# Efficient Late Binding of Dynamic Function Compositions

Lars Schütze
Chair for Compiler Construction
Technische Universität Dresden
Dresden, Germany
lars.schuetze@tu-dresden.de

Jeronimo Castrillon
Chair for Compiler Construction
Technische Universität Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

## Abstract

Adaptive software becomes more and more important as computing is increasingly context-dependent. Runtime adaptability can be achieved by dynamically selecting and applying context-specific code. Role-oriented programming has been proposed as a paradigm to enable runtime adaptive software by design. Roles change the objects' behavior at runtime and thus allow adapting the software to a given context. However, this increased variability and expressiveness has a direct impact on performance and memory consumption. We found a high overhead in the steady-state performance of executing compositions of adaptations. This paper presents a new approach to use run-time information to construct a dispatch plan that can be executed efficiently by the JVM. The concept of late binding is extended to dynamic function compositions. We evaluated the implementation with a benchmark for role-oriented programming languages leveraging context-dependent role semantics achieving a mean speedup of 2.79× over the regular implementation.

***CCS Concepts*** • **Software and its engineering** → **Software performance**; **Compilers**; *Context specific languages.*

***Keywords*** Role-Oriented Programming, Dispatch Optimization, Virtual Machine

**ACM Reference Format:**
Lars Schütze and Jeronimo Castrillon. 2019. Efficient Late Binding of Dynamic Function Compositions. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19), October 20–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3357766.3359543

## 1 Introduction

Ubiquitous computing leads to new challenges where context-dependent software is more and more important. Developing such software requires approaches that focus on objects, their context-dependent behavior and relations. Object-oriented programming (OOP) is the de facto standard approach to those problems today. This is because of the comprehensibility of object-oriented models and code which enables an intuitive representation of aspects of the real world. That is how classes, objects, functions and inheritance originated. For example, an aspect of the real world is that an object may appear in different roles at different times (i.e., contexts). To reflect the different roles of entities, design patterns have been proposed to achieve separation of concerns [7, 15]. Because these approaches can only compose or decompose in a single dimension by using delegation or inheritance, context-dependent concerns are tangled with the application and scattered over it.

With the advent of aspect-oriented programming (AOP) [33] and context-oriented programming (COP) [28] it was possible to separate behavioral concerns in multiple dimensions. However, the focus of these multi-dimensional separation of concerns (MDSOC) paradigms is on the cross-cutting nature of concerns and adaptation of classes.

Role-oriented programming (ROP) has been proposed as an extension to object-oriented programming to enable adaptive software by design [43, 47]. Classes represent the structural view of the program while context-dependent behavior is encapsulated in separate entities called *roles.* To model context-dependency *compartments* encapsulate roles and represent the context in which these roles can be active. Adaptation is achieved by attaching roles dynamically to objects to superimpose their behavior. Object Teams [23, 24] is one of the most mature role-oriented programming languages. It allows adapting Java programs available in source code or binary form dynamically at runtime.

It is common practice to lower MDSOC mechanisms to object-oriented mechanisms, which results in a verbose description that incurs high runtime overhead [3, 40]. The overhead is especially noticeable in the steady-state performance of dynamic role-oriented programming languages as the behavior of *every* object may be potentially adapted [46]. We argue that the major cause for the overhead is that current translation approaches have not been able to properly close the *semantic gap* between role-oriented mechanisms and object-oriented machine models preventing many possible optimisationz.

To close the semantic gap and reduce verbosity the execution environment must understand the enhanced execution semantics. We address this problem by applying the concept of late binding of virtually dispatched functions to function compositions using exact runtime type information. Functions superimposed by role functions are by default virtual, but may become static until superimposition is released resulting in no subsequent lookups. The runtime provides exact instructions to the execution environment about how to find and execute role functions by composing a directed acyclic graph (DAG) of function calls. This not only allows optimizing role dispatch in the sense of Just-in-Time (JIT) compilation by the execution environment but also improves the runtime and generated code in terms of lookup and reuse.

We demonstrate our approach by extending the static compiler, dynamic compiler and the runtime of Object Teams [23, 24]. We used a typical synthetic benchmark already reported in the literature to compare different language implementations of the role-oriented concept [46]. The benchmark uses many demanding role-oriented programming features such as multiple active contexts, deep roles (i.e., roles play roles), and exchanged function bodies that are not easily built with object-oriented design patterns. Our evaluation features a mean speedup of 2.79× over the original implementation when callsites can be reused and a mean slowdown of 9.76× if there is no possible reuse ever.

The paper introduces in Section 2 concepts similar to roles that could also benefit from the approach and gives an overview of how Object Teams implements roles. In Section 3 late binding for function compositions of role functions is presented and how a directed acyclic graph (DAG) can be built from that composition. The approach is evaluated in Section 4. Related approaches are discussed in Section 5. In Section 6 a conclusion is drawn and future work is highlighted.

## 2 Background

This section motivates multi-dimensional separation of concerns and assesses the cost of related separation of concerns approaches on modern object-oriented VMs. It introduces role-oriented programming and highlights the semantic gap.

### 2.1 Multi-Dimensional Separation of Concerns

Object-orientation excels at representing the structure of a domain but struggles at representing how objects collaborate dynamically. However, in different points in time different parts of the interface of an object is used by multiple other objects. The reason is that classes tend to exhibit behavior (i.e., functions) for multiple concerns and multiple functions have code for multiple concerns. This collaboration-scoped usage cannot be directly represented but is scattered over the program. While design patterns improve the quality of software architectures, they cannot solve the problem of tangling and scattering of concerns satisfactorily. To solve the problem, different approaches for separation of concerns have been proposed with varying degree of granularity ranging from adapting single objects or functions to classes and components.

***Aspect-Oriented Programming*** AOP decomposes cross-cutting concerns to encapsulate each concern separately. *Aspects* encapsulate such concerns and provide expressions to define interceptors, class extensions (inter-type declarations) and its own properties [12]. An aspect can alter the behavior of non-aspect parts of the program called *base* classes by applying *advices* which define the additional behavior. Alternative behavior can be applied at *join points* in the base program including function calls and property access. *Pointcuts* provide predicates that quantify over the set of existing join points and choose the set of join points where the execution of the advice is desired. Aspects are woven into the application using special compilers called *weavers*. Join points where advice invocation code may be woven in are called *join point shadows* [27].

