into JSON arrays or objects. This weakens some of the static guarantees: given a JSONValue, the compiler can't be 100% sure that it's only a boolean, number, null, string, array, or object, so it's possible for the JSON AST to be invalid. Again, we lose type safety without sum types.

There is a third approach for implementing sum types in languages without direct support, but it involves a great deal of boilerplate. In the next section I'll discuss how this can work.

The Expression Problem

Sometimes, when it comes up that a particular language doesn't support sum types (as most mainstream languages don't), people make the argument "You don't need sum types, you can just use an interface for the type and a class for each constructor."

That argument sounds good at first, but I'll explain generally that interfaces

and sum types have different (and somewhat opposite) use cases.

As I mentioned in the previous section, it's common in languages without sum types, such as Java and Go, to represent a JSON AST as follows:

```
interface JSONValue {}
class JSONBoolean implements JSONValue { bool value; }
class JSONString implements JSONValue { String value; }
class JSONNumber implements JSONValue { double value; }
class JSONArray implements JSONValue { ArrayList<JSONValue> elements; }
class JSONObject impleemnts JSONValue { HashMap<String, JSONValue>
properties; }
```

As I also mentioned, this structure does not rigorously enforce that the ONLY thing in, say, a JSON array is a null, a boolean, a number, a string, an object, or another array. Some other random class could derive from JSONValue, even if it's not a sensible JSON value. The JSON encoder wouldn't know what to do with it. That is, interfaces and derived classes here are not as type safe as sum types, as they don't enforce valid JSON.

With sum types, given a value of type JSONValue, the compiler and programmer know precisely which cases are possible. Thus, any code in the program can safely and completely enumerate the possibilities. Thus, we can **use** JSONValue anywhere in the program without modifying the cases at all. But if we add a new **case** to JSONValue, then we potentially have to update all uses. That is, it is much easier to use the sum type in new situations than to modify the list of cases. (Imagine how much code you'd have to update if someone said "Oh, by the way, all pointers in this Java program can have a new state: they're either null, valid, or lazy, in which case you have to force them. (Remember that nullable references are a limited form of sum types.) That would require a blood bath of code updates across all Java programs ever written.)

The opposite situation occurs with interfaces and derived classes. Given an interface, you don't know what class implements it — code consuming an interface is limited to the API provided by the interface. This gives a different degree of freedom: it's easy to add new **cases** (e.g. classes deriving from the

interface) without updating existing code, but your **uses** are limited to the functionality exposed by the interface. To add new methods to the interface, all existing implementations must be updated.

To summarize:

- With sum types, it's easy to add **uses** but hard to add **cases**.
- With interfaces, it's easy to add **cases** but hard to add **uses**.

Each has its place and neither is a great substitute for the other. This is known as The Expression Problem.

To show why the argument that sum types can be replaced with interfaces is weak, let us reduce the scenario to the simplest non-enumeration sum type: Maybe.

```
interface Maybe {
}

class Nothing implements Maybe {
}

class Just implements Maybe {
  Object value;
}
```

What methods should Maybe have? Well, really, all you can do is run different code depending on whether the Maybe is a Nothing or Just.

```
interface MaybeVisitor {
  void visitNothing();
  void visitJust(Object value);
}

interface Maybe {
  void visit(MaybeVisitor visitor);
```

```
}
```

This is the visitor pattern, which is another way to approximate sum types in languages without them. Visitor has the right maintainability characteristics (easy to add uses, hard to add cases), but it involves a great deal of boilerplate. It also requires two indirect function calls per pattern match, so it's dramatically less efficient than a simple discriminated union would be. On the other hand, direct pattern matches of sum types can be as cheap as a tag check or two.

Another reason visitor is not a good replacement for sum types in general is that the boilerplate is onerous enough that you won't start "thinking in sum types". In languages with lightweight sum types, like Haskell and ML and Rust and Swift, it's quite reasonable to use a sum type to reflect a lightweight bit of user interface state. For example, if you're building a chat client, you may represent the current scroll state as:

```
type DistanceFromBottom = Int
data ScrollState = ScrolledUp DistanceFromBottom | PeggedToBottom
```

This data type only has a distance from bottom when scrolled up, not pegged to the bottom. Building a visitor just for this use case is so much code that most people would sacrifice a bit of type safety and instead simply add two fields.

```
bool scrolledUp;
int distanceFromBottom; // only valid when scrolledUp
```

Another huge benefit of pattern matching sum types over the visitor pattern is that pattern matches can be nested or have wildcards. Consider a function that can either return a value or some error type. Haskell convention is that errors are on the Left and values are on the Right branch of an Either.

```
data FetchError = ConnectionError | PermissionError String |
JsonDecodeError
fetchValueFromService :: IO (Either FetchError Int)

-- Later on...

result <- fetchValueFromService
case result of
  Right value -> processResult value
  Left ConnectionError -> error "failed to connect"
  Left (PermissionError username) -> error ("try logging in, " ++
username)
  Left JsonDecodeError -> error "failed to decode JSON"
  _ -> error "unknown error"
```

Expressing this with the visitor pattern would be extremely painful.

