The Declarative Nature of Implicit Layer Activation

Stefan Ramson Jens Lincke Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam, Germany
{firstname.lastname}@hpi.de

ABSTRACT

Context-oriented programming (COP) directly addresses context variability by providing dedicated language concepts: *layers*, units of modularity, store context-dependent behavior. During runtime, layers can be applied dynamically depending on the current context of the program.

Various activation means for layers have been proposed. Most of them require developers to model context switches explicitly. In contrast, implicit layer activation (ILA) allows developers to bind the activation status of a layer to a boolean predicate. The associated layer stays automatically active as long as the given predicate evaluates to true.

Despite its declarative semantics, ILA is usually implemented in an imperative fashion. In this paper, we present and compare two implementation variants for ILA in ContextJS: an imperative and a reactive implementation. Furthermore, we discuss their trade-offs regarding code complexity as well as runtime overhead.

CCS CONCEPTS

• **Software and its engineering → Language features**; *Object oriented languages*; Multiparadigm languages;

KEYWORDS

Implicit Layer Activation, Reactive Programming, Active Expressions, Context-oriented Programming

ACM Reference format:
Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2017. The Declarative Nature of Implicit Layer Activation. In *Proceedings of COP'17, Barcelona , Spain, June 19-20, 2017,* 10 pages.
https://doi.org/10.1145/3117802.3117804

1 INTRODUCTION

Context variability is an inherent property to most modern software systems. However, wide-spread programming languages do not support concepts for context variability in a principled way. The context-oriented programming (COP) paradigm [3] directly addresses context variability by providing dedicated language concepts to describe contexts and context-dependent behavior. *Layers* allow to store all behavior related to a specific context in a single unit of modularity.

To apply context-dependent behavior, layers can be activated dynamically through various activation means. Most activation means require developers to model context switches in an explicit manner [5]. For example, using global layer activation, a layer becomes (de-)activated at a certain point in imperative control flow, usually guarded by a condition. Similarly, dynamic layer activation allows to explicitly activate a layer for the extent of a message send. Even integrations with event-based concepts, such as event transitions, require to emit the respective events explicitly.

In contrast to most activation means, implicit layer activation (ILA) [10] provides a mechanism to declaratively define contexts. By describing the extent of a context rather than context switches, ILA relieves the programmer from the task of manually describing the boundaries of contexts. While ILA offers promising properties, only few COP implementations support ILA [2, 4, 5, 10]. Despite its declarative semantics, ILA is usually implemented in an imperative fashion, similar to other activation means. In particular, the current layer composition stack is determined *imperatively* when calling a layered method. At this very point in time, the COP framework checks the conditions of all implicitly activated layers. However, this non-reactive approach is only possible, because most COP implementations are limited to adapting object and class methods [8]. These concepts are *passive* entities that only affect the program behavior when called explicitly. Therefore, the limitation to passive entities enables the COP framework to check the condition at a well-defined point in the program. In contrast, *active* entities, such as constraints, may initiate behavior by themselves without being called explicitly. Extending the concept of COP beyond method decoration requires to activate scoped entities at specific times.

Using a *reactive implementation*, a COP framework can *eagerly enable* implicitly activated layers and, thus, deal with active entities properly. Thus, a reactive implementation might pave the way to apply the concept of COP to other types of abstraction beyond partial methods. For example, a layer could be used to limit the scope of a constraint: when a condition becomes true, the corresponding layer becomes active and the constraint immediately takes effect [6], instead of waiting for an additional, explicit trigger.

Similar to active entities, *life-cycle callbacks*, such as `onActivate` and `onDeactivate`, should be executed immediately when a layer becomes active or inactive, respectively. As an example, an `onActivate` callback might set up some state required by the layer while the `onDeactivate` callback cleans up this additional state. An imperative implementation might lead to unintended behavior as it delays the execution of the callbacks unnecessarily. Instead, the life-cycle callbacks expect to be executed eagerly. Again, integrating ILA properly with reactive concepts, such as life-cycle callbacks, requires a reactive implementation for ILA itself.