A compiler for the aspect-oriented language consists of a module for evaluating pointcuts and the aforementioned weaver, beside the elements of a traditional compiler. After evaluating a pointcut, the join point shadows are forwarded to the weaver. But at weave-time it cannot be decided for all join point shadows whether pointcuts apply or not. Thus, for some join points advice invocation logic and guards are compiled into the application called *residuals*.

Since virtual machine does not understand aspect semantics, the aspect compiler produces a verbose description of aspects in an object-oriented paradigm which incurs high overhead [19]. The reason is, that function invocations and member accesses are typical locations for join point shadows that will be decorated with advice invocation logic (i.e., residues). Standard object-oriented optimizations such as late binding do not apply to the aspect-oriented execution semantics where advice code is implemented externally to the advised class or object, obscuring control flow. This results in a severe performance penalty ranging from two orders of magnitude in AspectWerkz [11] while Steamloom's [18] performance loss is always less than one decimal power [17].

Moreover, the focus of AOP is cross-cutting, class-wide aspects. For class-wide aspects Steamloom can generate efficient code compiling aspect invocation into the function body which incur minimal overhead. Object-centric or *instance-local* aspects on the other hand introduce performance penalties. A major reason is that Steamloom compiles different versions of the functions each associated to the respective object. Whenever such a function is a call target it

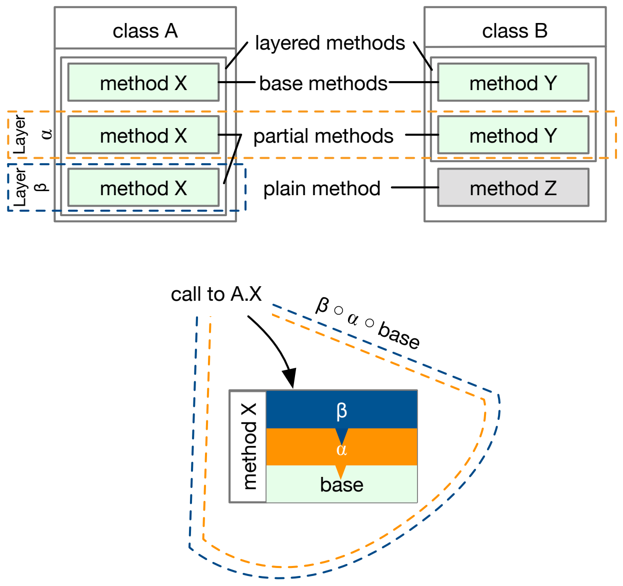


**Figure 1.** An overview of classes and layers in context-oriented programming and their influence on the execution semantics (adapted from [37, Fig.1]).

cannot be inlined because the compiler is not able to decide which version of the function to inline [18].

***Context-Oriented Programming*** COP aims at adapting the behavior of an application to a known context by providing contextual variations. In AOP context-sensitive adaptations have to be embedded in the predicates of a pointcut while COP provides dedicated language support. Similar to AOP the base program is altered at join points with method-level granularity. To achieve contextual variation, *layers* implement the context-dependent behavior in *partial methods*. Variations can replace the original functions, be executed before or after, or use the mechanisms of *proceed* to delegate to the next active layer. Layer activation and deactivation drives the contextual adaptation. This process is called *sideway composition* as the original inheritance hierarchy is enhanced orthogonally at runtime [29]. An overview is given in Figure 1.

In an object-oriented execution model function invocation is understood as a two-dimensional message send to the receiver object consisting of the name of the function to be executed and a list of parameters. In COP, however, the message is extended to four dimensions which also takes the sender object and the context of the actual message sent into account [28]. This means that COP resembles *multiple dispatch* which takes any argument into account. However, *single dispatch* as found in modern object-oriented programming languages such as C++ and Java just takes the runtime type of the receiver into account.

While there have been implementations of multiple dispatch in single dispatched languages like Java [50], context-aware execution semantics is often implemented using imperative control flow in libraries [45]. Layers and partial functions often are implemented with *proxy objects* encoding the semantics instead of the original functions [3]. The result is a severe performance hit introduced by sideway composition with up to 99.7% in ContextJS [37] or 95% in ContextPy [39].

## 2.2 Role-Oriented Programming

Previously discussed approaches achieve separation of concerns at the granularity of functions, classes or modules. The successful adoption of roles in software analysis and modeling [1, 42, 43] led to a demand for programming language support. Roles take the idea further, as "*no object is an island. All objects stand in relationship to others*" [8]. First approaches resulted in the role concept being hidden in the implementation of the host language [7, 14].

Role-oriented programming distinguishes between the base entities themselves and the roles they play in a collaboration. This provides explicit support for object collaboration in a way not normally supported by language features [20]. While base classes stay untouched the behavioral adaptations are implemented in *roles*. Roles encapsulated in *compartments* define clear boundaries when roles can be active enabling context-dependent behavior. Adaptation is achieved by attaching roles to objects whose behavior is superimposed by their roles. There are different variants of role-oriented programming languages providing different sets of role features [35].

In the following we will concentrate on the different approaches emerged in the past years [4–6, 13, 16, 24, 31, 32, 36, 41, 48].

***Contextual Roles with Object Teams*** Object Teams [23, 24] is the most mature role-oriented programming language supporting most of the features attributed to roles [35]. It extends the syntax of Java and introduces a new class keyword named `team class`. Teams can be instantiated to represent context and encapsulate roles. Roles are defined as inner classes of teams and have a slight extension to Java syntax in order to specify the `playedBy` relation from roles to the role-playing base classes. Roles in Object Teams define new or modified behavior of their base classes while their semantics is similar to crosscutting concerns in AOP [21]. Thus, every instance of the base class which plays a role in a team is affected whenever an instance of that team is active.

Object Teams uses a sophisticated mechanism to change the behavior of base classes via roles. While most approaches resort to structural typing where the signature of role functions must be identical to the signature of the base function, Object Teams provides mappings to state the *binding* from role functions to base functions such as argument permutations or arbitrary glue code.

Role functions may have two directions. A *callout* delegates a role function to a base function to reuse behavior of base classes. Role functions that alter the behavior of base classes are called *callins*. The adaptation can take place before, after, or even replace the original function.

There can be multiple active teams that provide roles that have bindings for the same base function. The Object Teams runtime keeps a stack of active teams where the latest activated team has the highest priority. If a callin replaces a base function it can also call back into the original function performing a *base call*. Whenever there is such a base call the next callin from the stack of active teams has precedence over the original function. This results in a recursive application of replace callins until there is no more active callin or there is no more base call executed.

