PIGRO

LAZY EVALUATION ON STEROIDS

github.com/erikvalkering/pigro

About me

- Erik Valkering
- Software Engineer @Mindesk (https://mindeskvr.com)
- https://github.com/erikvalkering
- https://medium.com/@eejv
- https://www.linkedin.com/in/erik-valkering-3729702/

Part 1

Starting out lazy

What is Lazy Evaluation anyway?

> In programming language theory, lazy evaluation, or call-by-need, is an evaluation strategy which ***delays the evaluation of an expression until its value is needed*** (non-strict evaluation) and which also ***avoids repeated evaluations*** (sharing). The sharing can reduce the running time of certain functions by an exponential factor over other non-strict evaluation strategies, such as call-by-name, which repeatedly evaluate the same function, blindly, regardless of whether the function can be memoized.

(Emphasis mine)

https://en.wikipedia.org/wiki/Lazy_evaluation

Disclaimer

- Presentation will be code-heavy (C++)
- Slideware: everything passed by value (i.e. no perfect forwarding)

Making Functions *Lazy*

```
auto lazy(auto f) {
    using result_t = decltype(f());

    auto cache = std::optional<result_t>{};
    return [=]() mutable {
        if (!cache)
            cache = f();

        return *cache;
    };
}
```

Making Functions *Lazy*

```
auto lazy(auto f) {
    using result_t = decltype(f());

    auto cache = std::optional<result_t>{};
    return [=]() mutable {
        if (!cache)
            cache = f();

        return *cache;
    };
}
```

Usage:

```
auto long_computation() -> int;

auto lazy_computation = lazy(long_computation);

auto answer_to_life = lazy_computation(); // may take a while...
assert(answer_to_life == 42);

// ...
auto universe_and_everything = lazy_computation(); // instantaneous!
```

Making Functions *Reactive*

```
auto lazy(auto f, auto... dependencies) {
    using result_t = decltype(f(dependencies()...));
    using dependencies_t = decltype(std::tuple{dependencies()...});

    auto cache = std::optional<result_t>{};
    auto dependencies_cache = std::optional<dependencies_t>{};
    return [=]() mutable {
        const auto args = std::tuple{dependencies()...};
        if (!cache || args != dependencies_cache) {
            cache = std::apply(f, args);
            dependencies_cache = args;
        }

        return *cache;
    };
}
```

Making Functions *Reactive*

```
auto lazy(auto f, auto... dependencies) {
    using result_t = decltype(f(dependencies()...));
    using dependencies_t = decltype(std::tuple{dependencies()...});

    auto cache = std::optional<result_t>{};
    auto dependencies_cache = std::optional<dependencies_t>{};
    return [=]() mutable {
        const auto args = std::tuple{dependencies()...};
        if (!cache || args != dependencies_cache) {
            cache = std::apply(f, args);
            dependencies_cache = args;
        }

        return *cache;
    };
}
```

Making Functions *Reactive*

Usage:

```
auto draw_mouse_cursor(const point_2d pos, const image &icon) -> ui_object;
auto get_mouse_pos() -> point_2d;
auto load_image(const std::string_view filename) -> image;

auto arrow = [] { return load_image("arrow.png"); };
auto mouse_cursor = lazy(draw_mouse_cursor, get_mouse_pos, arrow);

// Rendering loop for a graphics editor
while (true) {
    mouse_cursor();
}
```

Making Functions *Reactive*

Usage:

```
auto draw_mouse_cursor(const point_2d pos, const image &icon) -> ui_object;
auto get_mouse_pos() -> point_2d;
auto load_image(const std::string_view filename) -> image;

auto arrow = [] { return load_image("arrow.png"); };
auto mouse_cursor = lazy(draw_mouse_cursor, get_mouse_pos, arrow);

// Rendering loop for a graphics editor
while (true) {
    mouse_cursor();
}
```

Issues:

- **Verbosity:** the need to create a lambda for `arrow`
- **Performance:** the `image` is loaded repeatedly from disk

