Typed Design Patterns for the Functional Era

Will Crichton
wcrichto@brown.edu
Brown University
Providence, Rhode Island, USA

Abstract

This paper explores how design patterns could be revisited in the era of mainstream functional programming languages. I discuss the kinds of knowledge that ought to be represented as functional design patterns: architectural concepts that are relatively self-contained, but whose entirety cannot be represented as a language-level abstraction. I present four concrete examples embodying this idea: the Witness, the State Machine, the Parallel Lists, and the Registry. Each pattern is implemented in Rust to demonstrate how careful use of a sophisticated type system can better model each domain construct and thereby catch user mistakes at compile-time.

CCS Concepts: • **Software and its engineering → Design patterns**; **Functional languages**.

Keywords: design patterns, domain-driven design, rust

ACM Reference Format:
Will Crichton. 2023. Typed Design Patterns for the Functional Era. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture (FUNARCH '23), September 8, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3609025.3609477

1 Introduction

Where are all the functional programming design patterns? People have been clamoring for answers to this question for over a decade [1]. As functional programming concepts have crept into the mainstream of software engineering, that clamor has only grown. Yet, no catalog has emerged as the clear functional successor to the venerable *Design Patterns* [9]. This paper explores the question: what might a catalog of functional design patterns look like?

First, it is worth discussing the definition and purpose of a software design pattern. Norvig [20] argued that a design pattern should provide "descriptions of what experienced designers know (that isn't written down in the Language Manual)". The Gang of Four (GoF) further explain:

> "Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems. Our goal is to capture design experience in a form that people can use effectively. To this end we have documented some of the most important design patterns and present them as a catalog." [9, p. 4]

Put another way, a design pattern satisfies two key criteria:

1. It captures a recurring phenomenon (a pattern) in real-world system design.
2. It provides a software design strategy for managing this phenomenon that *is difficult to describe mechanistically.*

The first criterion points to why functional design patterns are still uncommon. There are not nearly as many large-scale systems written with functional languages compared to OOP languages. But this is slowly changing — the last decade has seen an explosion of systems implemented in functional languages[1] like Rust, Scala, Swift, and Clojure. I personally have close to a decade of experience with Rust, which underlies the perspective given in this paper. After working with Rust systems for web development, 3D graphics, data analytics, etc., I have seen many patterns of system design with various encodings into Rust. But I have never seen these patterns articulated concisely and collated into a single catalog.

The second criterion points to why the idea of design patterns has been tricky to translate into a functional setting. A common critique of the GoF patterns is that their complexity stems from an impoverished implementation language like C++. Norvig [20] showed that use of a dynamically-typed language like Lisp or Dylan can dramatically simplify some patterns to the point of triviality. Gibbons [10] showed that the use of a Haskell-esque type system enables patterns to be directly represented as well-typed library-level abstractions. From this perspective, one could argue that the Haskell or Rust standard libraries are design pattern catalogues *per se*, just communicated via code rather than prose.

However, it is undoubtedly true that an FP novice cannot simply read the Haskell or Rust language manuals and proceed to effectively design a complex system. A key skill that bridges the gap is understanding how to *systematically map*

[1] This list may raise some eyebrows as to my definition of "functional". I mean languages with features associated with the intellectual lineage of the lambda calculus.

FUNARCH '23, September 8, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0297-6/23/09.
https://doi.org/10.1145/3609025.3609477

domain concepts to language features. With object-oriented programming, developers have the problem of modeling a diverse range of domain concepts with a small set of load-bearing features. By contrast, functional languages offer a dizzying array of modeling tools: products, sums, newtypes, generics, modules, traits, type families, substructural types, lifetimes, and so on. FP novices therefore have a different problem: deciding which combination of many features is most appropriate for modeling a domain concept.

This paper articulates a vision for (typed) functional design patterns to match this new reality. I propose that functional design patterns should articulate concrete mappings from common domain concepts to particular configurations of functional language features. That vision is presented through four concrete patterns: the Witness, the State Machine, the Parallel Lists, and the Registry. The selection, structure, and substance of each pattern embodies my argument for what constitutes a useful software design pattern.