To implement the behavior Object Teams decoupled the definition and resolution of role functions by providing two different compilers. The concept of static and runtime compilation is introduced to account for base classes loaded at runtime (i.e., subclass of base class) that have not been seen by the static compiler.

First, the program is compiled to Java bytecode by the Object Teams compiler[1] whose task is to compile role function definitions inside teams and to type check role bindings. It also generates trampoline functions to implement translation polymorphism [25]. Every callin is associated with a unique identifier that is used within these functions to delegate to the correct role functions for base classes. Teams will be compiled into Java classes and roles will be nested classes of their team. Advanced inheritance mechanisms and role usages such as instance-based scoping in the style of family polymorphism is also handled by the compiler [22]. For every team the compiler generates metadata of the role bindings into the attribute section of the class bytecode that is read by the runtime compiler.

Second, a runtime compiler transforms every loaded class by peeking into the metadata section to identify teams and roles. Whenever a class is loaded that is subject of a callin (i.e., a base class) the respective function body is changed to point to the Object Teams runtime entry point. The original function is tagged with a unique identifier and its body is moved into an artificial function. A generated `switch` instruction uses the identifier to delegate to the original implementation.

The dynamic dispatch to role functions is not resolved by the JVM but verbosely compiled into the *control flow* of the program. The entry point into the Object Teams runtime retrieves all necessary data from the runtime and constructs the call stack. This includes the calling context, i.e., base instance, a copy of the stack of active teams and the actual arguments provided. Furthermore, the generated trampolines use the identifiers to delegate to the right role invocations that implement the proper lifting of base instances to their respective role instance. Executing replace callins results in a recursive descent over the stack of active teams.

On each call to the original function the whole procedure is repeated including preparing the stack, lifting, and delegate to the role functions. As a consequence, embedding the higher semantics of roles into the control flow results in many missed optimizations by the JIT compiler [46]. Recursion reduces the possibilities of method inlining as well as deep chains of method calls. As Figure 2b reveals, the indirection from the original function call to role functions is involved, which results in no possibilities for role function inlining. Function signatures with arrays of `Object` are a fundamental generic way to treat arbitrary sizes and types of arguments. However, as a consequence it incurs the overhead of constructing the arrays and boxing and unboxing primitive types to their equivalent class types (e.g., `int` to `Integer`). As a result Object Teams has a severe performance penalty of 59.9× compared to an object-oriented program using design patterns [46].

***Semantic Gap*** This mismatches between role-oriented semantics and object-oriented VMs discussed above also hold for other MDSOC approaches [3, 40]. While other MDSOC approaches provide external implementations of methods (i.e., partial methods in COP), roles have a deeper relationship superimposing behavior to their players on the level of individual objects. Consequently, playing a role changes the type of the player temporarily resulting in a possible different lookup for every role-playing object [32]. That is fundamentally different to dispatch in the JVM which optimises dispatch on the type of the receiver.

## 2.3 Dynamic Callsites in the JVM

The different kinds of dispatch offered by Java all require to know call targets and types at compile time. Dynamic languages do not know these in advance and have historically been implemented using reflective capabilities of the language. To ease the implementation of dynamic languages executed on the JVM the `invokedynamic` bytecode has been introduced [44, 49].

While other invoke bytecodes require target type, method name, and signature at compile time to statically type check callsites the invokedynamic bytecode just requires a signature. This reduces lookup to signature polymorphism instead of polymorphism on the receiver type. Any invocation has to conform to the signature which is enforced by the JVM.

To link a callsite, user defined code is executed which implements the discovery and returns a callsite object the JVM checks and executes. The user defined function (i.e., bootstrap method) can accept any additional argument. On

[1]The Eclipse Compiler for Object Teams (ecotj) is an extension of and compatible to the Eclipse Java Compiler (ejc).

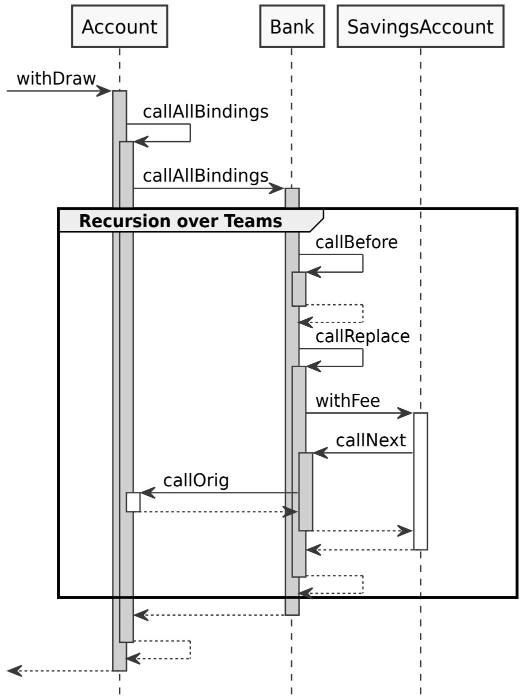
```
class Account {
  void withDraw(float amount) {...}
}

team class Bank {
 class CheckingsAccount
  playedBy Account {...}

 class SavingsAccount
  playedBy Account {

  callin void
   withFee(float amount) {
    base.withFee(FEE * amount);
  }
  withFee <- replace withDraw;
 }
}
```

**(a)** Object Teams source code showing how Accounts can have different behavior when withdrawing money. The `SavingsAccount` adds a fee for each withdraw.



**(b)** The UML sequence diagram of the dispatch in Object Teams. Grey coloured execution occurences mean framework code while white mean user code.

**Figure 2.** An example of how to define roles in Object Teams and how calls will be dispatched highlighting the intermediate functions to implement role dispatch.

subsequent invocations the bootstrap method will not be visited anymore.

Instead of pointing to a single function the callsite can consist of a graph of function invocations building a directed, acyclic graph (DAG). Such a DAG consists of `MethodHandles` that may represent actual functions bound to instances or are unbound placeholders defining the required `MethodType`[2] of objects that are passed in at runtime. Handles can be combined by generated adapters from an API of trusted system code. Whenever such a DAG can be constant folded to the root it can possibly be inlined by the JIT compiler.

Linked callsites may be invalidated which can also be triggered externally. Invalidation is expensive as the JVM has to deoptimize every function the callsite has been inlined.

