Incremental Overload Resolution in Object-Oriented Programming Languages

Tamás Szabó
itemis, Germany
Delft University of Technology, Netherlands

Edlira Kuci
Darmstadt University of Technology, Germany

Matthijs Bijman
Delft University of Technology, Netherlands

Mira Mezini
Darmstadt University of Technology, Germany

Sebastian Erdweg
Delft University of Technology, Netherlands

Abstract

Object-oriented programming languages feature static and dynamic overloading: Multiple methods share the same name but provide different implementations. Dynamic overloading (also know as dynamic dispatch) is resolved at run time based on the type of the receiver object. In this paper, we focus on static overloading in Featherweight Java, which is resolved at compile time based on the types of the method arguments.

The challenge this paper addresses is to incrementalize static overload resolution in IDEs. IDEs resolve overloaded methods for the developer to help them discern which implementation a method call refers to. However, as the code changes, the IDE has to reconsider previously resolved method calls when they are affected by the code change. This paper clarifies when a method call is affected by a code change and how to re-resolve method calls with minimal computational effort. To this end, we explore and compare two approaches to incremental type checking: co-contextual type checking and IncA.

Keywords Overload resolution, Type checking, Static analysis, Incremental computing, Object-oriented languages

ACM Reference Format:
Tamás Szabó, Edlira Kuci, Matthijs Bijman, Mira Mezini, and Sebastian Erdweg. 2018. Incremental Overload Resolution in Object-Oriented Programming Languages. In *Proceedings of 20th Workshop on Formal Techniques for Java-like Programs (FTfJP'18).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.475/123_4

FTfJP'18, July 2018, Amsterdam, Netherlands

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

1 Introduction

Static method overloading is an essential feature of object-oriented programming languages. Method overloading allows developers to provide the same method name to multiple method implementations. Developers typically use method overloading for one of two reasons. First, they use it to provide a more flexible interface to a single functionality by accepting different kinds of parameter types. A common example of this is constructor overloading, which allows users of a class to construct a class instance in different ways:

```
class Wheel {
  public Wheel(int sizeInch, Tire tire) { ... }
  public Wheel() { this(29, Tire.DEFAULT); }
}
```

Second, developers use method overloading to select one of multiple functionalities by dispatching over the number and types of arguments. For example, method overloading is a key component for the visitor pattern:

```
interface BikePartVisitor {
  void visit(Part part); // fall-back method
  void visit(Frame frame);
  void visit(Wheel wheel);
  void visit(Brake brake);
  // add specialized handlers for other parts as needed
}
```

The downside of method overloading is that it becomes difficult for developers to reason about method calls. In particular, the name of the called method does not provide sufficient information to understand which implementation will be invoked. While this is also true for dynamic dispatch via inheritance and overriding, overloaded methods are not governed by the Liskov Substitution Principle, such that a developer has to inspect the exact dispatch target to anticipate the effect of a call. To resolve a call to an overloaded method `e.m(a1,...,an)` in Featherweight Java (FJ), a developer has to take all of the following information into account:

- The compile-time type `C` of `e`: All methods named `m` in `C` are candidates for the resolution.

- The superclasses of `C`: All methods named `m` in superclasses of `C` are candidates for the resolution.
- The number of arguments `a1,...,an`: Only methods with `n` parameters are candidates for the resolution.
- The compile-time types `C1,...,Cn` of the arguments `a1,...,an`: Only methods that accept `C1,...,Cn` or superclasses thereof are candidates for the resolution.
- The distance in the class hierarchy between the argument types `C1,...,Cn` and the parameter types `D1,...,Dn` of candidate methods: Select the unique candidate with minimal aggregated distance or report an error if no unique minimal candidate exists.

Since this is a lot of information for a developer to trace manually, IDEs mirror the compiler's behavior and resolve overloaded methods automatically. Most editors handle overload resolution together with name resolution during type checking, since the target method determines the type of the method invocation.

