Delft University of Technology, Bachelor Seminar of Computer Science and Engineering

Undoing Software Engineering: Demodularization of a SGLR Parser for Performance Gains

Nik Kapitonenko
TU Delft

Abstract

JSGLR2 is a java implementation of the Scannerless Generalized LR-parsing (SGLR) algorithm. It employs a modular architecture. This architecture comes with a performance overhead for letting multiple components interact with each other. This paper looks into the size of the performance overhead penalty for the recovery parser variant. It does so by creating an 'inlined' version of the recovery parser variant. The inlined recovery variant is a JSGLR2 implementation that ignores the modular architecture, and hard-codes the components. The performance of the inlined variant is measured with a pre-existing evaluation suite. The results show that there is a performance increase between the original, and the inlined variant.

1 Introduction

1.1 Context

Parsing is a fundamental component of processing most programming and markup languages. As various projects can get a large amount of source code, it is important that parsing is as fast as possible to reduce build times. At the same time, it is convenient if a single parser implementation can handle a large variety of languages. JSGLR2 [Denkers, 2018] is a Scannerless Generalized LR-parser implementation that can parse any context free language at practical speeds. It manages the algorithm's complexity by splitting the algorithm into multiple components of a manageable size, each of which receive optimizations sourced from various scientific papers and other sources. The components are then composed together using dependency injection. A factory design pattern provides several different compositions, each providing a different feature set and optimizations. JSGLR2 shows a 3x speedup for parsing Java source code, compared to JSGLR1 [Denkers, 2018].

1.2 The work

However, to achieve the modularity, the JSGLR2 implementation makes use of bridging architecture unrelated to the actual paring algorithm itself. It is possible that the modular architecture provides an performance overhead, that can be avoided by removing unnecessary architecture supporting the modularization. This 'inlining' may simplify the way components interact on a lower abstraction level, which may improve parsing throughput.

This paper presents four inlined versions of one the JSGLR2 variants, called "recovery". A copy of the "recovery variant" is created, and refactors it be be less modular. The four inlined versions share a common base, but have slight differences in how they inline certain components. Next, using a dedicated evaluation suite, this paper presents the performance results of the original and the inlined variants. The inlined variants are compared to the base case and each other. From the results, conclusions are drawn on which inlined variants are the most successful.

1.3 Organization

The paper is organized as follows into several sections. Section 2 provides a more in-depth background on the JSGLR2 recovery parser. It gives an overview of SLGR parsing. Section 3 describes the methodology, and what the inlining process consisted of. It also discusses the differences between the four proposed variants, and why four different variants were created in the first place. Section 4 describes the evaluation suite and discusses the results. Section 5 touches on reproducibility of the results. Section 6 summarizes the paper, and suggests future work. Finally, the appendix contains most figures, including the results.

2 Background

2.1 LR Parsing

JSGLR2 is a Scannerless Generalized LR-parser. Visser [Visser, 1997] describes it as "the integration and improvement of scannerless parsing, generalized-LR parsing and grammar normalization". LR-parsing forms the base of the SLGR algorithm. LR-parsing is based on Knuth's LR parsing algorithm [Knuth, 1965]. It parses a subset of context free languages, noted as "LR(k)" grammars. It is based on consuming input tokens one by one, and updating a state machine with a stack. The state then tells which of the three functions to perform: shifting, reducing, or accepting [Denkers, 2018].

Before the parsing can begin, the state machine needs to be created. In JSGLR2, it is called the *parse table* [Denkers, 2018]. It represents the grammar of the language to be parsed.

As the parse table is not part of parsing itself, further discussion on its creation is outside the scope of this work. What is important to know, is that it is a map between a tuple of a state with a token, and an action. 'Shift' and 'Reduce' action state to which state to transition to next, while the 'Accept' action successfully ends parsing.

Parsing itself starts with a pre-processing step called scanning. It reads the input text, and turns it into a stream of tokens. A token represents a language primitive, such as an number, an operator, or a keyword. Scanning strips out white-space, layout, comments, and other features not needed for the parser. The tokens are then fed into the parser.

Parsing starts with the freshly generated stream of tokens, and a stack containing one element: the start state of the parse table. The parser then iteratively reads the next token, and uses it together with the topmost state in the stack to retrieve the next action.

The shift action pushes a tuple on top of the stack. The tuple consists of a new state, and the token that triggered this action. The shift action lets the parser consume the input until a reduce action can be applied.

The reduce action is where the 'comprehension' of the input happens. It pops off one or more states from the stack, and combines their associated tokens into a composite token called a parse node. The node represents what the tokens mean together. For example, if the three top-most tokens were "Constant 1", "Operation +", "Variable x", then a reduction that takes these three tokens could produce a parse node representing "Value (1 + x)". After reducing, the next state is looked up in the parse table. The input tuple is then the new topmost state, which was directly below the states that were removed for reducing, and the parse node as token. The found state is then put on top with the stack, together with the parse node. Effectively, reducing replaces simpler tokens with a single composite token.

The action state signifies that parsing is done. Usually, it is only accessible when the next token is End Of Input, which is appended at the end of a token stream. It tells the parser to inspect the stack, and see if is represents a successful parse. This is the case when the stack only contains two elements: the start state, and another state with an associated token. This token then represents the full syntax tree that the parser is supposed to generate. If there more than two states in the stack, then parser signals a failure.

2.2 Generalized parsing

While LR parsing is relatively fast and simple, it has one major downside: it fully deterministic. For each state and token, there can be only one action. This is an issue when the grammar description of a language contains ambiguities. For example, take the following grammar:

```
Start -> Expression
Expression -> <Expression> + <Expression>
Expression -> <Expression> * <Expression>
Expression -> x
```

Then, an input such as `"x + x * x"` presents an ambiguity: should it be parsed as `"(x + x) * x"`, or as `"x + (x * x)"`? When the parser reaches the middle `x`, it would not know whether to reduce the `"x + x"` into `"(x + x)"`, or to shift until the last `x`, and then reduce the multiplication first. The solution for this would be to designate some kind of operator precedence, where one action is preferred over the other.

However, in this example, if `"x + (x * x)"` is the proffered outcome, another issue appears. When deciding whether to shift or reduce the middle `x`, the parser so far has only consumed `"x + x"`. It does not know what the next character is. It might be *, in which case the parser should shift, or it might be actually End of Input, in which case the parser should reduce. Knuth solved this with lookahead: instead of using the currently parsed token to look up the next state in the parse table, The parser reads, but not consumes, the next `k` tokens and uses them all together to look the next state [Knuth, 1965].

