The Plan

- Motivation
- Patterns
  - Type Constructor
  - Functor
  - Applicative
  - Monoid
  - Monad
- Encapsulating Asynchronous API
Motivation

Create a Library or Language Extension to Encapsulate Asynchronous API
<table>
<thead>
<tr>
<th>AND combinator</th>
<th>OR combinator</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ Returns result after both functions finish (like <em>wait for all</em>)</td>
<td>▪ Return the first result that arrives (like <em>wait for any</em> or <em>select</em>)</td>
</tr>
<tr>
<td>▪ Call the first one, wait for result, then call the second one, combine results</td>
<td>▪ Start the two calls in parallel. When one of them returns, return its result</td>
</tr>
<tr>
<td>▪ Unless the functions have side effects, result is deterministic and independent of order</td>
<td>▪ Aha: can’t do without <em>threads</em>! Problems with the second thread termination!</td>
</tr>
<tr>
<td>▪ ANDs can be combined into chains as long as they all return the same types: result = a &amp; b &amp; c</td>
<td>▪ <strong>Nondeterministic</strong>, unless both are guaranteed to return the same value</td>
</tr>
<tr>
<td>▪ Can it be generalized to multiple types? (Yes!)</td>
<td>▪ ORs can be chained: result = a</td>
</tr>
</tbody>
</table>
A Lot of Questions

- How similar are those two?
- Is this some kind of algebraic question?
- Are they inherently different beasts?
- How do ANDs aggregate results?
Type Constructor

Constructing new types from existing ones
Built-in Type Constructors

- For any type $T$
  - $T*$ is a pointer to $T$
  - $T&$ is a reference to $T$
  - $T[]$ is an array of $T$
  - $T()$ is a function returning $T$
  - $T(int)$ is a function taking int and returning $T$
  - $T(*)(*)$ is a pointer to a function returning $T$
Templated Type Constructors

- `unique_ptr<T>`
- `shared_ptr<T>`
- `vector<T>`
- `function<T()>`
- `pair<T, int>` “Curried” type constructor
- `function<T(int)>`
Hiding a Value

Type Constructor:

▪ unique_ptr<T> p;
▪ vector<T> v;
▪ function<T(int)> f;

Access to Value:

▪ *p, p.get()
▪ v[0], v[3], *v.begin()
▪ f(0), f(3)
Functor

Acting on “hidden” values
unique_ptr

- Given: function
  int length(string);
- Given: unique_ptr<string> p;
- Calculate length of string inside p

```cpp
int len = -1;
if (p) len = length(p.get());
```

- Clever, but not general

```cpp
unique_ptr<int> len;
if (p) len.assign(new int(length(p.get())));
```
Lifting length

- Lifted length operates on `unique_ptr<string>` and produces `unique_ptr<int>`

```cpp
unique_ptr<int> liftedLength(unique_ptr<string> const & s) {
  unique_ptr<int> result;
  if (s) result.assign(new int(length(*s)));
  return result;
}
```
Lifting any Function

- Lifted f operates on `unique_ptr<A>` and produces `unique_ptr<B>`

```cpp
template<class A, class B>
unique_ptr<B> fmap(function<B(A)> f, unique_ptr<A> const & p) {
    unique_ptr<B> result;
    if (p) result.reset(new B(f(*p)));
    return result;
}
```
unique_ptr<string> p(new string(" Hi! "));
auto np = fmap<string, int>(&length,
    fmap<string, string>(&trim, p);
Lifting a Function to Vector Type

- Lifted $f$ operates on vector<$A>$ and produces vector<$B>$

```cpp
template<class A, class B>
vector<B> fmap(function<B(A)> f, vector<A> as) {
    vector<B> result;
    transform(as.begin(), as.end(), back_inserter(result), f);
    return result;
}
```

Haskell Digression

Finding Functional Roots
Maybe

- Some analogy to unique_ptr (a is arbitrary type)

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

- Maybe is a functor

```haskell
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

- Compare with unique_ptr