2 A Sample of Functional Design Patterns

Drawing inspiration from domain-driven design [7, 24], each design pattern is centered around a "domain pattern", i.e. a language-independent concept that appears in many software application domains. A domain pattern is described in terms of a "schema", in the sense used in psychological schema theory [22]. The knowledge captured in each design pattern is about how to model the domain pattern at both the type-level and expression-level of a particular functional programming language. The quality of a model is evidenced by the fact that incorrect uses of the model cause type errors rather than runtime errors (or worse, undefined behavior). Each pattern is described using the following structure:

1. **Schema:** the abstract elements (written in Small Caps) and relations that characterize the domain pattern.
2. **Examples:** a short list of instances of the pattern in real-world applications, briefly fit to the schema.
3. **Case study:** an extended case study of how to implement one instance of the pattern.
4. **Commentary:** some additional notes about the pattern.

The case studies are all implemented in Rust. That decision is in part due to my familiarity with the language. But it bears emphasizing that Rust is an excellent medium for articulating functional design patterns. Rust is an exceptionally practical language, designed for implementing production-grade systems rather than as a vehicle for research. Functional design patterns should focus on mappings that are *practical* as much as *feasible*, and the best way to determine practicality is under heavy stress from real-world use.

2.1 Witness

Schema: An Action that can only execute once a particular Condition is satisfied.

Examples: Access control (e.g., a user must login before seeing their profile), resource management (e.g., a computation can only occur if the machine has enough resources).

Case study: Consider a website that has normal and admin users, and an admin panel that should only be accessed (the Action) to users logged in as an admin (the Condition). The base API contains the following methods:

- `render_admin_panel() -> Html`: returns the HTML for the admin panel.
- `render_404() -> Html`: returns the HTML for a 404 page.
- `current_user() -> User`: returns the currently logged-in user.
- `is_admin(&User) -> bool`: returns true if the user is an admin.

Listing 1 provides both a correct and incorrect (but well-typed) implementation of a route for the admin panel. The correct admin panel route verifies that the current user is an admin, and returns a 404 otherwise. The issue with this implementation is that if we forget to check that the current user is an admin, or e.g. mess up the check, then the program could permit non-admins to see the admin panel.

```
#[route("/admin")]
fn admin_panel() -> Html {
  if is_admin(&current_user()) {
    render_admin_panel();
  } else {
    render_404();
  }
}

#[route("/admin")]
fn admin_panel_bad() -> Html {
  // Allowed to compile, leaks information
  render_admin_panel()
}
```

Listing 1. Admin panel with a boolean encoding of the admin flag on users.

An alternative encoding is shown in Listing 2. The key idea is to create a new data type that "witnesses" the proof of a capability, i.e. that a user is an admin. The `Admin` data type is wrapped in a module so it can only be constructed via the `try_admin` method. The `render_admin_panel` function is modified such that it takes an `Admin` as input. Therefore the only well-typed way to call the function is to generate a proof that the user (or at least *a* user) is an admin, and pass that proof to the renderer.

Commentary: While many type-driven design techniques are inspired by theorem proving, the witness technique is most directly drawn from that intellectual lineage. As applications become more security- and privacy-sensitive, design patterns can help developers write correct-by-construction

```rust
mod admin {
  pub struct Admin {}
  impl User {
    pub fn try_admin(&self) -> Option<Admin> {
      if is_admin(self) { Some(Admin {}) }
      else { None }
    }
  }
}

fn render_admin_panel(_admin: Admin) -> Html;

#[route("/admin")]
fn admin_panel() -> Html {
  if let Some(admin) = current_user().try_admin() {
    render_admin_panel(admin);
  } else {
    render_404();
  }
}

#[route("/admin"])
fn admin_panel_bad() -> Html {
   // Doesn't compile, missing witness
   render_admin_panel()
   // Doesn't compile, can't manually build witness
   render_admin_panel(Admin {})
}
```

Listing 2. Admin panel with a witness encoding of the admin permission.

code with respect to issues like access control. This pattern has been adopted in Rust web applications, most notably Rocket (rocket.rs), as well as some cryptography libraries.