## 3 Late Binding of Role Dispatch

Efficient execution of role functions requires removing or reducing the verbosity in implementing role dispatch. Current implementations exploit capabilities of the host language only instead the VM or JIT compiler resulting in less to none optimizations. However, a custom virtual machine might hold the best possible speedup but lacks broad applicability as it mandates to use a specific VM.

An alternative is to provide runtime feedback at language level via intrinsics to inform the compiler about possible optimizations that it can not infer by its heuristics such as `promote()` in RPython's VM PyPy [10]. This allows to communicate runtime constants, i.e., values that are dynamic over the course of the program but constant as long as some conditions hold. For example, roles played by an object are constant as long as there is no new role being played or dropped. With runtime feedback the compiler can use the knowledge to do lookup in constant time.

### 3.1 Runtime Feedback in Object Teams

The Object Teams static compiler produces metadata when type checking callins that is compiled into the Java bytecode as class attributes. These are read by the runtime compiler to identify and change base classes and function invocations. However, this metadata can also be used to identify and link callins directly from a callsite.

The stack of active teams is valuable runtime feedback because they provide the callins that change the behavior of base class instances. To include this feedback the code

[2]The signature describes the parameters and return types. It can be understood as `(ArgumentType [,ArgumentType]*)ReturnType` where the first argument type either defines the type of the receiver or the first argument if the function is static.

compiled by the Object Teams compilers needs to be adapted as well as the runtime to forward the values to the callsite. The result is a mutable callsite that can be relinked whenever a change happens to the role-play graph of the application. Otherwise, the callsite can be directly reused without reevaluation.

Prior, the Object Teams compiler enhanced the signature of base functions and callins to pass relevant stack frame data. That data was used to drive the dispatch. By directly linking callins signatures do not need to be enhanced and unnecessary argument conversions may be dropped (i.e., boxing and unboxing). The control-flow dependent nature of role dispatch is exchanged with a compilation strategy handing all optimization potential over to the JIT compiler.

Figure 3 gives an overview of how the runtime feedback influences control flow. The left side shows the twofold initialization phase. In the bootstrap phase the callsite is uninitialized and points to the generic trampoline that bootstraps the callsite by returning a mutable callsite that will be invoked afterwards. This initial bootstrap will happen at most once per callsite. The second phase is the actual initialization of the callsite.

There is no recursive function invocation anymore because the runtime *iterates* over active teams that actually provide adapted behavior and chains their callins appropriately. Thus, the control-flow dependent dispatch can be removed and replaced by a call graph directly observable and walkable by the JIT compiler. The callsite is guarded and will either link to the initialization or due to invalidation link to the relink function again.

In summary, this removes the verbose description of role dispatch and allows the JIT compiler to be able to optimize this function compositions.

### 3.2 Dispatch Plans for Fast Role Dispatch

A *dispatch plan* is a composition of role functions and necessary type conversions in order to generate a DAG at runtime that is directly observable and walkable by the JIT compiler. Callins are composed according to the semantics of Object Teams [26].

The signature (i.e., method type) of a base function without arguments is `(BaseType)ReturnType`. Each callin is signature polymorph to its base function w.r.t. translation polymorphism [25], i.e., `(RoleType)ReturnType`. That is, each base instance must be lifted to its role instance. Lifting can be directly implemented by filtering the callee using the compiler generated lifting function which returns the appropriate role instance.

Figure 4 gives an overview of how the method type of the base function has to be adapted to conform to the method type of a callin. Since role types represent dependent types (i.e., dependent on the team instance), Object Teams stores the binding of base instances to role instances inside the teams. Thus, for each callin `BIND-TEAM` has to be executed resulting in a *bound* method handle capturing the team instance. `FILTER-0` applies the bound lifting to a method handle of the base function and returns a method handle of the role function.

Constructing the DAG requires two steps. The process is highlighted in Figure 5 showing a DAG with a before callin and the original function. First, the dynamic compiler prepares the base function code to deliver the statically available information to the bootstrap method. This metadata is used to identify the base type, required lifting functions and registered callins. Second, the set of active teams is stored in a context object that is connected with the callsite. The DAG is valid as long as there is no change in the activation of teams that contribute callins to the callsite. At runtime, the base instance and its arguments are passed into the callsite and the DAG is executed.

Figure 3b shows the control flow of the actual invocation of the chain of callins of the example code of Figure 2a. For each replace callin there will be a new invokedynamic instruction *callNext*. That is, whenever the replace callin has a base call it will start iterating over active teams stored in the context of the callsite where it stopped in the last iteration. The whole chain will eventually stabilize resulting in a subsequent execution of callins without intervening framework code.

## 4 Evaluation

This section evaluates the runtime performance and characteristics of dispatch plans and compares it with the original implementation of Object Teams.

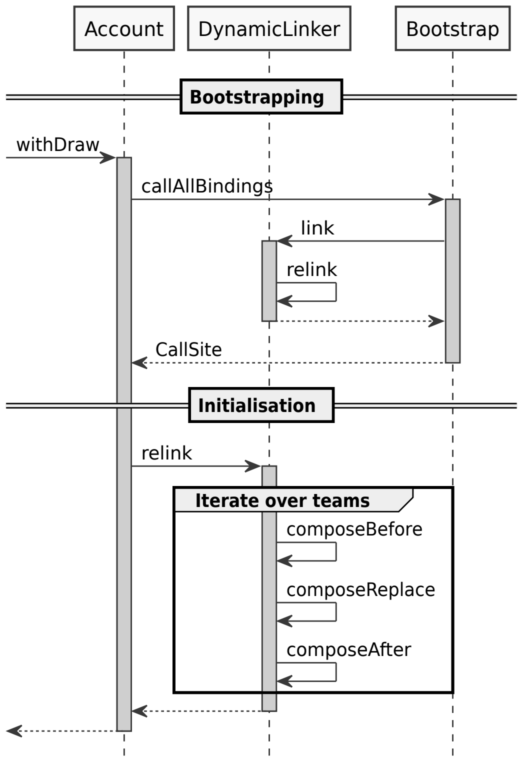
### 4.1 Benchmark Characterization

We used a typical synthetic benchmark already reported in the literature to compare different language implementations of the role-oriented concept [46]. The benchmark uses many demanding role-oriented programming features such as multiple active contexts, deep roles (i.e., roles play roles), and multiple callins that are not easily built with object-oriented design patterns.