Values as dependencies

```
auto lazy(auto f, std::invocable auto... dependencies) {
    // ...as before...
}
```

```
auto ensure_invocable(std::invocable auto dependency) {
    return dependency;
}

auto ensure_invocable(auto dependency) {
    return [=] { return dependency; };
}
```

```
auto lazy(auto f, auto... dependencies) {
     return lazy(f, ensure_invocable(dependencies)...);
}
```

Values as dependencies

```
auto lazy(auto f, std::invocable auto... dependencies) {
    // ...as before...
}
```

```
auto ensure_invocable(std::invocable auto dependency) {
    return dependency;
}

auto ensure_invocable(auto dependency) {
    return [=] { return dependency; };
}
```

```
auto lazy(auto f, auto... dependencies) {
     return lazy(f, ensure_invocable(dependencies)...);
}
```

> `std::invocable` is a C++20 ***concept*** which will match callable functions.

Values as dependencies

Usage (before):

```
auto draw_mouse_cursor(const point_2d pos, const image &icon) -> ui_object;
auto get_mouse_pos() -> point_2d;
auto load_image(const std::string_view filename) -> image;

auto arrow = [] { return load_image("arrow.png"); };
auto mouse_cursor = lazy(draw_mouse_cursor, get_mouse_pos, arrow);

// Rendering loop for a graphics editor
while (true) {
    mouse_cursor();
}
```

Values as dependencies

Usage:

```
auto draw_mouse_cursor(const point_2d pos, const image &icon) -> ui_object;
auto get_mouse_pos() -> point_2d;
auto load_image(const std::string_view filename) -> image;


auto mouse_cursor = lazy(draw_mouse_cursor, get_mouse_pos, load_image("arrow.png"));

// Rendering loop for a graphics editor
while (true) {
    mouse_cursor();
}
```

Values as dependencies

Usage:

```
auto draw_mouse_cursor(const point_2d pos, const image &icon) -> ui_object;
auto get_mouse_pos() -> point_2d;
auto load_image(const std::string_view filename) -> image;


auto mouse_cursor = lazy(draw_mouse_cursor, get_mouse_pos, load_image("arrow.png"));

// Rendering loop for a graphics editor
while (true) {
    mouse_cursor();
}
```

Values as dependencies

Usage:

```
auto draw_mouse_cursor(const point_2d pos, const image &icon) -> ui_object;
auto get_mouse_pos() -> point_2d;
auto load_image(const std::string_view filename) -> image;


auto mouse_cursor = lazy(draw_mouse_cursor, get_mouse_pos, load_image("arrow.png"));

// Rendering loop for a graphics editor
while (true) {
    mouse_cursor();
}
```

Issues:

- [x] **Verbosity:** ***values*** are now directly supported

Values as dependencies

Usage:

```
auto draw_mouse_cursor(const point_2d pos, const image &icon) -> ui_object;
auto get_mouse_pos() -> point_2d;
auto load_image(const std::string_view filename) -> image;


auto mouse_cursor = lazy(draw_mouse_cursor, get_mouse_pos, load_image("arrow.png"));

// Rendering loop for a graphics editor
while (true) {
    mouse_cursor();
}
```

Issues:

- [x] Verbosity: *values* are now directly supported
- [ ] Performance: the `image` is loaded repeatedly from disk

Values as dependencies

Usage:

```
auto draw_mouse_cursor(const point_2d pos, const image &icon) -> ui_object;
auto get_mouse_pos() -> point_2d;
auto load_image(const std::string_view filename) -> image;


auto mouse_cursor = lazy(draw_mouse_cursor, get_mouse_pos, lazy(load_image, "arrow.png"));

// Rendering loop for a graphics editor
while (true) {
    mouse_cursor();
}
```

Issues:

- ☑ **Verbosity:** ***values*** are now directly supported
- ☑ **Performance:** `load_image()` is now called lazily

Values as dependencies

Usage:

```
auto draw_mouse_cursor(const point_2d pos, const image &icon) -> ui_object;
auto get_mouse_pos() -> point_2d;
auto load_image(const std::string_view filename) -> image;


auto mouse_cursor = lazy(draw_mouse_cursor, get_mouse_pos, lazy(load_image, "arrow.png"));

// Rendering loop for a graphics editor
while (true) {
    mouse_cursor();
}
```