Like all IDE services, overload resolution has to be incremental and react to source-code changes. That is, when a code change occurs, the IDE has to incrementally update previously computed overload resolution results: It has to discard affected resolution results and recompute them. In prior work, we developed an approach for incremental type checking called co-contextual type checking [6, 9] that we used to develop efficient incremental checkers for PCF and FJ [8]. However, when considering FJ with overloading, co-contextual type checking failed to yield good incremental performance. Our first contribution in this paper is a detailed analysis of why co-contextual type checking fails to incrementalize overload resolution (Section 2).

Our second contribution is to clarify when a method call is affected by a code change and how to re-resolve affected method calls (Section 3). To this end, we study incremental type checking of FJ with overloading in a second framework, called IncA [15]. IncA is a framework for incremental static analysis based on an efficient incremental Datalog solver. In contrast to co-contextual type checking, IncA allows us to model dependencies between source-code artifacts precisely. This yields good incremental performance, which we evaluate through micro benchmarks. Both IncA and our overload resolution in IncA are available open-source.[1][2]

2 Co-Contextual Overload Resolution Does Not Work

Co-contextual type checking is a technique for incrementalizing type checkers. Its core strategy is to eliminate dependencies on the typing context, making type-checker results reusable in spite of context changes.

[1] https://github.com/szabta89/IncA
[2] https://github.com/seba–/inca-experiments

For example, the expression `v.visit(new Wheel())` has three context dependencies, namely on the binding of `v`, the declaration of `visit`, and the declaration of the nullary constructor of `Wheel`. A traditional type checker has to reconsider the expression when any of the three context dependencies changes, and such changes can produce ripple effects. In contrast, a co-contextual type checker assigns a typing to the expression without knowing the context. Whenever a traditional type checker looks up information in the context, the co-contextual checker generates a context requirement instead. That is, the typing judgment has the signature `typeOf: Exp -> (Type, ContextReqs, Constraints)`. An expression then is well-typed in any context that satisfies all context requirements and constraints (type equality, subtyping). Our example expression is well-typed in contexts that bind `v` to some type `U`, that declare a method `U.visit(Wheel)`, and that declare a nullary constructor `Wheel.init()`. Note that co-contextual typings do not need to be principal [16].

Co-contextual type checking can yield good incremental performance because the typing of expressions is not affected by context changes. For example, when changing the declaration of `v` or the nullary constructor of `Wheel`, the typing of our example expression remains valid. Thus, a co-contextual type checker only has to reconsider whether the new context satisfies the existing context requirements and constraints. Technically, a co-contextual type checker traverses the abstract syntax tree (AST) of the subject program bottom-up, computing the typing for each node by merging the requirements and constraints of sub-nodes. The key to *efficient* incremental co-contextual type checking is to discharge context requirements and constraints as early as possible (close to the leaves) and to represent them compactly otherwise. This way, it becomes cheap to embed an expression into varying contexts and to check whether these contexts satisfy the remaining requirements and constraints.

In previous work, we have demonstrated that the co-contextual approach yields good incremental performance for PCF with records, parametric polymorphism, and subtyping [6] and for FJ [9]. However, when adding overloading to FJ, co-contextual type checking fails to yield satisfactory performance. In the remainder of this section, we analyze what makes overloading particularly challenging and why co-contextual type checking cannot cope with it.

Our type checker has to follow three steps to resolve a call `e.m(a1,...,an)` to an overloaded method [3]:

1. Find candidate method declarations in the type of `e` and its superclasses. Candidates must have name `m` and accept `n` arguments.
2. Compute the distance between each candidate target and the call. The distance of a candidate is the sum of the distances of the parameters. The distance of a parameter is the distance of the parameter type and the corresponding argument type in the class hierarchy.