```cpp
template<class A, class B>
unique_ptr<B> fmap(function<B(A)> f, unique_ptr<A> a) {
    unique_ptr<B> result;
    if (p) result.assign(new B(f(*a)));
    return result;
}
```
Which Is It?

\[
\text{fmap} :: (a \to b) \to \text{Maybe } a \to \text{Maybe } b \\
fmap f \text{ Nothing} = \text{Nothing} \\
fmap f (\text{Just } x) = \text{Just } (f \ x)
\]

- But lifting should be

\[
\text{fmap} :: (a \to b) \to (\text{Maybe } a \to \text{Maybe } b)
\]
Currying and Partial Application

- Calling fmap
Currying

- Equivalent curried type signature:
  \( \text{fmap} :: (a \rightarrow b) \rightarrow ([a] \rightarrow [b]) \)

- \( \text{fmap} \) takes a function \( a \rightarrow b \) and returns a function \( [a] \rightarrow [b] \)
The Functor Pattern

- **Type constructor**
  - Recipe: Crate a new “enriched” type from any type
  - Encapsulating (hiding) values of underlying type

- **Lifting of functions**
  - Recipe: For any function a->b create a function acting on “enriched” types
  - Higher order function `fmap` that performs the above mapping

- **Advantages of using `fmap`**
  - No need to break encapsulation
  - Composability

- **Functor axioms: composition and identity**
Functors: Recap

template<class A, class B>
unique_ptr<B> fmap(function<B(A)> f, unique_ptr<A> p) {
    unique_ptr<B> result;
    if (p) result.reset(new B(f(*p)));
    return result;
}

template<class A, class B>
vector<B> fmap(function<B(A)> f, vector<A> as) {
    vector<B> result;
    transform(as.begin(), as.end(), back_inserter(result), f);
    return result;
}
Applicative

Applying functions
Binary Functions

- Lift function of two arguments:
  ```
  concat :: String -> String -> String
  liftedConcat :: Maybe String -> Maybe String -> Maybe String
  ```

- Lift by first argument:
  - Treat as curried function: `concat :: String -> (String -> String)`
  - Apply `fmap` to it:
    ```
    fmap concat :: Maybe String -> Maybe (String -> String)
    ```
  - End up with Maybe function. How to apply it to `Maybe String`?

- New operation, apply:
  ```
  apply :: Maybe (a -> b) -> Maybe a -> Maybe b
  apply (Just f) (Just x) = Just (f x)
  apply _ _ = Nothing
  ```
Apply

apply :: Maybe (a -> b) -> Maybe a -> Maybe b
apply (Just f) (Just x) = Just (f x)
apply _ _ = Nothing

This is an alternative notation for an enriched function, (as opposed to a function returning an enriched value, like `apply` above):
Applicative unique_ptr

```cpp
template<class A, class B>
unique_ptr<B> apply(unique_ptr<function<B(A)>> f, unique_ptr<A> p)
{
    unique_ptr<B> result;
    if (f && p)
        result.reset(new B((*f)(*p)));
    return result;
}
```

- Have to curry the function and apply `fmap` to get the first argument
- Or can be done in one operation with `fmap2`
fmap2 for unique_ptr

```cpp
template<class A, class B, class C>
unique_ptr<C> fmap2(function<C(A, B)> f, unique_ptr<A> p1, unique_ptr<B> p2)
{
    unique_ptr<C> result;
    if (p1 && p2)
        result.reset(new C(f(*p1, *p2)));
    return result;
}
```

unique_ptr<string> s1(new string("Hello "));
unique_ptr<string> s2(new string("Applicative!"));
auto r2 = fmap2<string, string, string>(&concat, move(s1), move(s2));
cout << *r2 << endl;
```
Applicative Functor Pattern

- Defines *apply*
  - Enriched function acting on enriched argument

- Defines *pure*:
  - A way to lift a regular value

- Examples
  - unique_ptr
  - vector
  - function

- Used, for instance, for lifting binary operators

- Applicative axioms

```cpp
template<class A>
unique_ptr<A> pure(A a) {
    return unique_ptr<A>(new A(a));
}
```
Asynchronous APIs
Motivation

- Windows 8: any API that can take more than 50 ms was made asynchronous
- Problem with async: Inversion of control
- Great support in C# (async, await, etc.)
- No solution in C++ yet
- Threads already messed up, async doesn’t look better
Async APIs as Continuations

- Call async API and pass it a handler function
- Later, the handler is called (from a different thread)

Abstracting this:
- An async function returns an Async object
- Async object encapsulated a “future” value
- To retrieve that value, pass it a function that will take this value and work with it (handler/continuation)
- Requirement: async functions should be composable: (first open file then read file, etc.)

```cpp
template<class A>
struct Async {
    virtual void andThen(function<void(A)> h) = 0;
};
```
Async is a Functor

- It’s a type constructor
- You can lift a function \( f: A \rightarrow B \) to:
  \[ \text{Async}\langle A \rangle \rightarrow \text{Async}\langle B \rangle \]
  - Create a new \( \text{Async}\langle B \rangle \) with \text{andThen} which
  - Takes a continuation \( k \) accepting \( B \)
  - Calls \( \text{Async}\langle A \rangle \)’s \text{andThen} with
  - An (anonymous) function which
    - Takes a value of type \( A \)
    - Applies \( f \) to it, producing a value of type \( B \)
    - Calls the continuation \( k \) with that value
- It produces another composable Async
Async is an Applicative Functor

- Define pure and apply
- pure: andThen calls continuation with given value
- apply: Takes
  - An Async \(af\) that hides a function \(f: A \rightarrow B\) and
  - An Async \(aa\) that hides a value \(a\) of type \(A\)
  - Create an Async hiding \(B\), whose andThen
    - Takes a continuation \(k\) to retrieve \(B\), and which
    - Calls andThen on \(af\)
    - And then calls andThen on \(aa\)
    - Passing each of them a lambda function that
      - If the other one is already present, calls \(k(f(a))\)
      - Otherwise stores the value (either \(f\) or \(a\))
      - Access to stored values must be protected by a lock
And Combinator