While this works for most practical languages, in theory, it is possible to devise a grammar that requires more than `k` tokens to resolve an ambiguity [Denkers, 2018]. *Generalized LR-Parsing* is an alternative solution to the ambiguity problem [Visser, 1997]. It allows a parse table to return multiple states from a lookup. Then, for each return action, the parser creates a new parallel stack, and applies a different action on each action. The parser then continues updating all stacks in parallel. Effectively, this means that the parser tries all actions at the same time. If a single stack ends up being wrong in some way, it can be discarded, while other stacks continue working. The parser fails only when all stacks fail.

To save space, the parallel stacks are stored in a Graph Structured Stack (GSS) [Denkers, 2018]. The states in the stacks are stored as nodes. Directed edges point to the next element below the top state. This lets parallel stacks share common states.

2.3 Scannerless parsing

Scannerless parsing, as the name implies, is a modification to the LR-parsing algorithm that removes the scanner. It changes the parsing algorithm to work on characters directly. One advantage of this is that now white-space and layout can be part of the language grammar. However, the main issue that scannerless parsing solves are context sensitive keywords. For example, take the string `int int = 1;` in a C-like language. Normally, the scanner would mark the second `int` as a keyword token, and then parsing would fail, as assigning a value to a keyword does not make any sense. However, with scannerless parsing, a grammar could be devised that would consider any alphanumeric characters between the first `int` and the = as a valid identifier. The big downside is that it significantly increases the size of the parse table.

2.4 Recovery parsing

The parsing techniques described so far assume a simple dichotomy: either the input text is syntactically valid and can be parsed, or there is an error and parsing should be aborted. Recovery parsing gives a middle ground. It tries to parse as much input as possible, and isolate out areas of text with errors. JSGLR2 uses a modification of the "islands and waters" approach [Moonen, 2001; Denkers, 2018]. During normal parsing, the parser keeps a history of the changes to the

stacks. Upon failure, the parser progressively undoes its actions, and skips over the characters where the error was encountered.

2.5 Spoofax Workbench

Spoofax is a suite of various tools and languages used to develop programming and mark-up languages [MetaBorg, 2016]. JSGLR2 is one of its components, providing the parsing functionality. Spoofax interacts with JSGLR2 by providing it with a parser generator, and a means to build abstract syntax trees Spoofax can used in other locations.

2.6 JSGLR

JSGLR2 implements all concepts mentioned so far using a modular architecture. The classes used to compose the recovery variant can be seen in Figure 1. The majority of these classes, such as `Parse State`, all managers, and `Reducer`, are singletons that provide a part of the SGLR parsing algorithm. A minority of classes, such as `Stack Node`, `Parse Forest` and `Derivation`, are data structures used by the parser. For example `Stack Link` and `Stack Node` are used to represent the stack, while `Character Node` and `Parse Node` store the parsed rules.

With the exception of `Parse Table`, all classes shown in figure 1 are subject to inlining. This is the case because the `Parse Table` is not defined in the JSGLR2 project, but defined and passed in externally.

3 Methodology and Contribution

The very first step is to define what an inlined variant should look like. This is what this section concerns itself with. The idea of inlining is as follows: keep the high-level algorithm the same, but remove the modularity; undo the idea that JSGLR2 can be assembled from arbitrary parts. The rationale behind this is that to enable modularity, all components need to support all possible parser variants. These variants each have unique features, such as recovery for the recovery variant. This means that the components need to have some way to support all those features. The result is that components contain control flow logic that is only exercised for certain variants. Yet, all variants need to suffer from the overhead of these additions.

3.1 Getting access

Before writing the inlined variant, an important preliminary needs to be satisfied first. There should be a way to retrieve the inlined recovery variant. Access is made possible with the insertion of the inlined variant into `JSGLR2Variant`. A new entry in `JSGLR2Variant.Preset` is created, called `recoveryInlined`. Instead of a `JSGLR2Variant`, this enum value returns a subclass called `RecoveryVariant`. It is stored as an inner class inside `JSGLR2Variant`, and overrides `JSGLR2Variant.getJSGLR2()` to return the inlined variant. `RecoveryVariant` also contains a parameterless constructor making it possible to create it without the need to pass in a `ParserVariant`, `ImploderVariant`, and `TokenizerVariant`, which are required to instantiate `JSGLR2Variant`. `RecoveryVariant`'s constructor simply passes `null` to the superclass constructor, signifying that the inlined variant is not modular.

3.2 Duplication and inheritance removal

Now that there a way to to get the inlined variant, the first major refactor can happen. It consists of four parts: duplicating the recovery variant, renaming the components, squishing inheritance, and removing generics. The first and second part provide the groundwork to work on. This way, the inlined variant is separate from the existing variants, so no existing functionality is affected. The new names are found in table 1. As for the location of the new files, `JSGLR2RecoveryInlined` is put in the `org.spoofax.jsglr2` package, the same place as where `JSGLR2Implementation` resides. All other components are put in a separate `org.spoofax.jsglr2.inlined` package, so that they would not pollute other namespaces.

3.3 Inheritance Removal

While the new files exist and can be used to create new variants, the third step is applied. For each inlined class. inheritance is removed by recursively retrieving the superclass, and inserting its methods. Methods that are overridden are ignored, unless the subclass used a `super` call. In that case, the superclass method body simply replaces the line with `super` in it. The end result is that with a few exceptions, the inlined classes do not have any super or subclasses anymore.

Inheritance Removal Exceptions

The exceptions to the inheritance squishing step are as follows: `JSGLR2RecoveryInlined` implements the `JSGLR2<IStrategoTerm>` interface, as it is the entry point into the entire JSGLR2 algorithm family. Without it, other parts of the Spoofax Workbench, which JSGLR2 is part of, wouldn't be able to use the inlined variant, including the evaluation suite that will be used for getting results in Section 4.

Another exception is `InlinedStackPath.Empty` and `InlinedStackPath.NonEmpty`. They inherit from `InlinedStackPath`, as they are used to represent a linked list where the last element holds a different type of value. While it is possible to rewrite this code to use a different, inheritance free way to represent stack paths, it has been decided to stick as close as possible to the original algorithm, to ensure that this change in behaviour would not affect the benchmark results.