2.2 State Machine

Schema: An object that exists in one of multiple States, which can Transition to other States in response to Events.

Examples: Files (open and closed), mutexes (unlocked and locked), shopping carts (empty, filled, purchased).

Case study: Consider a file model with three States: reading, end-of-file, and closed. Figure 1 shows a diagram of the model. An operating system provides a low-level API (i.e. a

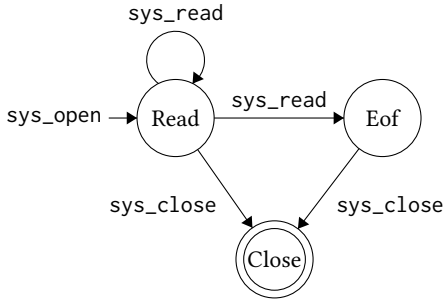

Figure 1. Model of the file state machine.

set of Transitions) that implements the model:

- `sys_open(&str) -> Descriptor`: creates a file descriptor.
- `sys_eof(&Descriptor) -> bool`: checks if the file has reached EOF, or panics if the file is closed.
- `sys_read(&mut Descriptor) -> Vec<u8>`: returns a buffer from the file, or panics if the file has reached EOF or closed.
- `sys_close(&mut Descriptor)`: closes the file, or panics if the file is already closed.

Consider the task of designing a `File` type that encapsulates the low-level API. Listing 3 shows a trivial wrapper. This wrapper provides no protection against incorrect usage. For example, one could attempt to read a file after reaching EOF which would panic, such as in Listing 4.

```rust
struct File { fd: Descriptor }
impl File {
  fn open(path: &str) -> File {
    File { fd: sys_open(path) }
  }
  fn eof(&self) -> bool {
    sys_eof(&self.fd)
  }
  fn read(&mut self) -> Vec<u8> {
    sys_read(&mut self.fd)
  }
  fn close(&mut self) {
    sys_close(&mut self.fd);
  }
}
```

Listing 3. File state machine that directly wraps the system API.

```rust
fn main() {
  let mut f = File::open("f.txt");
  // permits reading past EOF
  f.read(); f.read(); f.read();
  // permits closing multiple times
  f.close(); f.close();
  // permits reading after close
  f.read();
}
```

Listing 4. Example usage of the file state machine wrapper API.

One way to make the API less error-prone is to represent all states within one enumerated type, and then change the methods to return option types that indicate incorrect state transitions. Listing 5 shows an implementation of this idea. The enum representation ensures that panics cannot happen within the `File` implementation. However, a user may still encounter unexpected `None` values at runtime from an incorrect usage of the API as shown in Listing 6.

These runtime panics can be turned into compile-time errors by encoding each state as a distinct type, and then only

```
enum State { Read, Eof, Close }
struct File { fd: Descriptor, state: State }

impl File {
  fn open(path: &str) -> File {
    File { fd: sys_open(path), state: State::Read }
  }

  fn read(&mut self) -> Option<Vec<u8>> {
    match self.state {
      State::Read => {
        let buf = sys_read(&mut self.fd);
        if sys_eof(&self.fd) {
          self.state = State::Eof;
        }
        Some(buf)
      }
      State::Eof | State::Close => None
    }
  }

  fn close(&mut self) -> Option<()> {
    match self.state {
      State::Read | State::Eof => {
        sys_close(&mut self.fd);
        self.state = State::Close;
        Some(())
      }
      State::Close => None
    }
  }
}
```

Listing 5. File state machine with enum encoding.

```
fn main() {
  let mut f = File::open("hello.txt");
  while let Some(data) = f.read() {
    // process the data
  }
  f.close().unwrap();
  // can still get a runtime error,
  // but represented as a None
  f.read().unwrap();
  f.close().unwrap();
}
```

Listing 6. Example of the enum-based file state machine.

attaching the appropriate methods to each type. Listing 7 shows one such implementation with `FileRead` and `FileEof` states. The other key idea is that each state transition method invalidates the input object and returns the new state as a part of the output. In Rust, this concept is represented by taking ownership of the input state via `self` as opposed to a reference via `&self` or `&mut self`.