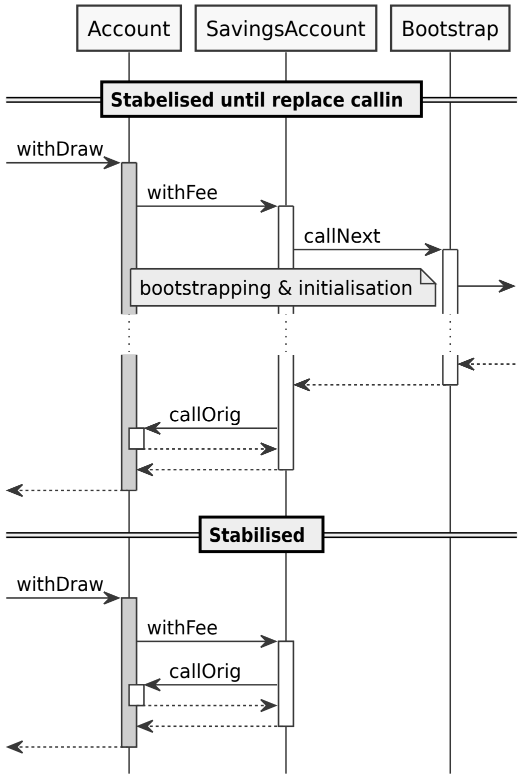
The benchmark describes a simple banking scenario. Persons and accounts are naturals implementing basic behavior. For example, accounts can withdraw and deposit money. A bank is a compartment (i.e., context) where persons can play the role of customers. Accounts play roles that change their behavior such as different fees involved in withdrawing money from a checking account.

The two variations *invalidation* and *reuse* evaluate different characteristics of context-dependent software. While the software must be adaptable it also has to deliver performance whenever there is a period of static behavior.

Figure 6 shows the measured part of the invalidation benchmark. In the most inner loop transactions are modeled as teams are activated and deactivated both triggering



**(a)** Bootstrapping and initialisation of a callsite



**(b)** The dispatch is already calculated up to the first replace callin that may introduce another invokedynamic call

**Figure 3.** A sequence diagram of how to dispatch a role function call in the presence of callins.

$$\frac{\begin{array}{c}(TeamType, BaseType)RoleType \\ team <: TeamType\end{array}}{(BaseType)RoleType} \text{(BIND-TEAM)}$$

$$\frac{\begin{array}{c}(BaseType)RoleType \\ (BaseType[, ArgumentType]^*)ReturnType\end{array}}{(RoleType[, ArgumentType]^*)ReturnType} \text{(FILTER-0)}$$

**Figure 4.** Semantics of lifting a base function to a role function

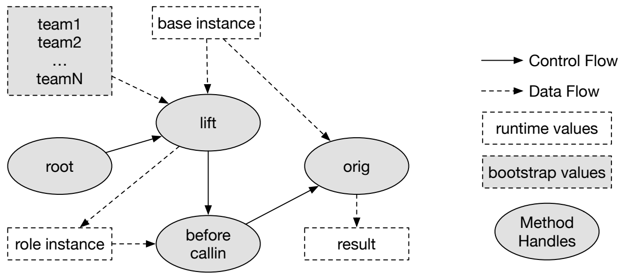


**Figure 5.** A DAG of a before callin and the original function.

an invalidation of the callsite. To evaluate reuse the benchmark does not model transactions but money is transferred directly from one account to the other.

The experiments have been performed on a 3.60GHz Intel Core i7-4790 CPU running Ubuntu 18.04 and OpenJDK JDK 9.0.4. For problem size N there are N persons having 2·N accounts (a CheckingAccount and a SavingsAccount). To reduce variation the benchmark has been repeated 100 times per data point with 3 iterations and each iteration has $N^2$ transactions. We are interested in the overall execution time. To observe if there are scalability problems, we measured with different problem sizes.

## 4.2 Results

The results, depicted Figure 7, show a comparison of the classic Object Teams dispatch and our proposed dispatch plans. The overall picture is that, in average, dispatch plans are slower by 9.76× if the role-play graph is constantly invalidated and built up again. But whenever reuse is possible, an average speedup of 2.79× can be achieved.

```
bank.activate();
for (Account from :
  bank.getSavingAccounts()) {
  for (Account to :
    bank.getCheckingAccounts()) {
    Transaction transaction =
      new Transaction();
    transaction.activate();
    transaction.execute(from,to,amount);
    transaction.deactivate();
  }
}
bank.deactivate();
```

**Figure 6.** The measured part of the Bank benchmark written in Object Teams/Java.

Invalidation is implemented by triggering a costly mechanism of the JVM. Every callsite is guarded by a `SwitchPoint` which can be invalidated by the runtime. In consequence, every callsite that is associated by that team will be invalidated. The benchmark has two invalidations per loop as activating a team changes the dispatch plan of the callsite as well as removing a team.

One main drawback is that dispatch plans currently cannot be reused across different callsites. Another drawback is that the late binding of role function compositions cannot be combined with polymorphic inline caches (PIC) [30] to cache and reuse compositions. The reason can be found in Figure 4 `BIND-TEAM`. To lift a base instance to its role instance, the lifting function of the respective *team instance* has to be called. This is due to the nature of *path-dependent* types. The call stack only provides the `BaseType` instance, while `BIND-TEAM` also requires the `TeamType` instance. Thus, the generated combination to lift the base instance is bound to the particular instance of the team. Because the JVM enforces signature polymorphism, the combination cannot be postponed.

For virtual callsites the JVM degrades dispatch to a lookup if there is too much variation (i.e., megamorphic callsite). The current approach requires a similar degradation mechanism whenever there is too much variation affecting a callsite. A future optimization would be to include a smart mechanism that decides when to use dispatch plans and when to use classical dispatch switching between both variants at runtime.

## 5 Related Work

This section compares the approach presented in the paper with related work of similar approaches.

While there are different approaches of VMs to identify and optimize *hot code* they all use heuristics that can fail leaving potential performance improvements unexplored. In ContextPyPy [39] the capabilities of the meta-tracing JIT compiler is used to mark code that must be promoted for compilation. That is, the steps of the interpreter are recorded. The sequence of instructions is called a *trace*. This way, the dispatch code for layer compositions can be efficiently executed. The VMs used in our approach employ *partial evaluation* [51] if a function has been found hot.