Issues:

- ☑ **Verbosity:** ***values*** are now directly supported
- ☑ **Performance:** `load_image()` is now called lazily

Reactivity all the way down

Usage:

```
auto draw_mouse_cursor(const point_2d pos, const image &icon) -> ui_object;
auto get_mouse_pos() -> point_2d;
auto load_image(const std::string_view filename) -> image;
auto get_drawing_mode() -> drawing_mode;

auto get_mouse_icon_filename(const drawing_mode mode) {
     return mode == drawing_mode::drawing ? "crosshair.png" : "arrow.png";
}

auto mode = lazy(get_drawing_mode);
auto filename = lazy(get_mouse_icon_filename, mode);
auto icon = lazy(load_image, filename);
auto mouse_cursor = lazy(draw_mouse_cursor, get_mouse_pos, icon);

// Rendering loop for a graphics editor
while (true) {
    mouse_cursor();
}
```

Reactivity all the way down

Usage:

```
auto draw_mouse_cursor(const point_2d pos, const image &icon) -> ui_object;
auto get_mouse_pos() -> point_2d;
auto load_image(const std::string_view filename) -> image;
auto get_drawing_mode() -> drawing_mode;

auto get_mouse_icon_filename(const drawing_mode mode) {
    return mode == drawing_mode::drawing ? "crosshair.png" : "arrow.png";
}

auto mode = lazy(get_drawing_mode);
auto filename = lazy(get_mouse_icon_filename, mode);
auto icon = lazy(load_image, filename);
auto mouse_cursor = lazy(draw_mouse_cursor, get_mouse_pos, icon);

// Rendering loop for a graphics editor
while (true) {
    mouse_cursor();
}
```

Reactivity all the way down

Usage:

```
auto draw_mouse_cursor(const point_2d pos, const image &icon) -> ui_object;
auto get_mouse_pos() -> point_2d;
auto load_image(const std::string_view filename) -> image;
auto get_drawing_mode() -> drawing_mode;

auto get_mouse_icon_filename(const drawing_mode mode) {
    return mode == drawing_mode::drawing ? "crosshair.png" : "arrow.png";
}

auto mode = lazy(get_drawing_mode);
auto filename = lazy(get_mouse_icon_filename, mode);
auto icon = lazy(load_image, filename);
auto mouse_cursor = lazy(draw_mouse_cursor, get_mouse_pos, icon);

// Rendering loop for a graphics editor
while (true) {
    mouse_cursor();
}
```

New issues:

- Performance: the `image` is constantly being compared, even if the drawing mode never changes

Part 2

Injecting some steroids

So what exactly is going on?

```
auto lazy(auto f, std::invocable auto... dependencies) {
    using result_t = decltype(f(dependencies()...));
    using dependencies_t = decltype(std::tuple{dependencies()...});

    auto cache = std::optional<result_t>{};
    auto dependencies_cache = std::optional<dependencies_t>{};
    return [=]() mutable {
        const auto args = std::tuple{dependencies()...};
        if (!cache || args != dependencies_cache) {
            cache = std::apply(f, args);
            dependencies_cache = args;
        }

        return *cache;
    };
}
```

So what exactly is going on?

```
auto lazy(auto f, std::invocable auto... dependencies) {
    using result_t = decltype(f(dependencies()...));
    using dependencies_t = decltype(std::tuple{dependencies()...});

    auto cache = std::optional<result_t>{};
    auto dependencies_cache = std::optional<dependencies_t>{};
    return [=]() mutable {
        const auto args = std::tuple{dependencies()...};
        if (!cache || args != dependencies_cache) {
            cache = std::apply(f, args);
            dependencies_cache = args;
        }

        return *cache;
    };
}
```

So what exactly is going on?

```
auto lazy(auto f, std::invocable auto... dependencies) {
    using result_t = decltype(f(dependencies()...));
    using dependencies_t = decltype(std::tuple{dependencies()...});

    auto cache = std::optional<result_t>{};
    auto dependencies_cache = std::optional<dependencies_t>{};
    return [=]() mutable {
        const auto args = std::tuple{dependencies()...};
        if (!cache || args != dependencies_cache) {
            cache = std::apply(f, args);
            dependencies_cache = args;
        }

        return *cache;
    };
}
```

> All the dependencies are unconditionally compared, even if they did **not** change.

