# How developers choose names in statically-typed functional programming languages

JAN CHRISTIANSEN, Flensburg University of Applied Sciences, Germany

To the best of our knowledge, this paper presents the first empirical study on the use of identifiers in functional programming languages. We conduct an exploratory case study by collecting identifiers from 400 non-archived, non-disabled, and non-outdated GitHub repositories with most stars that use Elm, Haskell, OCaml, or PureScript. In total, we extracted 3 830 575 identifiers from these projects to investigate four research questions related to the frequency and characteristic of identifiers in statically-typed functional programming languages.

CCS Concepts: • **Software and its engineering** → **Functional languages**; • **General and reference** → **Empirical studies**.

Additional Key Words and Phrases: Functional Programming Languages, Identifiers, Naming

**ACM Reference Format:**
Jan Christiansen. 2018. How developers choose names in statically-typed functional programming languages. *J. ACM* 37, 4, Article 111 (August 2018), 51 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

Consider the following function definition, which is the first Haskell code that you see, if you visit the Haskell website at haskell.org.

```
primes = filterPrime [2 ..]
  where
    filterPrime (p : xs) =
      p : filterPrime [x | x ← xs, x ‘mod‘ p /= 0]
```

While it is considered good practice to use identifiers that are descriptive, *p*, *xs* and *x* are used as identifiers in this very prominent Haskell definition. When teaching Haskell, subjectively, students coming from programming languages like Java often complain about very short identifiers used in Haskell code. In contrast the following function definition is the first PureScript code that you see, if you visit the PureScript website at purescript.org.

```
greet :: String → String
greet name = "Hello, " <> name <> "!"
```

This PureScript definition does not use a single-letter variable but the dictionary word *name*. Furthermore, the official style guide of the functional programming language Elm at elm-lang.org states the following.

Author's Contact Information: Jan Christiansen, jan.christiansen@hs-flensburg.de, Flensburg University of Applied Sciences, Flensburg, Germany.

ACM 1557-735X/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

> One character abbreviations are rarely acceptable, especially not as arguments for top-level function declarations where you have no real context about what they are.

In support of the Haskell example the identifiers *p*, *xs* and *x* are not arguments to top-level functions but arguments to a local function and a variable defined in a list comprehension. However, the Elm citation talks about "One character abbreviations". While the identifier *p* is probably the abbreviation of the word *prime*, it is very unlikely that the name *x* in the Haskell definition is even an abbreviation.

These examples raise the following more general research questions.

RQ1 What length do identifiers in statically-typed functional programming languages have?
RQ2 Does the length of identifiers differ for different statically-typed functional programming languages?
RQ3 Does the length of identifiers differ for different classes of identifiers?
RQ4 Which identifiers are used most often in statically-typed functional programming languages?

## 2 Related Work

Various studies have investigated the use of identifiers in computer programs, focusing on readability, comprehension or recall. Lawrie et al. [2006], Binkley et al. [2009], Scanniello and Risi [2013], Schankin et al. [2018], Hofmeister et al. [2019], and Etgar et al. [2022] conducted experiments with computer science students and/or professional developers. These studies vary the style of identifiers and ask participants to remember or to spot a specific identifier or to spot a defect within a program. For example, they compare underscore case with camel case or single letter with abbreviations and full-word identifiers and measure time and/or accuracy for the task. As another example, Feitelson et al. [2022] conducted an online survey asking participants to name variables, constants and data structures and, for example, show that the probability that two participants choose the same name is very low. Alpern et al. [2024] conducted a conceptual replication [Shull et al. 2008] of the study by Feitelson et al. and corroborated the results.

While these kinds of studies are somehow related to the paper at hand, we focus on the closest kind of related work: extracting information about identifiers from existing code. Caprile and Tonella [1999] conducted one of the first studies in this area, analyzing 3 304 function identifiers from ten C programs. Although not explicitly stated, they likely collected only binding identifier occurrences. They heuristically split identifiers into soft words[1] and manually classify soft words into one of seven categories like word contraction, number, or isolated character. Hard words are character sequences delimited by word markers such as underscores or camel-casing, while each hard word consists of one of more soft words. For example, in `get_filenum` the hard words are `get` and `filenum` and the soft words are `get`, `file`, and `num`.

Deissenboeck and Pizka [2006] extracted 5 907 576 identifiers from three open-source projects (Eclipse, Sun's Java JDK, and Tomcat) to analyze identifier usage. They found that identifiers constitute 33 % of all tokens in Eclipse and 30 % in Sun's Java JDK and in Tomcat. As their analysis operates on the token level they collect binding as well as bound identifier occurrences. They analyzed aspects like the amount of atomic and compound identifiers or the number of distinct words when compound identifiers are split into hard words. While Deissenboeck and Pizka mined existing code, their main contribution is a formal model of identifier quality to define when an identifier is "meaningful".