COP'17, June 19-20, 2017, Barcelona , Spain

ACM ISBN 978-1-4503-4971-0/17/06…$15.00
https://doi.org/10.1145/3117802.3117804

Contributions. Both examples above highlight advantages of a reactive implementation for ILA. However, imperative implementations are more prevalent. Thus, in this paper, we examine the different approaches to implement ILA. In particular, we make the following contributions:

- We extend ContextJS [7] with implicit layer activation (ILA) in two variants: an imperative implementation based on *an extended dispatch* and a reactive implementation based on *Active Expressions [9]*.
- We compare the two implementations regarding *code complexity* and *runtime overhead*.

Outline. The remaining of this paper is structured as follows. In section 2, we provide an overview of relevant prior work. In section 3, we present two implementation variants for ILA. Then, section 4 discusses the complexity of the presented implementations. Furthermore, we compare both implementation according to their runtime performance in section 5. Finally, section 6 concludes this work.

2 BACKGROUND

This section presents relevant prior work. In particular, we discuss the concept of ILA as well as Active Expressions as a means to implement ILA using reactive programming concepts.

2.1 Implicit Layer Activation

ILA [10] is an activation means with declarative semantics. To be precise, developers can associate a layer to an arbitrary object-oriented (OO) expression:

```
layer.activeWhile(expression)
```

By associating a layer with an expression, the layer is not activated or deactivated at a fixed time. Instead, the layer is active as long as the given condition holds, as depicted in the following example:

```
1  var shouldTrace = false;
2  new Layer().refineObject(Networking, {
3    fetch(url) {
4      console.log('fetch ' + url);
5      return proceed(url);
6    }
7  }).activeWhile(() => shouldTrace);
8
9  Networking.fetch('example.com'); // prints
       nothing
10 shouldTrace = true;
11 Networking.fetch('example.com'); // prints '
       fetch example.com'
```

Despite these declarative semantics, ILA is typically implemented in an imperative manner. An underlying system keeps track of all layered methods, such as the method `fetch` adapted in line 3. Then, when calling a layered method, the current layer composition stack is determined [5]. At this very point in time, the COP framework checks the conditions of all implicitly activated layers [1, 8]. If the condition evaluates to `true`, the method adaptation is taken into account for this method call, as the modified behavior in line 11 illustrates.

Considering the declarative semantics of ILA, a reactive implementation does not seem a stretch [6]: an underlying reactive framework may monitor variables referenced by the given condition. When such a variable changes, the condition is re-evaluated and the corresponding layer is activated or deactivated accordingly.

2.2 Active Expressions

Active Expressions [9] is a basic reactive programming concept designed to aid language designers when implementing reactive programming concepts in OO environments. In particular, Active Expressions relieve developers from the tedious task of change detection by hiding implementation details behind a unified abstraction: *OO expressions*. Developer provide the expression to be monitored to the reactive framework by calling the `aepxr` function:

```
aexpr(expression).onChange(callback)
```

The specified callback gets executed whenever the evaluation result of the expression changes.

Using Active Expressions, one can significantly reduce the implementation effort when creating new reactive programming concepts. As ILA fits well into the working principle of Active Expressions, we use Active Expressions in order to simplify the reactive implementation of ILA. The provided expression may contain any OO mechanism, such as information hiding and polymorphism. As a consequence, the resulting ILA implementation integrates well with OO environments.

3 IMPLEMENTING IMPLICIT LAYER ACTIVATION IN CONTEXTJS

Currently, ContextJS [7] does not support implicit layer activation (ILA) as an activation means[1]. However, due to the reactive and declarative nature of the web, ContextJS might benefit from this activation means. Thus, we show how to extend ContextJS with ILA, first using an imperative implementation, then using a reactive implementation with Active Expressions.