Paul Koerbitz comes to a similar conclusion.

Named Variants? or Variants as Types?

Now I'd like to talk a little about sum types are specified from a language design perspective.

Programming languages that implement sum types have to decide how the 'tag' of the sum type is represented in code. There are two main approaches languages take. Either the cases are given explicit names or each case is specified with a type.

Haskell, ML, Swift, and Rust all take the first approach. Each case in the type is given a name. This name is not a type – it's more like a constant that describes which 'case' the sum type value currently holds. Haskell calls the names "type constructors" because they produce values of the sum type. From the Rust documentation:

```
enum Message {
```

```
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}
```

Quit and ChangeColor are not types. They are values. Quit is a Message by itself, but ChangeColor is a function taking three ints and returning Message. Either way, the names Quit, ChangeColor, Move, and Write indicate which case a Message contains. These names are also be used in pattern matches. Again, from the Rust documentation:

```
fn quit() { /* ... */ }
fn change_color(r: i32, g: i32, b: i32) { /* ... */ }
fn move_cursor(x: i32, y: i32) { /* ... */ }
match msg {
    Message::Quit => quit()
    Message::ChangeColor(r, g, b) => change_color(r, g, b),
    Message::Move { x: x, y: y } => move_cursor(x, y),
    Message::Write(s) => println!("{}", s),
};
```

The other way to specify the cases of a sum type is to use types themselves. This is how C++'s boost.variant and D's std.variant libraries work. An example will help clarify the difference. The above Rust code translated to C++ would be:

```
struct Quit {};
struct ChangeColor { int r, g, b; };
struct Move { int x; int y; };
struct Write { std::string message; };
typedef variant<Quit, ChangeColor, Move, Write> Message;

msg.match(
  [](const Quit&) { quit(); },
  [](const ChangeColor& cc) { change_color(cc.r, cc.g, cc.b); },
  [](const Move& m) { move_cursor(m.x, m.y); },
  [](const Write& w) { std::cout << w.message << std::endl; }
);
```

Types themselves are used to index into the variant. There are several problems with using types to specify the cases of sum types. First, it's incompatible with nested pattern matches. In Haskell I could write something like:

```
type MouseButton = LeftButton | RightButton | MiddleButton |
ExtraButton Int
type MouseEvent = MouseDown MouseButton Int Int | MouseUp MouseButton
Int Int | MouseMove Int Int

-- ...

case mouseEvent of
  MouseDown LeftButton x y -> beginDrag x y
  MouseUp LeftButton x y -> endDrag x y
```

In C++, using the variant<> template described above, I'd have to do something like:

```
struct LeftButton {};
struct RightButton {};
struct MiddleButton {};
struct ExtraButton { int b; };
typedef variant<LeftButton, RightButton, MiddleButton, ExtraButten>
MouseButton;

struct MouseDown { MouseButton button; int x; int y; };
struct MouseUp { MouseButton button; int x; int y; };
struct MouseMove { int x; int y; };
typedef variant<MouseDown, MouseUp, MouseMove> MouseEvent;

// given: MouseEvent mouseEvent;

mouseEvent.match(
  [](const MouseDown& event) {
    event.match([](LeftButton) {
      beginDrag(event.x, event.y);
    });
  },
  [](const MouseUp& event) {
    event.match([](LeftButton) {
```

```
        endDrag(event.x, event.y);
      });
    }
  );
```

You can see that, in C++, you can't pattern match against MouseDown and LeftButton in the same match expression.

(Note: It might look like I could compare with == to simplify the code, but in this case I can't because the pattern match extracts coordinates from the event. That is, the coordinates are a "wildcard match" and their value is irrelevant to whether that particular branch runs.)