[1] Caprile and Tonella [1999] themselves do not use the term *soft word*.

Liblit et al. [2006] extracted 7 243 585[2] binding identifier occurrences from three programs (Sun's Java JDK, Gnumeric, a spreadsheet application written in C, and the source code of Windows 2003 Server, which is written in C and C++). They compare the average length of various identifier types, such as local variables, parameters and methods with file or global scope. Additionally, they discuss recurring grammatical structures in identifiers like noun phrases and verb phrases by using examples from the projects.

Lawrie et al. [2007] extracted 55 638 621 identifiers from 186 program versions across 76 projects, written in C, C++, Java, and Fortran. They collected "all program identifiers", that is, binding as well as bound identifier occurrences. They heuristically split identifiers into hard and soft words and provide basic statistics about the projects like unique identifiers and total number of hard and soft words. They classified words into categories like abbreviation and single letter and conducted statistical analyzes to draw conclusions. For example, they conclude that "modern programs include more dictionary soft words". They rate the quality of an identifier by calculating the ratio of soft to hard words and conclude that "modern programs contain higher quality identifiers".

Abebe et al. [2009] analyzed two software systems (ALICE and WinMerge) written in C++. They split identifiers into hard words (using non-literals and camel-casing for splitting) and apply stemming [Porter 1980] to compute vocabularies. A vocabulary contains the set of unique words that appear in the corresponding entity. They observe that the system vocabulary, which is the union of all vocabularies, and system size often exhibit a parallel evolution trend. However, most new identifiers introduce none or at most one new term in the vocabulary.

Haiduc and Marcus [2008] analyzed six graph theory libraries – two in C++ and four in Java – extracting domain-specific terms from comments and identifiers. They found that on average, 23 % of the domain terms used in the source code are present in comments only, whereas only 11 % in identifiers alone. That is, comments and identifiers contain a significant fraction of domain terms and comments may be more significant than identifiers regarding domain information.

The most closely related work and the work that sparked our interest in identifiers is a study by Beniamini et al. [2017]. They extracted binding identifier occurrences of local variables from 200 GitHub repositories with most stars for C, Java, JavaScript, Perl, and PHP. Their analysis includes histograms of identifier length and single-letter identifiers. Besides data mining Beniamini et al. conducted an online survey with students to comprehend and modify functions where identifiers used full words and a variant where some identifiers are replaced by single-letter variables. Another online survey asked participants what type they associate with different letters of the alphabet.

Swidan et al. [2017] conducted an exact replication [Shull et al. 2008] of Beniamini et al.'s study on a data set of 250 000 Scratch programs. Scratch is a block-based programming language for teaching basic programming concepts. Unlike the original study they analyzed both variable and procedure names, finding that procedure names tend to be longer. Instead of using a questionnaire to link single-letter variable names to types, they inferred type information from the values that are assigned to the variables. Additionally, they collected statistics about Scratch-specific features of procedure names, such as the number of spaces within a procedure name. They conducted statistical tests to compare distributions for Scratch with distributions from the original study.

Aman et al. [2021] extracted 637 077 binding identifier occurrences of local variables, including method parameters, from 1 000 Java repositories on GitHub with most stars. Given that they processed 472 665 Java source files, this results in approximately 1.3 variables per file, which seems unexpectedly low. In contrast, we extracted 702 879 identifiers from binding variable names, including function parameters, from 14 301 Haskell source files from 100 repositories. While variables in functional programming languages are quite different to variables in object-oriented

[2] By far the largest part (7 137 095) is extracted from the code of Windows 2003 Server.

languages, the discrepancy between these numbers is quite high. Aman et al. computed the variable scope in number of source lines and heuristically classified identifiers into categories like single letter, dictionary word or type-derived name. They found that about half of the variables with broad scope are compounded names and that the shares of single letter and dictionary word decrease while the share of compounded names increases for broad scopes compared to all variables.

Feitelson [2023] reanalyzed the data set collected by Aman et al., refining the scope classification by using six log-based scope sizes instead of two. This allowed for more precise observations, for example, he observes that the fact that variables with broad scope tend to be longer is part of a progression, and that the biggest difference is that variables with minimal scope are shorter. He also replicated the observation that long words tend to be abbreviated, starting at six characters. Additionally, he highlights a major limitation of the original study – the exclusion of class fields – arguing that developers often think of method vs. class scope when thinking about different scopes.

Gresta et al. [2023] extracted 2 603 381 identifiers (variables, parameters, and attributes) from 40 Java and 40 C++ projects on GitHub. They classified these identifiers into eight distinct naming patterns, such as names with trailing numbers or names that are identical to their type. Their analysis revealed that `value`, `result`, and `name` are the most recurring names and that single-letter names are widely used. Furthermore, they observe that single-letter names appear more often in conditionals or loops while names that identical to their type tend to appear in large-scope contexts like attributes. To identify naming practices in industry they conducted an online survey with 52 software developers.

