

Panbench: A Comparative Benchmarking Tool for Dependently-Typed Languages

3 **Anonymous author**

4 Anonymous affiliation

5 **Anonymous author**

6 Anonymous affiliation

7 — Abstract —

8 We benchmark four proof assistants (Agda, Idris 2, Lean 4 and Rocq) through a single test suite.
9 We focus our benchmarks on the basic features that all systems based on a similar foundations
10 (dependent type theory) have in common. We do this by creating an “over language” in which to
11 express all the information we need to be able to output *correct and idiomatic syntax* for each of our
12 targets. Our benchmarks further focus on “basic engineering” of these systems: how do they handle
13 long identifiers, long lines, large records, large data declarations, and so on.

14 Our benchmarks reveals both flaws and successes in all systems. We give a thorough analysis of
15 the results.

16 We also detail the design of our extensible system. It is designed so that additional tests and
17 additional system versions can easily be added. A side effect of this work is a better understanding
18 of the common abstract syntactic structures of all four systems.

19 **2012 ACM Subject Classification** Replace ccsdesc macro with valid one

20 **Keywords and phrases** Add keywords

21 **Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

22 1 Introduction

23 Production-grade implementations of dependently typed programming languages are complica-
24 tated pieces of software that implement many intricate and potentially expensive algorithms.
25 As such, large amounts of engineering effort has been dedicated to optimizing these com-
26 ponents. Unfortunately, engineering time is a finite resource, and this necessarily means
27 that other parts of these systems get comparatively less attention. This often results in
28 easy-to-miss performance problems: we have heard anecdotes from a proof assistant developer
29 that a naïve $O(n^2)$ fresh name generation algorithm used for pretty-printing resulted in 100x
30 slowdowns in some pathological cases.

31 This suggests that a benchmarking suite that focuses on these simpler components
32 could reveal some (comparatively) easy potential performance gains. Moreover, such a
33 benchmarking suite would also be valuable for developers of new dependently typed languages,
34 as it is much easier to optimize with a performance goal in mind. This is an instance of
35 the classic $m \times n$ language tooling problem: constructing a suite of m benchmarks for n
36 languages directly requires a quadratic amount of work up front, and adding either a new
37 test case or a new language to the suite requires an additional linear amount of effort.

38 Like most $m \times n$ tooling problems, the solution is to introduce a mediating tool. In our
39 case, we ought to write all of the benchmarks in an intermediate language, and then translate
40 that intermediate language to the target languages in question. There are existing languages
41 like Dedukti[2] or Informath[1] that attempt to act as an intermediary between popular proof
42 assistants, but these tools typically focus on translating the *content* of proofs, not exact
43 syntactic structure. To fill this gap, we have created the Panbench system, which consists of:



© Anonymous author(s);

licensed under Creative Commons License CC-BY 4.0

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:55



LIPICS Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

23:2 Panbench: A Comparative Benchmarking Tool for Dependently-Typed Languages

- 44 1. An extensible embedded Haskell DSL that encodes the concrete syntax of a typical
45 dependently typed language.
- 46 Reed: Cita-
47 tions here
- 48 2. A series of compilers for that DSL to Agda, Idris 2, Lean 4, and Rocq.
49 3. A benchmarking harness that can perform sandboxed builds of multiple revisions Agda,
50 Idris 2, Lean 4, Rocq.
51 4. An incremental build system that can produce benchmarking reports as static HTML
52 files or PGF plots¹.

51 2 Methodology

This is really documenting the 'experiment'. The actual details of the thinking behind the design is in Section 5.

this itemized
list should be
expanded into
actual text

- 53 54 55 56 57
- single language of tests
 - document the setup of tests, high level
 - document the setup of testing infrastructure, high level
 - linear / exponential scaling up of test 'size'

58 3 Results

59 4 Discussion

60 5 The Design of Panbench

61 At its core, Panbench is a tool for producing grammatically well-formed concrete syntax
62 across multiple different languages. Crucially, Panbench does *not* require that the syntax
63 produced is well-typed or even well-scoped: if it did, then it would be impossible to benchmark
64 how systems perform when they encounter errors. This seemingly places Panbench in stark
contrast with other software tools for working with the meta-theoretic properties of type
systems, which are typically concerned only with well-typed terms.

65 Reed: Cite
66 something

67 However, core task of Panbench is not that different from the task of a logical framework [4]:
68 Both systems exist to manipulate judgements, inference rules, and derivations: Panbench
69 just works with *grammatical* judgements and production rules rather than typing judgments
70 and inference rules. In this sense Panbench is a *grammatical* framework² rather than a logical
71 one.

72 This similarity let us build Panbench atop well-understood design principles. In particular,
73 a mechanized logical framework typically consists of two layers:

- 74 1. A layer for defining judgements à la relations.
75 2. A logic programming layer for synthesizing derivations.

76 To use a logical framework, one first encodes a language by laying out all of the judgements.
77 Then, one needs to prove an adequacy theorem on the side that shows that their encoding of
78 the judgements actually aligns with the language. However, if one wanted to mechanize this

¹ All plots in this paper were produced directly by Panbench!

² Not to be confused with *the Grammatical Framework* [7], which aims to be a more general meta-framework for implementing and translating between grammars.

79 adequacy proof, then a staged third layer that consists of a more traditional proof assistant
 80 would be required.

81 If we take this skeleton of a design and transpose it to work with grammatical constructs
 82 rather than logical ones, we will also obtain three layers:

- 83 1. A layer for defining grammars à la relations.
 84 2. A logic programming layer for synthesizing derivations.
 85 3. A staged functional programming layer for proving “adequacy” results.

86 In this case, an adequacy result for a given language \mathcal{L} is a constructive proof that all
 87 grammatical derivations written within the framework can be expressed within the concrete
 88 syntax of a language \mathcal{L} . However, the computational content of such a proof essentially
 89 amounts to a compiler written in the functional programming layer.

90 5.1 Implementing The Grammatical Framework

91 Implementing a bespoke hybrid of a logic and functional programming language is no small
 92 feat, and also requires prospective users to learn yet another single-purpose tool. Luckily,
 93 there already exists a popular, industrial-grade hybrid logic/functional programming language
 94 in wide use: GHC Haskell.

95 At first glance, Haskell does not contain a logic programming language. However, if we
 96 enable enough GHC extensions, the typeclass system can be made *just* rich enough to encode
 97 a simply-typed logical framework. The key insight is that we can encode each production
 98 rule using multi-parameter type classes with a single method. Moreover, we can encode our
 99 constructive adequacy proofs for a given set of production rules as instances that translate
 100 each of the productions in the abstract grammar to productions in the syntax of an actual
 101 language.

102 As a concrete example, consider the grammar of the following simple imperative language.

```
103 <expr> := x
104   | n
105   | <expr> '+' <expr>
106   | <expr> '*' <expr>
107
108 <stmt> := <var> '=' <expr>
109   | 'while' <expr> 'do' <stmt>
110   | <stmt> ';' <stmt>
```

112 We can then encode this grammar with the following set of multi-parameter typeclasses:

```
113
114 class Var expr where
115   var :: String → expr
116
117 class Lit expr where
118   lit :: Int → expr
119
120 class Add expr where
121   add :: expr → expr → expr
122
123 class Mul expr where
124   mul :: expr → expr → expr
```

23:4 Panbench: A Comparative Benchmarking Tool for Dependently-Typed Languages

```
126 class Assign expr stmt where
127   assign :: String → expr → stmt
128
129 class While expr stmt where
130   while :: expr → stmt → stmt
131
132 class AndThen stmt where
133   andThen :: stmt → stmt → stmt
134
```

135 This style of language encoding is typically known as the untyped variant of *finally*
136 *tagless*[3], and is well-known technique. However, our encoding is a slight refinement of
137 the usual tagless style. In particular, we restrict ourselves to a single class per production
138 rule, whereas other tagless encodings often use a class per syntactic category. This more
139 fine-grained approach allows us to encode grammatical constructs that are only supported
140 by a subset of our target grammars; see section 5.2 for examples.

141 Unfortunately, the encoding above has some serious ergonomic issues. In particular,
142 expressions like `assign "x"` (lit 4) will result in an unsolved metavariable for `expr`, as there
143 may be multiple choices of `expr` to use when solving the `Assign ?expr stmt` constraint. Luckily,
144 we can resolve ambiguities of this form through judicious use of functional dependencies[5],
145 as demonstrated below.

```
146 class Assign expr stmt | stmt -> expr where
147   assign :: String -> expr -> stmt
148
149 class While expr stmt | stmt -> expr where
150   while :: expr -> stmt -> stmt
151
152
```

153 5.2 Designing The Language

154 Reed: Too informal, just brain dumping.

155 Now that we've fleshed out how we are going to encode our grammatical framework into
156 our host language, it's time to design our idealized abstract grammar. All of our target
157 languages roughly agree on a subset of the grammar of non-binding terms, though their
158 treatment of binding forms and top level definitions diverges.

159 5.2.1 Binding Forms

160 As users of dependently typed languages are well aware, a binding form carries much more
161 information than just a variable name and a type. Moreover, this extra information can have
162 a large impact on typechecking performance, as is the case with implicit/visible arguments.
163 To make matters worse, languages often offer multiple syntactic options for writing the same
164 binding form, as is evidenced by the prevalence of multi-binders like $(x\ y\ z : A) \rightarrow B$. Though
165 such binding forms are often equivalent to their single-binder counterparts as *abstract* syntax,
166 they may have different performance characteristics, so we cannot simply lower them to a
167 uniform single-binding representation. To account for these variations, we have designed a
168 sub-language dedicated solely to binding forms. This language classifies the various binding
169 features along three separate axes: binding arity, binding annotations, and binding modifiers.

170 Binding arities and annotations are relatively self-explanatory, and classify the number of
171 names bound, along with the type of annotation allowed. Our target languages all have their

172 binding arities falling into one of three classes: *n*-ary, unary, or nullary. We can similarly
173 characterise annotations into three categories: required, optional, or forbidden.

174 This language of binders is encoded in the implementation as a single class `Binder` that is
175 parameterised by higher-kinded types `arity :: Type -> Type` and `ann :: Type -> Type`, and
176 provide standardized types for all three binding arities and annotations.

```
177 class Binder arity nm ann tm cell | cell -> nm tm where
178   binder :: arity nm -> ann tm -> cell
```

181 Production rules that involve binding forms are encoded as classes that are parametric over
182 a notion of a binding cell, as demonstrated below.

```
183 -- | Pi types .
184 class Pi cell tm | tm -> cell where
185   -- | Create a pi type over a list of binding cells .
186   pi :: [cell] -> tm -> tm
```

189 Decoupling the grammar of binding forms from the grammar of binders themselves allows us to be somewhat polymorphic over the language of binding forms when writing generators.
190 This in turn means that we can potentially re-use generators when extending Panbench with
191 new target grammars that may support only a subset of the binding features present in our
192 four target grammars.

194 Binding modifiers, on the other hand, require a bit more explanation. A binding modifier captures features like implicit arguments, which do not change the number of names bound nor their annotations, but rather how those bound names get treated by the rest of the system. Currently, Panbench only supports visibility-related modifiers, but we have designed the system so that it is easy to extend with new modifiers; EG: quantities in Idris 2 or irrelevance annotations in Agda.

200 The language of binding modifiers is implemented as the following set of Haskell type-classes.

```
202 class Implicit cell where
203   implicit :: cell -> cell
205
206 class SemiImplicit cell where
207   semiImplicit :: cell -> cell
```

209 This is a case where the granular tagless encoding shines. Both Lean 4 and Rocq have a form of semi-implicits³, whereas Idris 2 and Agda have no such notion. Decomposing the language of binding modifiers into granular pieces lets us write benchmarks that explicitly require support for features like semi-implicits. Had we used a monolithic class that encodes the entire language of modifiers, we would have to resort to runtime errors (or, even worse, dubious attempts at translation).

215 5.2.2 Top-Level Definitions

216 The question of top-level definitions is much thornier, and there seems to be less agreement
217 on how they ought to be structured. The largest source of divergence is type signatures.

³ We use the term *semi-implicit* argument to refer to an implicit argument that is not eagerly instantiated. In Rocq these are known as non-maximal implicits.

23:6 Panbench: A Comparative Benchmarking Tool for Dependently-Typed Languages

218 Languages that prioritize dependent pattern matching typically opt to have standalone type
219 signatures: this allow for top-level pattern matches, which in turn makes it much easier to
220 infer motives[6]. Conversely, languages oriented around tactics typically opt for in-line type
221 signatures and pattern-matching expressions. This appears to be largely independent of
222 Miranda/ML syntax split: Lean 4 borrows large parts of its syntax from Haskell, yet still
223 opts for in-line signatures.

224 This presents us with a design decision: should our idealized grammar use inline or
225 standalone signatures? As long as we can (easily) translate from one style to the other, we
226 have a genuine decision to make. We have opted for the former as standalone signatures
227 offer variations that languages with inline signatures cannot handle. As a concrete example,
228 consider the following Agda declaration:

```
230     id : (A : Type) -> A -> A
 231     id B x = x
```

233 In particular, note that we have bound first argument to a different name. Translating
234 this to a corresponding Rocq declaration then forces us to choose to use either the name from
235 the signature or the term. Conversely, using in-line signatures does not lead us to having to
236 make an unforced choice when translating to a separate signature, as we can simply duplicate
237 the name in both the signature and term.

238 However, inline signatures are not completely without fault, and cause some edge cases
239 with binding modifiers. As an example, consider the following two variants of the identity
240 function in Agda.

```
241     id : {A : Type} -> A -> A
 242     id x = x
 243
 244     id' : {A : Type} -> A -> A
 245     id' {A} x = x
```

248 Both definitions mark the `A` argument as an implicit, but the second definition *also* binds
249 it in the declaration. When we pass to inline type signatures, we lose this extra layer of
250 distinction. To account for this, we were forced to refine the visibility modifier system to
251 distinguish between “bound” and “unbound” modifiers. This extra distinction has not proved
252 to be too onerous in practice, and we still believe that inline signatures are the correct choice
253 for our application.

254 Unfortunately, we are not out of the woods yet: like binders, there are a plethora of
255 different top-level definition forms. The key insight we had was that top level definitions *are*
256 *binding forms*, and that both should share the same underlying language. This perspective
257 lets us simplify how we view some more baroque top-level bindings. As a contrived example,
258 consider the following Agda definition:

Reed: unicode :(

```
259
260 private instance abstract @irr @mixed foo bar : Nat -> _
261 foo _ = 0
262 bar zero = true
263 bar (suc _) = false
```

266 In our language of binders, this definition consists of a 2-ary annotated binding of the
267 names `foo`, `bar` that has had a sequence of binding modifiers applied to it, along with a
268 compound body which consists of a simple definition and a pattern-matching definition.

269 Reed: Talk about left-hand-sides
 270 .
 271 Reed: Present the panbench grammar in BNF
 272 .

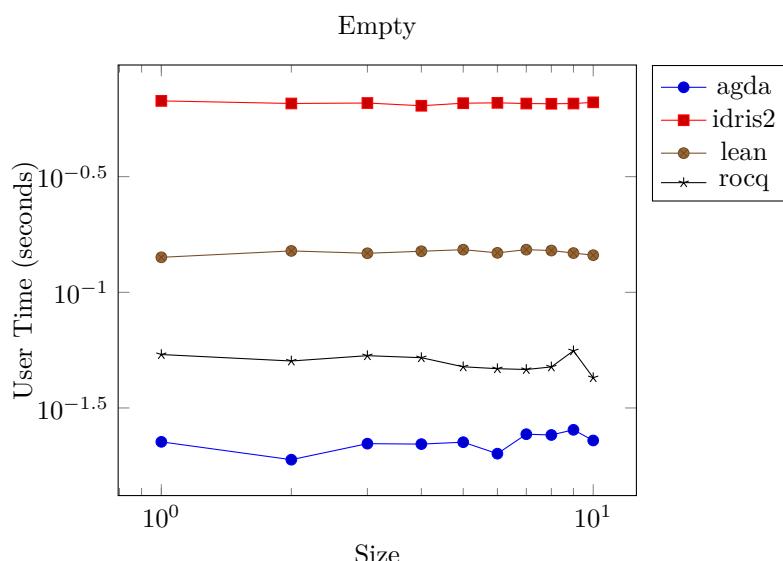
273 **6 Conclusion**

274 ————— **References** —————

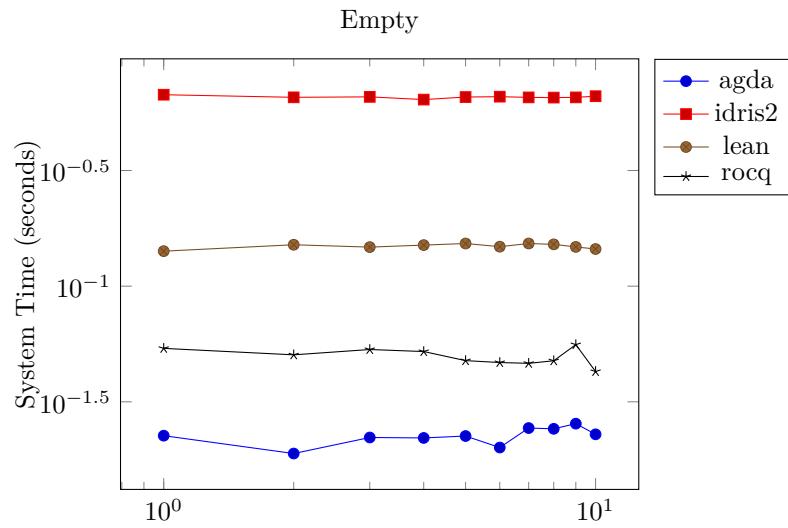
- 275 1 February 2026. URL: <https://github.com/GrammaticalFramework/informath>.
- 276 2 Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine
 277 Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti:
 278 a logical framework based on the $\lambda\pi$ -calculus modulo theory. (arXiv:2311.07185), November
 279 2023. arXiv:2311.07185 [cs]. URL: <http://arxiv.org/abs/2311.07185>, doi:10.48550/arXiv.
 280 2311.07185.
- 281 3 Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated:
 282 Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*,
 283 19(5):509–543, 2009. doi:10.1017/S0956796809007205.
- 284 4 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal
 285 of the ACM*, 40(1):143–184, January 1993. doi:10.1145/138027.138060.
- 286 5 Mark P. Jones. Type classes with functional dependencies. In Gert Smolka, editor, *Programming
 287 Languages and Systems*, page 230–244, Berlin, Heidelberg, 2000. Springer. doi:10.1007/
 288 3-540-46425-5_15.
- 289 6 Conor McBride and James Mckinna. The view from the left. *Journal of Functional Programming*,
 290 14(1):69–111, January 2004. doi:10.1017/S0956796803004829.
- 291 7 Arane Ranta. *Grammatical framework: programming with multilingual grammars*. CSLI
 292 studies in computational linguistics. CSLI Publications, Center for the Study of Language and
 293 Information, Stanford, Calif, 2011.