3. Select the unique candidate with minimal distance or report an error if no unique minimal candidate exists.

Overload resolution is difficult to incrementalize because it depends on non-local information. First, to find candidates, we need to determine and inspect the type of the receiver `e` and all its superclasses. This introduces a lot of dependencies from the call to the class table: A change to any of these types (e.g., edit extends clause, adding/removing method) affects overload resolution. Second, to compute the distance between a candidate and the call, we additionally need information about the subclasses of all parameter types that occur in the candidate. This introduces even more dependencies from the call to the class table: A change to the parameter list of a candidate, the argument list of a call, or the subclasses of a parameter type affects overload resolution. To incrementalize overload resolution efficiently, we need to find a way to re-resolve calls to overloaded methods with minimal computational effort.

Unfortunately, co-contextual type checking cannot provide efficient incremental overload resolution. The reason is that overload resolution requires global information (over the whole program) to make computational progress, but co-contextual type checking only provides local information when checking a node. Consequently, when implementing overload resolution co-contextually, the checker has to defer many computations until most of the program was traversed. In particular, there is not enough information available locally at a node to find the candidate method declarations or to compute their distance from the call. As a result, a co-contextual type checker cannot resolve overloaded methods until it has seen most of the class table, and any change to any of the classes will trigger a complete re-resolution: finding candidates, computing their distance, selecting the minimal candidate. We conclude that the co-contextual approach does not yield efficient incremental overload resolution.

3 Overload Resolution in IncA

After we failed to efficiently incrementalize overload resolution with the co-contextual approach, we turned our attention to the IncA incremental static analysis framework for the following reasons. First, IncA has been successfully used to incrementalize data-flow analyses to be used in interactive applications [15]. The key idea of IncA is to use relations to encode program analyses, and IncA allows precise handling of data dependencies between different parts of an analysis. This is exactly what the co-contextual approach failed to do. Second, IncA comes with a domain-specific language (DSL) for analysis definition, so we can immediately focus on overload resolution instead of the challenging technicalities of incrementalization. Finally, the DSL's expressive power seemed sufficient for expressing overload resolution in FJ. Our experiment with IncA was successful because we achieved tens of millisecond update times on considerable workloads. In this section, we review the IncA DSL and backend. Then, we make use of IncA to implement overload resolution. Finally, we present the results of our performance benchmarking.

IncA DSL The IncA DSL has its roots in Datalog, which is a logic programming language widely used for program analyses [7, 14]. There are several syntactical differences in IncA compared to Datalog, but it is common between the two languages that an analysis is ultimately encoded through inter-connected relations on the AST of the subject program. We demonstrate the IncA DSL and the relational nature through an example: Given an AST `node` in an FJ program, we want to look up its container method. We implement a `getContainerMethod` function in IncA as follows:

```
def getContainerMethod(node : Node) : Method = {
  parent := node.parent
  alt {
    assert parent instanceOf Method
    return parent
  } or {
    return getContainerMethod(parent)
  }
}
```

The function walks up along ancestors in the AST starting from `node` until we hit the container method. In Line 2, we access the parent of `node` with the `.parent` navigation, and we store the result in the `parent` variable. We use the `alt` construct to define two alternative computations for obtaining the output. Either parent is already a `Method`, and we return immediately, or we recurse on `parent`. The `assert` in the analysis code acts as a constraint: If it is satisfied for the given input, the switch alternative produces an output, otherwise, no output is returned from the alternative. This also makes sure that no output is returned for nodes that are not contained under methods. The distinction between input and output parameters allows natural forward or backward style analysis definitions [15]. However, parameters lose their direction in the backend, and the function becomes a relation `getContainerMethod` $\subseteq$ `Node` $\times$ `Method`, while the analysis result consists of tuples in this relation. A tuple `(n,m)` $\in$ `getContainerMethod` means that `m` is the container method of node `n`.