For example, the signature of `read` consumes the input file, and then uses an `Either` type to represent that one of two state transitions is possible. Either the file self-transitions to the Read state, or the file transitions to the Eof state. For the `close` method, no state is returned because because no operations exist on the Close state.

```
struct FileRead { fd: Descriptor }
struct FileEof { fd: Descriptor }
enum Either<L, R> { Left(L), Right(R) }

impl FileRead {
  fn open(path: &str) -> FileRead {
    FileRead { fd: sys_open(path) }
  }

  fn read(mut self) ->
    Either<(FileRead, Vec<u8>), FileEof>
  {
    if sys_eof(&self.fd) {
      Either::Right(FileEof { fd: self.fd })
    } else {
      let buf = sys_read(&mut self.fd);
      Either::Left((self, buf))
    }
  }

  fn close(mut self) {
    sys_close(&mut self.fd);
  }
}

impl FileEof {
  fn close(mut self) {
    sys_close(&mut self.fd);
  }
}
```

Listing 7. File state machine with typestate encoding.

```
fn main() {
  let mut f: FileRead = FileRead::open("f.txt");
  let f: FileEof = loop {
    f = match f.read() {
      Either::Left((f, data)) => {
        // ... process the data ...
        f
      }
      Either::Right(f) => break f
    }
  };
  // cannot call f.read() on FileEof
  f.close();
  // cannot call f.close() again since f is moved
}
```

Listing 8. Example of the typestate-based file state machine.

A drawback of the approach in Listing 7 is that both struct fields like `fd` and methods like `close` are duplicated between

the two state structs. Listing 9 shows an alternative approach. We return to having a single `File` type, but now it is parameterized[2] by a state type `S`. The implementation is similar to Listing 7, except every instance of `FileRead` is replaced by `File<Read>`. Additionally, the `close` method is now implemented once using a generic `impl` block over all states.

```
struct Read;
struct Eof;
struct File<S> { fd: Descriptor }

impl File<Read> {
  fn open(path: &str) -> File<Read> {
    /* Essentially same implementation */
  }

  fn read(mut self) ->
    Either<(File<Read>, Vec<u8>), File<Eof>>
  {
    /* Essentially same implementation */
  }
}

impl<S> File<S> {
  fn close(mut self) {
    sys_close(&mut self.fd);
  }
}
```

Listing 9. File state machine with generic typestate encoding.

Commentary: State machines are an extremely common idiom in system design, so it is a good instance of a kind of domain pattern that deserves treatment as a design pattern. The typestate technique has a long history [2, 23], and recent work has shown how typestate can be embedded in the type systems of existing languages [3, 6]. Libraries for session types use similar mechanisms [13, 21].

In the Rust ecosystem, typestate can be found in hardware programming, e.g. to represent the state of GPIO pins on a board as in the rppal library (golemparts/rppal) for Raspberry Pi. Typestate is also used by the Rocket web server framework (rocket.rs) to represent the different states of a server while it is starting up.

2.3 Parallel Lists

Schema: Two Lists of heterogeneous elements with a Parallel Relation between the elements.

Examples: `printf` (string format holes should match the arguments to fill the holes), web routes (a URL with multiple parameters should match the server callback receiving those parameters).

[2]The `File<S>` definition needs a field `PhantomData<S>` to prevent the compiler from complaining about an unused type, but we omit that detail here.

Case study: Consider a `printf`-style string formatter which takes a template List (string literals and holes) and an argument List (values to stringify into the holes). Listing 10 shows a simple implementation of printf (ignoring template parsing).

```
enum FElem {
  Str(String),
  Arg
}

fn format(
  tpl: Vec<FElem>, mut args: Vec<&dyn ToString>
) -> String {
  tpl.into_iter()
    .map(|elem| match elem {
      FElem::Str(s) => s,
      FElem::Arg => args.remove(0).to_string(),
    })
    .collect()
}
```

Listing 10. Formatting with enum/vector encoding.

```
fn main(){
  // Correct usage
  let tpl = vec![
    FElem::Str("Hello ".into()), FElem::Arg];
  let args = vec![&"World"];
  assert_eq!("Hello World", format(tpl, args));

  // Panics with too few arguments
  format(
    vec![FElem::Str("Hello ".into()), FElem::Arg],
    vec![]
  );

  // Silently ignores too many arguments
  format(
    vec![FElem::Str("Hello ".into()), FElem::Arg],
    vec![&"World", &"Again"]
  );
}
```

Listing 11. Example of enum/vector-encoded formatter.