294 **7 Charts**

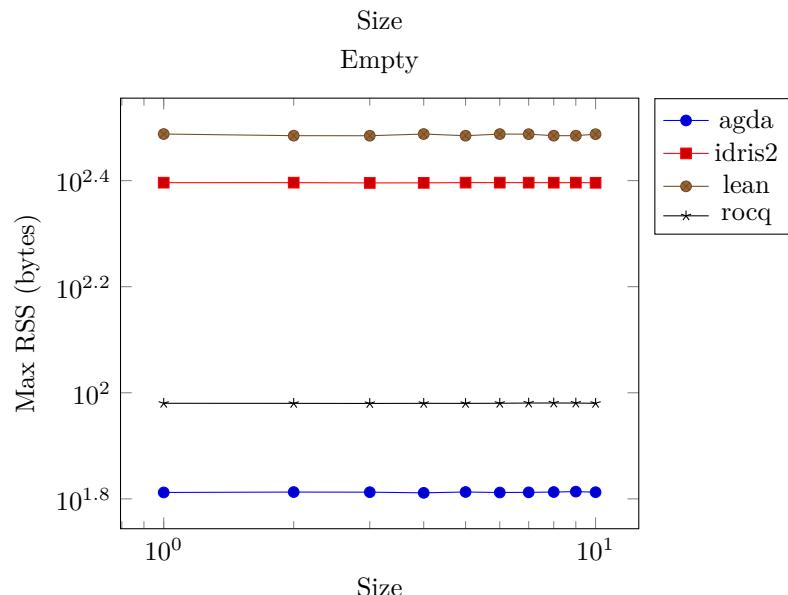
295 **7.1 Empty**



296

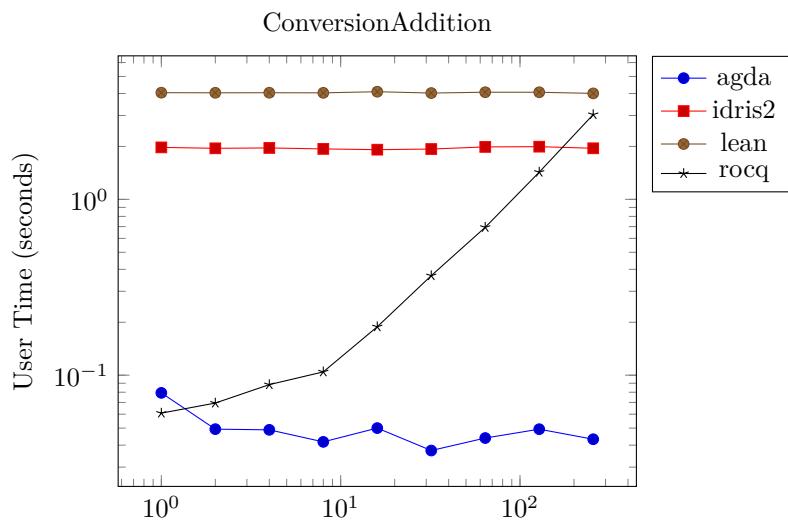


297

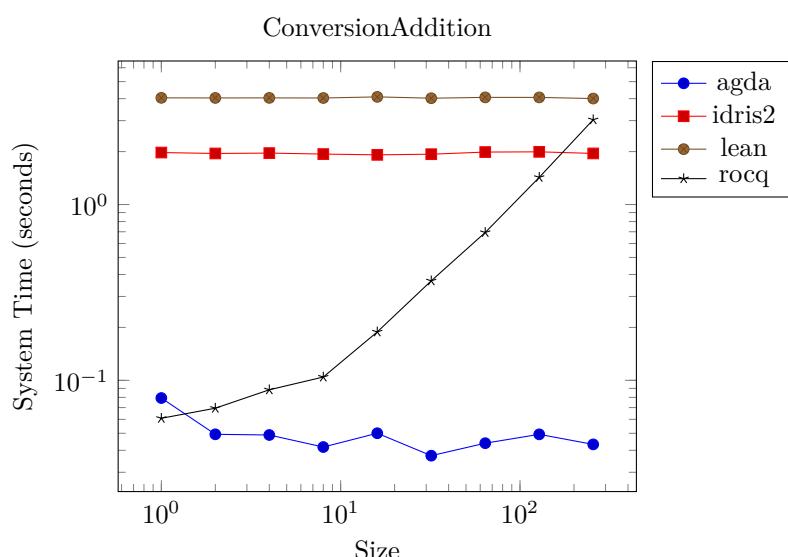


298

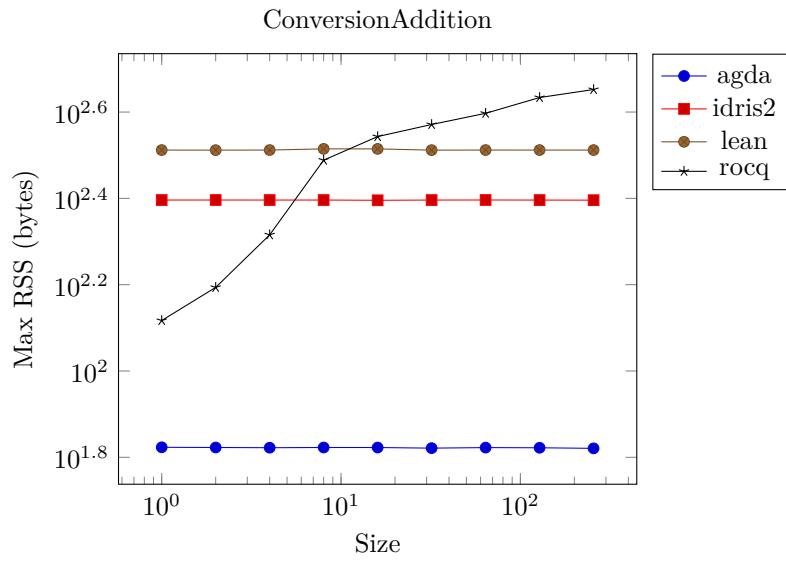
299 **7.2 ConversionAddition**



300

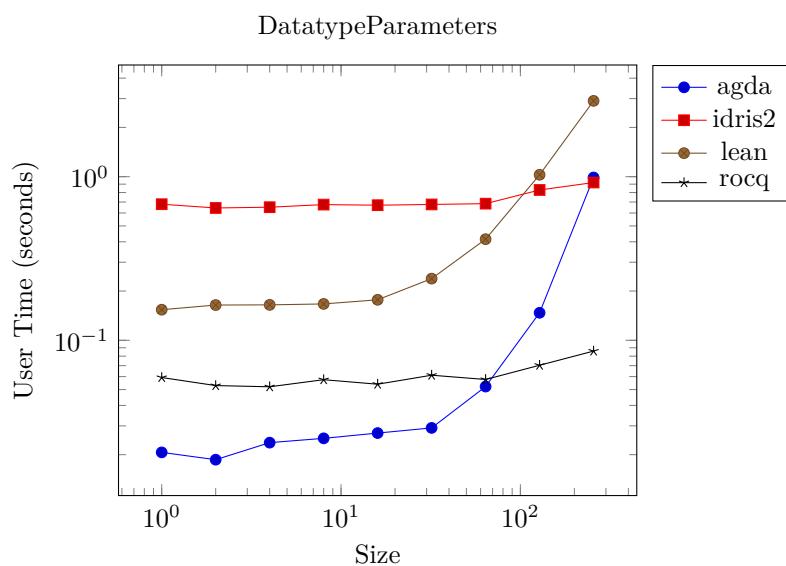


301



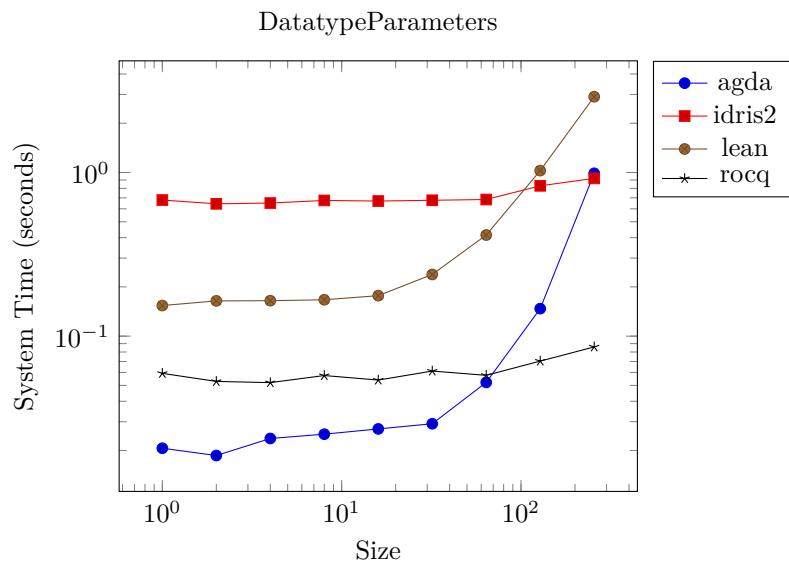
302

303 7.3 DatatypeParameters

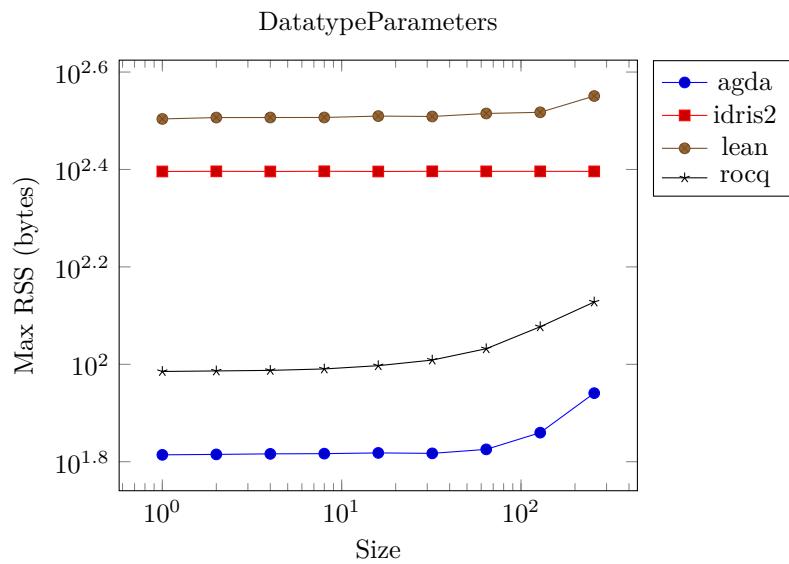


304

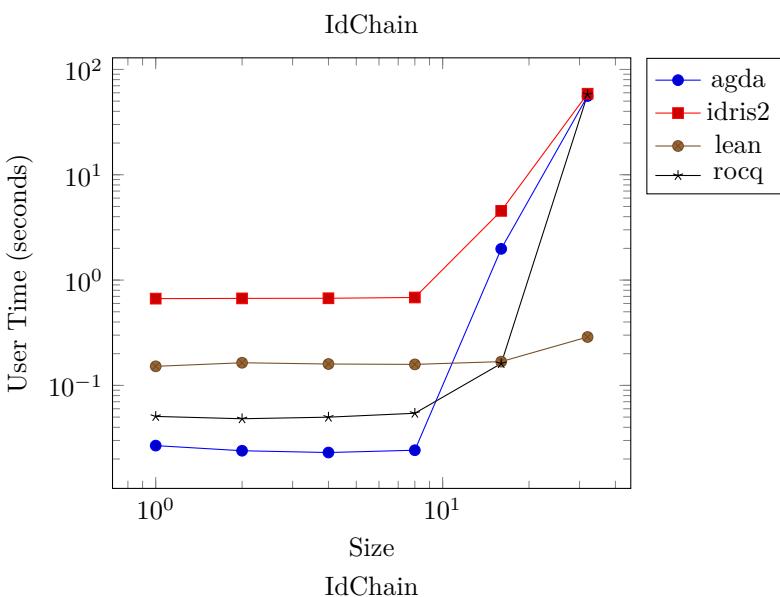
305



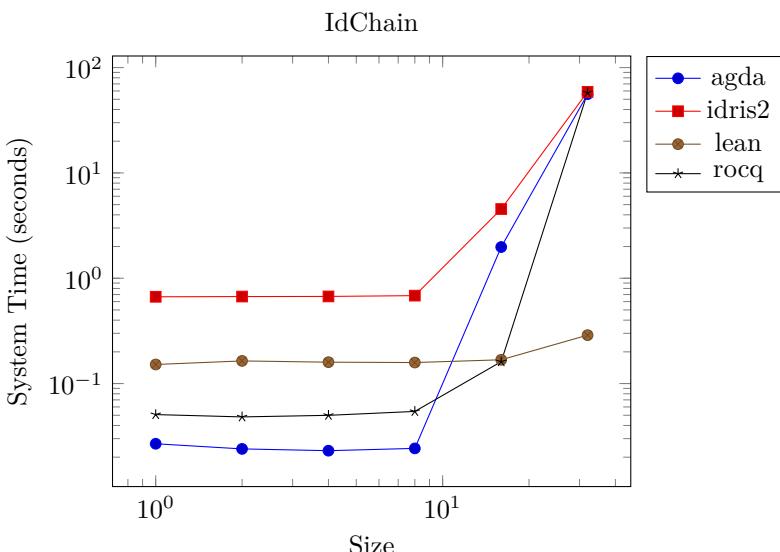
306



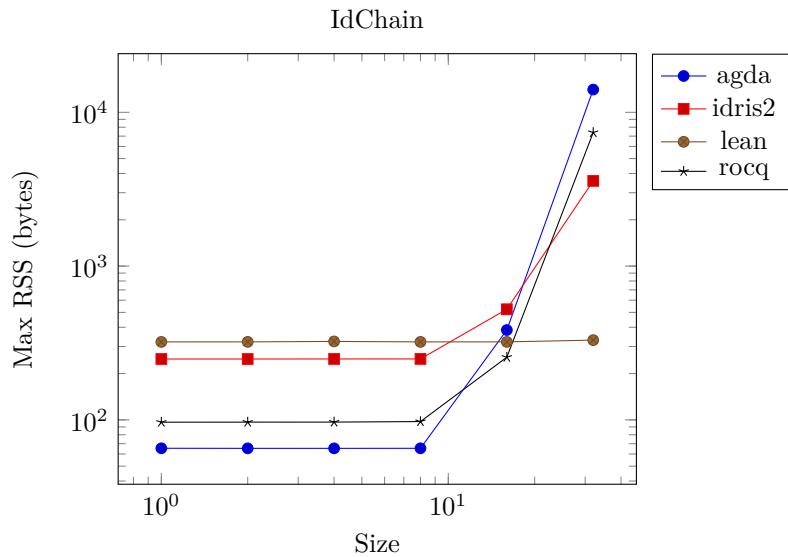
307 7.4 IdChain



308

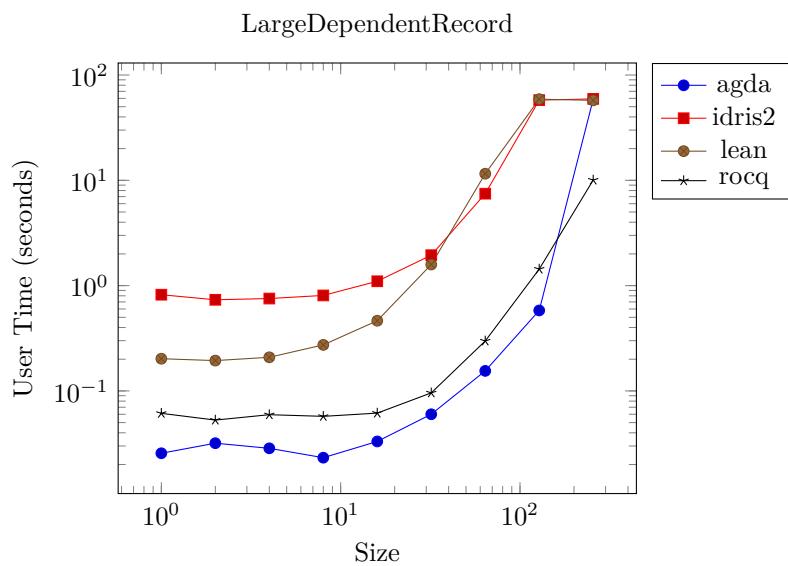


309

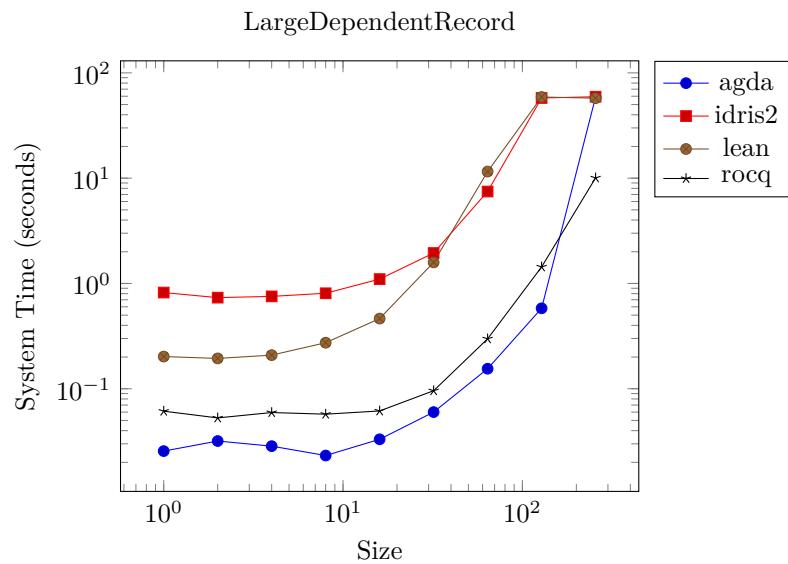


310

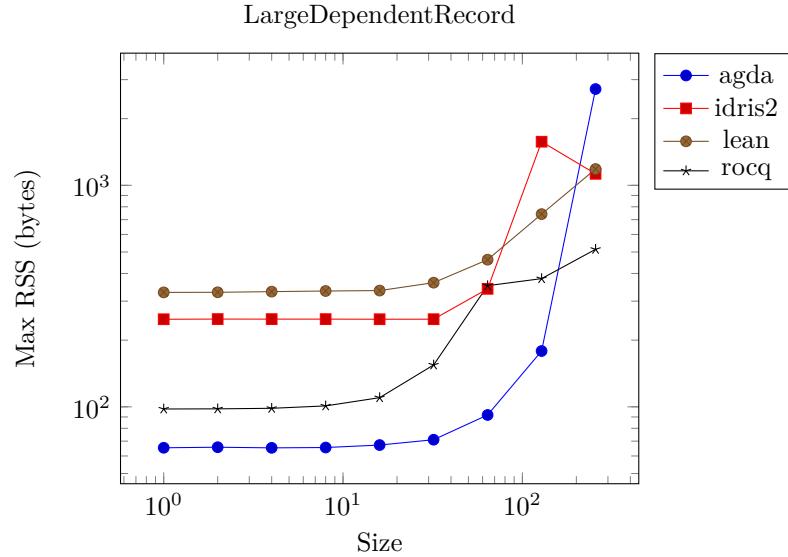
311 7.5 LargeDependentRecord



312

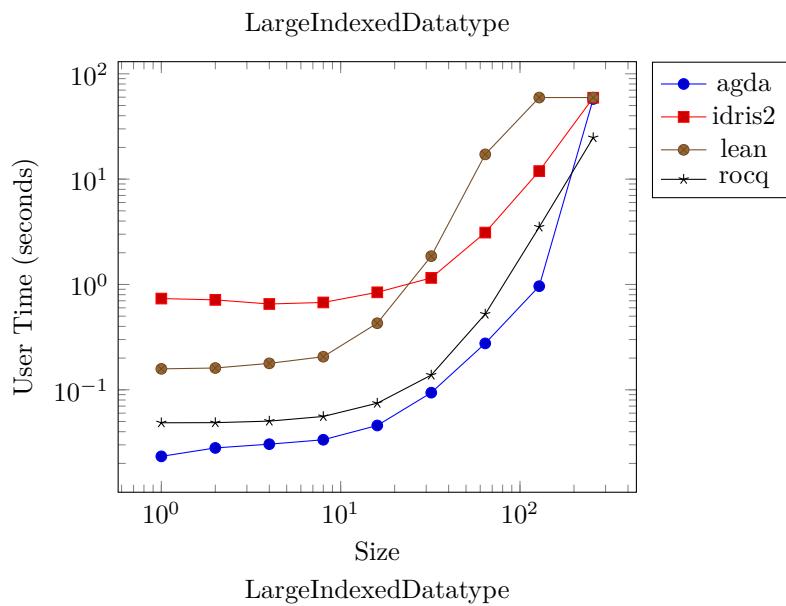
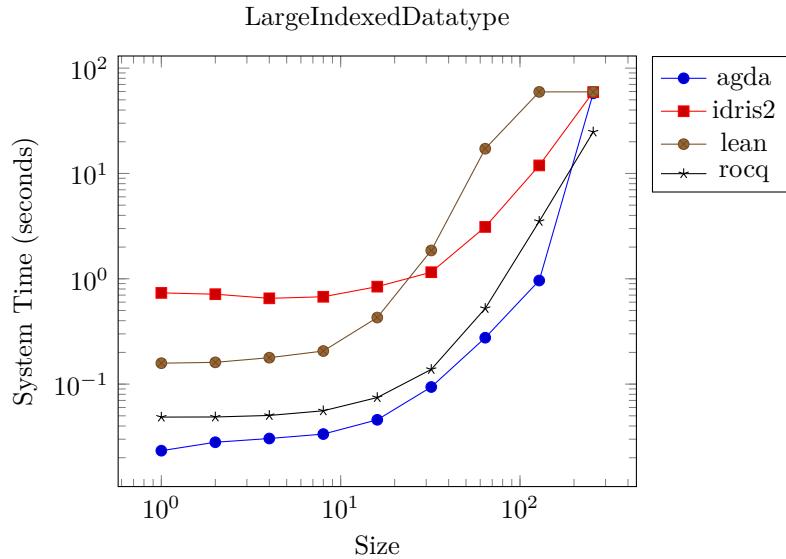