Similarly, `InlinedAmbiguityDetector`, `InlinedCycleDetector`, and `InlinedNonAssocDetector`, implement `IInlinedParseNodeVisitor`, as they use a variation of the visitor design pattern. Meanwhile, `InlinedParseNode` and `InlinedCharacterNode` have to implement the `IParseForest`, for two reasons. The first one is that they are both nodes in a parse forest. Some methods and data structures, such as the reducer and stack links, need to be able to function with both types of nodes. The second reason is because the parser component of the inlined variant implements `IParser`, the `parse` method it implements returns a `ParseResult`, which in turn contains a `IParseForest`.

Original Class	→	New Class
org.spoofax.jsglr2.JSGLR2Implementation	→	JSGLR2RecoveryInlined
org.spoofax.jsglr2.imploder.TokenizedStrategoTermImploder	→	InlinedImploder
org.spoofax.jsglr2.inputstack.InputStack	→	InlinedInputStack
org.spoofax.jsglr2.parseforest.AmbiguityDetector	→	InlinedAmbiguityDetector
org.spoofax.jsglr2.parseforest.CycleDetector	→	InlinedCycleDetector
org.spoofax.jsglr2.parseforest.NonAssocDetector	→	InlinedNonAssocDetector
org.spoofax.jsglr2.parseforest.ParseNodeVisitor	→	InlinedParseNodeVisitor
org.spoofax.jsglr2.parseforest.hybrid.HybridCharacterNode	→	InlinedCharacterNode
org.spoofax.jsglr2.parseforest.hybrid.HybridDerivation	→	InlinedDerivation
org.spoofax.jsglr2.parseforest.hybrid.HybridParseForestManager	→	InlinedParseForestManager
org.spoofax.jsglr2.parseforest.hybrid.HybridParseNode	→	InlinedParseNode
org.spoofax.jsglr2.parser.ForShifterElement	→	InlinedForShifterElement
org.spoofax.jsglr2.parser.Parser	→	InlinedParser
org.spoofax.jsglr2.parser.observing.IParserObserver	→	InlinedObserver*
org.spoofax.jsglr2.parser.observing.IParserNotification	→	InlinedObserving.IInlinedNotification*
org.spoofax.jsglr2.parser.observing.ParserObserving	→	InlinedObserving*
org.spoofax.jsglr2.recovery.BacktrackChoicePoint	→	InlinedBacktrackChoicePoint
org.spoofax.jsglr2.recovery.RecoveryDisambiguator	→	InlinedDisambugator
org.spoofax.jsglr2.recovery.RecoveryJob	→	InlinedRecoveryJob
org.spoofax.jsglr2.recovery.RecoveryObserver	→	InlinedObserver*
org.spoofax.jsglr2.recovery.RecoveryParseFailureHandler	→	InlinedParseFailureHandler
org.spoofax.jsglr2.recovery.RecoveryParseReporter	→	InlinedParseReporter
org.spoofax.jsglr2.recovery.RecoveryParseState	→	InlinedParseState
org.spoofax.jsglr2.recovery.RecoveryReducerOptimized	→	InlinedReducer
org.spoofax.jsglr2.reducing.ReduceManager	→	InlinedReduceManager
org.spoofax.jsglr2.stack.StackLink	→	InlinedStackLink
org.spoofax.jsglr2.stack.collections.ActiveStacksArrayList	→	InlinedActiveStacks
org.spoofax.jsglr2.stack.collections.ForActorStacksArrayDeque	→	InlinedForActorStacks
org.spoofax.jsglr2.stack.hybrid.HybridStackManager	→	InlinedStackManager
org.spoofax.jsglr2.stack.hybrid.HybridStackNode	→	InlinedStackNode
org.spoofax.jsglr2.stack.paths.EmptyStackPath	→	InlinedStackPath.Empty
org.spoofax.jsglr2.stack.paths.NonEmptyStackPath	→	InlinedStackPath.NonEmpty
org.spoofax.jsglr2.stack.paths.StackPath	→	InlinedStackPath
org.spoofax.jsglr2.tokens.StubTokenizer	→	*[removed]*

Table 1: Full list of all new renamed classes used for the inlined variants, as well as the full names of the original components.
*only for certain branches

The Observer Mechanism

`InlinedParser` implements `IParser<IParseForest>`, as means to satisfy `JSGLR2` interface, which provides a `IParser<?> parser()` method. However, notice that `JSGLR2` also provides a `void attachObserver(IParserObserver)` method, where `IParserObserver` is a subclass of of `IParser`. This method lets the user attach 'observers' to the parser, who hook in into the parser internals via the observer design pattern. Most methods inside the parser call an callback function of all attached observers. To let the observers modify the parsing logic, parser methods pass along the data structures they work with to the observer callbacks. Because of this, `IParserObserver` cannot be used with an inlined variant directly.

Code inspection reveals that the observer functionality is only used in four places: `RecoveryObserver` is used by the recovery variant to inject additional code into the 'shift' and 'reduce' steps. `ParserMeasureObserver` from the `measure` project is used to generate statistics during evaluation. `org.spoofax.jsglr2.cli` project contains several loggers to print out detailed information during the parsing. `org.spoofax.jsglr2.benchmarks` contains several observers used for data structure related tests.

To handle the observer mechanism, three solutions were considered:

1. **InlinedObserver:** The first option is to apply the same inlining logic to the `IParserObserver` as to all other components. Inlined versions of `IParserObserver`, `ParserObserving`, and `RecoveryObserver` are created and used. The upside of this solution is that it mimics the existing variant, resulting in only the inheritance removal step being measured. The downside is that it cannot be used with `void attachObserver(IParserObserver)`.
2. **No observers:** The observer mechanism is simply stripped out entirely. All references

to `IParserObserver`, `ParserObserving`, and `RecoveryObserver` are removed. `RecoveryObserver` has two method, `reducer` and `shift`. They are inserted directly into `InlinedReduceManager::reducer` and `InlinedParser::shifter`, where the two methods were indirectly called. The advantage of this solution is that the possible overhead of calling observer methods is gone. However, this solution has the same disadvantage as the the first one.

3. **FakeObserver:** This solution tries to make `void attachObserver(IParserObserver)` work by creating 'fake' components. Each parser component is used in `IParserObserver` gets an additional method called `getFake()`. Its return type is a corresponding non-inlined version of that component. For example, `InlinedDerivation::getFake()` would return `IDerivation`. However the actual return type is an anonymous class extending from the return type. This class has a single constructor that takes the inlined component and stores it in a private field. All relevant methods are overridden to call their equivalents in the inlined component. If such method takes or returns another parser component, they need to be converted as well. The result is that now non-inlined `IParserObserver`s can be used with the inlined recovery variant. The cost is that each inlined parser component now contains two classes: the true inlined version, and a fake version. This significantly bloats the codebase. Additionally, converting between the inlined and fake versions would induce a performance penalty.