Layer composition in Context JS [34] follows a similar approach of gradually inlining dispatch code. In the beginning wrapper methods are generated that forward to each partial method. On subsequent calls the partial methods are combined into one big method that is ultimately replacing the wrapper at all. This can be compared to the approach presented in the paper as the JIT compiler can constant fold the graph to ultimately inline the role dispatch code.

Steamloom [9] is a virtual machine which understands and optimizes aspect-oriented semantics. The Bytecode Augmentation Toolkit (BAT) [18] can query bytecode and insert and remove instructions from code. Steamloom understands the extended set of bytecodes and generates optimized aspect invocation code. Methods affected by aspects will be generated ad-hoc and replace existing versions of these methods. This delivers always the best performance without unnecessary code. Whenever there are instance-local aspects that only apply to single objects the affected methods will not be able to be inlined anymore as the compiler cannot decide which version to chose. Our approach is not affected by this limitation. First, roles always represent instance-local aspects because of their close relation to the playing objects. Second, our approach is concerned with the callsites themselves and inlining the dispatch code to role functions. If the enclosing method is inlined elsewhere by the VM is out of our scope.

Layered method dispatch with invokedynamic [2] comes closest to our approach presented in this paper. They do not construct a graph of method calls but construct the composition of layers by storing the handles to each partial method in a list. For each partial method there is its own callsite object that will be swapped in and out when layers get activated and deactivated or a `proceed` is called which returns the next callsite object pointing to the next partial method. In contrast, our approach allows to communicate the whole execution to the VM.

Dispatch Chains [38] is a generalization of polymorphic inline caches (PIC). Dynamic languages often use reflective capabilities of the VM to implement the flexible dynamic dispatch mechanisms but sacrificing performance. Dispatch chains can be used with meta-tracing JIT compilers or partial evaluation and perform equally well for both approaches. Function calls are defined by the name of the function, the argument list and the type of the callee. In a dynamic language these can change over the course of the program exhibiting

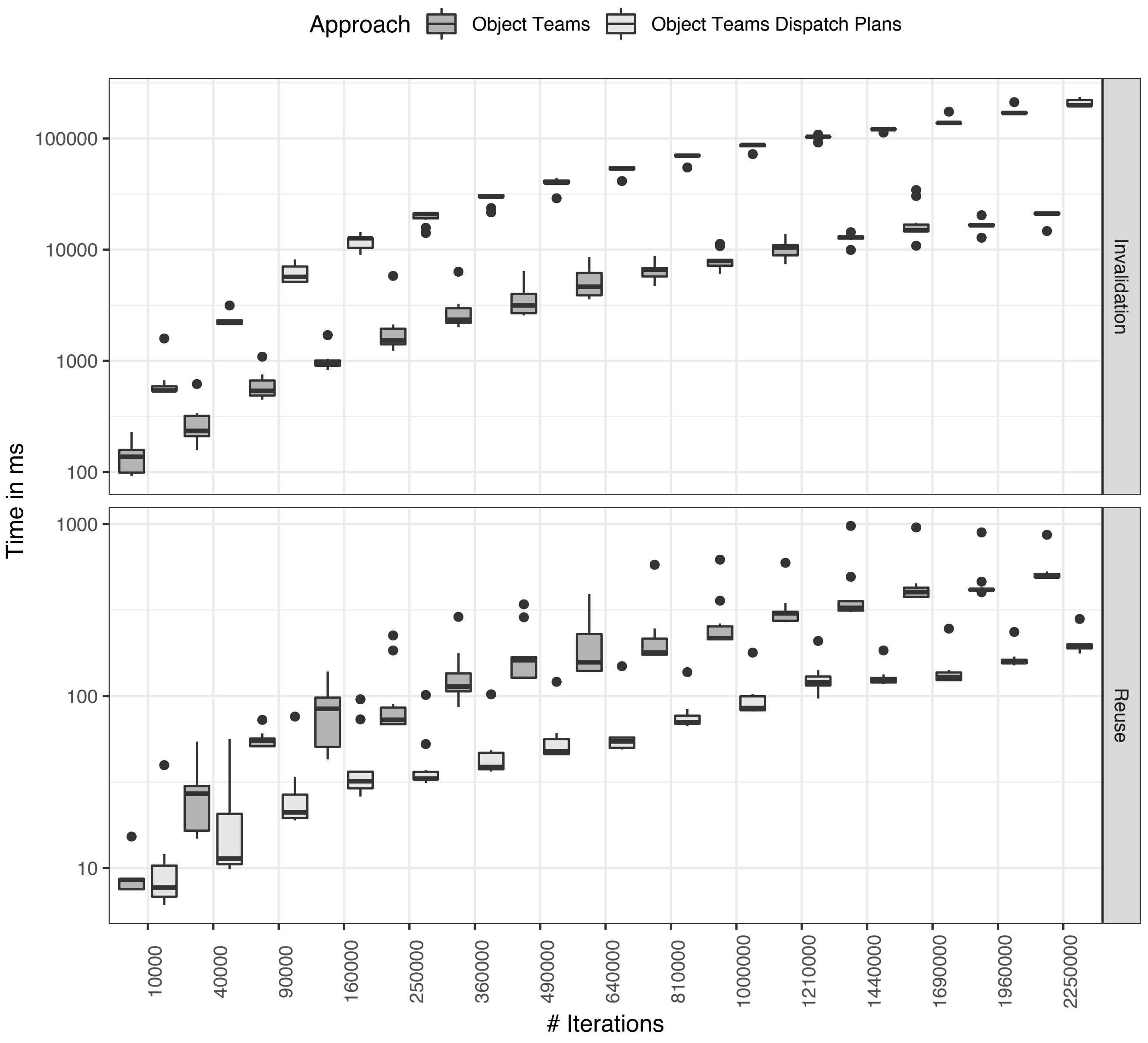


**Figure 7.** A boxplot with whiskers showing the results of classic Object Teams dispatch and Object Teams with Dispatch Plans. Runtime in ms with logarithmic scale while iterations increase quadratic. Dots represent outliers.

many types of callees. While PIC is limited to one family of an inheritance hierarchy, dispatch chains are able to build chains on different levels such as the name of the function or the type of the callee. However, they do not cope with the semantics of role functions. Especially, the composition of callins of different teams for one function that are able to replace and call recursively deeper into the hierarchy is better represented in a graph structure and iterative approach as presented in this paper.