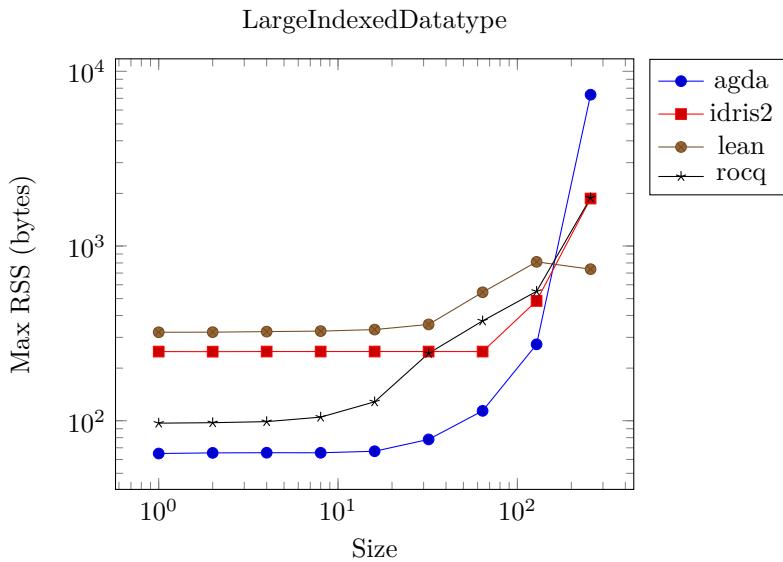
313



314

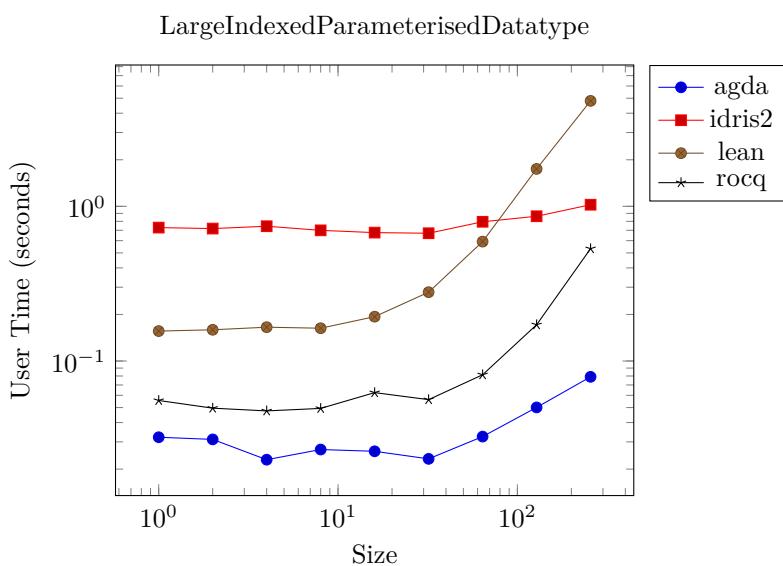
315 **7.6 LargeIndexedDatatype**

316317

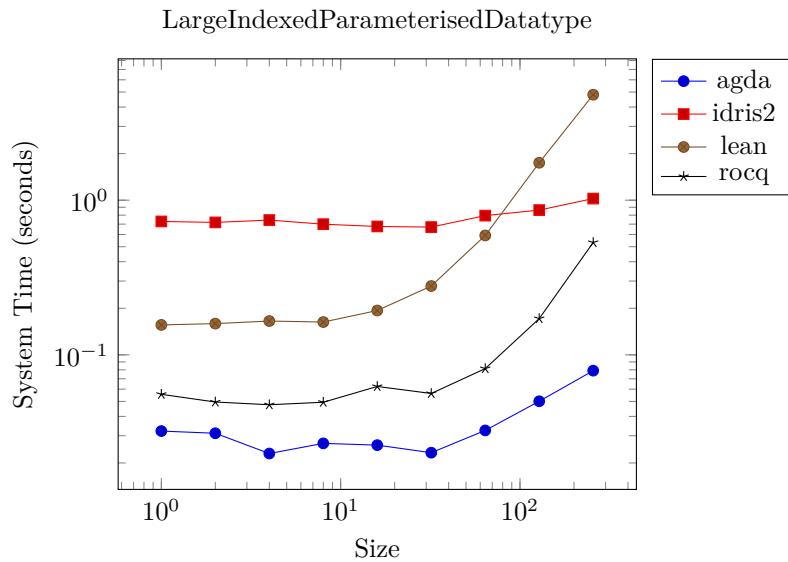


318

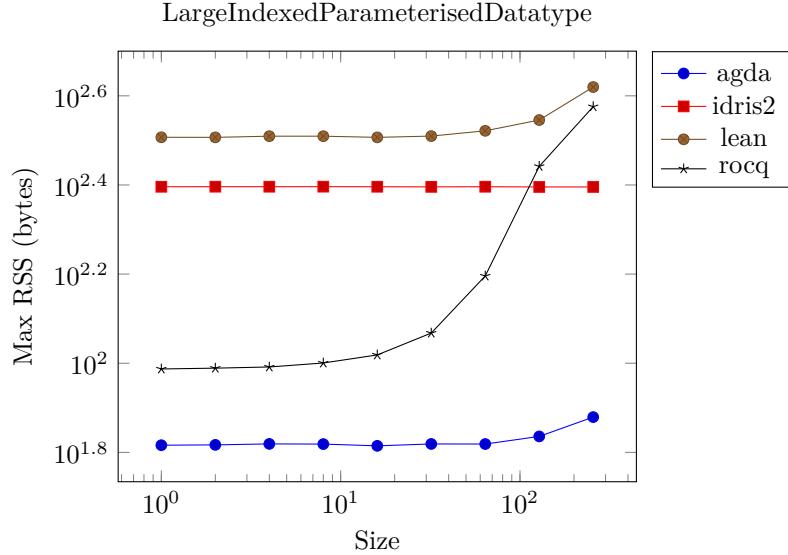
319 7.7 LargeIndexedParameterisedDatatype