3.1 Imperative Implementation

ContextJS supports multiple activation means, including global activation and dynamic activation for the extent of a function call. When calling a layered method, the `currentLayers` function is responsible for computing an appropriate layer composition, either by using a cached result or, if necessary, by determining a new one using global and dynamic layers:

```
1 export function currentLayers() {
2   // parts omitted for readability
3   if (!current.composition) {
4     current.composition = composeLayers(
          LayerStack);
5   }
6   return current.composition;
7 }
```

[1] https://github.com/LivelyKernel/ContextJS accessed on April 16th, at commit 938e117; npm package: contextjs in version 2.0.0

For our extension[2], we add a separate list of layers to represent layers potentially activated through ILA, called `implicitLayers`. To implicitly activate a layer, we add the method `activeWhile` to the class `Layer`:

```
1 activeWhile(condition) {
2   if (!implicitLayers.includes(this)) {
3     implicitLayers.push(this);
4   }
5   this.implicitlyActivated = condition;
6   return this;
7 }
```

This method has two responsibilities. First, in line 3, it adds the layer to the list of implicitly activated layers, if necessary. Second, it stores the provided argument `condition`, a boolean function to specify whether the layer should be active, as seen in line 5. Using the list of implicitly activated layers, we can get all layers that are actually activated by filtering this list for layers with their conditions evaluating to true, as done by the `getActiveImplicitLayers` function:

```
1 function getActiveImplicitLayers() {
2   return implicitLayers.filter(layer =>
        layer.implicitlyActivated());
3 }
```

Using the `getActiveImplicitLayers` function, we can now adjust the computation of the current layer composition in `currentLayers`:

```
1 export function currentLayers() {
2   // part omitted for readability
3   var current = LayerStack[LayerStack.length
        - 1];
4   if (!current.composition) {
5     current.composition = composeLayers(
        LayerStack);
6   }
7   return current.composition@@@.concat(
      getActiveImplicitLayers())@@@;
8 }
```

To include implicitly activated layers, we append all layers activated through ILA to the already computed layer composition. As a result, the returned layer composition contains dynamically activated, globally activated, and implicitly activated layers. Note that we cannot cache implicitly activated layer, because we have no means to invalidate the cache on changes to the condition.

3.2 Reactive Implementation

In contrast to the imperative implementation, we do not introduce a separate data structure for implicitly activated layers. Instead, we treat implicitly activated layers as being globally active as long as their condition evaluates to true. As a result, we can reuse the existing layer composition algorithm once a layer becomes active.

[2]https://github.com/active-expressions/programming-contextjs-plain accessed on April 16th 2017, at commit 22deb54

Table 1: Code complexity of the presented implementations compared to the existing ContextJS version in terms of the number of AST nodes.

AST nodes (sloc)	Complete	Difference to ContextJS
Unmodified ContextJS	2485 (568)	
Imperative Implementation	2557 (580)	72 (12)
Reactive Implementation	2525 (575)	40 (7)

To do so, the `activeWhile` method has to setup dependencies to detect changes to the given condition and update the layer accordingly. For this implementation[3], we wrap the given `condition` in an Active Expression. Using this Active Expression, we can easily implement the appropriate reactive behavior:

```
1 activeWhile(condition) {
2   aexpr(condition)
3     .onBecomeTrue(() => this.beGlobal())
4     .onBecomeFalse(() => this.beNotGlobal()
        );
5   return this;
6 }
```

Using the `onBecomeTrue` (line 3) and `onBecomeFalse` (line 4) methods, the layer is eagerly activated or deactivated whenever the expression result becomes true or false, respectively. Thus, the layer is automatically taken into account as a globally activated layer by the existing layer composition algorithm. Additionally, those methods automatically adjust the initial state of the layer depending on the current result of the given expression.

4 IMPLEMENTATION COMPLEXITY

We quantitatively compare both implementations of ILA in terms of code complexity. As a measurement for code complexity, we count the total number of abstract syntax tree (AST) nodes in each implementation. We summarize our measurements in Table 1. According to Table 1, the reactive implementation based on Active Expressions has a lower complexity compared to the imperative implementation.

However, more important than these quantitative results is the way both implementations introduce the concept of ILA to the ContextJS library. The imperative implementation introduces an additional layer type: implicitly activated layers. Furthermore, the imperative implementation requires knowledge about the underlying dispatch mechanism in order to extend the layer composition algorithm to take implicitly activated layers in account. In contrast, the reactive variant reuses the existing semantics by only modifying layers through already exposed methods. Thus, the reactive variant only requires knowledge about the usage of ContextJS, not about its internal working principles.