Out of these, the last one was attempted at first, but then dropped as infeasible. The other two solutions, however, were both applied, by splitting the development. One maintains the observers as in solution one, the second branch applies solution two. The two variants will be further referred to as "Inlined Observers", the other one as "No Observers" The reasoning behind this is that the observer mechanism is hypothesised to be a notable contributor to the modularity overhead, so evaluation of both versions would help confirm or deny this.

3.4 ParserMeasureObserver

`ParserMeasureObserver` is part of the measurements project, and is used for parser testing. It uses the observer mechanism to count how much each method in the parser was called. This is useful when comparing different parser variants. This includes the inlined ones. While for the "Inlined Observers" inlined variant it would be possible to create a "Inlined Measure Observer", the "No Observers" variant cannot support it directly. Therefore, a third inlined variant was created, called "Integrated Measurements". It is similar "Inlined Observers" variant. The difference between these two is that `ParserObservering` is renamed to `StatCounter`, and instead of letting it store arbitrary observers, it instead contains `ParserMeasureObserver` logic.

Inheritance Removal Rationale

The idea behind removing inheritance concerns itself with static and dynamic dispatch. In many object oriented languages, these are the two ways a method can be called inside a compile binary. Static dispatch is simple enough: during compilation, the location of the called method is calculated, and the resulting memory address is directly added to the call instruction. During runtime, method calls are as simple as a jump instruction plus some stack setup and teardown overhead. The major downside of it is that it only works if the target method is known during compile time. This is the case when inheritance is used. This is because the runtime type of an object may actually be a subclass, and therefore, the called method may be overridden. Therefore, the system needs to check to which class the object whose method is called belongs to, and look up the appropriate method location. For java, the official specification implies that it supports both dispatches, via the `invokespecial` and `invokevirtual`, respectively [Gosling *et al.*, 2021]. Therefore, it is possible that removing inheritance would improve performance.

3.5 Removing Generics

The last step to make the inlined variant functional is to swap out all references to parser components from generics to the specific inlined versions. Non-inlined parser components use generics to refer to each other. For example, the full type definition of `RecoveryParseState` is `RecoveryParseState<InputStack extends IInputStack, StackNode extends IStackNode>`. So, all occurences of `InputStack` have been replaced with `InlinedInputStack`, and `StackNode` with `InlinedStackNode`.

3.6 Marking Classes as Final

The last change was marking all classes as final. In java, this prevents a class from having subclasses. This might help the compiler use static dispatch. To see if it indeed helps the compiler, they were put in the fourth inlined variant, simply called "No Observers and Final Classes". As the name implies, it is built upon the "No Observers" variant.

4 Evaluation, Results, Discussion

4.1 Evaluation suite

The evaluation of the inlined parser variants is done with an external evaluation suite simply called "JSGLR-evaluation". It is a a collection of scripts operated via a Makefile. It is responsible for setting up the test sources, running the benchmarks, doing statistical measurements, and generating the graphs. This evaluation suite was originally designed to compare the various modular parser variants. During the development of the inlined variants, the suite was adjusted as well to support the inlined versions, and to exclude the modular variants other than recovery. It was used to generate the figures.

4.2 Sources

The test sources are single source files picked from public projects written in Java, webdsl[1], and sdf3[2]. All benchmarks and measurements are run per language, but all projects are evaluated together in one run. The metadata for the files used can be found in table 3.

Results

Figures 2 - 5 show four bar graphs. The right column shows time spent parsing per language. The left column normalizes the result by showing the throughput in characters parsed per second. The top row only shows parsing, while the bottom row includes the imploder. The original, non-modular variant is marked as 'recovery', and is brown colored, while the inlined variant is called 'recoveryInlined' and uses blue bars. The show that with the exception of the inlined variant with the integrated measurement observer, all inlined parsers have better performance. The variant with built-in measurement observer scores terribly. When looking at throughput including imploding, Figure 3 shows a decrease of 28.9% for Java, and 27.4%, 25.0% for Webdsl, SDF3, respectively. Without imploding, the decreases[3] are 15.8%, 29.6%, and 26.5%. The high parse times of the variant with integrated measurements is can be attributed to the fact that `ParserMeasureObserver` uses hash maps to count distinct components created over the course of parsing a test file. As an example, during testing, the measurements project reported that over 30 000 stack nodes were used during parsing, per language. So the measure observer had a hash map that grew to contain all these elements.

The variant with inlined observers (Figure 2) differs from the regular recovery variant only by having no inheritance, and it shows that the inheritance is relatively small penalty. In terms of parsing throughput, the inlined variant shows a percentage increase of 9.0% for Java, 6.5% for WebDSL and 8.3% for SDF3. When excluding imploding, the results are 3.0%, 1.9%,and 6.2%, respectively.

Much more interesting are the results that forgo observers altogether. Figure 4 shows a significant improvement of 24.0%, 32.4%, and 31.9% with imploding, 33.4%, 35.0%, 17.6% without imploding. This heavily implies that the observer mechanism poses a significant overhead on the parser, and that any attempts to improve the performance the JSGLR2 parser should remove the observers first.

The variant not discussed yet is the one that builds upon the "no observers" variant with making all classes final. With imploding, it shows an increase of 32.8%, 32.2%, and 40.4%, while just parsing results in 29.6% 38.5%, and 9.3%. If the two inlined variants are compared[4], the relative improvement is -11.5 %, +10.0%, and -47.3%! That is, the variant with final classes shows worse performance. If imploding is included, then the results are 29.9%, 21.2%, 35.5%.

The most likely reason for this discrepancy is that according to table 3, Java and SDF3 have relatively small sample size, of only three files each. These three files have a total size of 10739 and 5019 bytes, respectively. Meanwhile, WebDSL has 10 files, totalling 49194 bytes. Therefore, Java and SDF3 measurements must have a relatively high variance between runs. WebDSL is the only language that shows a relative improvement in both cases, so therefore it seems to be the most trustworthy result.