320

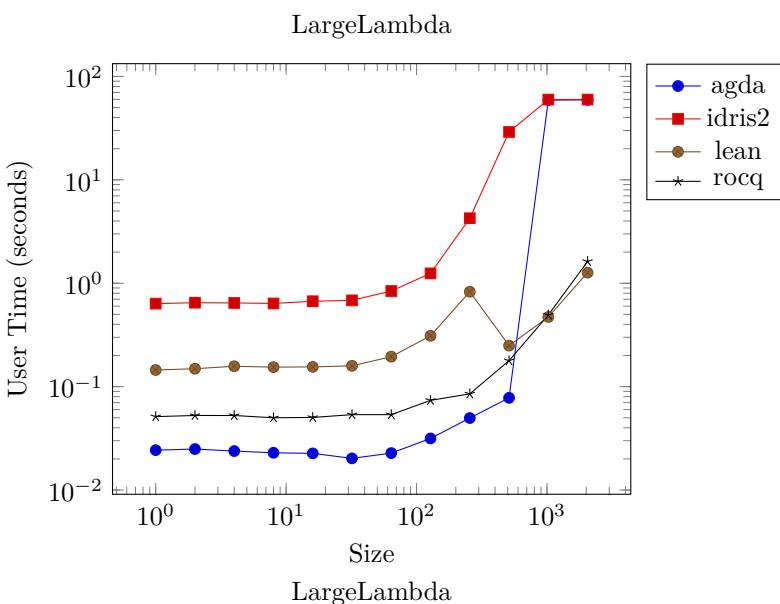


321

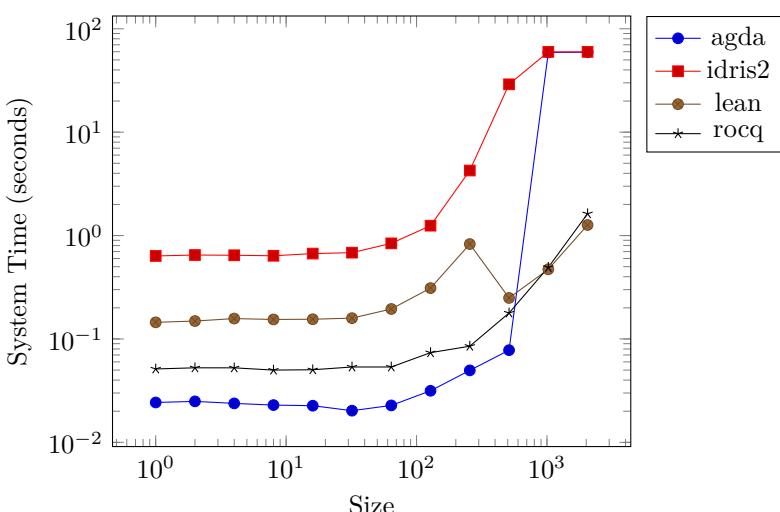


322

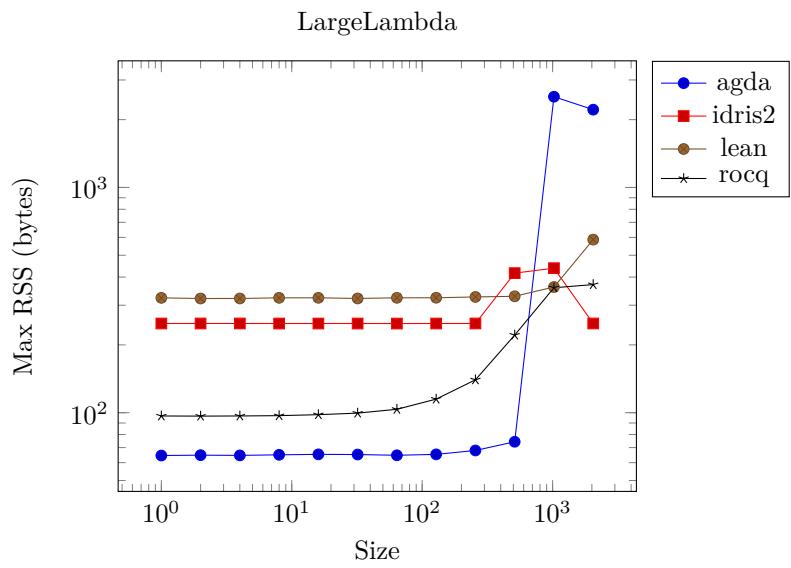
323 7.8 LargeLambda



324

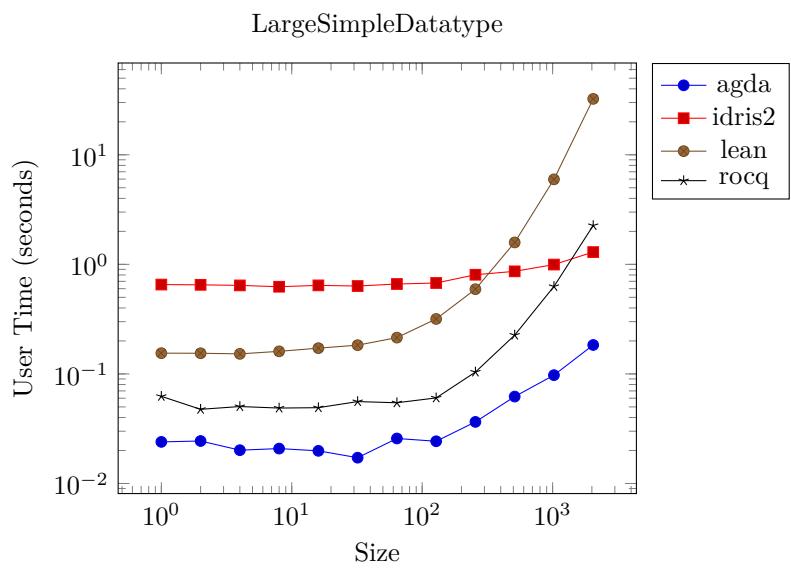


325

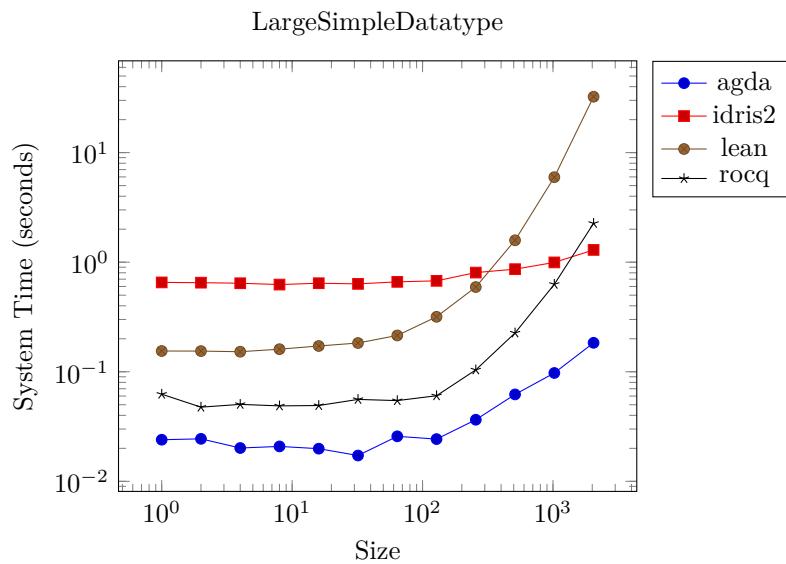


326

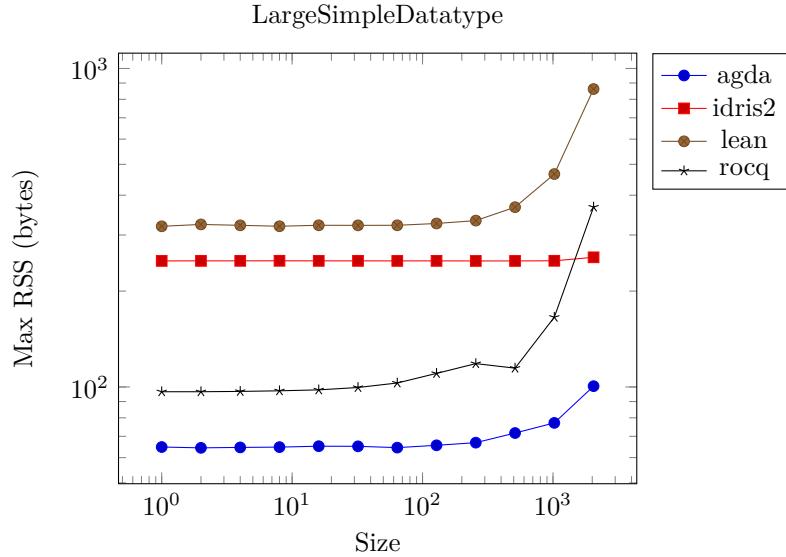
7.9 LargeSimpleDatatype



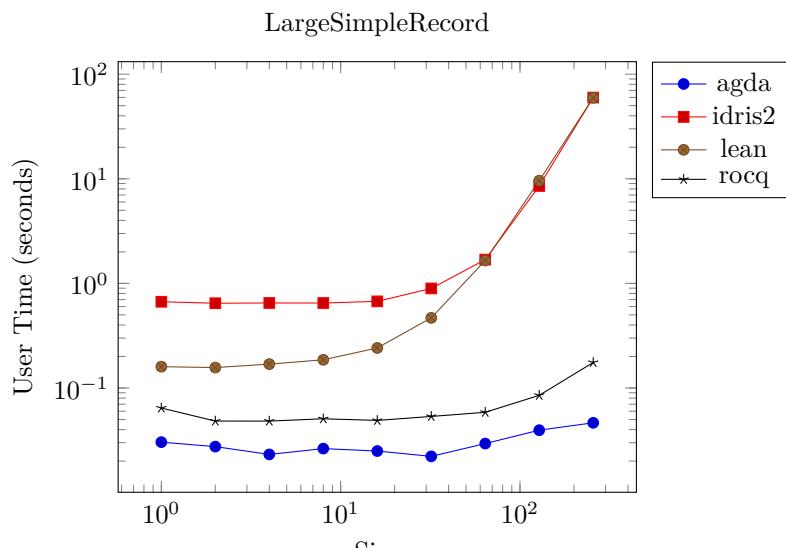
328



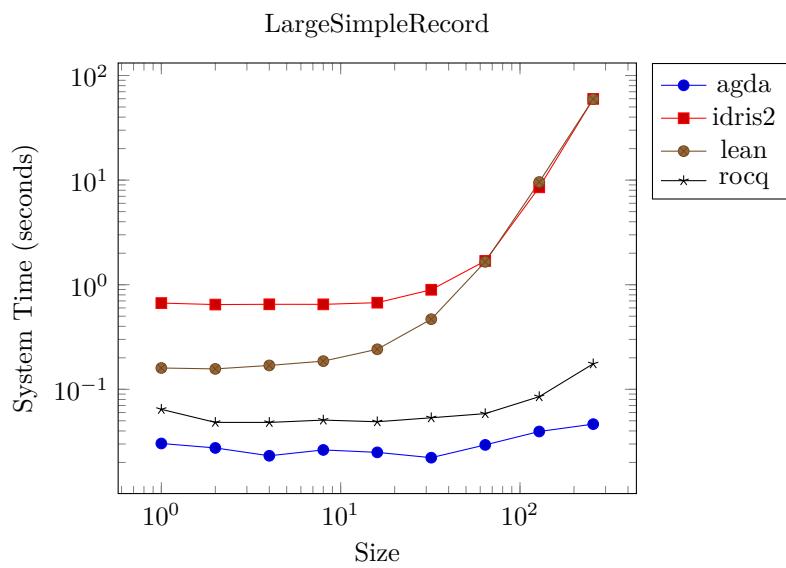
329



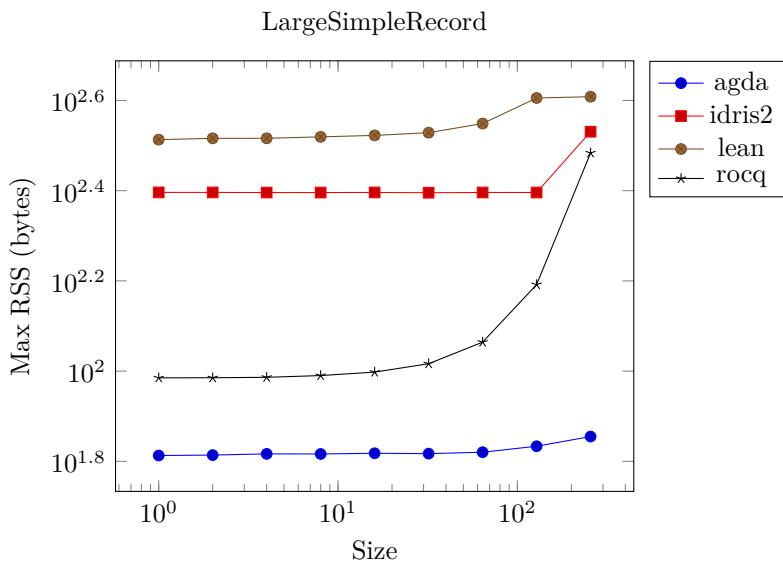
330

331 7.10 LargeSimpleRecord

332

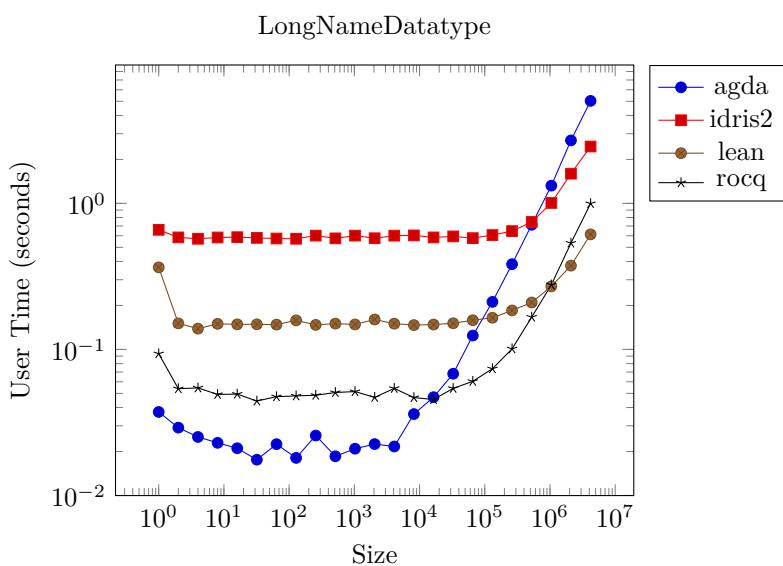


333

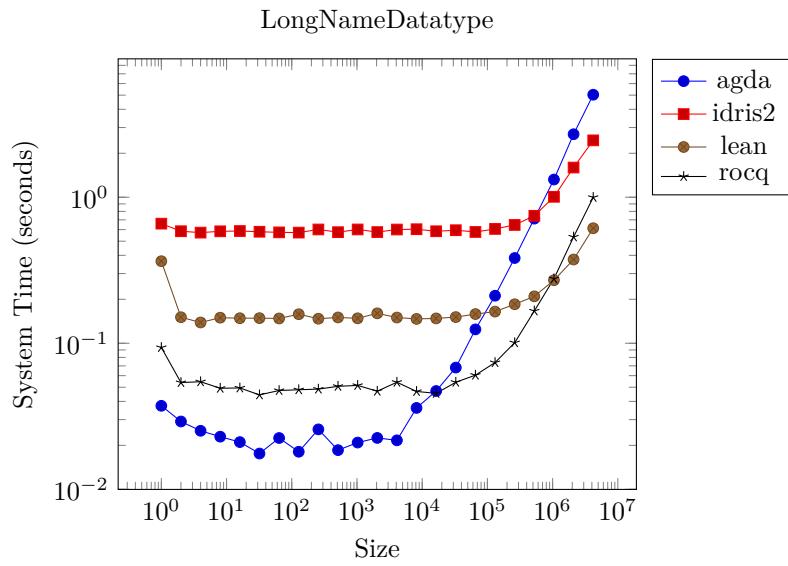


334

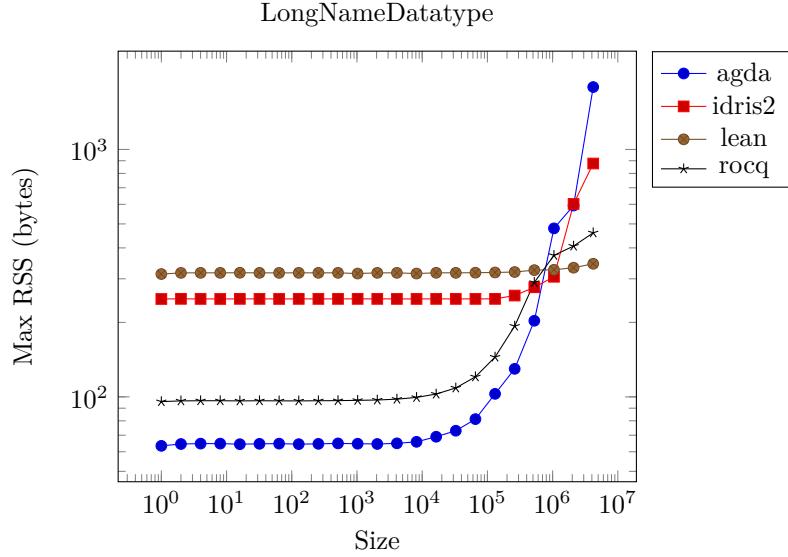
335 7.11 LongNameDatatype



336

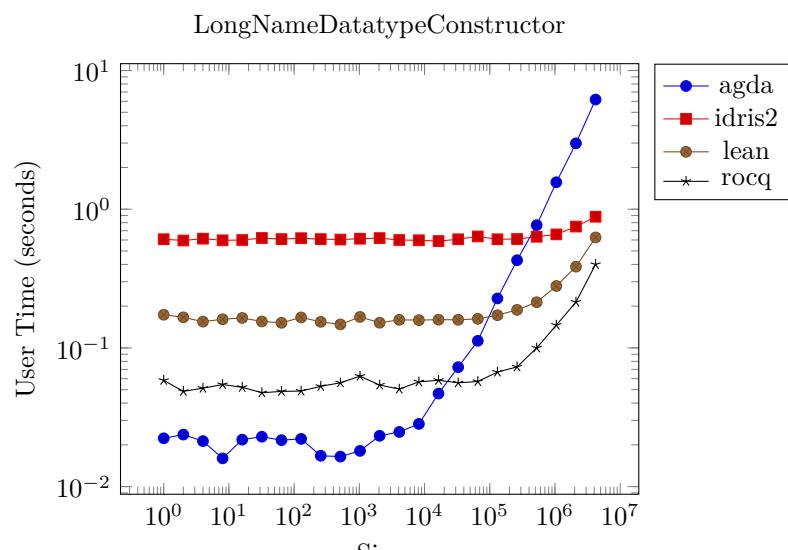
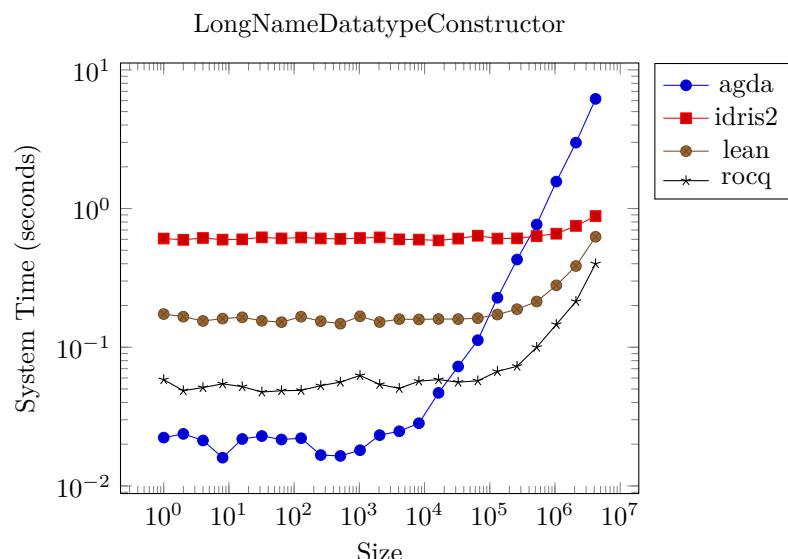


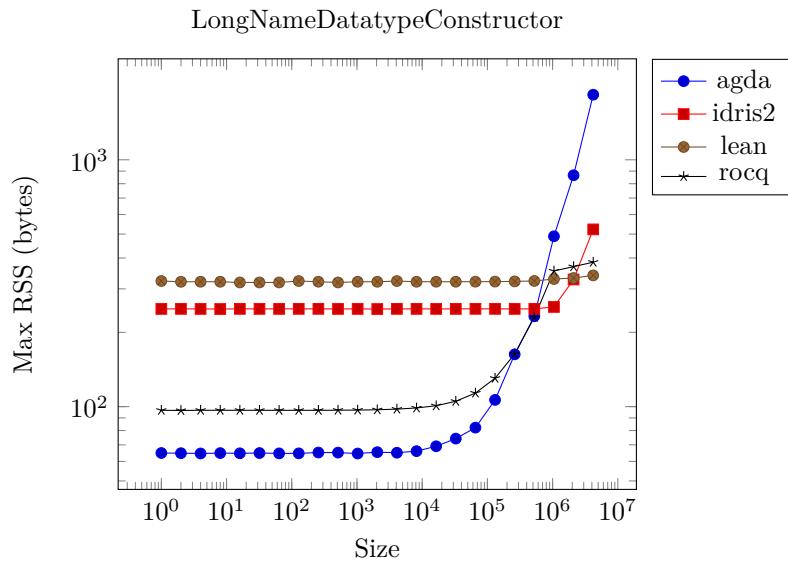
337



338

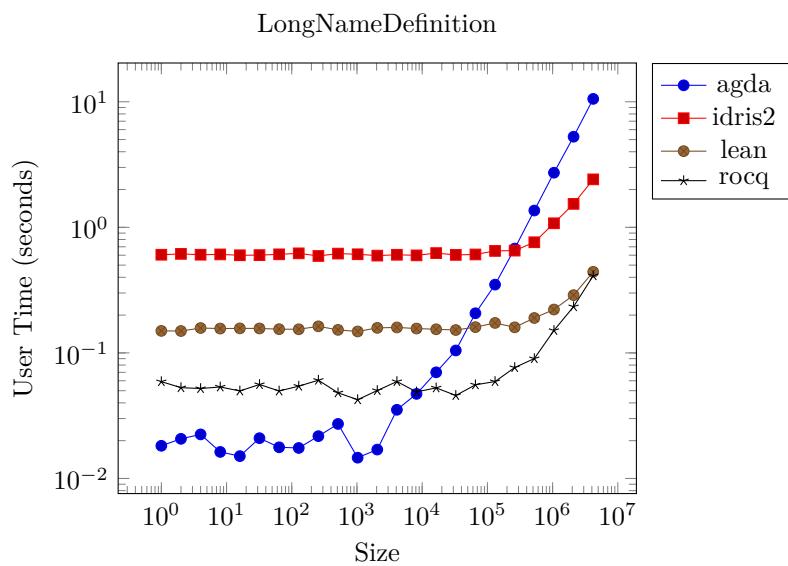
³³⁹ **7.12 LongNameDatatypeConstructor**