In addition to relating program elements from the AST, the IncA DSL allows us to define custom lattices and aggregations over lattice values in the analysis code. This is important for the overload resolution because we *compute* and *aggregate* numerical distances between arguments and parameters as the analysis runs. To this end, we define a `Nat` lattice in IncA representing the natural numbers and operations over lattice values as follows:

```
lattice Nat {
 constructors { Value(int) | Top }
 def bot() : Nat = return Value(0)
 def top() : Nat = return Top
 def leq(l : Nat, r : Nat) : boolean = {
  match (l, r) with {
   case (Value(v1), Value(v2)) => return v1 <= v2
   case _ => return false
 }}
 def glb(l : Nat, r : Nat) : Nat = {
  match (l, r) with {
   case (Value(v1), Value(v2)) => return v1<=v2 ? l:r
   case (_, Top) => return l
   case (Top, _) => return r
 }}
 def sum(l : Nat, r : Nat) : Nat = {
  match (l, r) with {
   case (Value(v1), Value(v2)) => return Value(v1 + v2)
   case _ => return Top
 }}
 ...
}
```

The `Nat` lattice has two kinds of lattice values: `Value` simply wraps a Java `int`, while `Top` represents ∞. The implementation of the lattice operations is straightforward. We will use them later when implementing overload resolution. For more details on the IncA DSL, we refer the reader to [15]. In the following, we show how the incremental solver of IncA computes and incrementally maintains an analysis result.

IncA Solver IncA translates analysis functions into a computation network. For example, Figure 1C shows a computation network built for the `getContainerMethod` function. The network has two kinds of nodes. First, *Input nodes* (grey) let IncA access the AST, e.g. `Method` lists all methods, while `Parent` enumerates the binary parentship between AST nodes. Second, *computation nodes* (white) combine the results of nodes that they depend on through relational algebra, e.g. by joining, filtering, or computing union. For example, `getContainerMethod_alt1` filters, `getContainerMethod_alt2` joins, and `getContainerMethod` computes union based on the results of the previous two.

The computation network immediately lends itself to efficient evaluation. Every node in the network caches its result tuples. The unit of incrementalization is a tuple in a relation: A change in the subject program triggers tuple insertions and deletions at input nodes. From here, a change propagation takes place: Each node updates its cache and propagates the *delta* on the outgoing edges to dependent nodes. The change propagation goes on as long as some computation node changes, that is, until the IncA solver reaches a fixpoint. This is important when there are recursive dependencies, such as, between the `getContainerMethod_alt1` and `getContainerMethod` nodes. Note that computing the initial analysis result also follows the same approach, but in that case IncA reads the

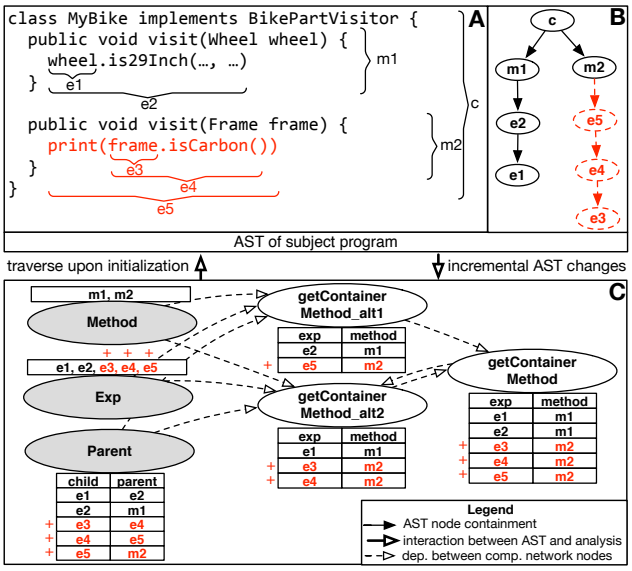

Figure 1. Computation network (C) of the `getContainerMethod` function evaluated on the AST (B) of a subject program (A). Red color shows newly inserted entities. Tuples of unary relations are shown in one row.

relevant parts of the AST, which then triggers a (transitive) batch insertion of many tuples.