Let's fix that (1) - Core Algorithm

```
auto lazy_core(auto f, lazy_function auto... dependencies) {
    using result_t = decltype(f(dependencies(nullptr).value...));


    auto cache = std::optional<result_t>{};

    return [=](std::nullptr_t) mutable {
        auto changed = !cache || (dependencies(nullptr).is_changed || ...);
        if (changed) {
            auto result = f(dependencies(nullptr).value...);

            changed = result != cache;
            cache = result;
        }

        return LazyResult{*cache, changed};
    };
}
```

Let's fix that (1) - Core Algorithm

```
auto lazy_core(auto f, lazy_function auto... dependencies) {
    using result_t = decltype(f(dependencies(nullptr).value...));


    auto cache = std::optional<result_t>{};

    return [=](std::nullptr_t) mutable {
        auto changed = !cache || (dependencies(nullptr).is_changed || ...);
        if (changed) {
            auto result = f(dependencies(nullptr).value...);

            changed = result != cache;
            cache = result;
        }

        return LazyResult{*cache, changed};
    };
}
```

Let's fix that (1) - Core Algorithm

```
auto lazy_core(auto f, lazy_function auto... dependencies) {
    using result_t = decltype(f(dependencies(nullptr).value...));


    auto cache = std::optional<result_t>{};

    return [=](std::nullptr_t) mutable {
        auto changed = !cache || (dependencies(nullptr).is_changed || ...);
        if (changed) {
            auto result = f(dependencies(nullptr).value...);

            changed = result != cache;
            cache = result;
        }

        return LazyResult{*cache, changed};
    };
}
```

> **Note:** In the `pigro` library, the double call to `dependencies(nullptr)` is optimized to a single call.

Let's fix that (1) - Core Algorithm

```
auto lazy_core(auto f, lazy_function auto... dependencies) {
    using result_t = decltype(f(dependencies(nullptr).value...));


    auto cache = std::optional<result_t>{};

    return [=](std::nullptr_t) mutable {
        auto changed = !cache || (dependencies(nullptr).is_changed || ...);
        if (changed) {
            auto result = f(dependencies(nullptr).value...);

            changed = result != cache;
            cache = result;
        }

        return LazyResult{*cache, changed};
    };
}
```

> **Note:** In the `pigro` library, the double call to `dependencies(nullptr)` is optimized to a single call.

Issues:

- [x] **Performance:** now skipping the comparisons and instead check whether any of the dependencies has changed

Let's fix that (2) - Some Helpers

```
template<typename T, std::same_as<bool> B>
struct LazyResult {
    T value;
    B is_changed;
};

template<typename F>
concept lazy_function = requires(F f) {
    { f(nullptr).value };
    { f(nullptr).is_changed };
};

template<typename F>
concept lazy_function_with_facade = lazy_function<F> && std::invocable<F>;

auto lazy_value(auto value, auto is_changed) {
    return [=](std::nullptr_t) {
        return LazyResult{value, is_changed};
    };
}
```

Usage:

```
auto f = lazy_value(42, true);    // always considered to be changed
auto g = lazy_value(1729, false); // always considered up-to-date
```

Let's fix that (3) - Ensuring laziness

```
auto ensure_lazy_function(lazy_function auto dependency) {
    return dependency;
}

auto ensure_lazy_function(lazy_function_with_facade auto dependency) {
    return dependency;
}

auto ensure_lazy_function(std::invocable auto dependency) {
    return lazy(
        [=](int) mutable { return dependency(); },
        lazy_value(0, true)
    );
}

auto ensure_lazy_function(auto dependency) {
    return lazy_value(dependency, false);
}
```