Even though the percentage values likely have a high error margin, the conclusion is still sound: The variant with no observers and final classes shows the best performance increases of all inlined variants. This means that removing inheritance and observer mechanism, marking the classes as final, and lowering the access modifiers has a net positive result on parsing performance.

5 Responsible Research

5.1 Software sources

New findings are generally more useful when they can be consistently reproduced, or at least not consistently disproved. To ensure that the reader can confirm the results for themselves, the source code for the inlined JSGLR2 fork can be found online at https://github.com/certified-potato/jsglr2-inlined, while the evaluation suit is hosted at https://github.com/certified-potato/jsglr2evaluation. The four variants can be retrieved from repository via four branches: `inlined_observers`, `integrated_measurements`, `no_measurements` and `with_privates`. They correspond to the variants with inlined observers, built-in `ParserMeasureObserver`, purged observers, and purged observers with all classes marked final, respectively.

5.2 Evaluation device

The results were evaluated on a HP ZBook Studio G5 notebook. The CPU is an Intel®Core™i7-8750H at 2.20 GHz. The L1, L2, and L3 caches are 384KiB, 1536KiB, and 9MiB, respectively. The system memory is 16GiB SODIMM DDR4 Synchronous 2667 MHz (0.4 ns). To ensure consistent results, Intel®Turbo Boost has been disabled. The java installation used to run the benchmarks is OpenJDK Runtime Environment (build 1.8.0_292-8u292-b10-0ubuntu1 20.10-b10) with OpenJDK 64-Bit Server VM (build 25.292-b10, mixed mode).

5.3 Ommitted data

Some results were omitted, as they were considered not relevant enough to include in this paper. The first set of discarded data are the benchmarks of the parser with inlined observer, that was modified to also attach a `InlinedMeasureObserver`. It used to confirm that low performance of the parser version with integrated measure observer is indeed caused by the measurement logic itself. However, this discarded variant has no other use, and does not provide any other useful information.

[1] https://webdsl.org

[2] SDF3 is a domain specific language developed for, and used in the Spoofax Workbench. It is used to write grammars for programming languages

[3] Calculated as $(\text{new} - \text{old})/\text{old} \cdot 100\%$, where new is the throughput of the inlined variant, and old of the non-inlined recovery variant.

[4] (percent increase variant with final classes% increase variant with no observers)
$100\% - 100\%$

Another piece of data that is not shown here, are the benchmarks that include non-recovery variants. The reason is simple: other variants have a different logic, and are therefore not comparable to the recovery variant, both the regular and the inlined versions.

Similarly, comparisons with third party parsers, such as ANTLR[5], and tree-sitter[6], are omitted as well.

Lastly, the measurements from the measurement project were omitted as well. The reason is because it shows how much each method in the parser is called. Inlining does not change the logical algorithm, so all values for all variants were the same.

6 Conclusions and Future Work

6.1 Summary

The goal of this work was to measure the overhead of the modular architecture on the recovery variant JSGLR2 parser. This variant has been extracted, and refactored to stand on its own, without making use of the existing framework. Four different inlined variants have been created, all sharing a common base, but having slight differences, particularly on how to handle the parser observer mechanism. The four variants all have been evaluated relative to the original, non-inlined recovery variant. The result show that removing inheritance, deleting the observer mechanism completely, and marking methods as package-protected or private, and classes as final, has a net benefit on performance.

6.2 Future work

Future work would consist of applying additional inlining techniques. The first one would be combining classes that have an one to one relation. For example, `InlinedParseState` and the various Manage could be be inserted into `InlinedParser`. This could be used together with the idea of combining methods together. Coman [Coman, 2021] has made similar work, so those ideas could be combined.

Another improvement would be to increase the sample size of the files used for benchmarking. The improvement itself is trivial, as it only requires to change the evaluation configuration suite. However this does increase the running time of the evaluation suite. So better planning to schedule in longer benchmarks would be beneficial.

References

[Coman, 2021] Mara Coman. Performance impact of the modular architecture in the incremental sglr parsing algorithm. Bachelor's thesis, Delft University of Technology, Delft, 2021.

[Denkers, 2018] Jasper Denkers. A modular sglr parsing architecture for systematic performance optimization. Master's thesis, Delft University of Technology, Delft, 2018.

[Gosling *et al.*, 2021] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *Java 16 JVM Specification*, feb 2021.

[Knuth, 1965] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[MetaBorg, 2016] MetaBorg. The spoofax language workbench, 2016.

[Moonen, 2001] L. Moonen. Generating robust parsers using island grammars. *Proceedings Eighth Working Conference on Reverse Engineering*, pages 13–22, 2001.

[Visser, 1997] Eelco Visser. Scannerless generalized-lr parsing. Technical report, University of Amsterdam, Amsterdam, Aug 1997.