We demonstrate the change propagation on the subject program in Figure 1A. Assume we just inserted the body of the `visit(Frame)` method. In response to this change, the AST of the subject program also changes with three newly inserted expressions as shown in Figure 1B. In the computation network, the change propagation starts from the `Exp` and `Parent` input nodes. The delta in the contents of `getContainerMethod` yield the changes in the analysis result, which clients can query through an API in IncA [15].

Overload resolution in IncA We use the IncA DSL to implement overload resolution for FJ, and we rely on the IncA computation network to efficiently incrementalize the analysis. Our approach consists of the exact same steps as with co-contextual type checking: (1) Find all candidates, (2) compute distance sums, and (3) select the unique candidate with minimal distance.

Figure 2 shows an excerpt of the analysis implementation in IncA. `findMinimalMethodMain` counts the number of candidates with minimum distance, and if there is only one, it returns the method. `findMinimalMethod` drives the whole search because it selects the methods with the minimum distance from all candidate methods. `getMinimalMethodDistance` uses the previously defined `Nat` lattice and aggregates result tuples with `glb` which will select the tuple that has the lowest number associated. Finally, `getDistance` computes the distance between the arguments and parameters. There are two things to note in the body of `getDistance`. First, Line 17-18 compute the cross product between the arguments and

```
def findMinimalMethodMain(call : MethodCall, class : ClassDec) : MethodDec = {                1
    assert count findMinimalMethod(call, class) == 1                                          2
    return findMinimalMethod(call, class)                                                     3
}                                                                                             4
def findMinimalMethod(call : MethodCall, class : ClassDec) : MethodDec = {                    5
    candidate := getCandidateMethods(call, class)                                             6
    candidateDistance := getDistance(call, candidate)                                         7
    minDistance := getMinimalMethodDistance(call, class)                                      8
    assert minDistance == candidateDistance                                                   9
    return candidate                                                                          10
}                                                                                             11
def getMinimalMethodDistance(call : MethodCall, class : ClassDec) : Nat/glb = {               12
    method := getCandidateMethods(call, class)                                                13
    return getDistance(call, method)                                                          14
}                                                                                             15
def getDistance(call : MethodCall, method : MethodDec) : Nat/sum = {                          16
    parameter := method.params                                                                17
    argument := call.args                                                                     18
    assert parameter.index == argument.index                                                  19
    return getArgParamDistance(argument, parameter)                                           20
}                                                                                             21
```

Figure 2. Excerpt of the overload resolution in IncA.

the parameters, but then we restrict the pairs to have the same index (which means that they are at the same position) in Line 19. Second, the function uses `sum` aggregation over the `Nat` lattice, which will sum up the distances. Without aggregation, for a `call` and `method` value pair, there would be as many `(call,method,`v_{index}`)` $\in$ `getDistance` result tuples as the number of argument-parameter pairs.

Compared to co-contextual type checking, IncA employs fine-grained incrementalization, and the evaluation in the computation network is not governed by the AST structure but by the data dependencies induced by the analysis definition. Assume we change the type of a method parameter. IncA will only consider calls that have a matching name, and, for these calls, it will only re-compute the argument-parameter distance at the affected position. Because IncA also incrementalizes the aggregator functions, it can efficiently derive both the new distance `sum`s and the `glb` aggregation result. Based on the updated aggregation results, IncA can then update the result of the `findMinimalMethodMain` function if the changed method is the new resolution result. Next, we exactly consider this kind of update scenario to benchmark the performance of overload resolution in IncA.

Evaluation Our goal is to use incremental overload resolution in IDEs to aid developers. To this end, the incremental update time is the most important performance characteristic. In addition, an incremental system relies on extensive caching, so we want to get a feeling about the memory consumption of the solution. We ran benchmarks to examine these aspects of our solution.