To the best of our knowledge, no prior research has examined identifiers in functional programming languages. The most closely related work is the study by Beniamini et al. [2017], which analyzed variable length in JavaScript projects – a language significantly influenced by functional programming languages.

## 3 Method

Following Easterbrook et al. [2008] on selecting appropriate research methods, we conduct an exploratory case study to enhance understanding of identifiers in statically-typed functional programming languages. While we generally support reductionism, we consider a pragmatist approach more suitable for initial investigations into identifiers in functional programming languages.

### 3.1 Data Collection

To answer our research questions, we collected code from GitHub, selecting the 100 public repositories with most stars for Elm, Haskell, OCaml, and PureScript. We exclude archived, disabled, and outdated repositories, where the information whether a repository is marked as archived or disabled by its owner is provided by the GitHub API. We define a repository to be outdated if the last commit on the default branch[3] is before January 1st 2020, that is, the code has not been updated for at least five years.

Filtering outdated repositories involves a trade-off. Excluding them removes highly starred repositories that many developers find valuable but increases the likelihood that the collected code reflects current programming practices. Since star counts accumulate over time, sorting by stars often favors older repositories. Notably, some archived repositories reference replacement repositories where development continues. However, the replacement repository is often not included because it has much fewer stars than the original repository due to its younger age.

To collect repositories, we queried the GitHub API with the option `language` with the appropriate value and `sort:stars` and fetch the first 100 results that are neither archived, nor disabled, nor

[3] Our tool clones the default branch of a repository.

outdated. We prioritized highly starred repositories over a random sample to focus on identifiers in active, mature projects. Figures 1 to 4 summarize the mined repositories as of January 1st, 2025. These tables show the name of the repository, the number of stars, and the number of variable names and definition names that were extracted. The category of variables contains all identifiers introduced in pattern matching in function parameters, case expressions, parameters of lambda expressions, and do notation[4]. The category of definitions contains all identifiers of function definitions and constant definitions, as well as record selectors. We exclude operators from the collection. The number of operators is very small, for all languages fewer than 1 % of all definitions.