³⁴⁰³⁴¹

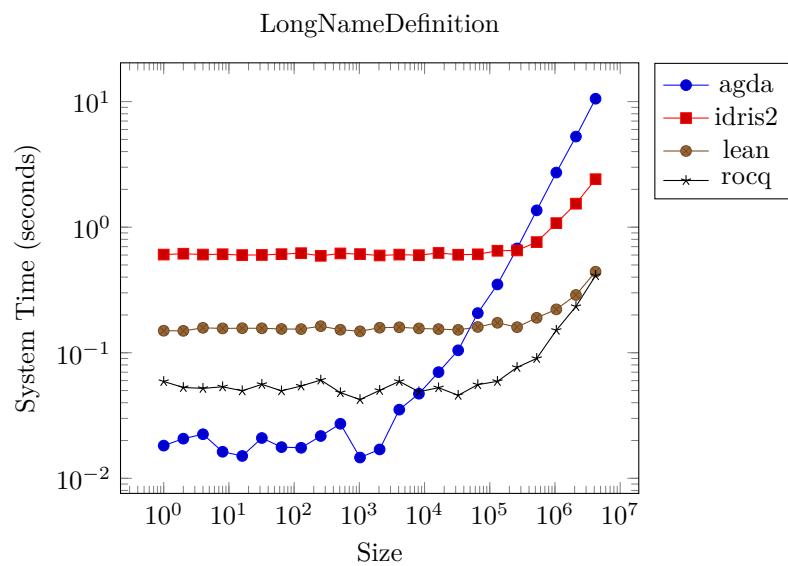


342

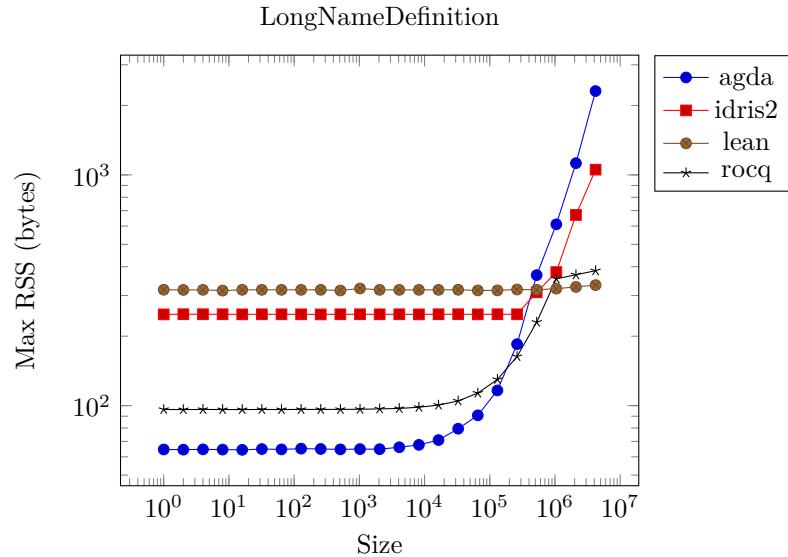
7.13 LongNameDefinition



343

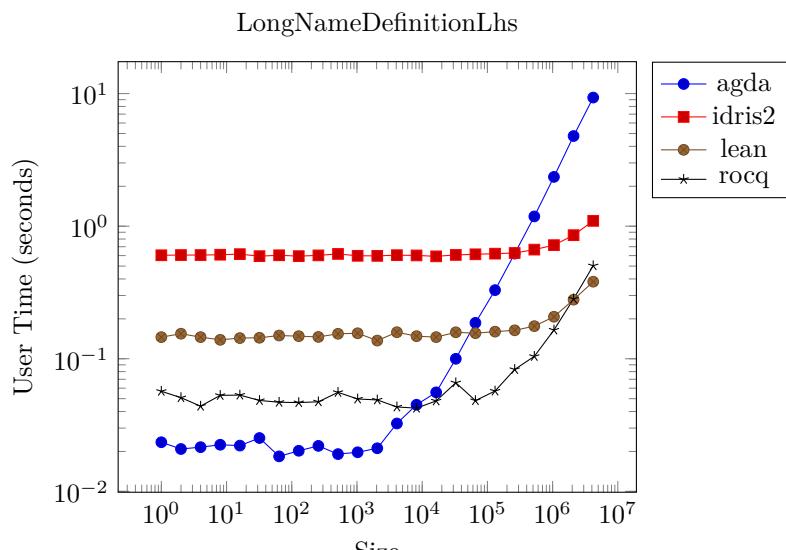


345

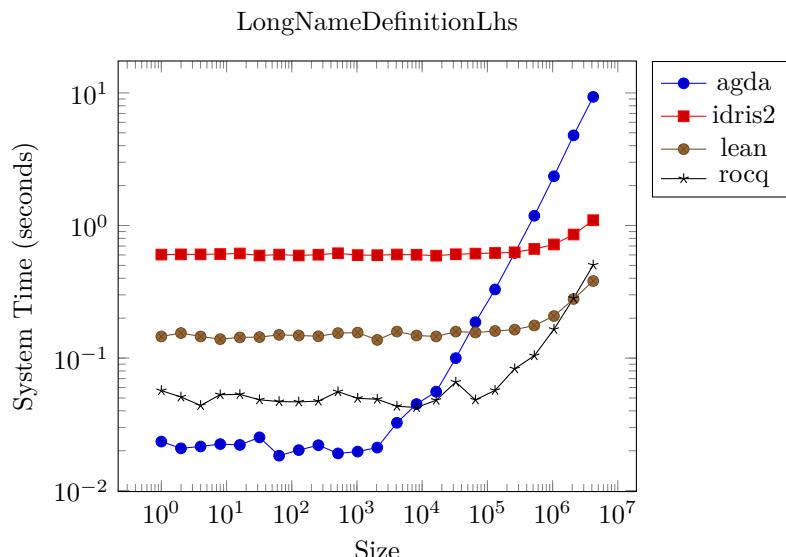


346

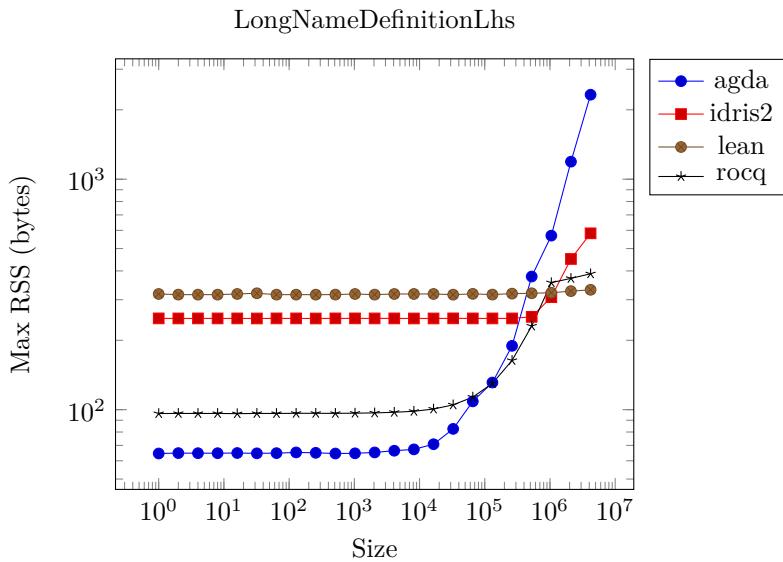
347 7.14 LongNameDefinitionLhs



348

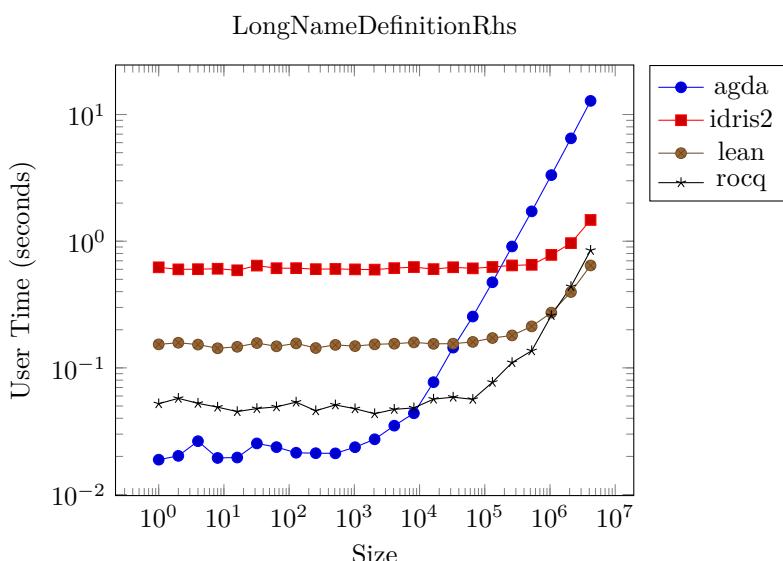


349

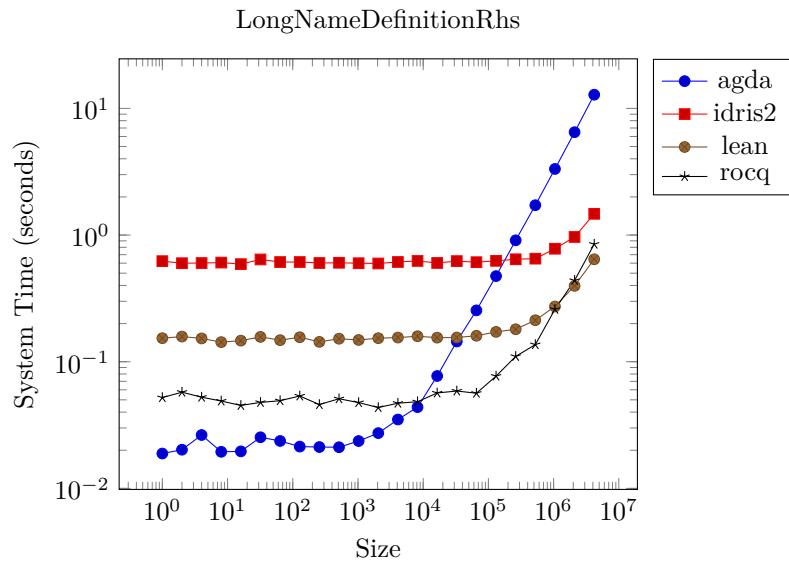


350

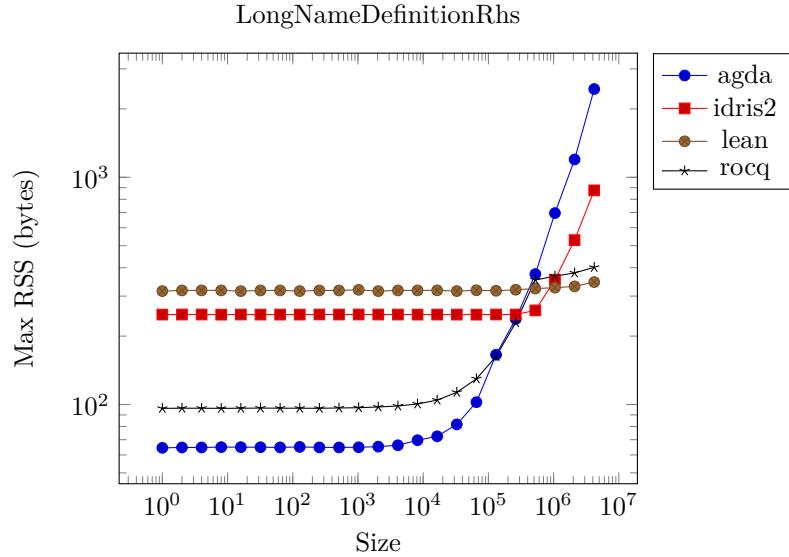
351 7.15 LongNameDefinitionRhs



352

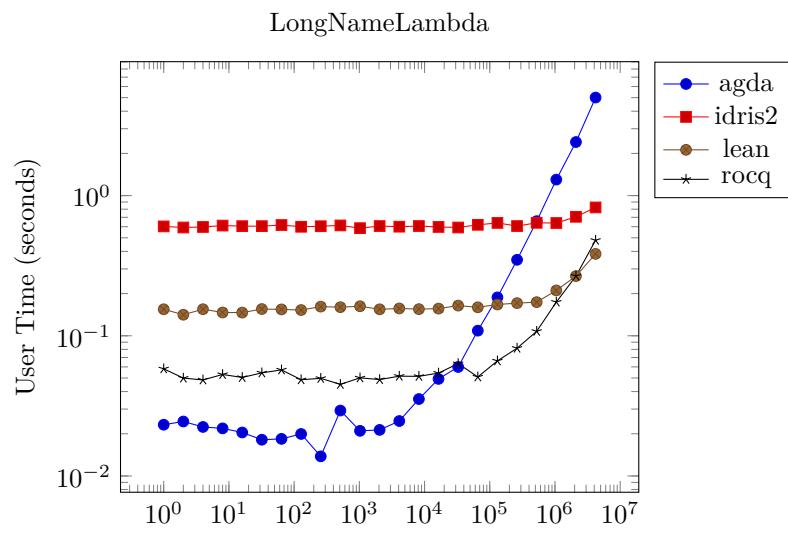


353

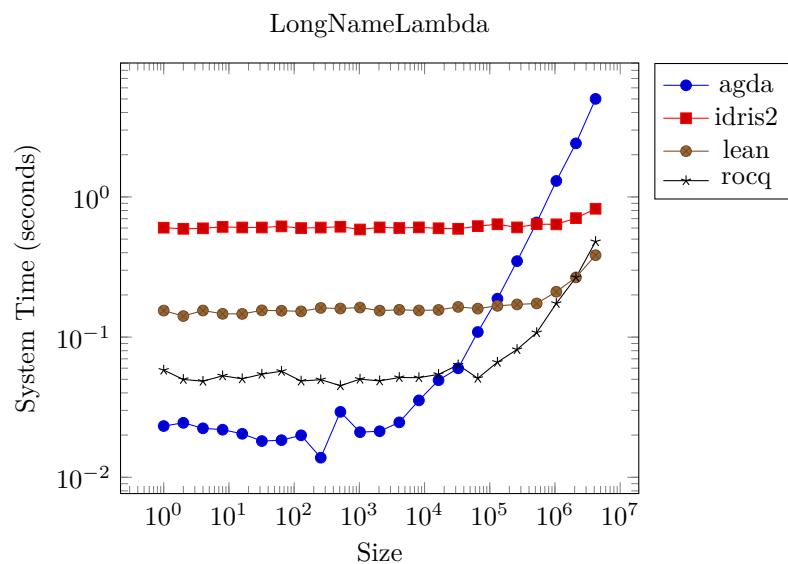


354

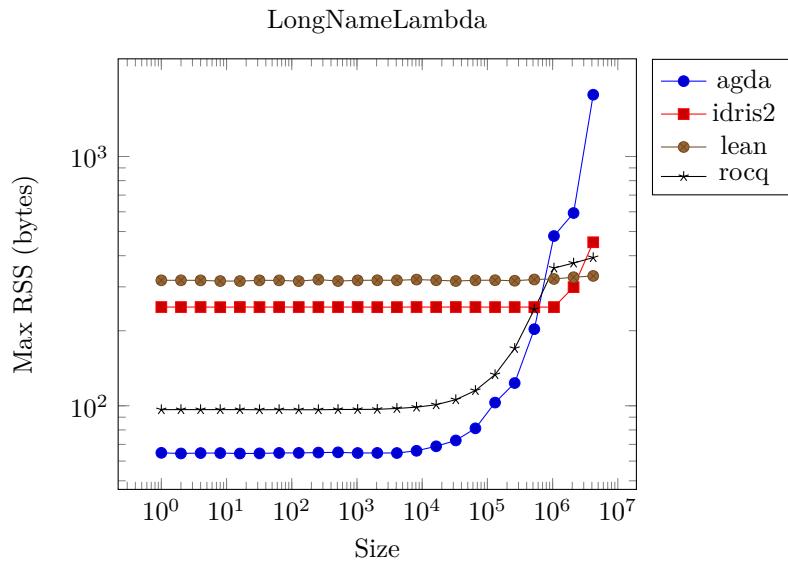
355 7.16 LongNameLambda



356

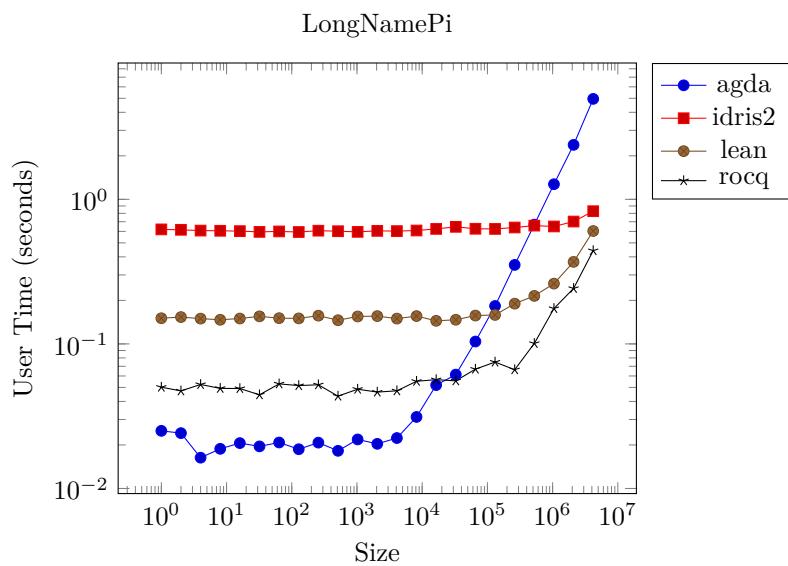


357

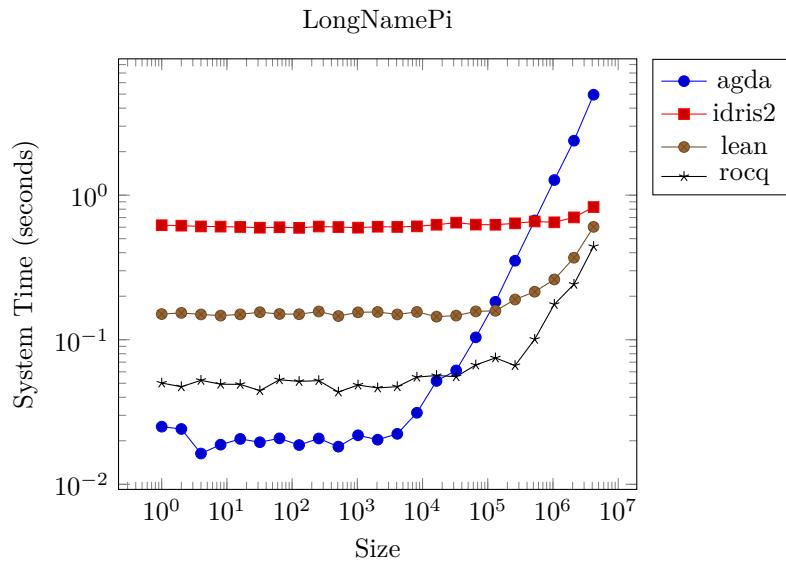


358

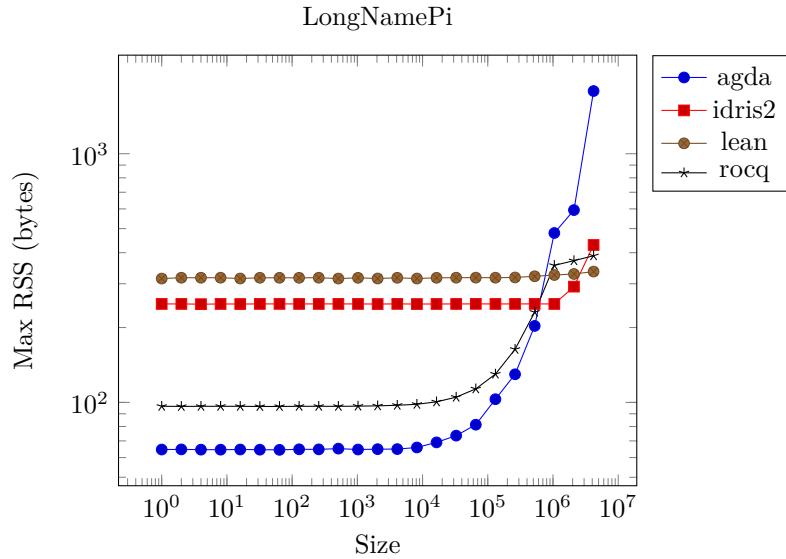
7.17 LongNamePi



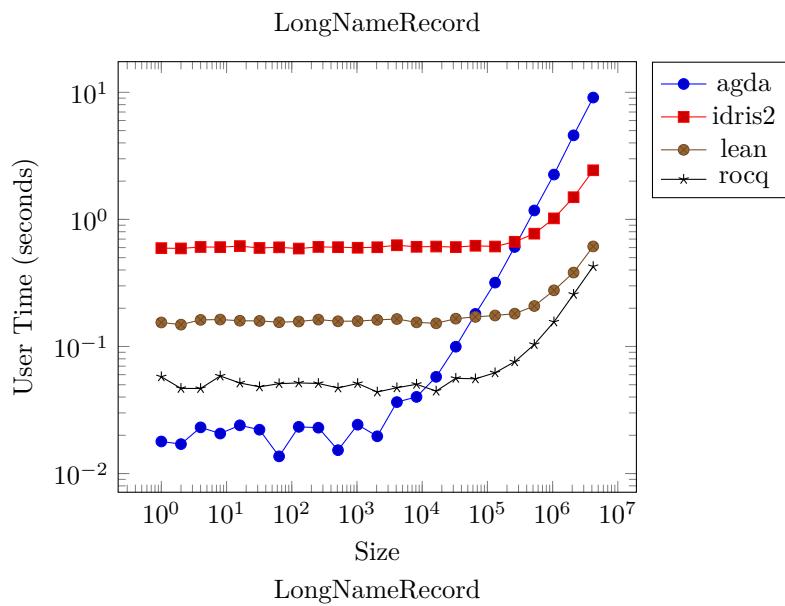