[5] https://www.antlr.org/

[6] https://tree-sitter.github.io/tree-sitter/

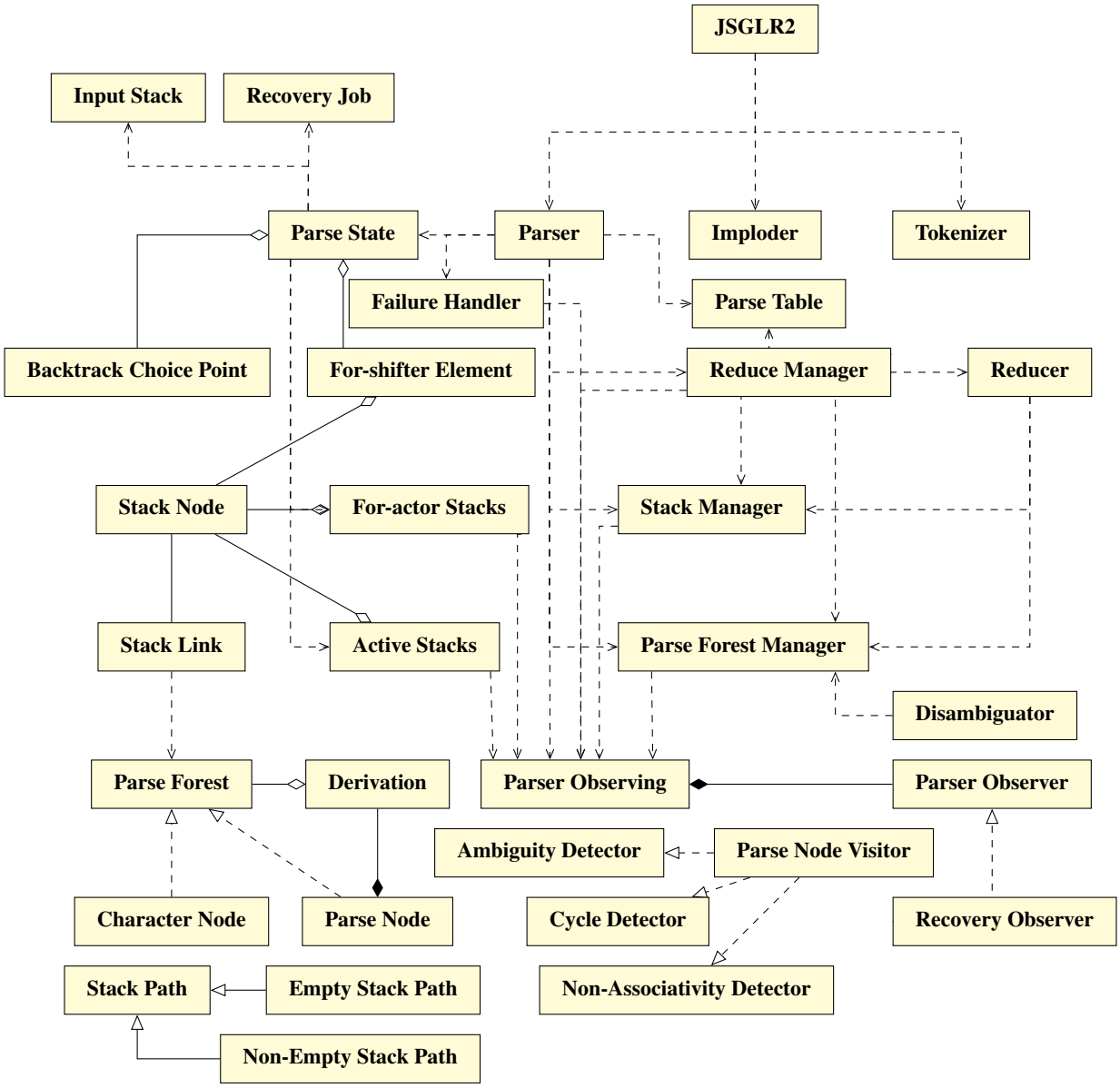
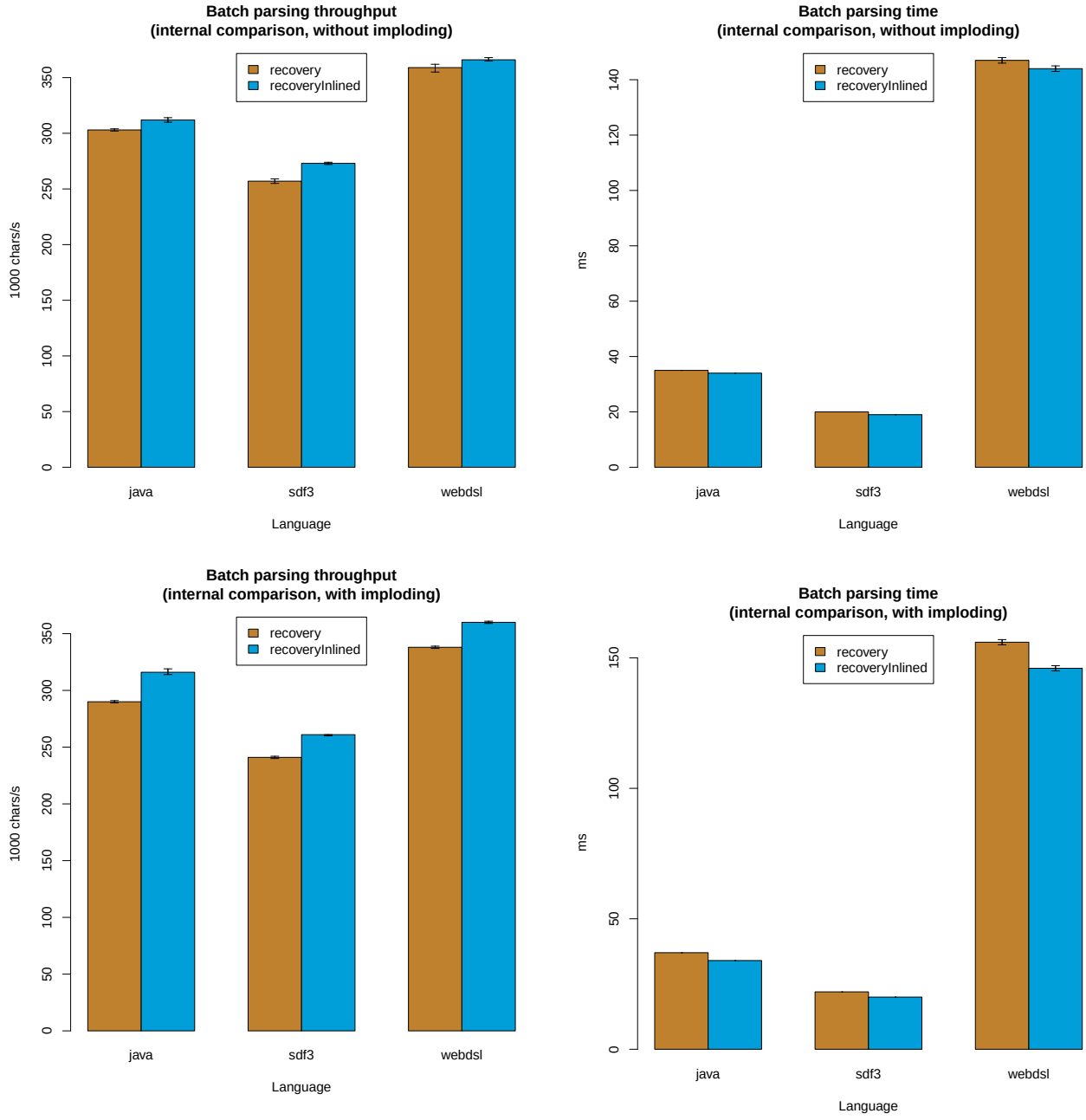