Let's fix that (4) - Wrapping Up

```
auto facade(lazy_function auto f) {
    return [=]<typename... Args>(Args... args) mutable {
        if constexpr (std::same_as<std::tuple<Args...>, std::tuple<std::nullptr_t>>)
            return f(nullptr);
        else if (sizeof...(Args) == 0)
            return f(nullptr).value;
    };
}

auto lazy(auto f, auto... dependencies) {
    return facade(lazy_core(f, ensure_lazy_function(dependencies)...));
}
```

Overview - Under 70 lines of code

```cpp
template<typename F>
concept lazy_function = requires(F f) {
    { f(nullptr).value };
    { f(nullptr).is_changed };
};

template<typename T, std::same_as<bool> B>
struct LazyResult {
    T value;
    B is_changed;
};

auto lazy_core(auto f, lazy_function auto... dependencies) {
    using result_t = decltype(f(dependencies(nullptr).value...));

    auto cache = std::optional<result_t>{};
    return [=](std::nullptr_t) mutable {
        auto changed = !cache || (dependencies(nullptr).is_changed || ...);
        if (changed) {
            auto result = f(dependencies(nullptr).value...);

            changed = result != cache;
            cache = result;
        }

        return LazyResult{*cache, changed};
    };
}

template<typename F>
concept lazy_function_with_facade = lazy_function<F> && std::invocable<F>;

auto facade(lazy_function auto f) {
    return [=]<typename... Args>(Args... args) mutable {
        if constexpr (std::same_as<std::tuple<Args...>, std::tuple<std::nullptr_t>>)
            return f(nullptr);
        else if (sizeof...(Args) == 0)
            return f(nullptr).value;
    };
}
```

```cpp
auto lazy_value(auto value, auto is_changed) {
    return [=](std::nullptr_t) {
        return LazyResult{value, is_changed};
    };
}

auto ensure_lazy_function(lazy_function auto dependency) {
    return dependency;
}

auto ensure_lazy_function(lazy_function_with_facade auto dependency) {
    return dependency;
}

auto ensure_lazy_function(std::invocable auto dependency) {
    return lazy(
        [=](int) mutable { return dependency(); },
        lazy_value(0, true)
    );
}

auto ensure_lazy_function(auto dependency) {
    return lazy_value(dependency, false);
}

auto lazy(auto f, auto... dependencies) {
    return facade(lazy_core(f, ensure_lazy_function(dependencies)...));
}
```

https://godbolt.org/z/q8oG5da1W

Comparison with hand-coded solution

HAND-CODED

```
auto should_draw_mouse_cursor = false;
auto cache_pos = std::optional<point_2d>{};
auto cache_icon = std::optional<image>{};
auto cache_filename = std::optional<std::string>{};
auto cache_mode = std::optional<drawing_mode>{};

// Rendering loop for a graphics editor
while (true) {
    const auto pos = get_mouse_pos();
    if (cache_pos != pos) {
        cache_pos = pos;
        should_draw_mouse_cursor = true;
    }

    const auto mode = get_drawing_mode();
    if (cache_mode != mode) {
        cache_mode = mode;

        const auto filename = get_mouse_icon_filename(mode);
        if (cache_filename != filename) {
            cache_filename = filename;

            const auto icon = load_image(filename);
            if (cache_icon != icon) {
                cache_icon = icon;

                should_draw_mouse_cursor = true;
            }
        }
    }

    if (should_draw_mouse_cursor /* && cache_pos && cache_icon */) {
        draw_mouse_cursor(*cache_pos, *cache_icon);
    }
}
```

LAZY & REACTIVE

```
auto mode = lazy(get_drawing_mode);
auto filename = lazy(get_mouse_icon_filename, mode);
auto icon = lazy(load_image, filename);
auto mouse_cursor = lazy(draw_mouse_cursor, get_mouse_pos, icon);

// Rendering loop for a graphics editor
while (true) {
    mouse_cursor();
}
```

- Much shorter
- Declarative
- Less error-prone
- Better maintainable
- Logic can be factored out easily

Conclusion

Using lazily evaluated functions in a declarative and reactive way will make your code easier to reason about, easier to maintain, and less prone to errors.

Thank you!

https://github.com/erikvalkering/pigro