360



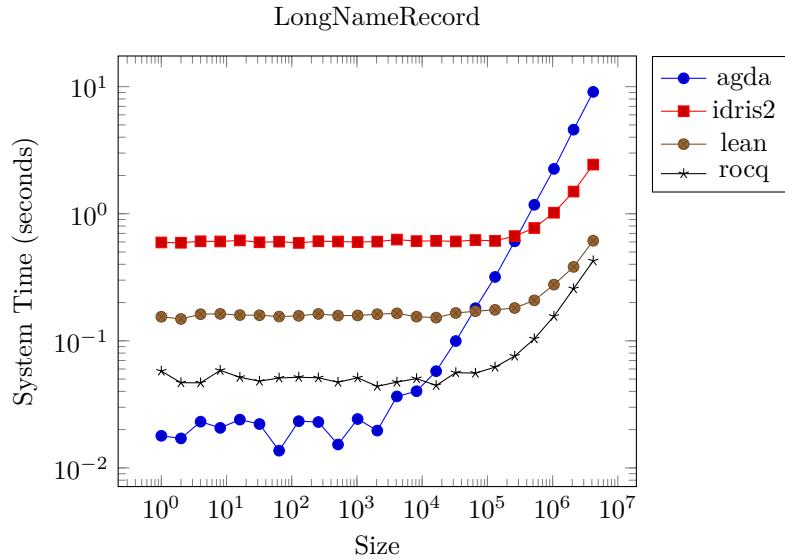
361



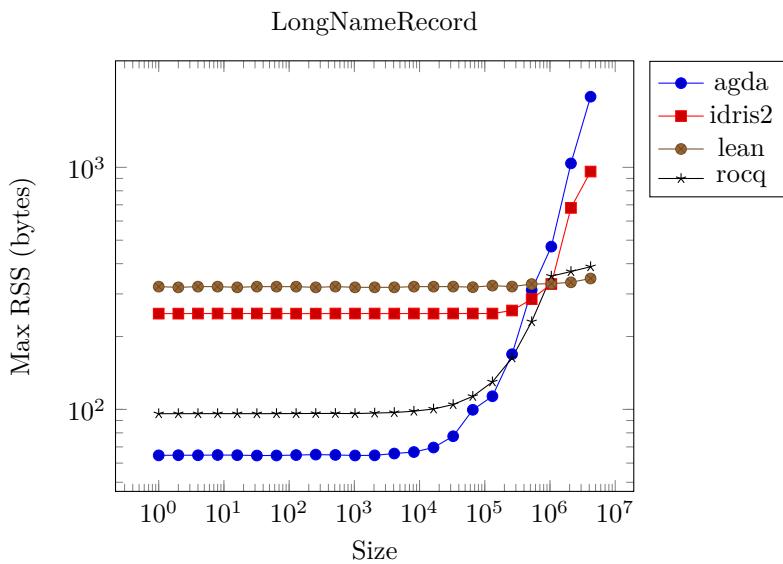
362

363 7.18 LongNameRecord

364

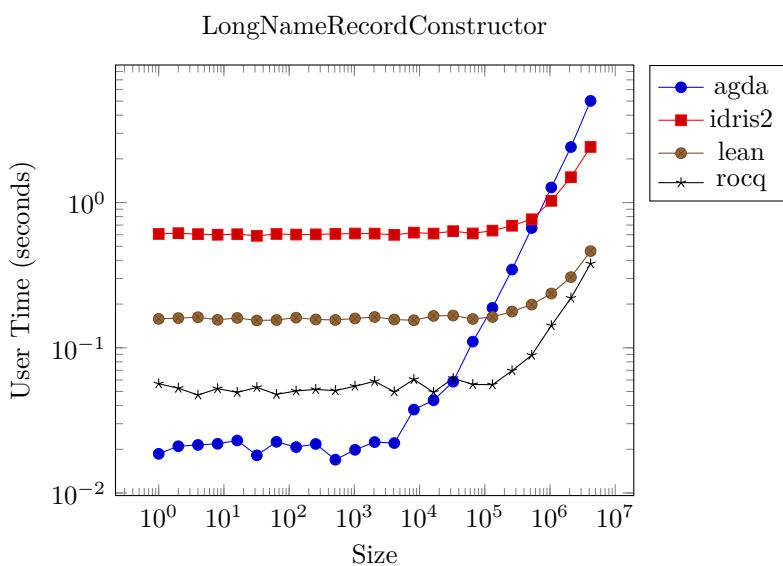


365

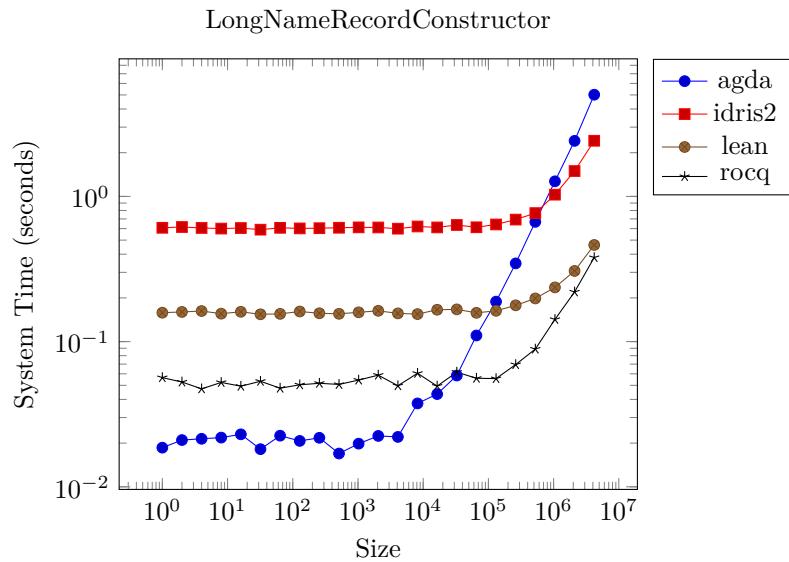


366

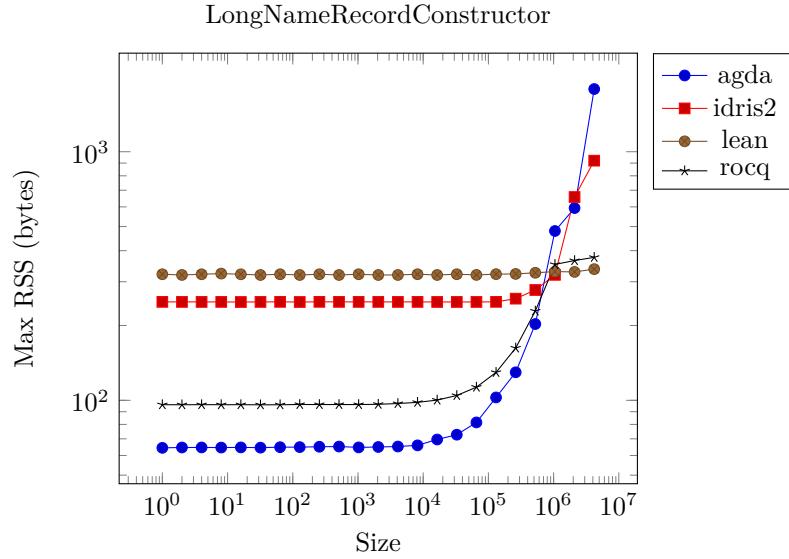
367 7.19 LongNameRecordConstructor



368

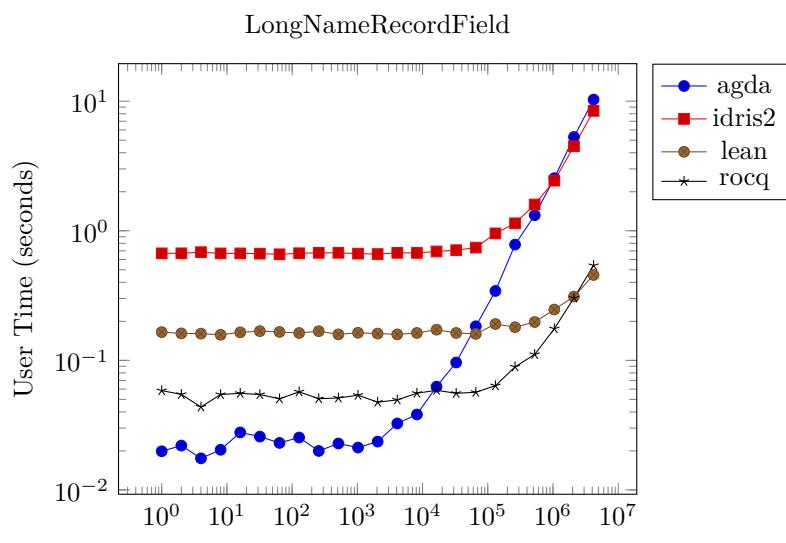


369

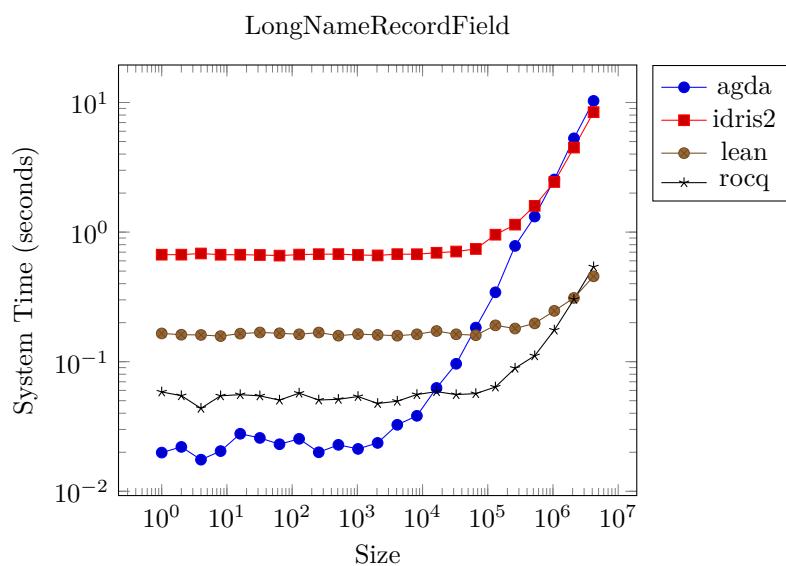


370

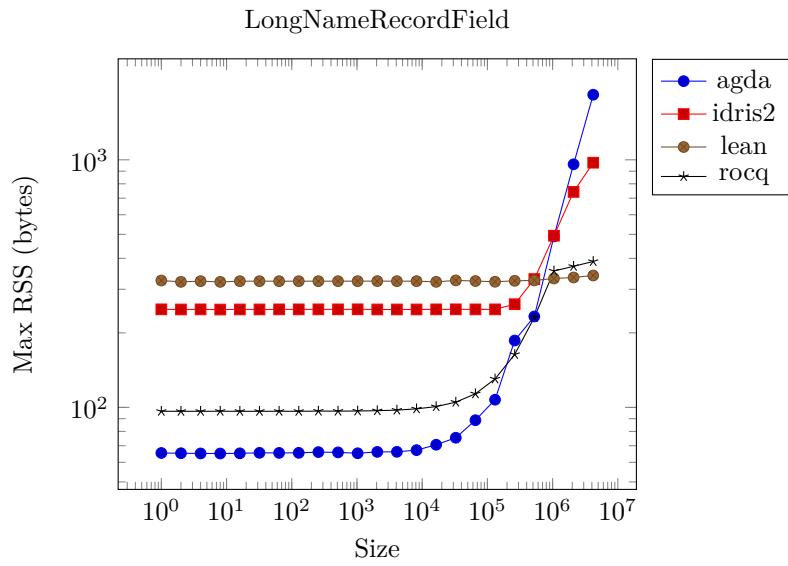
371 7.20 LongNameRecordField



372

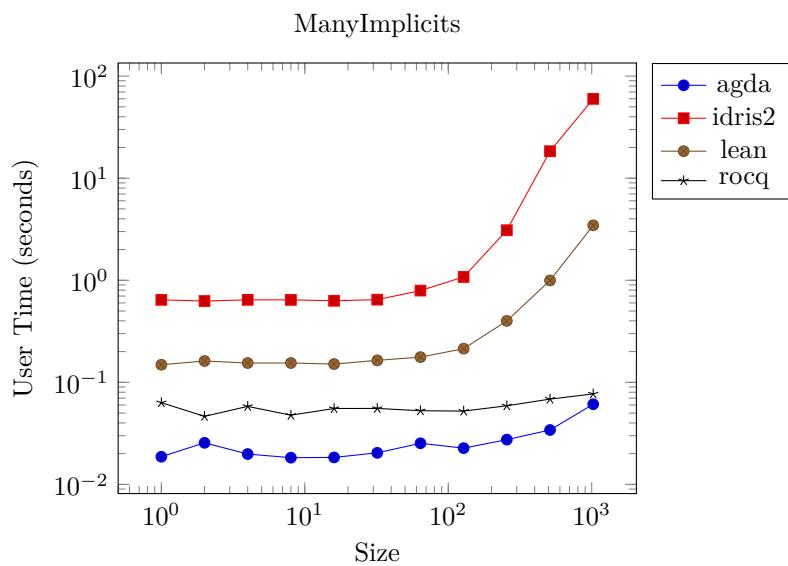


373

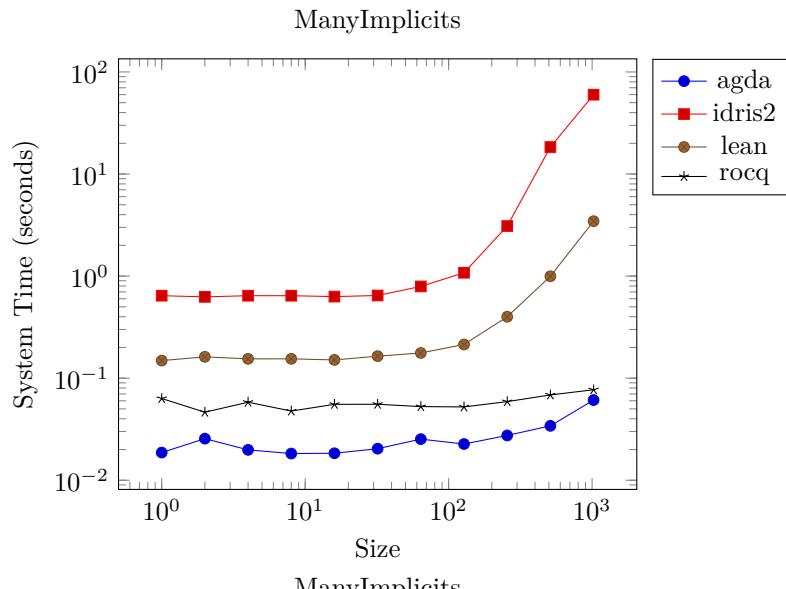


374

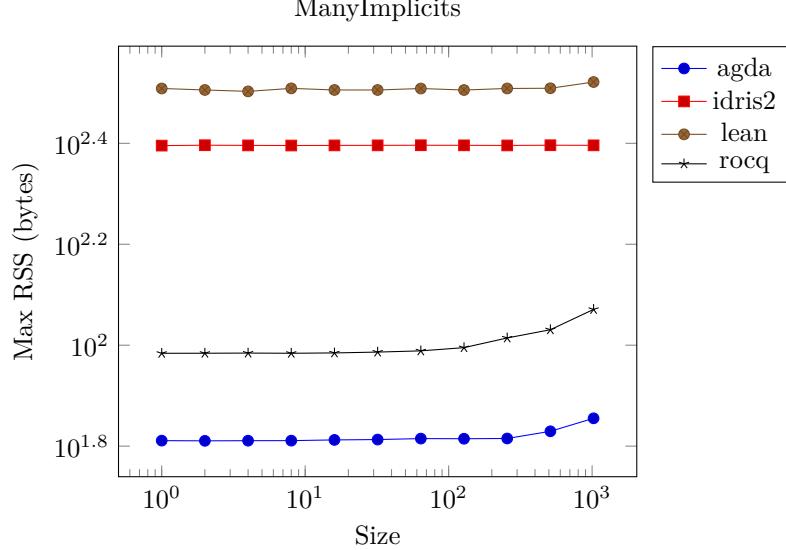
7.21 ManyImplicits



375



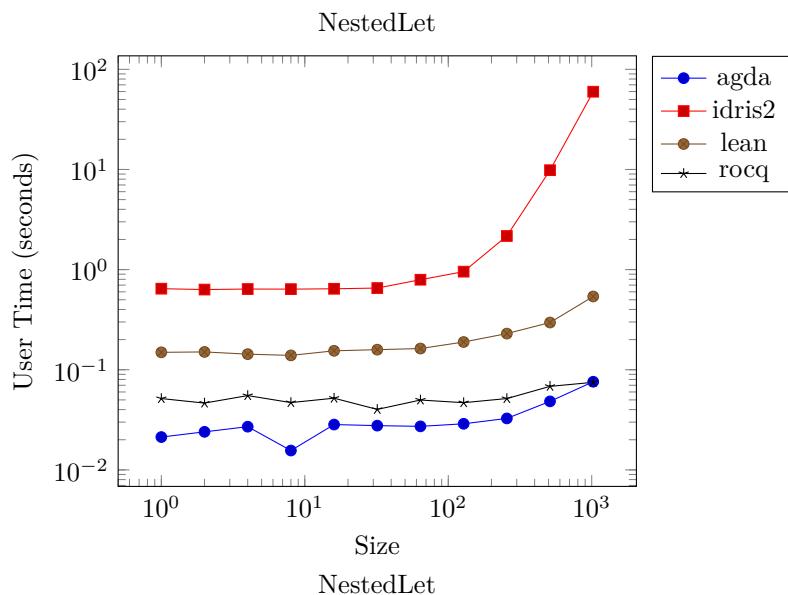
377



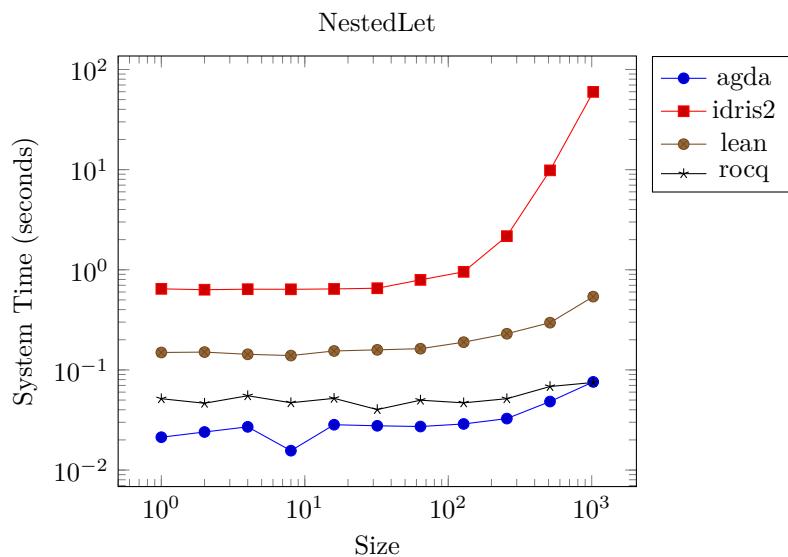
378

7.22 NestedLet

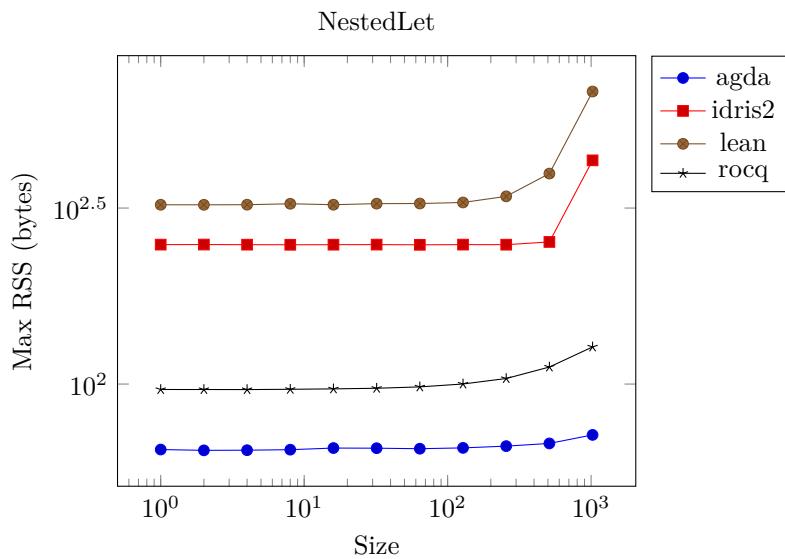
379



380

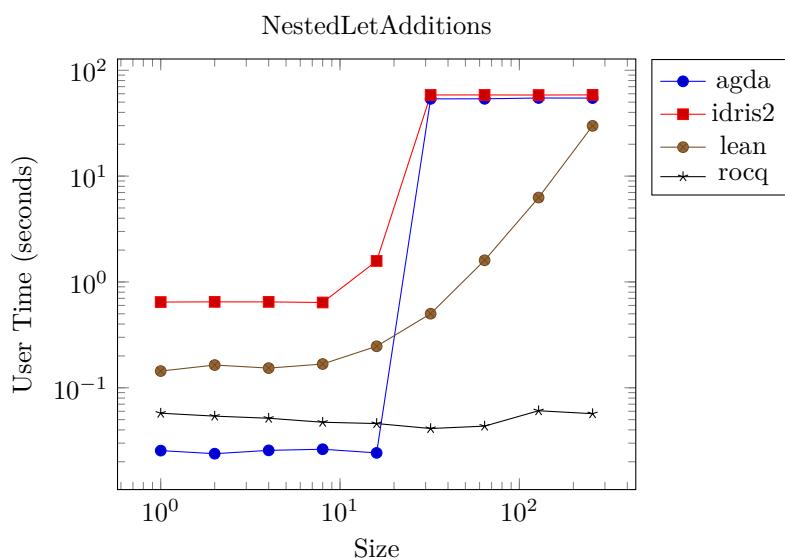