An API client can use this interface incorrectly which results in runtime errors. Listing 11 shows how too few arguments will cause a runtime panic (because `Vec::remove` will panic if there are no elements in the vector). More insidiously, extra arguments are simply ignored.

The key observation is that printf has a Parallel Relation between the template and arguments. The number of `Var` holes should match the size of the `args` vector. (A more sophisticated model might also specify that type-specific formatting modifiers like `".1f"` should only be used for floats.)

To catch user mistakes at compile-time, the type system must be aware of the length and type of elements in each

list. One technique for this is the *heterogeneous list*, or h-list. An h-list is a list of elements with mixed types whose size is known at compile-time. Listing 12 shows how to implement a type-safe formatter using h-lists.

```
1  struct FStr(String);
2  struct FArg;
3
4  struct HNil;
5  struct HCons<H, T> { head: H, tail: T }
6
7  trait Format<Args> {
8    fn format(&self, args: Args) -> String;
9  }
10
11 impl Format<HNil> for HNil {
12   fn format(&self, _args: HNil) -> String {
13     "".to_string()
14   }
15 }
16
17 impl<ArgList, FmtList> Format<ArgList>
18 for HCons<FStr, FmtList>
19 where FmtList: Format<ArgList>
20 {
21   fn format(&self, args: ArgList) -> String {
22     let HCons { head: FStr(head), tail } = self;
23     head.to_string() + &tail.format(args)
24   }
25 }
26
27 impl<T, ArgList, FmtList> Format<HCons<T, ArgList>>
28 for HCons<FArg, FmtList>
29 where FmtList: Format<ArgList>, T: ToString,
30 {
31   fn format(&self, args: HCons<T, ArgList>)
32     -> String
33   {
34     args.head.to_string() +
35       &self.tail.format(args.tail)
36   }
37 }
```

Listing 12. Printf with h-list encoding.

The core h-list uses an inductive cons-list representation of `HNil` and `HCons<H, T>` types. For example, lines 2-3 of Listing 13 show the creation of h-lists through an `hlist!` macro. The type of `tpl` is `HCons<FStr, HCons<FArg, HNil>>`. The formatting function is then expressed as an inductive computation over the template and argument h-lists. Unfortunately, the implementation is far less straightforward than the enum/vector encoding in Listing 10. In short, the computation is defined through the `format` method wrapped in the `Format` trait. This trait is implemented for template h-lists, i.e. h-lists whose head types are either `FArg` or `FStr`. The base case of `HNil` is the empty string (lines 11-15). The

```
1  fn main() {
2    let tpl = hlist![FStr("Hello ".into()), FArg];
3    let args = hlist!["World"];
4    assert_eq!("Hello World", tpl.format(args));
5
6    // Type error with too few args
7    tpl.format(hlist![])
8
9    // Type error with too many args
10   hlist![].format(hlist!["extra"])
11 }
```

Listing 13. Example of h-list-encoded printf.

inductive case of `HCons<FStr, ...>` (lines 17-25) extracts the inner string from `FStr` and recurses.

The key is the inductive case of `HCons<FArg, ...>` (lines 27-37) which captures the parallelism between the template and argument lists. The `Format` trait is only implemented when the argument list is an `HCons<T, ...>` where `T: ToString`. The implementation converts `T` to a string, and then recurses on *both* `self.tail` (the format list) and `args.tail` (the argument list). The resulting design catches the issues in Listing 11 at compile-time, as shown in Listing 13. Passing both too many and too few arguments results in a type error.

Commentary: The Parallel List pattern is a good example of the trade-off between complexity and safety. The enum/vector encoding is simpler to read for the implementor, but it can lead to more bugs at runtime. However, although the h-list approach catches bugs, the type errors can potentially be incomprehensible, especially to novices.