|  | Repo Name | Stars | Variables | Definitions |  | Repo Name | Stars | Variables | Definitions |
|---|---|---|---|---|---|---|---|---|---|
| 1 | elm/core | 2 805 | 815 | 413 | 51 | stoeffel/elm-verify-examples | 166 | 238 | 200 |
| 2 | elm/elm-lang.org | 1 986 | 791 | 669 | 52 | harehare/textusm | 165 | 2 963 | 3 437 |
| 3 | eikek/docspell | 1 691 | 6 767 | 8 722 | 53 | joakin/elm-canvas | 163 | 589 | 611 |
| 4 | ohanhi/elm-native-ui | 1 540 | 259 | 418 | 54 | lukewestby/elm-http-builder | 162 | 65 | 69 |
| 5 | azimuttapp/azimutt | 1 505 | 10 473 | 6 802 | 55 | bryanjenningz/25-elm-examples | 160 | 289 | 238 |
| 6 | mdgriffith/elm-ui | 1 359 | 1 891 | 1 800 | 56 | n1k0/tooty | 157 | 801 | 500 |
| 7 | rtfeldman/elm-css | 1 241 | 1 167 | 2 457 | 57 | mxgrn/pairs.one | 156 | 246 | 296 |
| 8 | wende/elchemy | 1 146 | 941 | 389 | 58 | funk-team/funkLang | 156 | 7 811 | 8 638 |
| 9 | eikek/sharry | 934 | 1 357 | 1 821 | 59 | arturopala/elm-monocle | 153 | 228 | 133 |
| 10 | gdotdesign/elm-ui | 923 | 1 100 | 1 554 | 60 | rl-king/elm-hnpwa | 152 | 92 | 81 |
| 11 | icidasset/diffuse | 816 | 2 244 | 2 375 | 61 | finos/morphir | 152 | 12 526 | 5 662 |
| 12 | cultureamp/react-elm-components | 779 | 0 | 0 | 62 | mweiss/elm-rte-toolkit | 149 | 2 162 | 1 368 |
| 13 | dillonkearns/elm-graphql | 778 | 3 535 | 7 873 | 63 | w0rm/elm-physics | 147 | 1 179 | 1 470 |
| 14 | terezka/elm-charts | 745 | 2 532 | 3 439 | 64 | cmditch/elm-ethereum | 146 | 710 | 775 |
| 15 | ellie-app/ellie | 737 | 1 251 | 998 | 65 | rtfeldman/elm-validate | 144 | 62 | 43 |
| 16 | dillonkearns/elm-pages | 662 | 6 167 | 4 866 | 66 | w0rm/elm-mogee | 143 | 342 | 365 |
| 17 | ravichugh/sketch-n-sketch | 553 | 15 648 | 8 503 | 67 | Bogdanp/elm-ast | 143 | 293 | 264 |
| 18 | erkal/kite | 549 | 822 | 1 075 | 68 | iancanderson/PurpleTrainElm | 138 | 169 | 195 |
| 19 | gampleman/elm-visualization | 520 | 2 260 | 2 677 | 69 | mdgriffith/elm-codegen | 137 | 4 092 | 2 875 |
| 20 | roovo/obsidian-card-board | 519 | 2 276 | 1 610 | 70 | NoRedInk/elm-json-decode-pipeline | 136 | 21 | 10 |
| 21 | jah2488/elm-companies | 469 | 0 | 1 | 71 | jschomay/elm-narrative-engine | 135 | 341 | 137 |
| 22 | terezka/line-charts | 456 | 1 274 | 1 927 | 72 | elm-community/list-extra | 135 | 420 | 157 |
| 23 | Lattyware/massivedecks | 441 | 2 458 | 2 386 | 73 | elm-community/js-integration-examples | 135 | 26 | 26 |
| 24 | NixOS/nixos-search | 425 | 466 | 362 | 74 | mdgriffith/elm-animator | 135 | 1 907 | 1 375 |
| 25 | stil4m/elm-analyse | 415 | 1 573 | 1 137 | 75 | RoganMurley/GALGAGAME | 135 | 519 | 556 |
| 26 | w0rm/elm-flatris | 408 | 110 | 145 | 76 | exercism/elm | 134 | 354 | 350 |
| 27 | gingko/client | 366 | 2 887 | 2 129 | 77 | NoRedInk/noredink-ui | 133 | 4 567 | 5 614 |
| 28 | elm-land/elm-land | 359 | 1 147 | 1 451 | 78 | lindsaykwardell/vite-elm-template | 132 | 4 | 5 |
| 29 | aforemny/elm-mdc | 354 | 2 114 | 2 864 | 79 | dmotz/emdash | 130 | 589 | 335 |
| 30 | cotoami/cotoami | 348 | 1 884 | 1 186 | 80 | mdgriffith/elm-optimize-level-2 | 130 | 949 | 645 |
| 31 | evancz/guide.elm-lang.org | 323 | 183 | 87 | 81 | dmy/elm-doc-preview | 128 | 582 | 360 |
| 32 | jamesmacaulay/elm-graphql | 312 | 288 | 362 | 82 | lucamug/elm-ecommerce | 127 | 378 | 356 |
| 33 | Malax/elmboy | 311 | 779 | 1 121 | 83 | passiomatic/sunny-land | 126 | 238 | 305 |
| 34 | rogeriochaves/spades | 301 | 117 | 121 | 84 | JetBrains/origami | 126 | 1 310 | 1 048 |
| 35 | ohanhi/elm-shared-state | 301 | 92 | 129 | 85 | sporto/elm-patterns | 123 | 0 | 0 |
| 36 | Zinggi/NoKey | 276 | 2 548 | 1 609 | 86 | arowM/elm-form-decoder | 123 | 231 | 210 |
| 37 | Arcitectus/Sanderling | 274 | 841 | 1 334 | 87 | elm-explorations/webgl | 117 | 130 | 322 |
| 38 | jfmengels/elm-review | 263 | 7 707 | 6 096 | 88 | cedricss/elm-batteries | 117 | 30 | 35 |
| 39 | elm-explorations/test | 236 | 1 910 | 1 210 | 89 | emanchado/narrows | 115 | 954 | 1 069 |
| 40 | pine-vm/pine | 227 | 14 961 | 11 292 | 90 | niksilver/elm-explained | 115 | 24 | 91 |
| 41 | passiomatic/elm-designer | 222 | 1 503 | 1 237 | 91 | avh4/elm-program-test | 92 | 1 056 | 527 |
| 42 | LiaScript/LiaScript | 220 | 4 294 | 2 975 | 92 | JetBrains/open-radiant | 91 | 2 106 | 1 810 |
| 43 | ianmackenzie/elm-3d-scene | 205 | 2 012 | 2 685 | 93 | albertdahlin/elm-posix | 91 | 89 | 66 |
| 44 | alexkorban/elmstatic | 196 | 85 | 116 | 94 | elm/json | 91 | 44 | 53 |
| 45 | ianmackenzie/elm-geometry | 183 | 6 668 | 8 092 | 95 | mapwatch/mapwatch | 90 | 1 192 | 1 344 |
| 46 | gicentre/elm-vega | 181 | 1 620 | 2 571 | 96 | tmcw/flair | 87 | 161 | 252 |
| 47 | mdgriffith/elm-markup | 179 | 2 083 | 1 089 | 97 | Zokka-Dev/zokka-compiler | 87 | 3 795 | 4 768 |
| 48 | Viir/bots | 178 | 5 293 | 8 834 | 98 | choonkeat/elm-webapp | 58 | 123 | 120 |
| 49 | mohsenasm/swarm-dashboard | 172 | 159 | 190 | 99 | MartinSStewart/ascii-collab | 33 | 4 382 | 7 711 |
| 50 | zwilias/elm-demystify-decoders | 170 | 2 | 20 | 100 | Gizra/elm-keyboard-event | 22 | 31 | 45 |