[3]https://github.com/active-expressions/programming-contextjs-aexpr accessed on April 16th 2016, at commit 07437e8

5 PERFORMANCE EVALUATION

To identify the performance penalties implied by the different implementation variants described in section 3, we provide and discuss multiple micro benchmark scenarios in the following. For each scenario, we compare the imperative implementation, described in subsection 3.1, with the Active Expression-based implementation, described in subsection 3.2. Active Expressions allow to choose between multiple underlying implementation strategies. Thus, we compare against two strategies. First, the *interpretation* strategy uses dynamic interpretation to punctually insert property accessors into the system space. Second, the *compilation* strategy performs a heavy-weight source code transformation to notify about changes in the system state.

In subsection 5.1, we discuss our benchmark setup, test suite, and statistical methods. Appendix A provides the source code of the benchmark suite.

5.1 Performance Benchmark Setup and Statistical Methods

All benchmarks were executed on the following system:

- CPU and memory: Intel(R) Core(TM) i7-6650U CPU @ 2.20GHz 2.21 GHz, 4 Logical cores; 16.0 GB Main Memory
- System software: Windows 10 Pro (OS Build 15063)
- Runtime: Google Chrome version 57.0.2987.133; benchmarks executed using Karma test runner version 1.2.0 and Mocha test framework version 3.0.2
- Transpiler and bundler: babel-cli 6.11.4 (no es2015 preset) and rollup 0.34.8,
- Libraries under test:
  - *programming-contextjs-plain* at commit 1360e1a[4]
  - *programming-contextjs-aexpr* at commit 07437e8[5]
- Benchmark suite: *aexpr-ila-benchmark* at commit 6a6395b[6].

We measured the execution time of a benchmark by wrapping the benchmark in a function and measuring the time between calling the function and it returning. Each benchmark configuration was iterated 100 times, with only the final 30 iterations taken into account for the overall performance measurement to mitigate the effects of the V8 just-in-time compiler (JIT).

Statistical Methods. We make no assumptions on the underlying distribution and provide Tukey boxplots in Figure 1 to 4 to visualize the median and variation of the measured timings. Exact median timings are given in Table 2 to 5. Slowdowns are computed by dividing the median execution times of the measurements to compare. Confidence bounds of this statistic are given by the 2.5-th and 97.5-th percentile of the bootstrap distribution of the computed ratio.

5.2 Overhead for Initial Association

First off, we analyze the initial cost to create associations between a layer and a declarative context. For this purpose, we measure the time to associate 10,000 layers with the same context expression. As a running example, we use the following expression for all benchmarks:

```
() => context.enabled()
```

The variable `context` references an instance of the class `Context` with a single Boolean property representing whether the current is active (full implementation provided in Appendix A). The `enabled` method provides access to this status. Thus, we execute `layer.activeWhile(() => context.enabled())` 10,000 times.

Discussion. As Figure 1 reveals, the imperative implementation has the lowest runtime for creating associations to declarative contexts. This result is to be expected, as the imperative implementation only adds the given layer to a global array when creating the association. In contrast, both reactive strategies have to set up their respective dependency mechanisms in order to monitor for changes. Accordingly, they impose high overhead as shown by the relative slowdowns in Table 2. While the compilation strategy runs the given expression in native JavaScript, the interpretation strategy uses a full-fledged JavaScript-in-JavaScript interpreter to determine

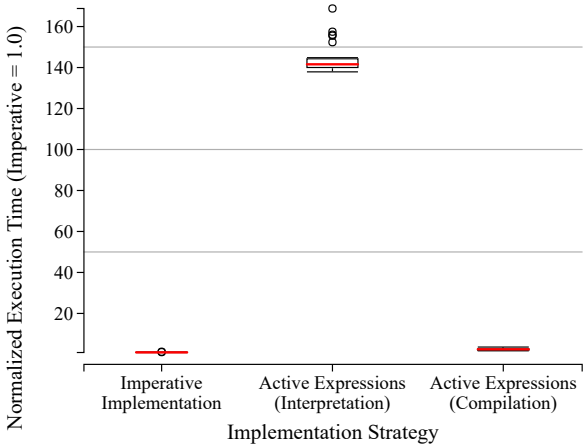

Figure 1: Execution times for declaratively associating 10,000 layers with a context using ILA. The thick red line indicates the median, the upper and lower edges of the box are the second and third quartile and the end of the whiskers are the most outlying values in a 1.5 inter-quartile range distance from the second and third quartile.