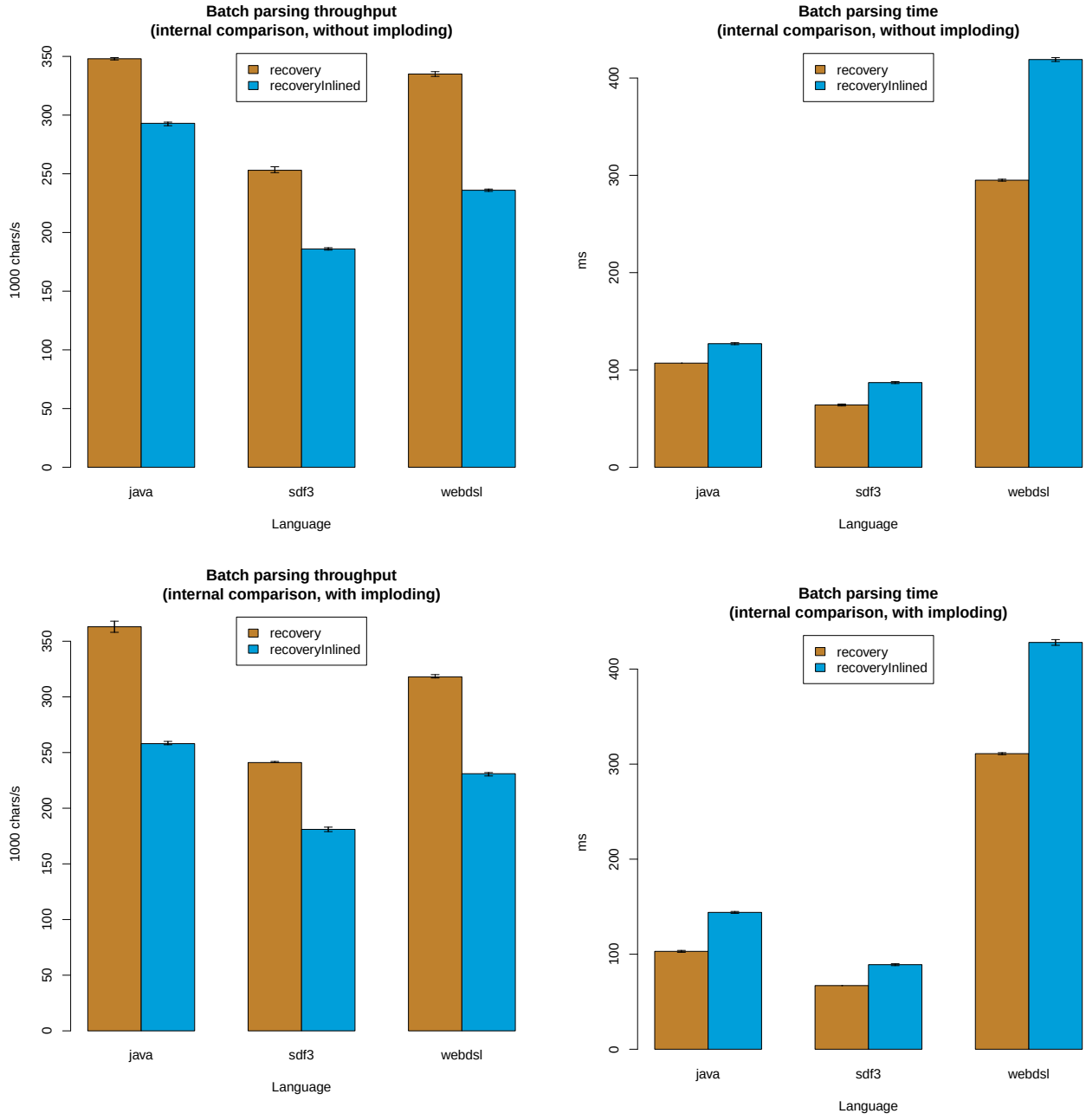
Figure 1: Simplified UML diagram of the recovery variant parser. It only shows which classes own which other classes, and inheritance in cases when the multiple subclasses are used together. Not shown is how the classes relate to each other in terms of method calls, nor the full inheritance tree.

Table 2: Counts of components, method calls and other parts of the parser used during parsing.