Fig. 1. Non-archived, non-disabled, and non-outdated Elm repositories with most stars on GitHub

GitHub defines the language of a repository based on the majority of lines of code. For example, the repository `nammayatri/nammayatri` contains a PureScript frontend and a Haskell backend but appears only in the PureScript list, despite having enough starts for both lists. However, because 48.3 % of the lines of code are PureScript code and only 33.3 % are Haskell code the GitHub

[4] Let declarations in do notation are classified as local definition.

| | Repo Name | Stars | Variables | Definitions | | Repo Name | Stars | Variables | Definitions |
|---|---|---|---|---|---|---|---|---|---|
| 1 | koalaman/shellcheck | 36 689 | 5 689 | 5 293 | 51 | srid/neuron | 1 521 | 1 192 | 835 |
| 2 | jgm/pandoc | 35 238 | 23 201 | 12 643 | 52 | yi-editor/yi | 1 512 | 6 496 | 4 353 |
| 3 | PostgREST/postgrest | 24 151 | 1 788 | 1 571 | 53 | ndmitchell/hlint | 1 485 | 2 807 | 1 421 |
| 4 | hadolint/hadolint | 10 567 | 938 | 1 008 | 54 | clash-lang/clash-compiler | 1 454 | 20 663 | 11 201 |
| 5 | github/semantic | 8 990 | 3 597 | 7 740 | 55 | HuwCampbell/grenade | 1 451 | 844 | 686 |
| 6 | purescript/purescript | 8 616 | 14 331 | 5 798 | 56 | google/haskell-trainings | 1 396 | 332 | 297 |
| 7 | elm/compiler | 7 585 | 22 472 | 8 352 | 57 | input-output-hk/plutus-pioneer-program | 1 388 | 206 | 291 |
| 8 | unisonweb/unison | 5 859 | 33 905 | 15 447 | 58 | erebe/greenclip | 1 369 | 187 | 87 |
| 9 | carp-lang/Carp | 5 576 | 6 233 | 2 873 | 59 | lettier/gifcurry | 1 366 | 971 | 790 |
| 10 | digitallyinduced/ihp | 4 966 | 4 895 | 4 800 | 60 | avh4/elm-format | 1 315 | 4 397 | 1 802 |
| 11 | facebook/Haxl | 4 285 | 1 222 | 464 | 61 | fossas/fossa-cli | 1 314 | 7 755 | 6 845 |
| 12 | kmonad/kmonad | 4 177 | 633 | 429 | 62 | haskell-beginners-2022/course-plan | 1 305 | 0 | 0 |
| 13 | system-f/fp-course | 4 146 | 0 | 0 | 63 | faylang/fay | 1 283 | 2 872 | 1 300 |
| 14 | facebook/duckling | 4 097 | 12 495 | 8 008 | 64 | haskell/aeson | 1 258 | 3 805 | 1 703 |
| 15 | commercialhaskell/stack | 3 995 | 7 768 | 5 447 | 65 | BartoszMilewski/Publications | 1 257 | 0 | 0 |
| 16 | HigherOrderCO/Kind | 3 599 | 3 760 | 972 | 66 | quchen/articles | 1 256 | 181 | 66 |
| 17 | idris-lang/Idris-dev | 3 440 | 31 593 | 7 357 | 67 | google/codeworld | 1 246 | 5 047 | 3 078 |
| 18 | xmonad/xmonad | 3 389 | 861 | 460 | 68 | ucsd-progsys/liquidhaskell | 1 206 | 51 280 | 24 349 |
| 19 | koka-lang/koka | 3 350 | 23 371 | 9 261 | 69 | fullstack-development/developers-roadmap | 1 197 | 0 | 0 |
| 20 | krispo/awesome-haskell | 3 118 | 0 | 0 | 70 | ndmitchell/ghcid | 1 144 | 480 | 246 |
| 21 | simonmichael/hledger | 3 107 | 6 244 | 5 728 | 71 | GaloisInc/cryptol | 1 139 | 17 640 | 7 028 |
| 22 | IntersectMBO/cardano-node | 3 087 | 12 613 | 5 464 | 72 | reanimate/reanimate | 1 135 | 3 181 | 2 306 |
| 23 | ghc/ghc | 3 073 | 15 739 | 8 127 | 73 | evincarofautumn/kitten | 1 107 | 3 270 | 1 048 |