Also, it's so verbose! Most C++ programmers I know would give up some type safety in order to fit cleanly into C++ syntax, and end up with something like this:

```
struct Button {
  enum ButtonType { LEFT, MIDDLE, RIGHT, EXTRA } type;
  int b; // only valid if type == EXTRA
};

struct MouseEvent {
  enum EventType { MOUSE_DOWN, MOUSE_UP, MOUSE_MOVE } type;
  Button button; // only valid if type == MOUSE_DOWN or type ==
MOUSE_UP
  int x;
  int y;
};
```

Using types to index into variants is attractive – it doesn't require adding any notion of type constructors to the language. Instead it uses existing language functionality to describe the variants. However, it doesn't play well with type inference or pattern matching, especially when generics are involved. If you pattern match using type names, you must explicitly spell out each fully-qualified generic type, rather than letting type inference figure out what is what:

```
auto result = some_fallible_function();
// result is an Either<Error, std::map<std::string, std::vector<int>>>
result.match(
  [](Error& e) {
    handleError(e);
  },
  [](std::map<std::string, std::vector<int>>& result) {
    handleSuccess(result);
  }
);
```

Compare to the following Haskell, where the error and success types are inferred and thus implicit:

```
result <- some_fallible_action
case result of
  Left e ->
    handleError e
  Right result ->
    handleSuccess result
```

There's an even deeper problem with indexing variants by type: it becomes illegal to write `variant<int, int>`. How would you know if you're referring to the first or second int? You might say "Well, don't do that", but in generic programming that can be difficult or annoying to work around. Special-case limitations should be avoided in language design if possible – we've already learned how annoying `void` can be in generic programming.

These are all solid reasons, from a language design perspective, to give each case in a sum type an explicit name. This could address many of the concerns raised with respect to adding sum types to the Go language. (See also this thread). The Go FAQ specifically calls out that sum types are not supported in Go because they interact confusingly with interfaces, but that problem is entirely sidestepped by named type constructors. (There are other reasons retrofitting sum types into Go at this point is challenging, but their interaction with interfaces is a red herring.)

It's likely not a coincidence that languages with sum types and type constructors are the same ones with pervasive type inference.

Summary

I hope I've convinced you that sum types, especially when paired with pattern matching, are very useful. They're easily one of my favorite features of Haskell, and I'm thrilled to see that new languages like Rust and Swift have them too. Given their utility and generality, I expect more and more languages to grow sum types in one form or another. I hope the language authors do some reading, explore the tradeoffs, and similarly come to the conclusion that Haskell, ML, Rust, and Swift got it right and should be copied, especially with respect to named cases rather than "union types". :)

To summarize, sum types:

- provide a level of type safety not available otherwise.
- have an efficient representation, more efficient than vtables or the visitor pattern.
- give programmers an opportunity to clearly describe possibilities.
- with pattern matching, provide excellent safety guarantees.
- are an old idea, and are finally coming back into mainstream programming!

I don't know why, but there's something about the concept of sum types that makes them easy to dismiss, especially if you've spent your entire programming career without them. It takes experience living in a sum types world to truly internalize their value. I tried to use compelling, realistic examples to show their utility and I hope I succeeded. :)

Endnotes

Terminology

In this article, I've used the name sum type, but tagged variant, tagged union, or discriminated union are fine names too. The phrase sum type originates in type theory and is a denotational description. The other names are operational in that they describe the implementation strategy.

Terminology is important though. When Rust introduced sum types, they had to name them something. They happened to settle on `enum`, which is a bit confusing for people coming from languages where enums cannot carry payloads. There's a corresponding argument that they should have been called `union`, but that's confusing too, because sum types aren't about sharing storage either. Sum types are a combination of the two, so neither keyword fits exactly. Personally, I'm partial to Haskell's `data` keyword because it is used for both sum and product types, sidestepping the confusion entirely. :)

More Reading

If you're convinced, or perhaps not yet, and you'd like to read more, some great articles have been written about the subject:

Wikipedia has two excellent articles, distinguishing between Tagged Unions and Algebraic Data Types.

Sum types and pattern matching go hand in hand. Wikipedia, again, describes pattern matching in general.

TypeScript's union types are similar to sum types, though they don't use named type constructors.

FP Complete has another nice introduction to sum types.

For what it's worth, Ada has had a pretty close approximation of sum types for decades, but it did not spread to other mainstream languages. Ada's implementation isn't *quite* type safe, as accessing the wrong case results in a runtime error, but it's probably close enough to safe in practice.

Much thanks goes to Mark Laws for providing valuable feedback and corrections. Of course, any errors are my own.

 Chad Austin / July 9, 2015 / c++, functional, haskell

15 thoughts on “Sum Types Are Coming: What You Should Know”

 **Ted Mielczarek**

July 10, 2015 at 8:08 am

I’ve just started learning Rust and I’ve found the Option and Result types to be fantastic. So many terrible APIs in other languages are the result of trying to solve “how do you return a value or indicate an error” but working around the lack of sum types. These so naturally solve the problem that I’m amazed that so many languages have been so popular without them! Having your error-handling statically checked is the icing on the cake.