381

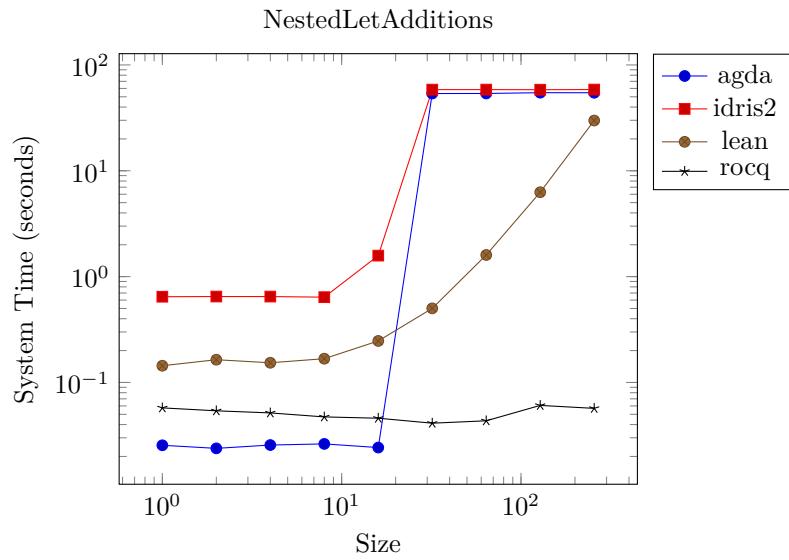


382

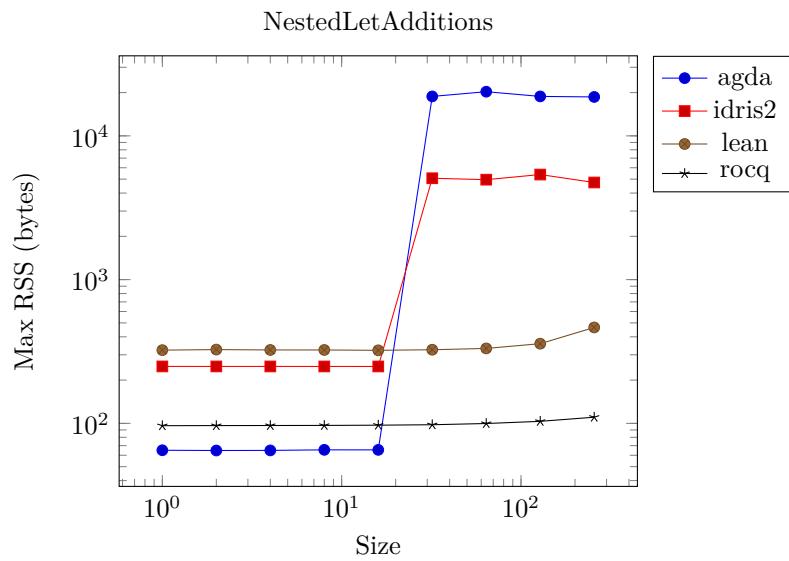
383 7.23 NestedLetAdditions



384

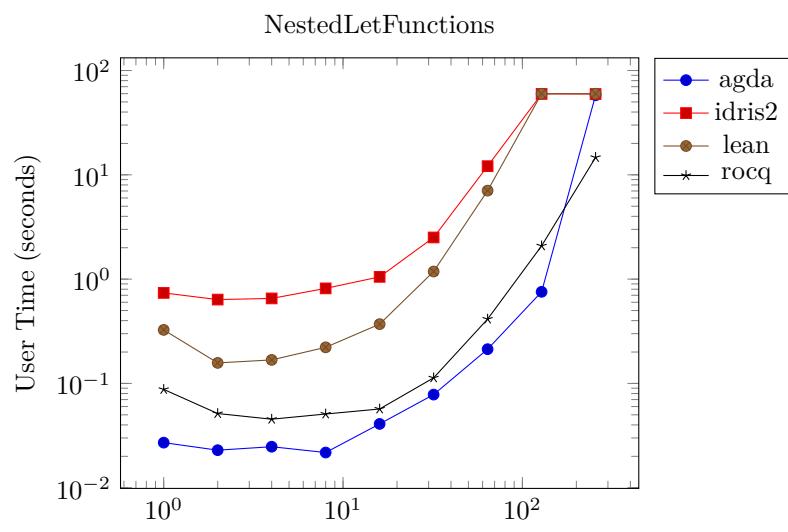


385

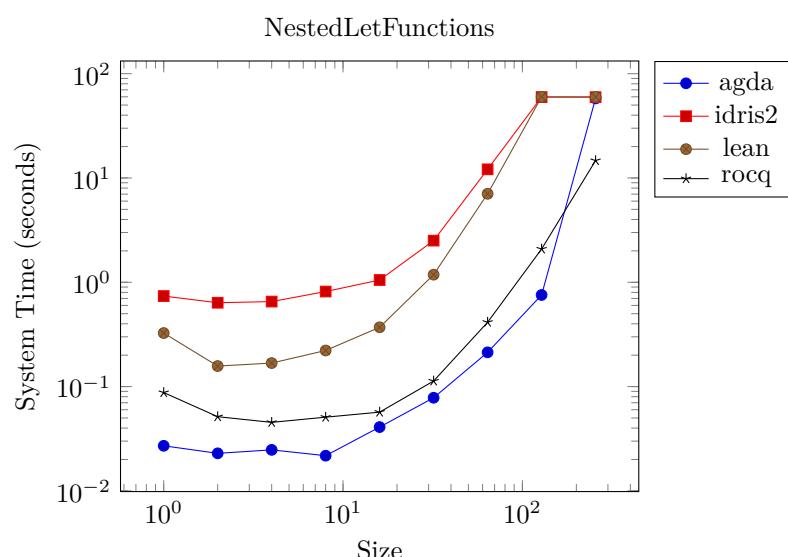


386

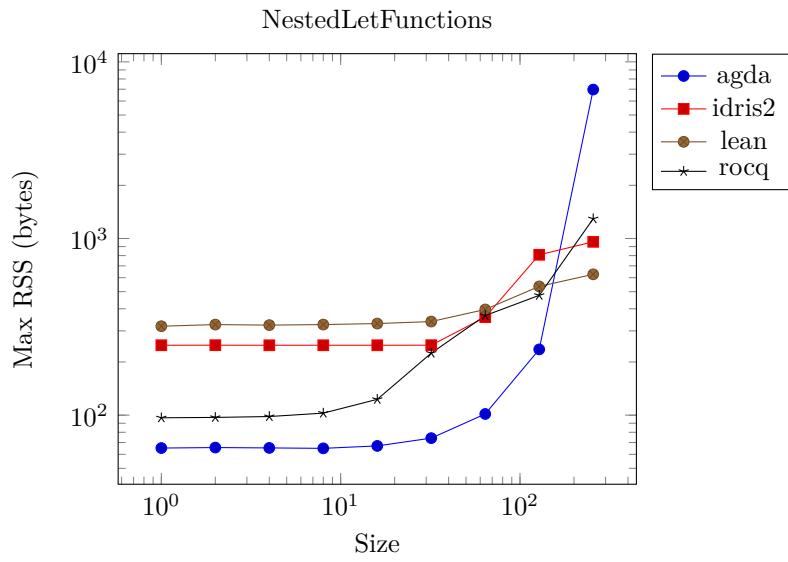
387 7.24 NestedLetFunctions



388

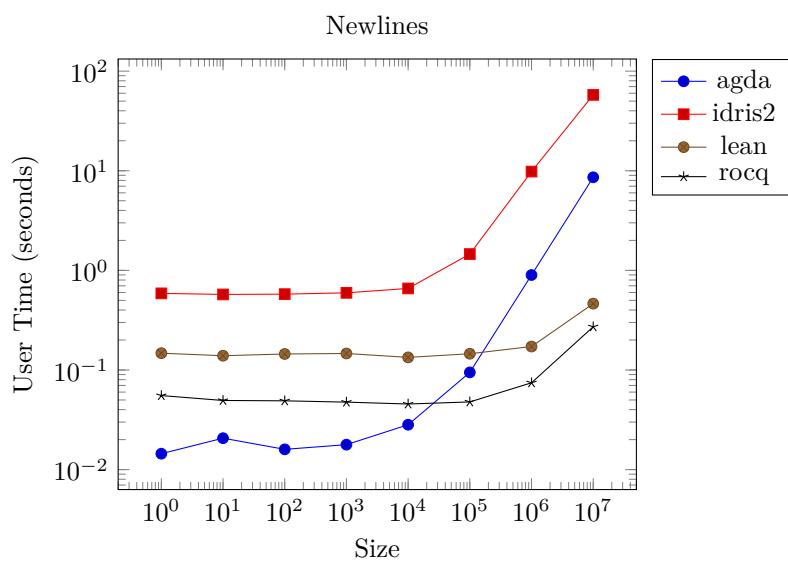


389

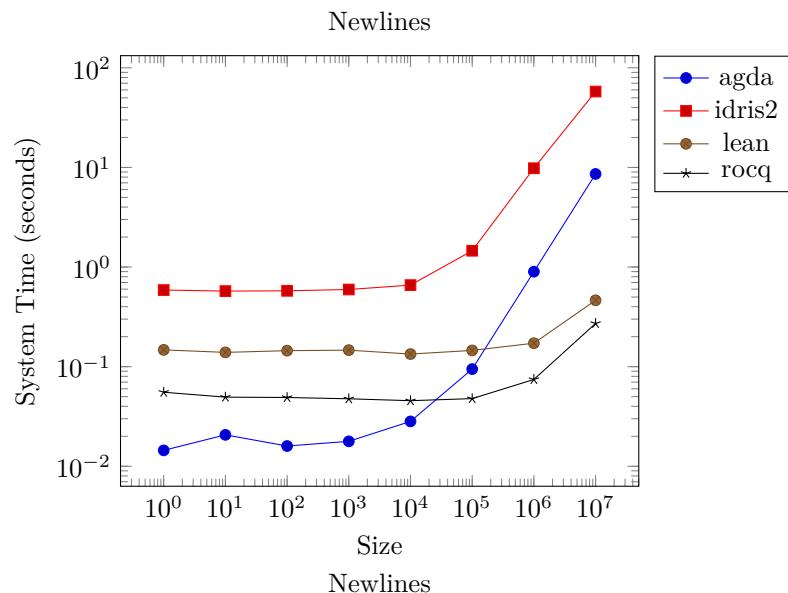


390

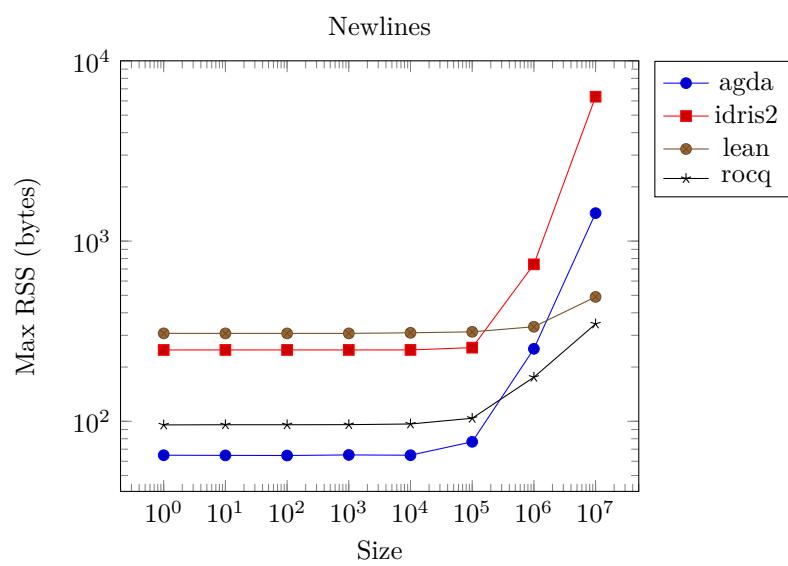
7.25 Newlines



392

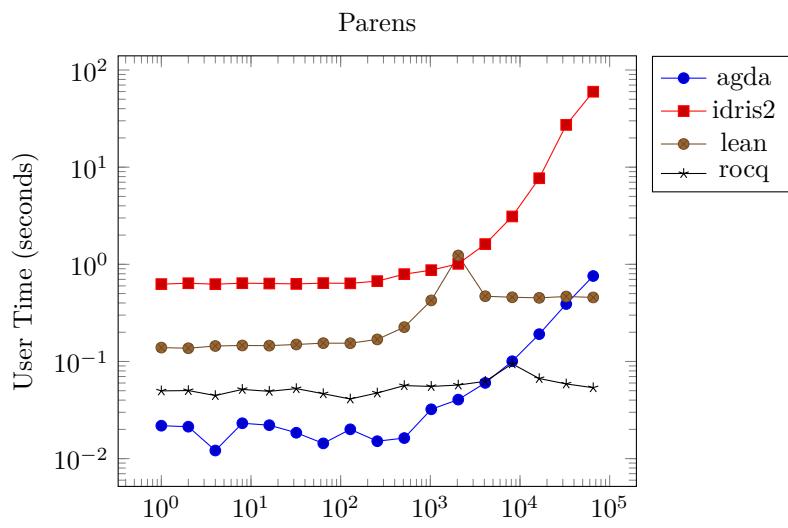


393

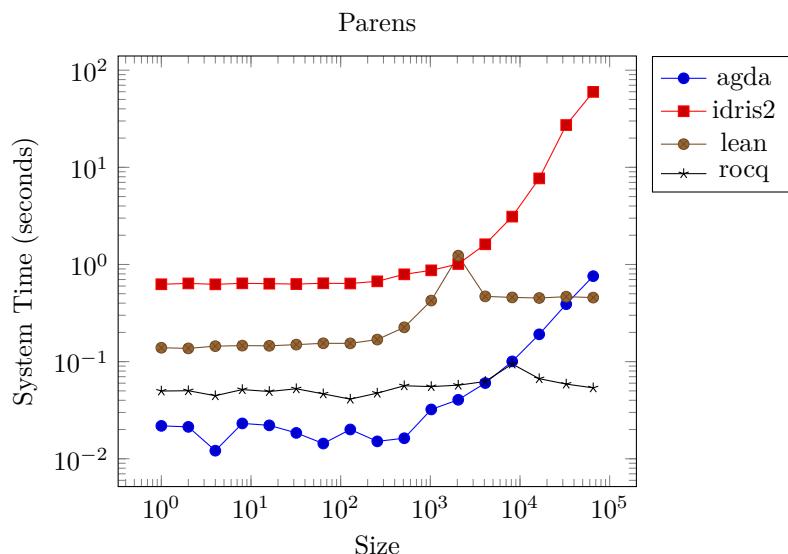


394

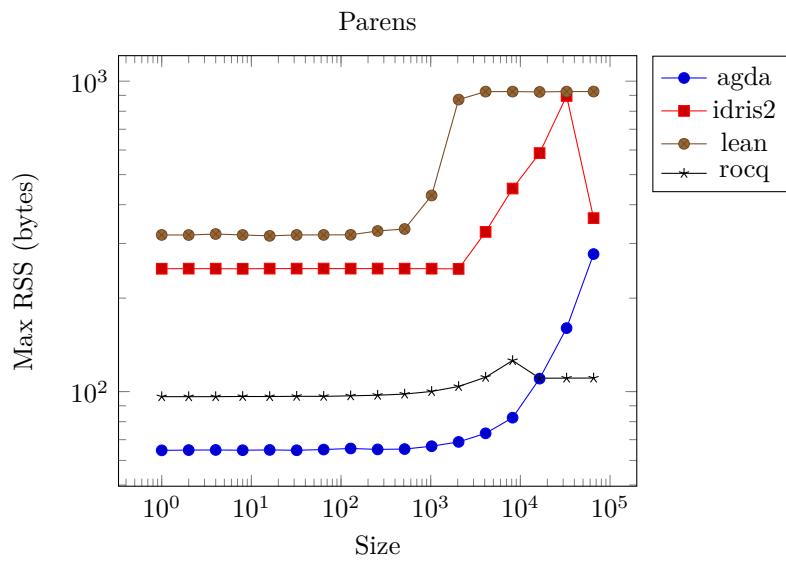
395 7.26 Paren



396

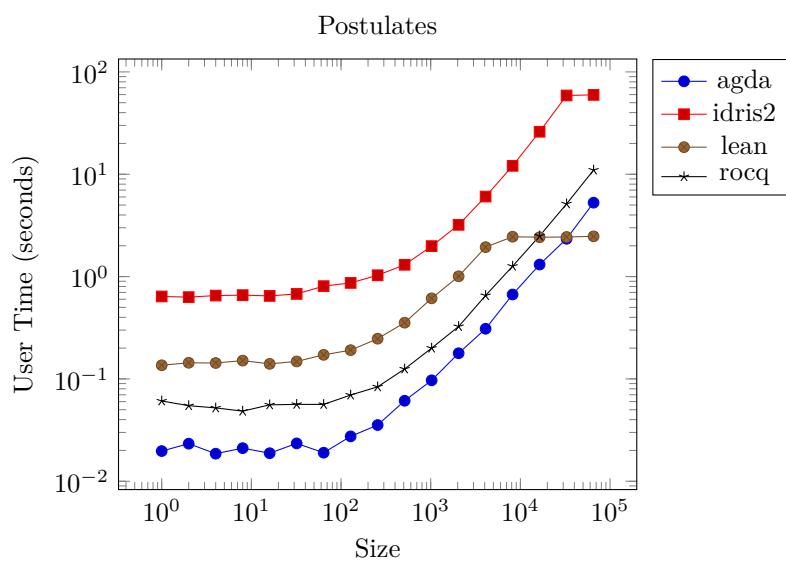


397

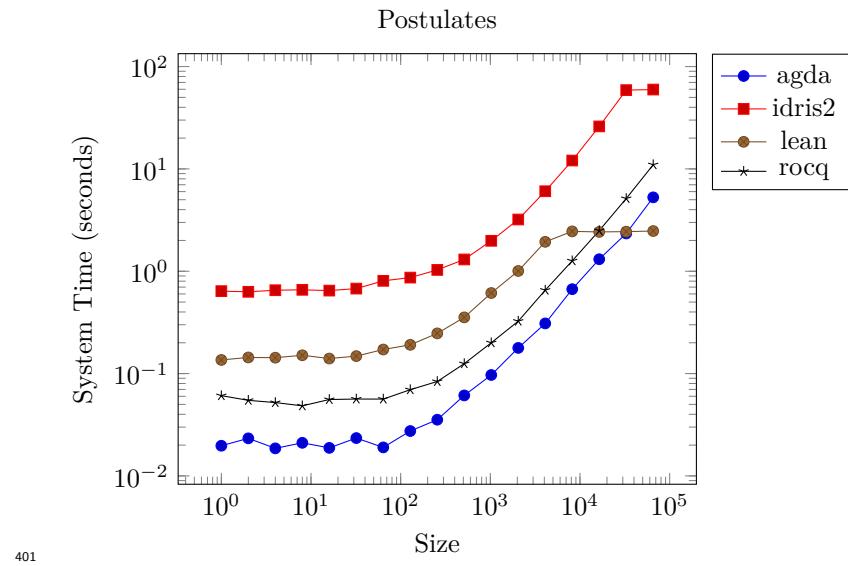


398

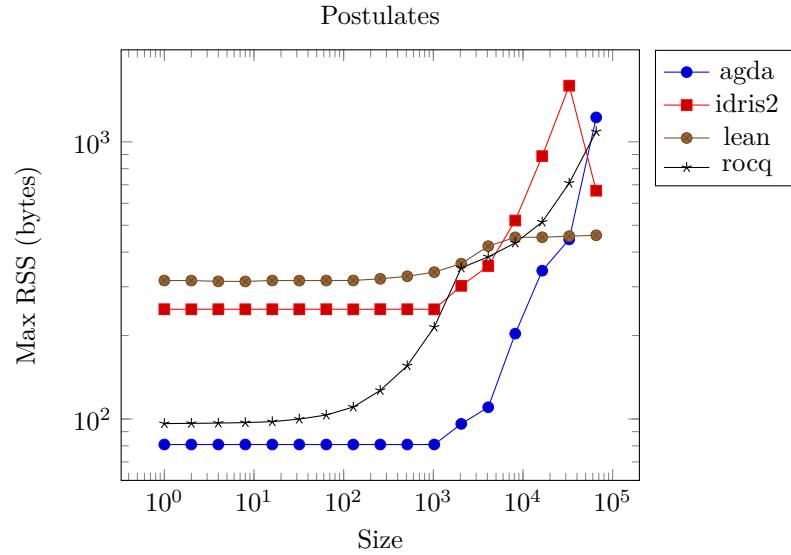
399 7.27 Postulates



400

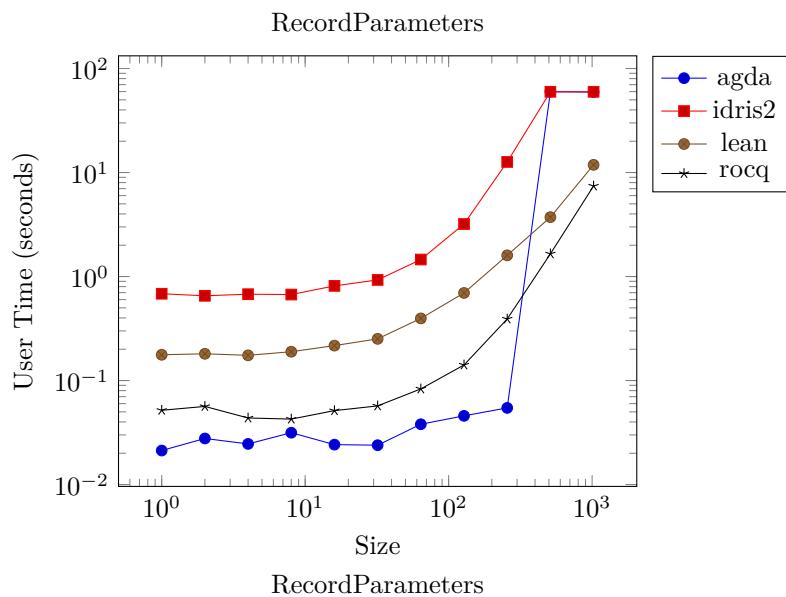


401