| 24 | SimulaVR/Simula | 2 994 | 2 348 | 911 | 74 | hasktorch/hasktorch | 1 085 | 31 303 | 16 953 |
| 25 | crytic/echidna | 2 791 | 1 762 | 983 | 75 | graninas/software-design-in-haskell | 1 084 | 0 | 2 |
| 26 | haskell/haskell-language-server | 2 738 | 10 503 | 6 379 | 76 | uber/queryparser | 1 077 | 5 807 | 3 214 |
| 27 | jaspervdj/hakyll | 2 716 | 1 370 | 800 | 77 | reflex-frp/reflex | 1 073 | 2 460 | 1 050 |
| 28 | yesodweb/yesod | 2 652 | 3 930 | 2 408 | 78 | matterhorn-chat/matterhorn | 1 055 | 4 331 | 3 104 |
| 29 | wireapp/wire-server | 2 638 | 35 688 | 20 608 | 79 | polysemy-research/polysemy | 1 041 | 975 | 378 |
| 30 | ghcjs/ghcjs | 2 607 | 10 607 | 4 503 | 80 | grin-compiler/grin | 1 029 | 7 897 | 3 464 |
| 31 | sdiehl/wiwinwlh | 2 580 | 54 | 56 | 81 | tonymorris/fp-course | 1 028 | 1 329 | 491 |
| 32 | b3nj5m1n/xdg-ninja | 2 559 | 109 | 47 | 82 | phuhl/linux_notification_center | 1 019 | 550 | 373 |
| 33 | agda/agda | 2 539 | 48 908 | 16 773 | 83 | kowainik/learn4haskell | 1 010 | 29 | 35 |
| 34 | jaspervdj/patat | 2 473 | 1 029 | 634 | 84 | haskell/stylish-haskell | 995 | 922 | 581 |
| 35 | diku-dk/futhark | 2 443 | 42 842 | 15 676 | 85 | thma/LtuPatternFactory | 995 | 490 | 354 |
| 36 | dmjio/miso | 2 210 | 543 | 873 | 86 | leksah/leksah | 976 | 7 654 | 2 950 |
| 37 | jgm/gitit | 2 164 | 1 081 | 766 | 87 | aviaviavi/toodles | 973 | 272 | 188 |
| 38 | dapphub/dapptools | 2 103 | 3 650 | 2 028 | 88 | tweag/ormolu | 972 | 1 892 | 1 209 |
| 39 | ekmett/lens | 2 035 | 7 638 | 2 170 | 89 | NixOS/nixfmt | 970 | 1 084 | 332 |
| 40 | lamdu/lamdu | 1 857 | 5 389 | 3 614 | 90 | obsidiansystems/obelisk | 968 | 1 611 | 903 |
| 41 | haskell-servant/servant | 1 836 | 2 084 | 1 226 | 91 | maralorn/nix-output-monitor | 964 | 712 | 590 |
| 42 | BurntSushi/erd | 1 808 | 164 | 156 | 92 | B-Lang-org/bsc | 963 | 0 | 0 |
| 43 | fosskers/aura | 1 780 | 1 066 | 980 | 93 | lierdakil/pandoc-crossref | 959 | 641 | 390 |
| 44 | scotty-web/scotty | 1 726 | 372 | 316 | 94 | Gabriella439/turtle | 945 | 891 | 549 |
| 45 | haskell/cabal | 1 635 | 27 371 | 16 348 | 95 | mrkkrp/megaparsec | 928 | 1 162 | 602 |
| 46 | nmattia/niv | 1 634 | 454 | 355 | 96 | pcapriotti/optparse-applicative | 915 | 767 | 457 |
| 47 | jtdaugherty/brick | 1 615 | 1 766 | 1 783 | 97 | egison/egison | 913 | 3 499 | 1 169 |
| 48 | google-research/dex-lang | 1 595 | 14 250 | 3 366 | 98 | agentm/project-m36 | 913 | 6 928 | 2 890 |
| 49 | tensorflow/haskell | 1 583 | 917 | 756 | 99 | AccelerateHS/accelerate | 912 | 9 383 | 3 262 |
| 50 | IntersectMBO/plutus | 1 582 | 17 300 | 8 887 | 100 | samtay/tetris | 906 | 141 | 149 |

Fig. 2. Non-archived, non-disabled, and non-outdated Haskell repositories with most stars on GitHub

API does not yield this repository when searching for Haskell repositories. Similarly, `simplex-chat/simplex-chat` was initially part of the list of Haskell repositories but disappeared as the share of Kotlin code in the repository raised to 32.7 % and exceeded the share of Haskell code, which is 31.1 %.