Kiselyov et al. [15] pioneered the h-list approach for Haskell, which has since been adopted by libraries such as Servant [17]. H-lists are used in Rust by the web server library Warp (seanmonstar/warp) and a popular MySQL client (blackbeam/rust-mysql-simple). Many Rust libraries like Diesel (diesel.rs) and Bevy (bevyengine.org) emulate h-lists by implementing a trait for all tuples up to some size N, usually $N = 16$ or 32. However, this approach seriously reduces the comprehensibility of type errors (rust-lang/rfcs#2397). Yet another approach would be language-level variadic generics, but that feature is unlikely to be implemented in Rust soon (rust-lang/rfcs#376).

2.4 Registry

Schema: Objects that map Keys to heterogeneous Values, and users can register Keyed Requests for Values.

Examples: Event systems (callbacks associated with event names), dependency injection (named component injected into functions requesting components by name).

Case study: Consider an event system that contains events (the Value) with names (the Key), such as `OnClick` to represent a mouse click. Clients can register callbacks for an event (the Keyed Request) which can be triggered with an event

of the same name. The system should be open-world—the set of events can be extended by the API client.

A straightforward encoding of this model (e.g., as you can find in Javascript's DOM API) would represent event names as strings and event listeners as functions consuming a dynamically-typed event payload. Listing 14 shows the Rust implementation of this model.

```
type Listener = Box<dyn Fn(&dyn Any)>;

#[derive(Default)]
struct Events {
  listeners: HashMap<String, Vec<Listener>>
}

impl Events {
  fn register(&mut self, ev: &str, f: Listener) {
    self.listeners.entry(ev.into())
      .or_default().push(f);
  }

  fn trigger(&self, ev: &str, data: &dyn Any) {
    if let Some(fs) = self.listeners.get(ev) {
      for f in fs { f(data); }
    }
  }
}
```

Listing 14. Event registry with string-key encoding.

```
struct OnClick { mouse_x: usize, mouse_y: usize }
let mut events = Events::default();

// Correct example of register and trigger usage
events.register("click", Box::new(|ev| {
  let ev = ev.downcast_ref::<OnClick>().unwrap();
  assert_eq!(ev.mouse_x, 1);
}));
let ev = OnClick { mouse_x: 1, mouse_y: 3 };
events.trigger("click", &ev);

// Error #1: Wrong event name at register time
events.register("clack", ...);

// Error #2: Wrong event type in callback
events.register("click", Box::new(|ev| {
  let ev = ev.downcast_ref::<OnKeyPress>().unwrap();
}));

// Error #3: Wrong event name at trigger time
events.trigger("clack", ...);

// Error #4: Wrong event type passed to trigger
events.trigger("click", &OnKeyPress { /* ... */ });
```

Listing 15. Example usage of the string-key event encoding.

In this implementation, an event listener is a boxed function consuming `&dyn Any`, the Rust trait for dynamic typing. An event registry is a hashmap from strings to vectors of listeners. However, this implementation shifts significant complexity and risk onto the API client. Listing 15 shows an example client usage. Observe that the event listener must assert that the event payload is typed as `OnClick` via the `downcast_ref` method. The `register` and `trigger` calls must both use a matching `"click"` string. The `trigger` call must also provide a payload of the appropriate type. If the API client messes up any of these conditions, a runtime error will occur.