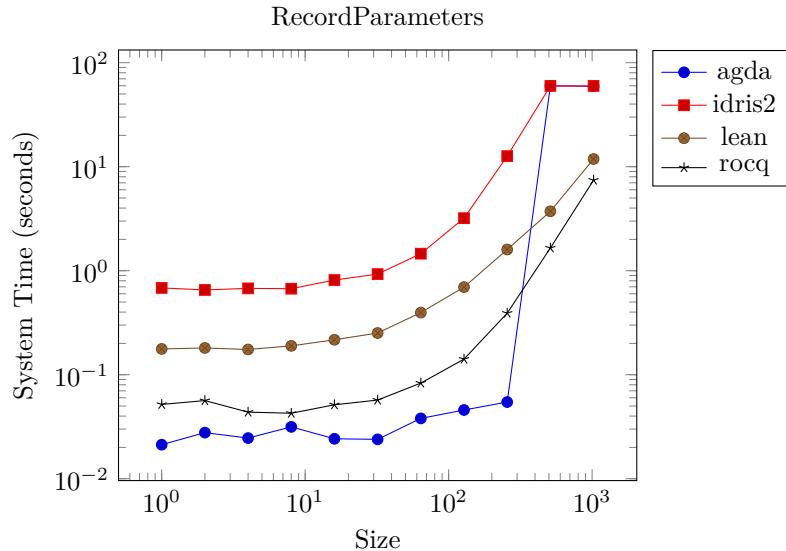


402

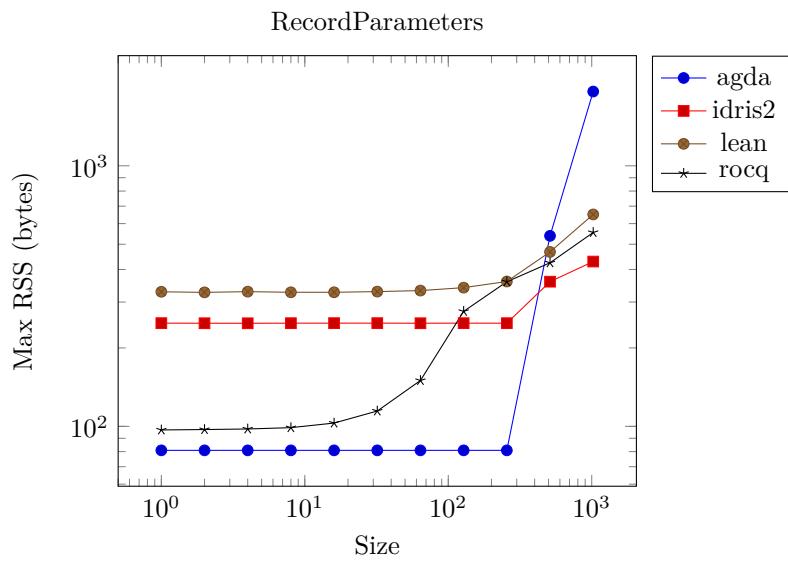
403 7.28 RecordParameters



404

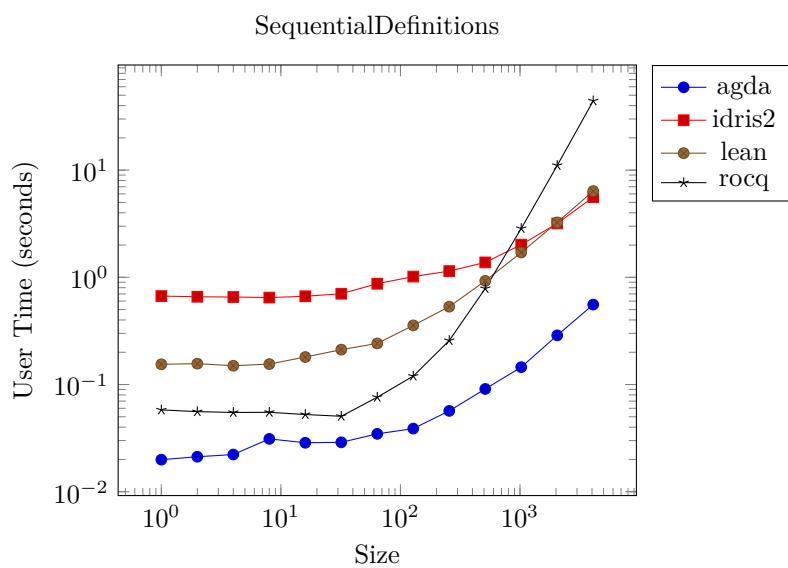


405

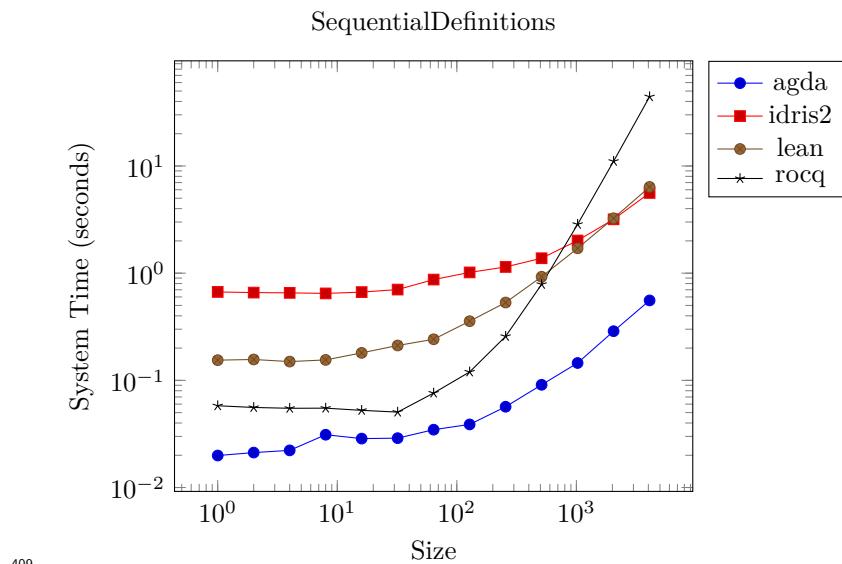


406

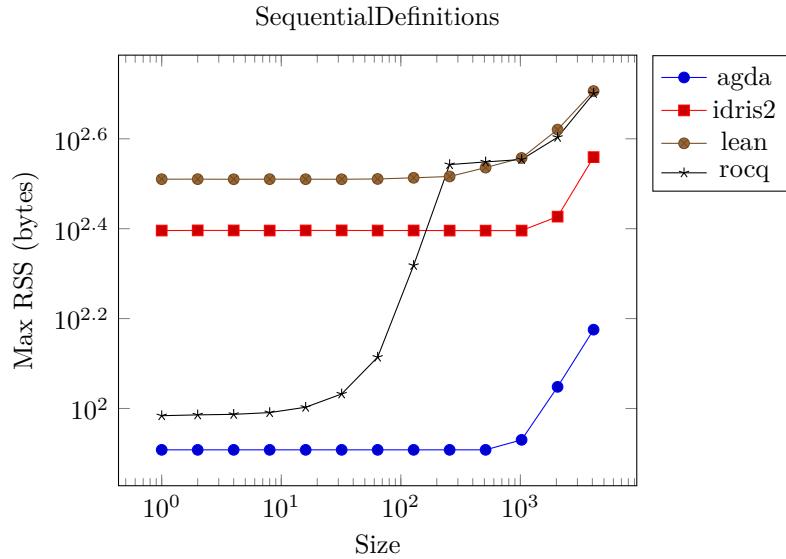
407 7.29 SequentialDefinitions



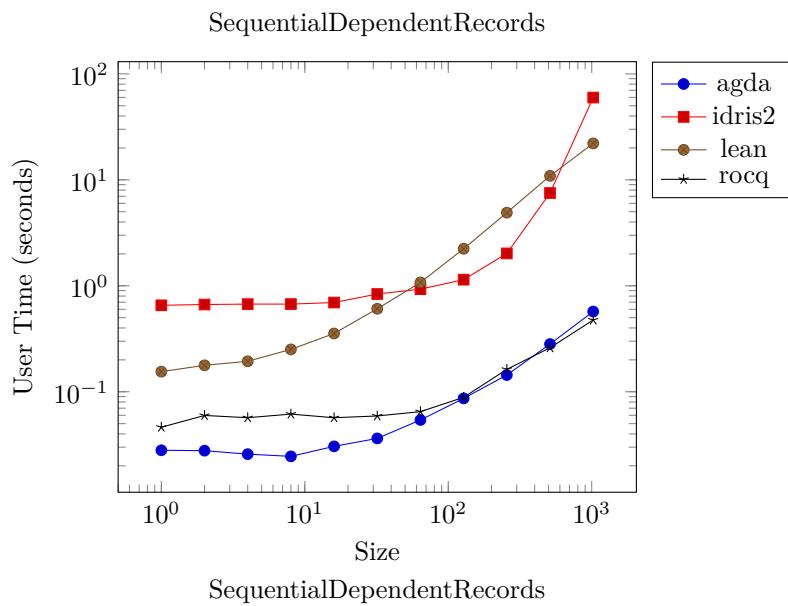
408



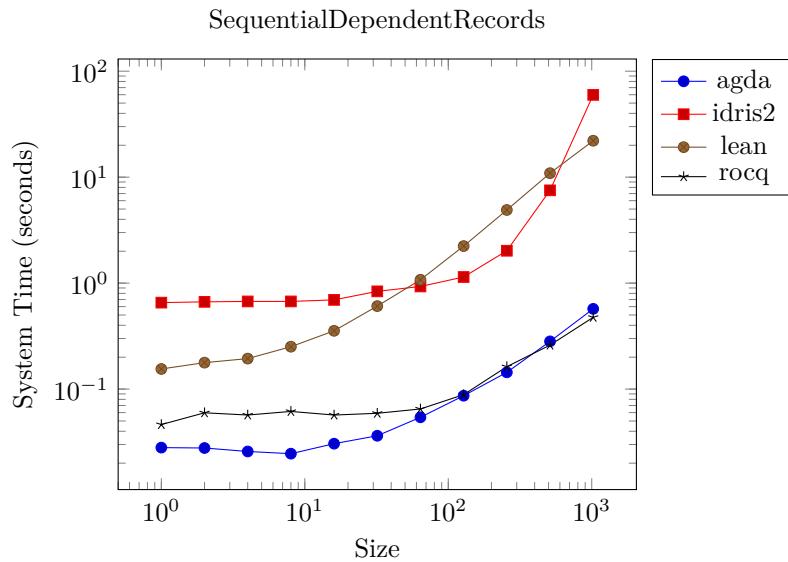
409



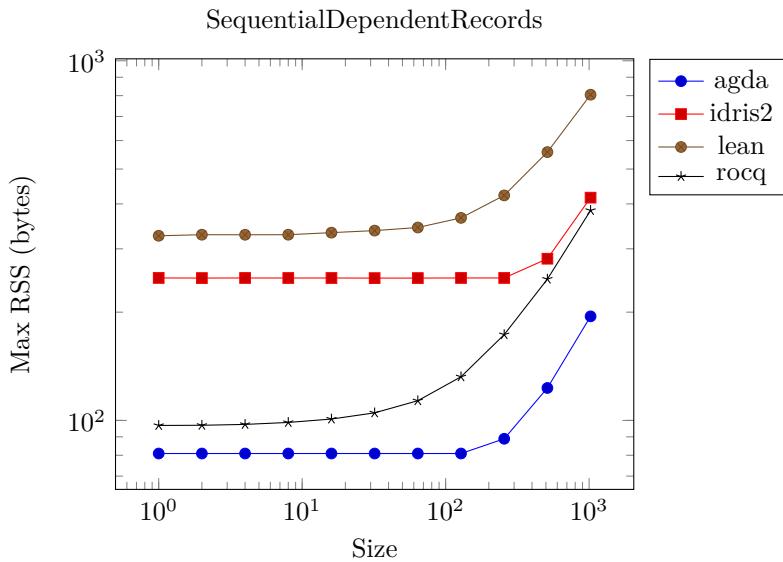
410

411 7.30 SequentialDependentRecords

412

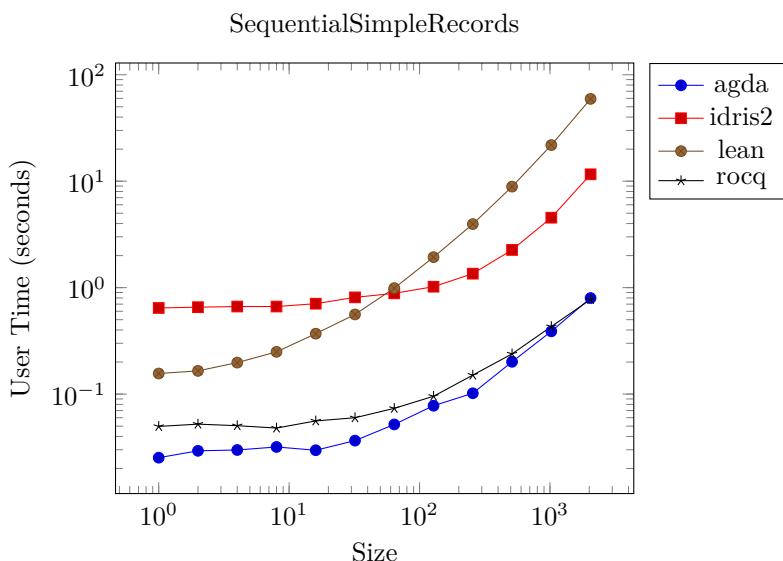


413

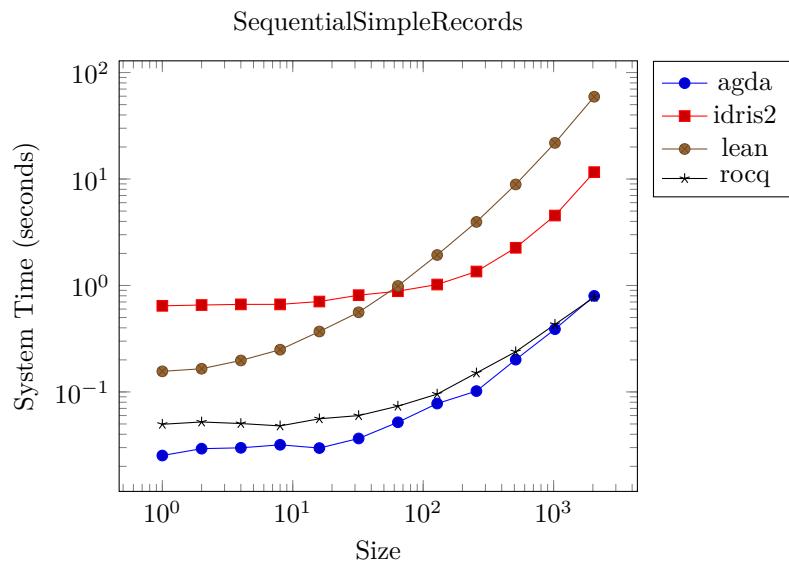


414

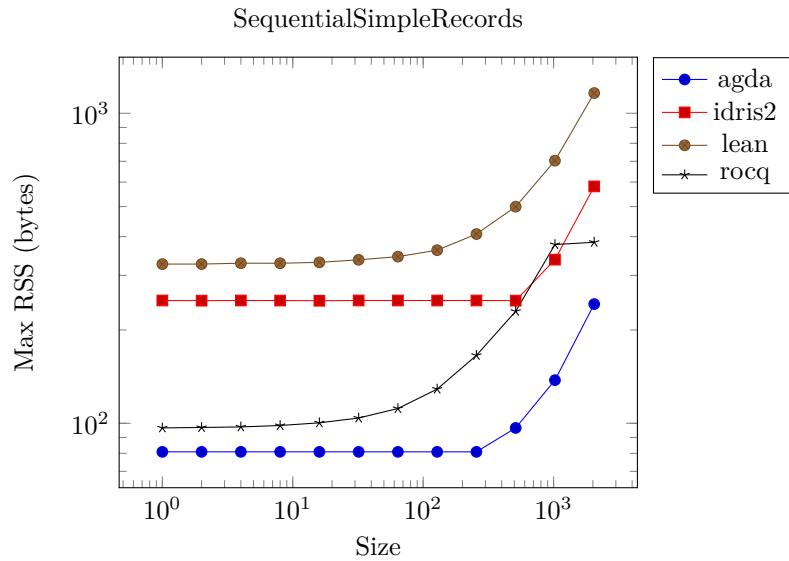
415 7.31 SequentialSimpleRecords



416

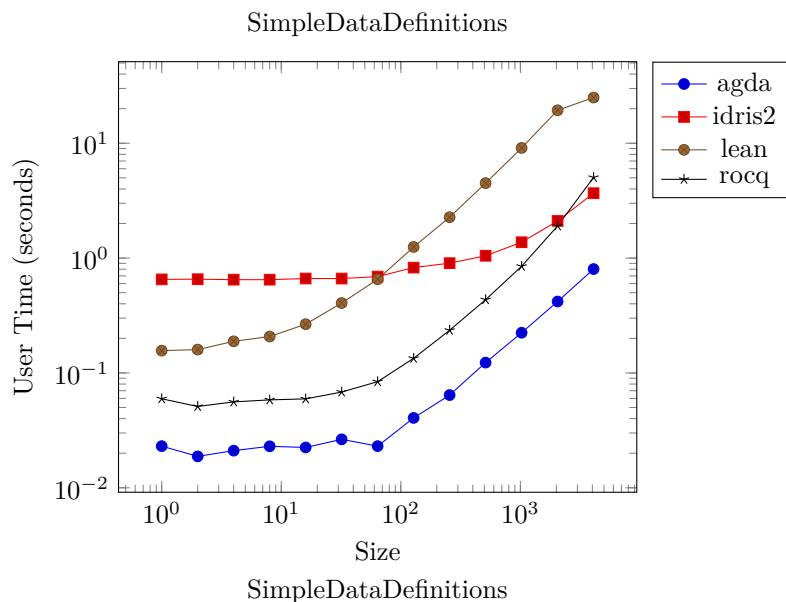


417

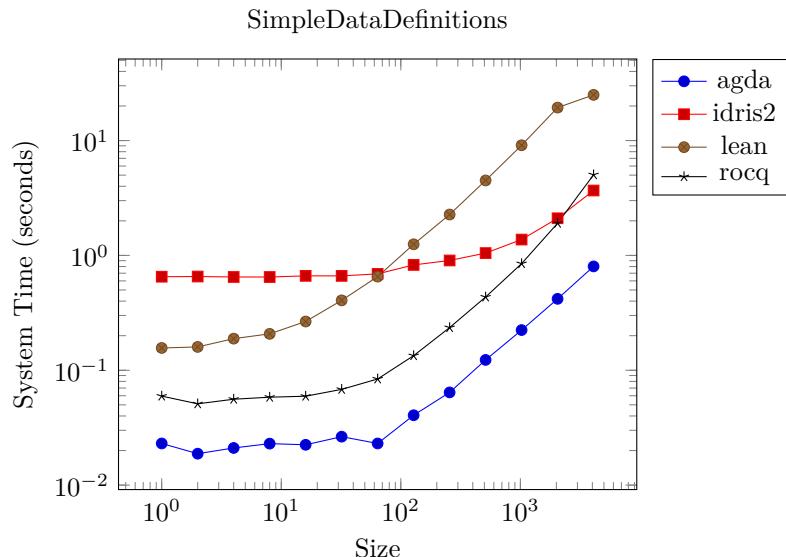


418

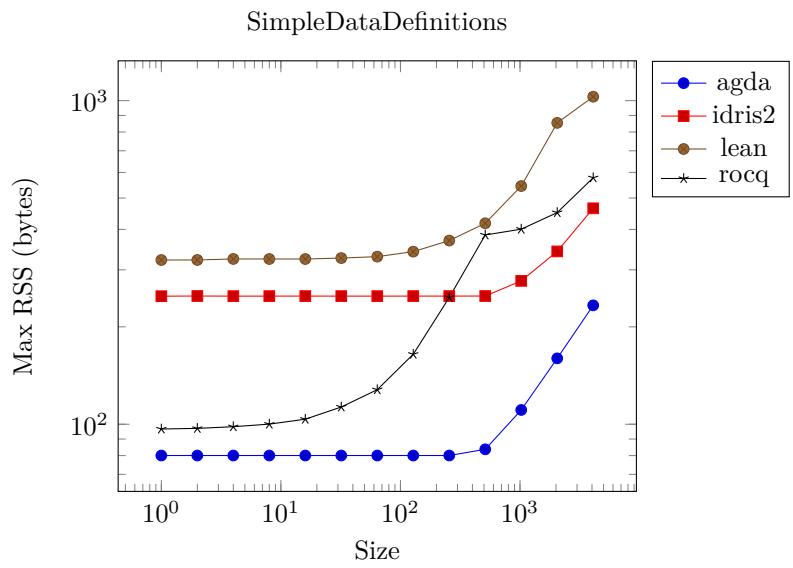
419 7.32 SimpleDataDefinitions



420



421



422