We have implemented a Haskell application to clone the repositories, extract source files, and collect identifiers from abstract syntax trees. As Singer et al. [2008] noted, code analysis for empirical studies presents unexpected challenges. To minimize errors, we implemented the entire processing pipeline in Haskell, allowing us to use a uniform AST traversal technique [Mitchell and Runciman 2007] across multiple languages and maintain a single data type for collected identifiers to reduce inconsistencies. While Haskell-based parsers exist for Elm, Haskell, and PureScript, no complete open-source OCaml parser in Haskell was available. To address this, we implemented a Haskell parser capable of processing a debug output from the OCaml compiler. Even with full parser support, most parsers target only the latest language version. However, repositories sometimes use older versions – for instance, some PureScript repositories use older language versions despite our

| | Repo Name | Stars | Variables | Definitions | | Repo Name | Stars | Variables | Definitions |
|---|---|---|---|---|---|---|---|---|---|
| 1 | facebook/flow | 22 116 | 47 805 | 40 408 | 51 | riot-ml/riot | 591 | 1 206 | 991 |
| 2 | facebook/infer | 15 041 | 45 266 | 32 044 | 52 | mukul-rathi/bolt | 582 | 3 156 | 557 |
| 3 | semgrep/semgrep | 10 853 | 62 579 | 46 522 | 53 | janestreet/learn-ocaml-workshop | 576 | 492 | 633 |
| 4 | reasonml/reason | 10 167 | 4 682 | 3 308 | 54 | ocaml-multicore/eio | 575 | 4 025 | 3 022 |
| 5 | facebook/pyre-check | 6 890 | 19 763 | 17 573 | 55 | mahsu/MariOCaml | 542 | 0 | 0 |
| 6 | astrada/google-drive-ocamlfuse | 5 621 | 1 967 | 2 135 | 56 | inhabitedtype/httpaf | 537 | 897 | 963 |
| 7 | ocaml/ocaml | 5 560 | 51 543 | 34 536 | 57 | linoscope/CAMLBOY | 525 | 799 | 1 183 |
| 8 | coq/coq | 4 895 | 98 541 | 71 151 | 58 | janestreet/bonsai | 524 | 4 604 | 3 302 |
| 9 | janestreet/magic-trace | 4 749 | 1 514 | 1 656 | 59 | dfinity/motoko | 524 | 19 911 | 12 429 |
| 10 | batsh-dev-team/Batsh | 4 321 | 817 | 470 | 60 | ocaml-batteries-team/batteries-included | 522 | 2 389 | 1 722 |
| 11 | bcpierce00/unison | 4 271 | 7 629 | 6 765 | 61 | 0install/0install | 512 | 4 867 | 4 531 |
| 12 | mirage/mirage | 2 589 | 2 414 | 2 113 | 62 | o1-labs/snarky | 499 | 2 619 | 2 591 |
| 13 | comby-tools/comby | 2 437 | 3 311 | 4 396 | 63 | esumii/min-caml | 496 | 0 | 0 |
| 14 | dmtrKovalenko/odiff | 2 082 | 213 | 295 | 64 | vult-dsp/vult | 495 | 0 | 0 |
| 15 | BinaryAnalysisPlatform/bap | 2 081 | 33 081 | 23 939 | 65 | c-cube/ocaml-containers | 495 | 4 909 | 2 879 |
| 16 | MinaProtocol/mina | 2 017 | 33 518 | 43 954 | 66 | moonbitlang/moonbit-compiler | 479 | 44 354 | 69 424 |
| 17 | CatalaLang/catala | 2 009 | 8 148 | 5 849 | 67 | ocaml-ppx/ppx_deriving | 472 | 847 | 600 |
| 18 | mirage/irmin | 1 861 | 12 850 | 11 584 | 68 | mirage/alcotest | 461 | 733 | 646 |
| 19 | airbus-seclab/bincat | 1 712 | 9 422 | 8 451 | 69 | jrh13/hol-light | 441 | 0 | 0 |
| 20 | ocaml/dune | 1 652 | 41 316 | 33 463 | 70 | uber/NEAL | 428 | 0 | 0 |
| 21 | aantron/dream | 1 636 | 1 997 | 1 983 | 71 | terrateamio/terrateam | 423 | 700 | 555 |
| 22 | ocaml/merlin | 1 594 | 30 272 | 32 479 | 72 | ocaml-multicore/effects-examples | 423 | 589 | 362 |