## 6 Conclusions

Context-dependent software is more and more important. The role concept is a promising candidate to build context-dependent software as contexts and behavioral adaptations can be directly represented in the language. This allows for a flexible software development process as well as a better context-dependent software. In general, however, role language implementations suffer from a high runtime overhead when dispatching compositions of adaptations. In this paper we analyzed this for the concrete case of Object Teams, the most mature role-oriented programming language. To reduce the overhead in Object Teams, we propose constructing explicit *dispatch plans*. By using the concept of late binding and signature polymorphism, compositions of role functions can be built at runtime. Runtime feedback helps communicating values to the JIT compiler normally not identified by the heuristics leading to bigger optimization potential. This effectively reduces the semantic gap between role-oriented mechanisms and object-oriented machine models. For a demanding role-based benchmark, we showed that an average speedup of 2.79× can be achieved in the best case (high reuse case). In the worst case, with repetitive callsite invalidation, we observed an average slowdown of 9.76×. We are confident that it is possible to identify when the worst case appears, so that the more efficient implementation of the dispatch can be decided at runtime.

In future work, we will further analyze the effect of dispatch plans on a larger set of benchmarks under real world conditions. We will also look into how path-dependent types can be fit into the signature polymorphic approach. As a result a polymorphic inline cache could be build which will speedup the approach dramatically.

## Acknowledgments

This work has been funded by the German Research Foundation within the Research Training Group Role-based Software Infrastructures for continuous-context-sensitive Systems (GRK 1907).

## References

[1] Egil P. Andersen and Trygve Reenskaug. 1992. System Design by Composing Structures of Interacting Objects. In *ECOOP '92 European Conference on Object-Oriented Programming*. Vol. 615. Springer-Verlag, Berlin/Heidelberg, 133–152. https://doi.org/10.1007/BFb0053034

[2] Malte Appeltauer, Michael Haupt, and Robert Hirschfeld. 2010. Layered Method Dispatch with INVOKEDYNAMIC: An Implementation Study. ACM Press, 1–6. https://doi.org/10.1145/1930021.1930025

[3] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. 2009. A Comparison of Context-Oriented Programming Languages. ACM Press, 1–6. https://doi.org/10.1145/1562112.1562118

[4] Matteo Baldoni, Guido Boella, and Leendert van der Torre. 2006. Roles as a Coordination Construct: Introducing powerJava. *Electronic Notes in Theoretical Computer Science* 150, 1 (March 2006), 9–29. https://doi.org/10.1016/j.entcs.2005.12.021

[5] Stephanie Balzer, Thomas R. Gross, and Patrick Eugster. 2007. A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships. In *ECOOP 2007 – Object-Oriented Programming*. Vol. 4609. Springer Berlin Heidelberg, Berlin, Heidelberg, 323–346. https://doi.org/10.1007/978-3-540-73589-2_16

[6] Fernando Sérgio Barbosa and Ademar Aguiar. 2012. Modeling and Programming with Roles: Introducing JavaStage. *Frontiers in Artificial Intelligence and Applications* (2012), 124–145. https://doi.org/10.3233/978-1-61499-125-0-124

[7] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. 1997. The Role Object Pattern. In *Proceedings of the 1997 Conference on Pattern Languages of Programs (PLoP 97)*.

[8] Kent Beck and Ward Cunningham. 1989. A Laboratory for Teaching Object-Oriented Thinking. In *Object-Oriented Programming Systems, Languages and Applications Conference*. ACM, New Orleans, LA, 1–6.

[9] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. 2004. Virtual Machine Support for Dynamic Join Points. ACM Press, 83–92. https://doi.org/10.1145/976270.976282

[10] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems - ICOOOLPS '11*. ACM Press, Lancaster, United Kingdom, 1–8. https://doi.org/10.1145/2069172.2069181

[11] Jonas Bonér. 2004. AspectWerkz – Dynamic AOP for Java.

[12] Johan Brichau, Michael Haupt, Nicholas Leidenfrost, Awais Rashid, Lodewijk Bergmans, Tom Staijen, Istvan Nagy, Anis Charfi, Christoph Bockisch, Ivica Aracic, Vaidas Gasiunas, Klaus Ostermann, Lionel Seinturier, Renaud Pawlak, Mario Südholt, Davy Suvee, Theo D'Hondt, Peter Ebraert, Wim Vanderperren, Shiu Lun Tsang, Monica Pinto, Lidia Fuentes, Eddy Truyen, Adriaan Moors, Maarten Bynes, Wouter Joosen, Shmuel Katz, Adrian Coyler, Helen Hawkins, Andy Clement, and Olaf Spinczyk. 2005. *Survey of Aspect-Oriented Languages and Execution Models*. AOSD-Europe-VUB-01 Deliverable D12. Vrije Universiteit Brussel.

[13] Chengwan He, Zhijie Nie, Bifeng Li, Lianlian Cao, and Keqing He. 2006. Rava: Designing a Java Extension with Dynamic Object Roles. IEEE, 7 pp.–459. https://doi.org/10.1109/ECBS.2006.57

[14] Martin Fowler. 1997. Dealing with Roles. In *Proceedings of the 1997 Conference on Pattern Languages of Programs (PLoP 97)*.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass.

[16] Kasper B. Graversen and Kasper Østerbye. 2003. Implementation of a Role Language for Object-Specific Dynamic Separation of Concerns. In *SPLAT: Software Engineering Properties of Languages for Aspect Technologies*.

[17] Michael Haupt and Mira Mezini. 2004. Micro-Measurements for Dynamic Aspect-Oriented Systems. In *Object-Oriented and Internet-Based Technologies*. Vol. 3263. Springer Berlin Heidelberg, Berlin, Heidelberg, 81–96. https://doi.org/10.1007/978-3-540-30196-7_7

[18] Michael Haupt, Mira Mezini, Christoph Bockisch, Tom Dinkelaker, Michael Eichberg, and Michael Krebs. 2005. An Execution Layer for Aspect-Oriented Programming Languages. ACM Press, 142. https://doi.org/10.1145/1064979.1065000

[19] Michael Haupt and Hans Schippers. 2007. A Machine Model for Aspect-Oriented Programming. In *ECOOP 2007 – Object-Oriented Programming*, Vol. 4609. Springer Berlin Heidelberg, Berlin, Heidelberg, 501–524. https://doi.org/10.1007/978-3-540-73589-2_24

[20] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. 1990. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. *ACM SIGPLAN Notices* 25, 10 (Oct. 1990), 169–180. https://doi.org/10.1145/97946.97967

[21] Stephan Herrmann. 2003. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World*. Vol. 2591. Springer Berlin Heidelberg, Berlin, Heidelberg, 248–264. https://doi.org/10.1007/3-540-36557-5_19

[22] S. Herrmann. 2004. Sustainable Architectures by Combining Flexibility and Strictness in Object Teams. *IEE Proceedings - Software* 151, 2 (2004), 57. https://doi.org/10.1049/ip-sen:20040168

[23] Stephan Herrmann. 2005. Programming with Roles in ObjectTeams/-Java. In *AAAI Fall Symposium on Roles- an Interdisciplinary Perspective*.