For our measurements, we used the Meta Programming System (MPS)[3], which is a language workbench used to implement (domain-specific) programming languages. In the workbench, we synthesized FJ benchmarks that consist of class hierarchies with the shape of binary trees. Each class contains two methods. A method `m` with two parameters that have the type of the containing class, and a method `c` that contains a call with the name `m`. Each call has two arguments, which are references to fields of the containing class. We defined the types of the fields randomly from all the classes in the benchmark. This makes sure that there are random distances between arguments and parameters. During the benchmarks, we vary the depth of the binary tree, this way controlling the number of classes, methods, and calls in the subject program.

Given a subject program, we first perform a non-incremental run of the IncA analysis. This is when IncA builds up the computation network and the initial caches. We measure the time spent on this operation. Then, we perform 5 random code changes in each benchmark. A code change manipulates the types of both parameters of a randomly selected `m` method by changing them to a randomly selected class. Such a program change has an influence on overload resolution because the distances may change. We measure the average time it takes to update the analysis result.

We also measure the memory requirement of the system. Before and after initializing the analysis, we measure the memory requirement of the JVM, and subtract the former value from the latter one. Based on our benchmarks, the

[3] https://perma.cc/LN73-3879

memory consumption of IncA roughly remains the same throughout a benchmark.

We ran our measurements on an Intel Core i7 at 2.7 GHz with 16 GB of RAM, running OSX 10.12.6, Java 1.8.0_121, and MPS version 2017.3.5. Our benchmarking setup is available online.[4]

The following table shows our measurement results for tree depth 6-9 (class hierarchies containing 63, 127, 255, and 511 classes):

Tree depth	Non-inc. (ms)	Inc. (ms)	Memory (MB)
6	1360	3	108
7	1461	3	258
8	2594	5	411
9	12612	14	1466

The numbers in the table show good incremental performance: These are the kind of numbers we need for interactive applications in IDEs. The only larger values are the initialization time and memory consumption for depth 9. A deeper investigation revealed that, in this case, certain computation nodes have an excessive cache size due to inefficient orderings of relational algebra operators. We believe that further engineering work in IncA can reduce these numbers. To better gauge these numbers, we also put them into perspective with other metrics and other IncA analyses:

- We also implemented data-flow analyses with IncA, for example, strong-update points-to analysis [11] and string analyses [5] for Jimple, which is an intermediate program representation of the Soot analysis framework [10]. On average, the update times of those analyses are also in the ballpark of a few tens of milliseconds, with initialization times of up to one minute.
- MPS used roughly 2GB of memory during our benchmarking. This means that the largest benchmark induces 75% of this memory consumption in addition.
- According to a survey among program analysis users [4], one of the primary reasons that hinders the adoption of static analysis tools is slow speed. We argue that overloading is fundamental for object oriented languages, so overload resolution must run with sub-second updates times, otherwise it breaks the development flow in the IDE. Our IncA overload resolution delivers the required update times.
- Arzt and Bodden implemented several incremental data-flow analyses (e.g. reaching definitions, uninitialized variables) with the Reviser framework and continuously evaluated them over a series of commits to several repositories [1]. The update times are in the ballpark of a few tens of seconds, although commits represent much larger code changes than our programmatically triggered ones (affecting only single methods and parameters). We believe that our benchmarking scenario is representative for in-IDE applications because developers modify the code bases through a series of small code changes typically.

[4] https://github.com/seba–/inca-experiments

4 Related Work

Incrementalizing program analyses in the context of IDEs has a long tradition because an analysis that runs in the background can only be useful if it does not break the development flow with lengthy pauses. For example, the Eclipse JDT compiler[5], the Facebook Hack[6] and Flow[7] languages, and the Java base language in MPS[8] all use incremental type checking. Unfortunately, the inner workings of these solutions are not well documented. Also, these are one-off solutions specific to a given programming language, while our goal is to find a generic solution that is applicable to programming languages in general that feature overloading. To this end, we deliberately experimented with extending co-contextual type checking or using a generic analysis framework.