Table 2: Benchmark timings and relative slowdowns for declaratively associating 10,000 layers with a context using ILA. Slowdowns given as ratio of medians with 95% confidence intervals.

	timing [ms]	slowdown (vs Imperative)
Imperative	28.09	
Reactive (Interpretation)	3976.06	141.57 [138.74 - 144.77]
Reactive (Compilation)	70.29	2.50 [2.14 - 2.64]

[4]https://github.com/active-expressions/programming-contextjs-plain accessed on April 16th 2017
[5]https://github.com/active-expressions/programming-contextjs-aexpr accessed on April 16th 2017
[6]https://github.com/active-expressions/aexpr-ila-benchmark accessed on April 16th 2017

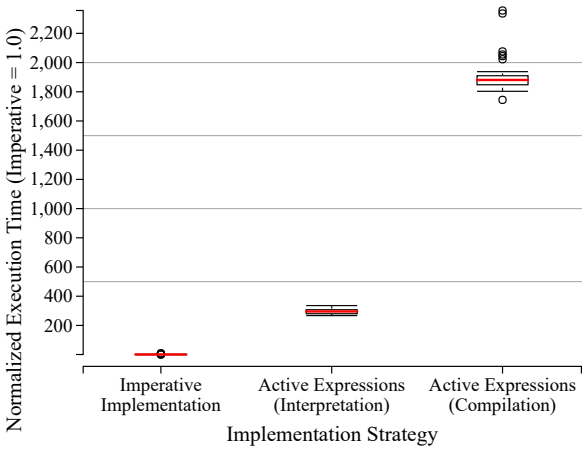

Figure 2: Performance benchmark results for a high ratio of context switches to invocations of context-dependent behavior. The exact ratio is 1000 to 1. All results are normalized by the imperative median. The normalization is the quotient of execution time of the respective implementation and the time of the imperative solution.

Table 3: Benchmark timings and relative slowdowns for frequently switching contexts.

	timing [ms]	slowdown (vs Imperative)
Imperative	0.42	
Reactive (Interpretation)	125.63	295.61 [284.28 - 305.44]
Reactive (Compilation)	799.34	1880.81 [1851.14 - 1924.57]

relevant dependencies, which explains the very high impact of the interpretation strategy. The relative overhead compared to the imperative implementation is subject to the complexity of the given expression.

5.3 Frequent Context Switches

The above measurements show only the initial overhead of the respective implementation strategy. In the following, we identify the overhead that is imposed by implicit context switches. Continuing the previous scenario, we enable and disable a context object implicitly associated with a layer by ILA. We disable and re-enable this context 500 times each. Then, we test the expected semantics by calling context-dependent behavior:

```
expect(adaptee.call()).to.equal(expected);
```

Thus, 1,000 context switches occur before using context-dependent behavior once. We measure the time it takes to repeat this process 100 times.

Discussion. According to the results in Figure 2, both reactive implementations impose a very high overhead for frequent context switches. As highlighted in Table 3, the interpretation strategy is over 2 orders of magnitude slower and the compilation strategy is

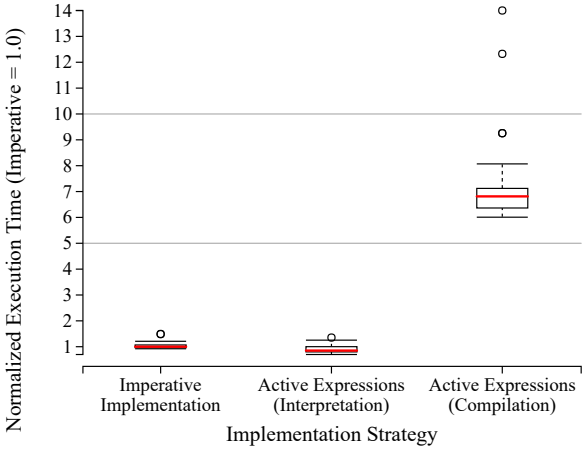

Figure 3: Performance benchmark results for calling context-dependent behavior 1,000 times before switching contexts. All results are normalized by the median of the imperative implementation.