[24] Stephan Herrmann. 2007. A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java. *Applied Ontology* 2, 2 (2007), 181–207.

[25] Stephan Herrmann, Christine Hundt, and Katharina Mehner. 2004. *Translation Polymorphism in Object Teams.* Technical Report Bericht-Nr. 2004/05. Technische Universität Berlin, Berlin.

[26] Stephan Herrmann, Christine Hundt, and Marco Mosconi. 2011. OT/J Language Definition v1.3.

[27] Erik Hilsdale and Jim Hugunin. 2004. Advice Weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development - AOSD '04.* ACM Press, Lancaster, UK, 26–35. https://doi.org/10.1145/976270.976276

[28] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-Oriented Programming. *The Journal of Object Technology* 7, 3 (2008), 125. https://doi.org/10.5381/jot.2008.7.3.a4

[29] Robert Hirschfeld, Hidehiko Masuhara, and Atsushi Igarashi. 2013. L: Context-Oriented Programming with Only Layers. In *Proceedings of the 5th International Workshop on Context-Oriented Programming - COP'13.* ACM Press, Montpellier, France, 1–5. https://doi.org/10.1145/2489793.2489797

[30] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *ECOOP'91 European Conference on Object-Oriented Programming.* Vol. 512. Springer-Verlag, Berlin/Heidelberg, 21–38. https://doi.org/10.1007/BFb0057013

[31] Tetsuo Kamina and Tetsuo Tamai. 2009. Towards Safe and Flexible Object Adaptation. ACM Press, 1–6. https://doi.org/10.1145/1562112.1562116

[32] Tetsuo Kamina and Tetsuo Tamai. 2010. A Smooth Combination of Role-Based Language and Context Activation. In *FOAL 2010 Proceedings.*

[33] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *ECOOP'97 — Object-Oriented Programming.* Vol. 1241. Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242. https://doi.org/10.1007/BFb0053381

[34] Robert Krahn, Jens Lincke, and Robert Hirschfeld. 2012. Efficient Layer Activation in Context JS. IEEE, 76–83. https://doi.org/10.1109/C5.2012.20

[35] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. 2014. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Software Language Engineering.* Vol. 8706. Springer International Publishing, Cham, 141–160. https://doi.org/10.1007/978-3-319-11245-9_8

[36] Max Leuthäuser and Uwe Aßmann. 2015. Enabling View-Based Programming with SCROLL: Using Roles and Dynamic Dispatch for Etablishing View-Based Programming. ACM Press, 25–33. https://doi.org/10.1145/2802059.2802062

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

[38] Stefan Marr, Chris Seaton, and Stéphane Ducasse. 2015. Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and without Compromises. ACM Press, 545–554. https://doi.org/10.1145/2737924.2737963

[39] Tobias Pape, Tim Felgentreff, and Robert Hirschfeld. 2016. Optimizing Sideways Composition: Fast Context-Oriented Programming in ContextPyPy. ACM Press, 13–20. https://doi.org/10.1145/2951965.2951967

[40] Andrei Popovici, Thomas Gross, and Gustavo Alonso. 2002. Dynamic Weaving for Aspect-Oriented Programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development - AOSD '02.* ACM Press, Enschede, The Netherlands, 141. https://doi.org/10.1145/508386.508404

[41] Michael Pradel and Martin Odersky. 2008. SCALA ROLES A Lightweight Approach towards Reusable Collaborations. In *ICSOFT 2008 - Proceedings of the 3rd International Conference on Software and Data Technologies.* 13–20.

[42] Trygve Reenskaug, Per Wold, and Odd Arilc Lehne. 1996. *Working with Objects: The OOram Software Engineering Method.* Manning, Greenwich. OCLC: 613322213.

[43] Dirk Riehle and Thomas Gross. 1998. Role Model Based Framework Design and Integration. ACM Press, 117–133. https://doi.org/10.1145/286936.286951

[44] John R. Rose. 2009. Bytecodes Meet Combinators: Invokedynamic on the JVM. ACM Press, 1–11. https://doi.org/10.1145/1711506.1711508

[45] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. 2012. Context-Oriented Programming: A Software Engineering Perspective. *Journal of Systems and Software* 85, 8 (Aug. 2012), 1801–1817. https://doi.org/10.1016/j.jss.2012.03.024

[46] Lars Schütze and Jeronimo Castrillon. 2017. Analyzing State-of-the-Art Role-Based Programming Languages. In *Proceedings of the International Conference on the Art, Science, and Engineering of Programming - Programming '17.* ACM Press, Brussels, Belgium, 1–6. https://doi.org/10.1145/3079368.3079386

[47] Friedrich Steimann. 2000. On the Representation of Roles in Object-Oriented and Conceptual Modelling. *Data & Knowledge Engineering* 35, 1 (Oct. 2000), 83–106. https://doi.org/10.1016/S0169-023X(00)00023-9

[48] Nguonly Taing, Thomas Springer, Nicolás Cardozo, and Alexander Schill. 2016. A Dynamic Instance Binding Mechanism Supporting Run-Time Variability of Role-Based Software Systems. ACM Press, 137–142. https://doi.org/10.1145/2892664.2892687

[49] Christian Thalinger and John Rose. 2010. Optimizing Invokedynamic. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java - PPPJ '10.* ACM Press, Vienna, Austria, 1. https://doi.org/10.1145/1852761.1852763

[50] David Ungar, Harold Ossher, and Doug Kimelman. 2014. Korz: Simple, Symmetric, Subjective, Context-Oriented Programming. ACM Press, 113–131. https://doi.org/10.1145/2661136.2661147

[51] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. ACM Press, 662–676. https://doi.org/10.1145/3062341.3062381