- Call two (or more) async APIs and wait until all complete
- Each call returns an Async object
- Combine them into a compound Async object
  - What value should it produce?
  - In C# it produces a vector of values: not general enough
- Solution: Applicative. Use values as arguments to a user-provided function
  - Lift the function using *pure*
  - Apply it to Async objects using *apply* (chain the *applys* if necessary)
  - Basic case: function that pushes its arguments on a vector
Or Combinator and Monoid

- Start two async operations, wait for the one to finish first

- Problems
  - Nondeterminism (unless both evaluate the same thing)
  - Cancellation of the second request

- Monoid structure
  - Operation that’s associative
    - Here, the OR combinator
  - A “zero” element
    - An Async that never calls the continuation passed to it
Chaining Async Calls

- Call one async function: returns an Async
- With the result of the first Async, call another async function
- Example: open file then read from it (need file handle from the call to open)

a >>= b >>= c >>= …
Monad
Monadic Functions

- Start with applicative functor (In Haskell, rename `pure` to `return`)

- Monadic functions generate enriched values from plain values
  - Functions that may fail produce Maybe
  - Non-deterministic functions produce lists of possibilities

- How can we chain monadic functions?

```haskell
f :: String -> Maybe String
g :: String -> Maybe Int
```

- One possibility: Lift g

```haskell
fmap g :: Maybe String -> Maybe (Maybe Int)
```

- Collapse Maybe squared?
Monadic bind

- Alternatively, define \textit{bind} operation
  - Takes a monadic value (e.g., the result of the first monadic function)
  - And the second monadic function, and
  - Returns a new monadic value

\[
\text{bind} :: m\ a \to (a \to m\ b) \to m\ b
\]

- Bind for \texttt{unique\_ptr}

\[
\text{template<class A, class B>}
\text{unique\_ptr<A> bind(unique\_ptr<A> p,}
\text{ function<unique\_ptr<B>(A)> f)}
\]
Implementation of bind

```cpp
template<class A, class B>
unique_ptr<B> bind(unique_ptr<A> p,
                    function<unique_ptr<B>(A)> f)
{
    if (p) return f(*p);
    else    return unique_ptr<B>();
}
```

- binds may be chained
  - f returns unique_ptr<A>
  - g takes A and returns unique_ptr<B>
  - h takes B and returns unique_ptr<C>
  - etc.
Monad

- Type constructor
- *return*: the same as Applicative *pure*
  - Lifts arbitrary value to a monadic value
- *bind*
  - Applies monadic function to a monadic value
- Monadic axioms (they exist)
Async is a Monad: Bind

a is an async value so it has the `andThen` method that takes a continuation.

aFun is a function that returns an async value (of type B).

bind takes the two above and returns another async value (which has a method `andThen` that takes a continuation).
Example of Bind

`aOpen` takes a path and returns an async file handle `aFh`.

We bind `aFh` to `aRead` to obtain an async object with a method `andThen`.

And then:

- Bind `aFh` to `aRead` to obtain an async object with a method `andThen`.
bind aFh aRead fh

aFh

andThen fh

λ fh

parse buf

Later, the system calls our lambda with the actual file handle fh λ fh aRead fh fh aRead fh

Later, the system calls our lambda with the actual file handle

Here’s what our lambda does

It passes fh to aRead and gets an asynch object. It calls its andThen method with the parse continuation. Later the system calls parse with buf.
What the Client Code Looks Like

- In C++ it’s awful, but the logic is simple:
  - Call `asyncOpenFile` (returns immediately)
  - Bind it with function `asyncReadFile`
  - Start the proceedings with `andThen`, passing it `parseFile`

- In Haskell there is syntactic sugar, the `do` notation

```haskell
readFile path =
  do
    fh <- asyncOpenFile path
    buf <- asyncReadFile fh
    return buf

main = readFile "foo.bin" `andThen` parseFile
```

- It hides inversion of control!
Summary

- Async is a:
  - Functor: apply a function to a future value
  - Applicative: create the And combinator (select)
  - Monoid: create the Or combinator
  - Monad: chain async functions

- These combinators may be applied in any combination before the execution starts

- To activate execution, call the andThen method of the top level Async

- With syntactic sugar, the code could look the same as in the synchronous case
Conclusion

- In C++ Next, how about
  - Implementing powerful general purpose support (syntactic sugar) for these patterns
  - Building an Async library on top of it
- The same methods may be used to implement task-based concurrency (and mix it with Async) and functional reactive programming.
More Details

- My blogs at fpcomplete.com
- See also BartoszMilewski.com
- Conal Elliott: http://conal.net/papers/push-pull-frp/push-pull-frp.pdf, the same techniques applied to functional reactive programming (GUI programming)