Table 4: Benchmark timings and relative slowdowns for frequently invoking context-dependent behavior.

	timing [ms]	slowdown (vs Imperative)
Imperative	17.63	
Reactive (Interpretation)	14.98	0.85 [0.80 - 0.96]
Reactive (Compilation)	120.12	6.81 [6.39 - 7.08]

over 3 orders of magnitude slower. This high overhead is to be expected, because the imperative implementation does not invoke any additional behavior when switching contexts implicitly. In contrast, both reactive strategies activate the respective layer globally, thus, invalidating the current layer composition. The compilation strategy has a higher overhead than the interpretation strategy, because the applied source code transformation affects all computations. In contrast, the interpretation strategy uses property accessors to punctually intercept the program execution.

5.4 Frequent Message Sends

The previous experiment hints a high overhead for reactive implementations when switching contexts frequently. In the following benchmark, we examine the overhead introduced by each implementation strategy when frequently calling context-dependent behavior. In particular, we switch context 10 times, with invoking context-dependent behavior 1,000 times after each context switch.

Discussion. As Figure 3 reveals, both, the imperative and the interpretation-based implementation have similar performance for frequent invocation of context-dependent behavior. According to Table 4, the reactive variant using the interpretation strategy is slightly faster than the imperative implementation. The reason is that the reactive implementation utilizes the caching mechanism of ContextJS. When calling context-dependent behavior subsequently,

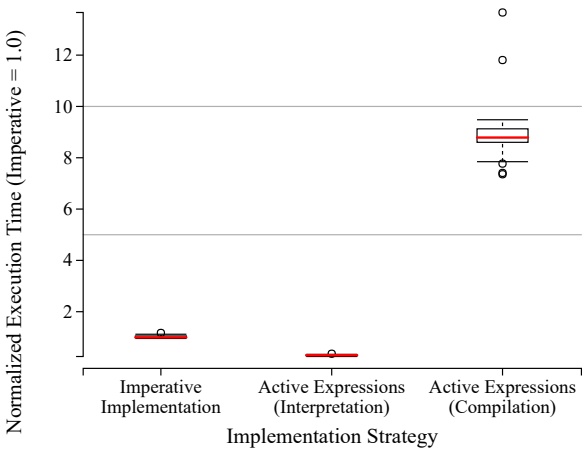

Figure 4: Performance measurements for calling a method with 1,000 implicitly activated layers 1,000 times, normalized by the median of the imperative implementation.

the dispatch mechanism checks whether the layer composition became invalid since the last dispatch. If not, ContextJS can reuse the existing layer composition. Because the reactive implementations update layers on change, no additional checks are required. In contrast, the imperative implementation has to check the current status of each implicitly activated layer on dispatch. Because the imperative implementation cannot anticipate context switches, the layer composition needs to be recomputed for each method invocation.

Even with this conceptual advantage, the compilation strategy is considerably slower than the imperative implementation. This result highlights the high performance overhead imposed by the source code transformation. The reason for this high overhead is that the invocation of detection hooks, such as access to object members, is *highly polymorphic*, and, therefore hard to optimize by JITs. As every access to a property and every call of a member function is wrapped, this strategy can cause severe performance penalties.

5.5 Multiple Layers

So far, we only measured the influence of a single layer on context switches and method invocations. In the following benchmark, we are interested in the influence of a higher number of implicitly activated layers when invoking context-dependent behavior. Thus, we create 1,000 layers, each associated with a different context using ILA. We enable each context and measure the time to invoke the same context-dependent method 1,000 times. Additionally, we check against the expected behavior.

Discussion. As shown in Figure 4 and Table 5, the interpretation-based strategy provides a better performance compared to the imperative implementation in this scenario. This result is to be expected, because the imperative strategy needs to reevaluate the expressions for each implicitly activated layer on each dispatch, as explained in subsection 5.4. Thus, the interpretation strategy performs about three times faster than the imperative implementation.