 **Paddy3118**

July 10, 2015 at 9:10 pm

It may not predate Ada but Pascal has union types going back decades. (A quick google sees this mention for example: http://stackoverflow.com/questions/29514436/understanding-union-types)

Chad Austin •

July 10, 2015 at 9:13 pm

Excellent point! Actually a previous revision of the article mentioned Pascal but since Pascal's union types aren't memory-safe I removed them... but maybe I should have left that note in. Thanks for commenting!

John Haugeland

July 10, 2015 at 9:35 pm

Sum types come from the imperative community, which generally calls them [tagged|labelled] [structs|tuples|records|unions]. They were relatively common in the 1970s.

They were generally done away with because C++-style polymorphism is more useful in practice.

John Haugeland

July 10, 2015 at 9:42 pm

Other old places that you can find them include Algol 68, COM (as IVariant,) CORBA (as Discriminator,) the Pascal family (ada, modula-2, oberon, delphi, arguably c#, etc,) ML, arguably stuff in the C++ standard library, SQL, Visual Basic/VBA/VBS, OLE, LotusScript, QModemScript, REXX, and HyperCard, just off of the top of my head. Many of those are older than Haskell, some by 20+ years.

C has unions, and you're just expected to tag them with an enum. Most people complaining that C people are struggling to figure this out just don't know any real C people.

The original K&R book uses tagged unions in chapter 2.

sacundim

July 10, 2015 at 9:51 pm

"Haskell calls the names 'type constructors' because they produce values of the sum type."

No, here you're actually talking about data constructors. "Either" is a type constructor. "Left" and "Right" are data constructors.

Chad Austin •

July 10, 2015 at 11:16 pm

Good catch! Thanks, I'll correct it.

Luke deGruchy

July 11, 2015 at 5:18 am

Ceylon (which you didn't mention) was essentially built on union types:

http://ceylon-lang.org/documentation/1.1/tour/types/

solaster

July 11, 2015 at 6:17 am

So many complex things just because the C way is "unsafe"? If you care that much about that safety, just introduce an accessor function to check the type:

enum EventType {
CLICK,
PAINT
};

```
struct ClickEvent {
int x, y;
};
struct PaintEvent {
Color color;
};

struct Event {
enum EventType type;
union {
struct ClickEvent click;
struct PaintEvent paint;
};
};

struct ClickEvent *get_clickevent(struct Event *e)
{
assert(e != NULL);
assert(e->type == CLICK);
return &e->click;
};
```

I personaly rarely use this kind of construct anyway, I prefer avoiding it when it makes my code complex...

MartinMcDonough

August 15, 2015 at 12:27 pm

When I first started learning Mercury, I had thought that unification and discriminated unions (the Mercury name for sum types) were a logic programming feature, not a more functional feature.

Now I pretty much have no idea what logic programming is :)

Steven Sagaert

September 18, 2015 at 3:40 am

The recent Julia language also has them (called Union).

jbgi

October 20, 2015 at 1:53 pm

Note that lightweight definition of ADT with structural pattern matching can be achieve in Java with the help of annotations processors. like https://github.com/derive4j/derive4j or https://github.com/sviperll/adt4j/

Pingback: Designing a Delightful Functional Programming Language | Chad Austin

Carolos

February 15, 2018 at 11:06 pm

Kotlin has sealed classes.

Chris Warburton

February 16, 2018 at 7:09 am

I really enjoyed this; especially the balance between theory and practice, i.e. including examples of “this is the equivalent” *and* “this is what people would actually write”.

One thing I think it’s lacking is showing how we can provide nice APIs to encapsulate the required pattern matching.

It seems like many programmer’s first encounter with sum types is

`Option`/`Maybe`, and they don't see much benefit compared to null checking because they'll immediately pattern-match on any optional value they have; then wrap up their results into another optional value; then pattern-match that; then wrap up their result; and so on.

I think it's crucial to show beginners that such destructing-then-constructing is a bad idea, and can be avoided by using functions like `map`/`flatMap`/`join`/etc. This is nice for a few reasons:

– It keeps error-handling separate from the "happy path", so we can write the details of our code in a "straightforward" way using non-optional values, then our "top-level" can use `map`/etc. to plumb things together and handle the error case (similar to exceptions).

– It interacts very nicely with interfaces/polymorphism: if we handle optional values with `map`, then the same code will *also* work for `Either` values (which is useful for attaching error messages, for example).

– There's a nice progression of ideas which can get us *very* far: it initially seems strange that we can `map`/`flatMap`/etc. over optional values; until we realise that "results with an optional value" are the same as "lists with at most one element"! If we relax this "at most one element" constraint, we get a new perspective on our familiar "lists of values": they're the same as "results with many possible values"! At this point we read Wadler's "How to Replace Failure by a List of Successes" and discover that we've accidentally invented logic programming :)