We chose IncA because it is incremental, it is independent of the subject language, and it supports recursive aggregation over lattices. However, we also examined other frameworks. For example, Flix [12] and Bloom^L support recursive aggregation, but Flix is non-incremental, while Bloom^L only supports insertions and no deletions, which is not sufficient for applications in IDEs. QL [2] has a scalable commercial backend, but without support for lattice-based aggregation.

There is increased research attention towards giving semantics to and dealing with incomplete programs, as proposed by Omar et al. [13]. IncA fits into this research direction because the declarative analyses with the Datalog solver executed in a structured program editor (MPS) naturally supports incomplete or uncompilable subject programs.

5 Conclusions

We examined overload resolution for FJ in depth, and we tried out two approaches for its incrementalization in this paper. We failed with co-contextual type checking, but we were successful with IncA. They key difference between these two approaches is that IncA allows to build new relations and model the dependencies precisely between them, whereas co-contextual type checking relies on the AST directly. IncA can efficiently maintain the relations based on a delta in the AST, while co-contextual type checking can only collapse the constraints after recomputing them for the whole AST. IncA delivers millisecond update times, which is exactly what we need in interactive applications in an IDE.

[5] https://perma.cc/5QUY-9XGQ
[6] https://perma.cc/D77Q-L5B2
[7] https://perma.cc/5JWN-K8HR
[8] https://perma.cc/LN73-3879

References

[1] Steven Arzt and Eric Bodden. 2014. Reviser: Efficiently Updating IDE-/IFDS-based Data-flow Analyses in Response to Incremental Program Changes. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 288–298. https://doi.org/10.1145/2568225.2568243

[2] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[3] Lorenzo Bettini, Sara Capecchi, and Betti Venneri. 2009. Featherweight Java with dynamic and static overloading. *Sci. Comput. Program.* 74, 5-6 (2009), 261–278. https://doi.org/10.1016/j.scico.2009.01.007

[4] M. Christakis and C. Bird. 2016. What developers want and need from program analysis: An empirical study. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 332–343.

[5] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. 2011. Static Analysis of String Values. In *Proceedings of the 13th International Conference on Formal Methods and Software Engineering (ICFEM'11)*. Springer-Verlag, Berlin, Heidelberg, 505–521. http://dl.acm.org/citation.cfm?id=2075089.2075132

[6] Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 880–897. https://doi.org/10.1145/2814270.2814277

[7] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Found. Trends databases* 5, 2 (Nov. 2013), 105–195. https://doi.org/10.1561/1900000017

[8] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *Transactions on Programming Languages and Systems (TOPLAS)* (2001). https://doi.org/10.1145/503502.503505

[9] Edlira Kuci, Sebastian Erdweg, Oliver Bracevac, Andi Bejleri, and Mira Mezini. 2017. A Co-contextual Type Checker for Featherweight Java. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs)*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 18:1–18:26. https://doi.org/10.4230/LIPIcs.ECOOP.2017.18

[10] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15. 35.

[11] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to Analysis with Efficient Strong Updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 3–16. https://doi.org/10.1145/1926385.1926389

[12] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 194–208. https://doi.org/10.1145/2908080.2908096

[13] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 11:1–11:12. https://doi.org/10.4230/LIPIcs.SNAPL.2017.11

[14] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the First International Conference on Datalog Reloaded (Datalog'10)*. Springer-Verlag, Berlin, Heidelberg, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

[15] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the Definition of Incremental Program Analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 320–331. https://doi.org/10.1145/2970276.2970298

[16] J. B. Wells. 2002. The Essence of Principal Typings. In *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings (Lecture Notes in Computer Science)*, Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo (Eds.), Vol. 2380. Springer, 913–925. https://doi.org/10.1007/3-540-45465-9_78