Table 5: Benchmark timings and relative slowdowns for invoking context-dependent behavior 1,000 times with 1,000 implicitly activated layers.

	timing [ms]	slowdown (vs Imperative)
Imperative	226.72	
Reactive (Interpretation)	68.80	0.30 [0.29 - 0.31]
Reactive (Compilation)	1992.30	8.79 [8.44 - 9.06]

Interestingly, the relative overhead between the imperative implementation and the compilation strategy is similar to the results in subsection 5.4. One possible reason is again the V8 JIT. Because we execute the same code in a loop, the execution always follows the same code pathes. Thus, the assumptions made by the JIT are rarely invalidated. In contrast, the source code transformation of the compilation strategy introduces highly polymorphic code that is rather difficult to optimize.

*
**

The presented benchmark highlights the potential provided by reactive implementations of ILA. In particular, systems with long living layers and frequent invocations of context-dependent behavior might benefit from such an implementation. However, the benchmarks also indicate potential performance problems. In particular, the production of JIT-unfriendly code represents a major source of performance issues. Further studies on the effect of the different implementation variants are needed, especially with regard to real-world COP applications.

6 CONCLUSION

In this paper, we presented two possible implementations for implicit layer activation (ILA): an imperative and a reactive one. Our comparison shows that the reactive implementation matches the declarative semantics of ILA more closely. The runtime overhead of the two implementations highly depends on the specific usage scenario: the imperative implementation is suitable for a system with frequent context switches, while the reactive implementation is more suitable for systems with frequent invocations of context-dependent behavior.

A reactive implementation seems viable and offers interesting possibilities: the eager (de-)activation of layers allows for the integration of ILA with layer life-cycle callbacks and active entities, for example by scoping the effect of constraints [6].

REFERENCES

[1] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. 2009. A Comparison of Context-oriented Programming Languages. In *International Workshop on Context-Oriented Programming (COP)*. ACM, New York, NY, USA, Article 6, 6 pages. DOI: https://doi.org/10.1145/1562112.1562118

[2] Engineer Bainomugisha, Jorge Vallejos, Coen De Roover, Andoni Lombide Carreton, and Wolfgang De Meuter. 2012. Interruptible context-dependent executions: a fresh look at programming context-aware applications. In *Symposium on New Ideas in Programming and Reflections on Software (Onward!), 2012*. ACM, 67–84. DOI: https://doi.org/10.1145/2384592.2384600

[3] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-oriented Programming. *Journal of Object Technology (JOT)* 7, 3 (March 2008), 125–151.

DOI: https://doi.org/10.5381/jot.2008.7.3.a4

[4] Hiroaki Inoue and Atsushi Igarashi. 2016. A library-based approach to context-dependent computation with reactive values: suppressing reactions of context-dependent functions using dynamic binding. In *15th International Conference on Modularity (MODULARITY), 2016*. 50–54. DOI: https://doi.org/10.1145/2892664.2892669

[5] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2015. Generalized layer activation mechanism through contexts and subscribers. In *14th International Conference on Modularity (MODULARITY), 2015*. ACM, 14–28. DOI: https://doi.org/10.1145/2724525.2724570

[6] Stefan Lehmann, Tim Felgentreff, and Robert Hirschfeld. 2015. Connecting Object Constraints with Context-oriented Programming: Scoping Constraints with Layers and Activating Layers with Constraints. In *7th International Workshop on Context-Oriented Programming (COP)*. ACM, Article 1, 6 pages. DOI: https://doi.org/10.1145/2786545.2786549

[7] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. 2011. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Science of Computer Programming (SCICO)* 76, 12 (2011), 1194–1209. DOI: https://doi.org/10.1016/j.scico.2010.11.013

[8] Kim Mens, Rafael Capilla, Nicolás Cardozo, and Bruno Dumas. 2016. A taxonomy of context-aware software variability approaches. In *Workshop on Live Adaptation of Software SYstems (LASSY), March 14 - 18, 2016 (MODULARITY Companion 2016)*. ACM, 119–124. DOI: https://doi.org/10.1145/2892664.2892684

[9] Stefan Ramson and Robert Hirschfeld. 2017. Active Expressions: Basic Building Blocks for Reactive Programming. *The Art, Science, and Engineering of Programming (<Programming>)* 1, Issue 2 (2017). DOI: https://doi.org/10.22152/programming-journal.org/2017/1/12