To develop a safer alternative, the key idea is to unify the representation of event-identity in the API. Rather than using strings to talk about types, we instead use the type directly, facilitated with the `TypeId` API that provides a unique, hashable identifier for any type. Listing 16 shows an implementation of `Events` that maps from type IDs to type-erased vectors of listeners. The `register` and `trigger` methods no longer take a string event name as input, but instead use the generic type `E` to refer to events.

```
trait Listener<E> = Fn(&E) -> () + 'static;
type ListenerVec<E> = Vec<Box<dyn Listener<E>>>;

#[derive(Default)]
struct Events {
  listeners: HashMap<TypeId, Box<dyn Any>>
}

impl Events {
  fn register<E: 'static, F: Listener<E>>(
    &mut self, f: F
  ) {
    let fs = self.listeners
      .entry(TypeId::of::<ListenerVec<E>>())
      .or_insert_with(||
        Box::new(Vec::<ListenerVec<E>>::new()));
    fs.downcast_mut::<ListenerVec<E>>().unwrap()
      .push(Box::new(f));
  }

  fn trigger<E: 'static>(&self, ev: &E) {
    let fs_opt = self.listeners
      .get(&TypeId::of::<ListenerVec<E>>());
    if let Some(fs) = fs_opt {
      let fs = fs.downcast_ref::<ListenerVec<E>>()
        .unwrap();
      for f in fs { f(ev); }
    }
  }
}
```

Listing 16. Event registry with type-key encoding.

Listing 17 shows an example usage of this alternative API. Observe that all the previous failure points are gone. A callback's payload must be consistent with the associated event.

A triggered event's payload must also be consistent with the associated event. Any typos are caught by the compiler.

```
struct OnClick { mouse_x: usize, mouse_y: usize }
let mut events = Events::default();

events.register(|ev: &OnClick| {
  assert_eq!(ev.mouse_x, 1);
});
let ev = OnClick { mouse_x: 1, mouse_y: 3 };
events.trigger(&ev);

// Can't associate this with OnClick
events.register(|ev: &OnKeyPress| { /* ... */ });

// Only triggers OnKeyPress callbacks
events.trigger(&OnKeyPress { /* ... */ });

// Typos caught by the compiler
events.trigger(&OnClack { /* ... */ });
```

Listing 17. Example usage of the type-key event encoding.

Commentary: The string-key event system is an instance of the broader problem of "stringly-typed" systems. For example, most developers learn early on that a fixed domain of objects is better represented by an enumerated type rather than a string. But that lesson can be difficult to extrapolate into open-world domains like an event system. Typed design patterns like the Registry can highlight the mechanisms needed in such a case, like the `TypeId` API.

In the Rust ecosystem, typed registries can be found in many crates. The Matrix client for Rust (matrix-org/matrix-rust-sdk) uses a typed registry for event handling. The Bevy game engine (bevyengine.org) uses a typed registry for implementing a typed entity-component-system architecture. The Shaku library (AzureMarker/shaku) provides a generic dependency injection framework using typed registries. An implementation of the Liquid template language in Rust (cobalt-org/liquid-rust) uses typed registries to hold external plugins. The core data structure that maps type IDs to boxed values can be found neatly encapsulated in the AnyMap library (chris-morgan/anymap).

3 Related Work

Having now seen four patterns in full detail, we should step back and consider how this style of presentation relates to prior work on teaching functional system design. Several books articulate design patterns for individual functional languages: *Scala Design Patterns* [12], *Functional Programming Patterns in Scala and Clojure* [4], *Scala Functional Programming Patterns* [14], *Haskell Design Patterns* [16], *Scala Design Patterns* [19], and *Functional Design and Architecture* [11].

These books do not articulate design patterns in the same sense I use within this paper. For example, much of the content of these books is (a) reimplementing GoF patterns like visitors, (b) explaining standard library APIs like monoids and functors, or (c) explaining basic functional programming concepts like currying. That content is assuredly valuable to its readers, but I do not envision that the N^{th} tutorial on monads should be considered a functional design pattern.

My philosophy is more ideologically aligned with books that articulate a modeling-oriented design practice. For example, *How to Design Programs* [8], *Domain Modeling Made Functional* [24], and *Type-Driven Development with Idris* [5] all fall under this umbrella. The difference between these books and the present work is, to an extent, stylistic. These books present a monolithic narrative that is intended to be consumed linearly from the first to the last page. But this style is less well-suited to demand-driven learning where a developer may want to simply find the best patterns for the problem at hand.

4 Discussion

One goal of this paper is to advance my particular vision for functional design patterns. But another is to prompt a discussion within the relevant academic and industrial circles about the key question: what kinds of knowledge, if any, ought to be articulated as typed functional design patterns? Or put another way, what do developers need to know that isn't currently documented about how to design effective systems in functional languages?