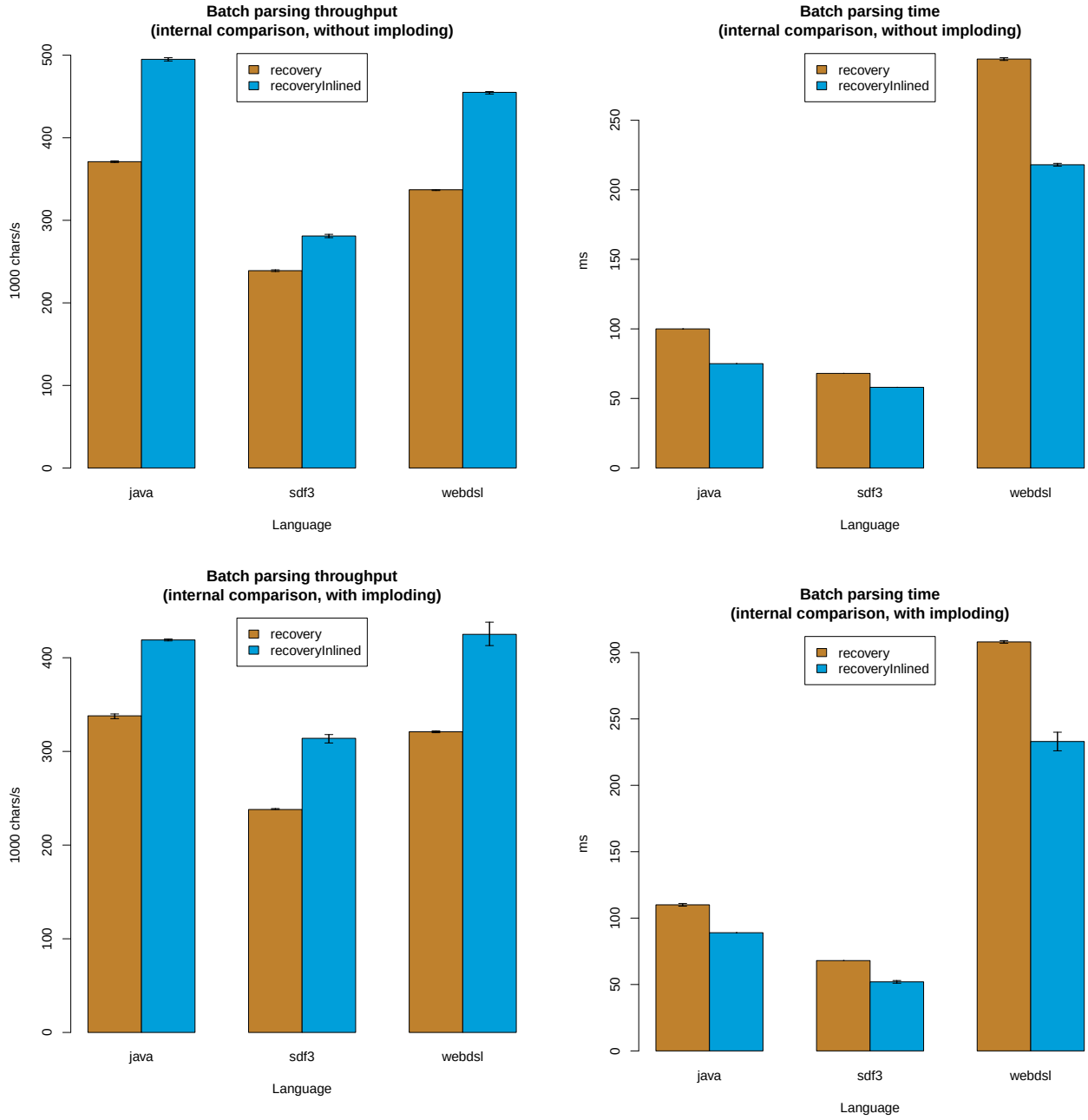
(a) Foo

Figure 2: Results of the inlined variant with *inlined observers*



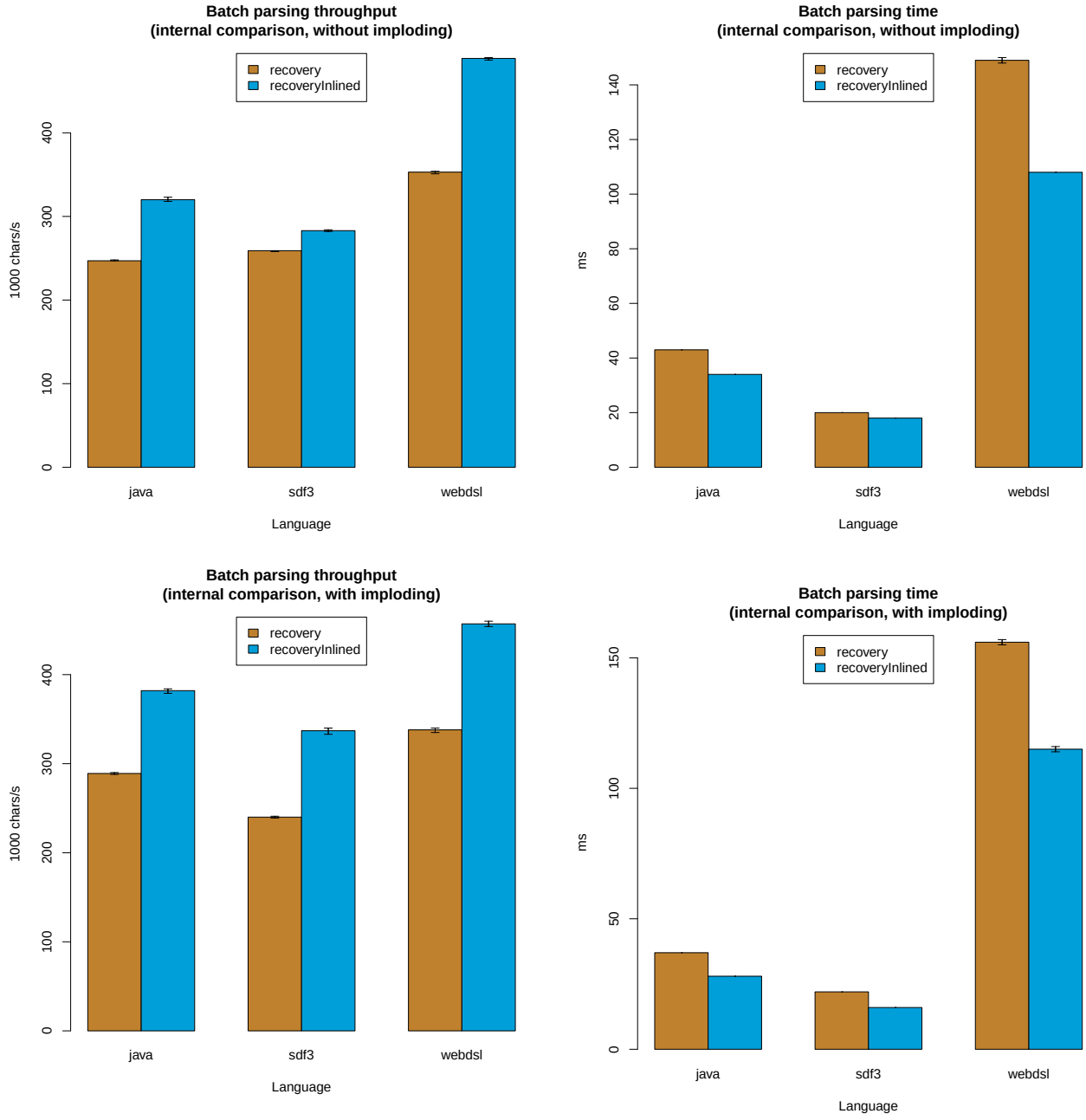
(a) Foo

Figure 3: Results of the inlined variant with that has `ParserMeasureObserver` built-in



(a) Foo

Figure 4: Results of the inlined variant with *observers mechanism removed*



(a) Foo

Figure 5: Results of the inlined variant with *observers mechanism removed and all classes made final*

Language	Source	Files	Lines	Size (bytes)
Java	apache-commons-lang	1	57	2331
	netty	1	56	1858
	spring-boot	1	167	6550
WebDSL	webdsl-yellowgrass	1	6	100
	webdsl-elib-example	1	7	95
	webdsl-elib-ace	1	131	3775
	webdsl-elib-tablesorter	1	107	4075
	webdsl-elib-utils	1	25	648
	webdsl-elib-bootstrap	1	1196	34501
	webdsl-elib-unsavedchanges	1	96	4108
	webdsl-elib-timeline	1	52	964
	webdsl-elib-timezones	1	109	3724
	webdsl-elib-holder	1	17	602
SDF3	nabl	1	14	136
	dynsem	1	145	3649
	flowspec	1	63	1434

Table 3: Metadata of source files used for benchmark tests