[10] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. 2007. Context-oriented Programming: Beyond Layers. In *International Conference on Dynamic Languages (ICDL), 2007*. ACM, 143–156. DOI: https://doi.org/10.1145/1352678.1352688

A PERFORMANCE BENCHMARK SOURCE CODE

In the following, we describe important parts of the source code used for the benchmark described in section 5. As described in subsection 5.2, all benchmarks rely on instances of the class `Context` to represent context information. The following source code is the complete implementation of that class:

```
class Context {
  constructor() {
    this.disable();
  }
  enable() { this.state = true; }
  disable() { this.state = false; }
  enabled() { return this.state; }
}
```

Furthermore, the benchmarks described in subsection 5.3, 5.4, and 5.5 use the class `Adaptee`:

```
class Adaptee {
  call() { return -1; }
}
```

In those benchmarks, we adapt the behavior of the `call` method with layers activated through ILA.

The source code of the benchmarks in section 5 is given in Listing 1 to 4. For readability and conciseness, we omitted framework-specific code required by the Karma test runner and the Mocha test framework, as described in subsection 5.1. Whole source code can be found in its corresponding repository[7].

Listing 1: Source code for the benchmark described in subsection 5.2.

```
let bool = false;
let layers = [];

perfTest("Overhead for Initial Association", {
  setupRun() {
    layers.length = 0;
    for(let i = 0; i < 10000; i++) {
      layers.push(new Layer());
    }
  },
  run() {
    layers.forEach(layer => {
      layer.activeWhile(aexpr(() => bool))
    });
  },
  teardownRun() {
    resetLayers(layers);
  }
});
```

[7] *aexpr-ila-benchmark* at commit 6a6395b; https://github.com/active-expressions/aexpr-ila-benchmark accessed on April 16th 2017

Listing 2: Source code for the benchmark described in subsection 5.3.

```
let context, adaptee, layer;

perfTest("Frequent Context Change", {
  setupRun() {
    context = new Context();
    adaptee = new Adaptee();
    layer = new Layer()
        .refineObject(adaptee, {
          call() {
            return 42;
          }
        })
        .activeWhile(aexpr(() => context.
            enabled()));
    },
  run() {
    for(let i = 0; i < 100; i++) {
      for(let j = 0; j < 500; j++) {
        context.disable();
        context.enable();
      }
      expect(adaptee.call()).to.equal(42);
    }
  },
  teardownRun() {
    resetLayers([layer]);
  }
});
```

Listing 3: Source code for the benchmark described in subsection 5.4.

```
let context, adaptee, layer;

perfTest("Frequent Message Sends", {
  setupRun() {
    context = new Context();
    adaptee = new Adaptee();
    layer = new Layer()
        .refineObject(adaptee, {
          call() {
            return 42;
          }
        })
        .activeWhile(aexpr(() => context.
            enabled()));
  },
  run() {
    for(let i = 0; i < 5; i++) {
      context.enable();
      for(let j = 0; j < 1000; j++) {
        expect(adaptee.call()).to.equal(42);
      }
      context.disable();
      for(let j = 0; j < 1000; j++) {
        expect(adaptee.call()).to.equal(-1);
      }
    }
  },
  teardownRun() {
    resetLayers([layer]);
  }
});
```

Listing 4: Source code for the benchmark described in subsection 5.5.

```
let contexts = [], adaptee;
let layers = [];

perfTest("Multiple Layers", {
  setupRun() {
    let numberOfLayers = 1000;

    adaptee = new Adaptee();
    layers.length = 0;
    for(let i = 0; i < numberOfLayers; i++)
        {
      (index => {
        let context = new Context();
        let layer = new Layer()
            .refineObject(adaptee, {
              call() {
                return index;
              }
            })
            .activeWhile(aexpr(() =>
               context.enabled()));

        contexts.push(context);
        layers.push(layer);
      })(i);
    }
    contexts.forEach((context, index) => {
      context.enable();
    });
  },
  run() {
    for(let i = 0; i < 1000; i++) {
      expect(adaptee.call()).to.be.
          greaterThan(-1);
    }
  },
  teardownRun() {
    resetLayers(layers);
  }
});
```