To that end, we should briefly consider the negative space of issues unaddressed by this paper. For instance, this paper has focused primarily on design patterns that catch incorrect programs at compile-time. But performance is also an important aspect of system design. Within the Rust community, there are excellent resources for how to profile and optimize small pieces of code like *The Rust Performance Book* [18]. However, there are comparatively few resources for how to architect a system to trade-off correctness, performance, maintainability, compile-times, etc. at larger scales. As just one example of a performance pattern, many Rust libraries use interning strategies (with a range of implementations) to reduce the memory footprint of commonly-allocated objects. Such performance patterns deserve to be documented in their own right.

Regardless, I believe it is high time to revisit design patterns now that the functional era is upon us. I hope to be a part of writing *Functional Design Patterns*, and I invite you to join me in making a Gang of more than just One.

References

[1] 2011. Where are all the functional programming design patterns? https://softwareengineering.stackexchange.com/q/89273.

[2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-Oriented Programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 1015–1022. https://doi.org/10.1145/1639950.1640073

[3] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (dec 2017), 29 pages. https://doi.org/10.1145/3158093

[4] Michael Bevilacqua-Linn. 2013. *Functional Programming Patterns in Scala and Clojure.* The Pragmatic Bookshelf.

[5] Edwin Brady. 2017. *Type-Driven Development with Idris.* Manning, New York, USA.

[6] José Duarte and António Ravara. 2021. Retrofitting Typestates into Rust. In *Proceedings of the 25th Brazilian Symposium on Programming Languages* (Joinville, Brazil) *(SBLP '21)*. Association for Computing Machinery, New York, NY, USA, 83–91. https://doi.org/10.1145/3475061.3475082

[7] Eric Evans. 2003. *Domain-driven design.* Addison-Wesley Educational, Boston, MA.

[8] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to design programs: an introduction to programming and computing.* MIT Press.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design patterns.* Addison Wesley, Boston, MA.

[10] Jeremy Gibbons. 2006. Design Patterns as Higher-Order Datatype-Generic Programs. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming* (Portland, Oregon, USA) *(WGP '06)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1159861.1159863

[11] Alexander Granin. 2020. *Functional Design and Architecture.* Leanpub.

[12] John Hunt. 2013. *Scala Design Patterns.* Springer, Pennsylvania, USA.

[13] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session Types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming* (Vancouver, BC, Canada) *(WGP 2015)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/2808098.2808100

[14] Atul S. Khot. 2015. *Scala Functional Programming Patterns.* Packt Publishing, Birmingham, UK.

[15] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell* (Snowbird, Utah, USA) *(Haskell '04)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/1017472.1017488

[16] Ryan Lemmer. 2015. *Haskell Design Patterns.* Packt Publishing, Birmingham, UK.

[17] Alp Mestanogullari, Sönke Hahn, Julian K. Arni, and Andres Löh. 2015. Type-Level Web APIs with Servant: An Exercise in Domain-Specific Generic Programming. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming* (Vancouver, BC, Canada) *(WGP 2015)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2808098.2808099

[18] Nicholas Nethercote. 2020. The Rust Performance Book. https://nnethercote.github.io/perf-book/.

[19] Ivan Nikolov. 2016. *Scala Design Patterns.* Packt Publishing, Birmingham, UK.

[20] Peter Norvig. 1998. Design Patterns in Dynamic Languages. https://www.norvig.com/design-patterns/.

[21] Riccardo Pucella and Jesse A. Tov. 2008. Haskell Session Types with (Almost) No Class. *SIGPLAN Not.* 44, 2 (sep 2008), 25–36. https://doi.org/10.1145/1543134.1411290

[22] David Rumelhart. 1980. Schemata: The building blocks of cognition. In *Theoretical Issues in Reading Comprehension.* Routledge.

[23] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* SE-12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

[24] Scott Wlaschin. 2017. *Domain Modeling made Functional. Tackle Software Complexity with Domain-Driven Design and F#.* The Pragmatic Bookshelf.

Received 2023-06-01; accepted 2023-06-28