| 23 | andrejbauer/plzoo | 1 464 | 1 777 | 627 | 73 | LaurentMazare/ocaml-torch | 416 | 11 276 | 15 567 |
| 24 | savonet/liquidsoap | 1 459 | 11 212 | 13 434 | 74 | FStarLang/karamel | 403 | 5 932 | 4 768 |
| 25 | ocaml/opam | 1 266 | 15 261 | 11 257 | 75 | mazeppa-dev/mazeppa | 397 | 1 362 | 762 |
| 26 | owlbarn/owl | 1 228 | 27 741 | 21 150 | 76 | mmmdbybyd/CLNC | 393 | 0 | 0 |
| 27 | realworldocaml/book | 1 197 | 139 500 | 121 636 | 77 | Camilotk/ocaml4noobs | 389 | 7 | 12 |
| 28 | gfngfn/SATySFi | 1 186 | 6 183 | 4 781 | 78 | aantron/lambdasoup | 386 | 569 | 438 |
| 29 | austral/austral | 1 144 | 3 804 | 2 635 | 79 | PataphysicalSociety/soupault | 385 | 1 918 | 1 389 |
| 30 | janestreet/core | 1 128 | 9 486 | 9 811 | 80 | janestreet/incr_dom | 385 | 1 041 | 1 319 |
| 31 | moby/vpnkit | 1 119 | 4 972 | 4 179 | 81 | leostera/minttea | 376 | 569 | 722 |
| 32 | leostera/caramel | 1 067 | 19 443 | 15 671 | 82 | yallop/ocaml-ctypes | 375 | 1 663 | 1 481 |
| 33 | ocsigen/js_of_ocaml | 971 | 17 713 | 13 688 | 83 | mirage/ocaml-git | 362 | 4 492 | 3 290 |
| 34 | janestreet/incremental | 887 | 2 027 | 2 445 | 84 | shentufoundation/deepsea | 362 | 7 024 | 3 171 |
| 35 | janestreet/base | 879 | 12 803 | 9 443 | 85 | oxidizing/sihl | 360 | 1 960 | 2 639 |
| 36 | matijapretnar/eff | 865 | 5 583 | 3 239 | 86 | c-cube/qcheck | 358 | 3 769 | 2 170 |
| 37 | melange-re/melange | 865 | 40 875 | 32 624 | 87 | pqwy/notty | 352 | 1 101 | 846 |
| 38 | ocaml-community/utop | 852 | 1 034 | 662 | 88 | ott-lang/ott | 351 | 0 | 0 |
| 39 | ocaml/ocaml-lsp | 785 | 13 278 | 17 446 | 89 | coq/vscoq | 350 | 1 724 | 2 324 |
| 40 | rgrinberg/opium | 766 | 1 115 | 758 | 90 | xapi-project/xen-api | 348 | 46 432 | 44 239 |
| 41 | cs3110/textbook | 766 | 165 | 78 | 91 | ocamllabs/vscode-ocaml-platform | 345 | 2 562 | 2 483 |
| 42 | ocsigen/lwt | 727 | 3 612 | 3 251 | 92 | stedolan/malfunction | 343 | 901 | 433 |
| 43 | mirage/ocaml-cohttp | 717 | 2 822 | 2 442 | 93 | mirage/mirage-tcpip | 343 | 2 005 | 1 800 |
| 44 | janestreet/hardcaml | 681 | 5 525 | 5 370 | 94 | teikalang/teika | 336 | 784 | 793 |
| 45 | ocaml-multicore/ocaml-effects-tutorial | 667 | 0 | 0 | 95 | links-lang/links | 333 | 19 702 | 15 347 |
| 46 | inhabitedtype/angstrom | 662 | 919 | 507 | 96 | ocaml-community/yojson | 329 | 250 | 285 |
| 47 | jaredly/reason-language-server | 657 | 64 534 | 34 977 | 97 | binsec/binsec | 329 | 21 270 | 17 378 |
| 48 | ocaml-ppx/ocamlformat | 645 | 10 218 | 5 983 | 98 | ocaml/odoc | 327 | 15 056 | 10 771 |
| 49 | coccinelle/coccinelle | 632 | 1 446 | 1 395 | 99 | EasyCrypt/easycrypt | 327 | 22 086 | 18 604 |
| 50 | andreas/ocaml-graphql-server | 621 | 955 | 658 | 100 | RedPRL/redtt | 204 | 5 128 | 3 288 |

Fig. 3. Non-archived, non-disabled and non-outdated OCaml repos with most stars on GitHub

exclusion of outdated projects. To mitigate validity threats, our tool logs unparsed source files, and we manually examined parsing failures.

After we are able to parse as many source files as possible, we have to collect identifiers from the abstract syntax tree. We do not collect all literal occurrences of identifiers but only binding or defining identifier occurrences. For example, consider the Haskell function from the introduction, where we have colored identifiers that are collected by our tool.

```
primes = filterPrime [2 ..]
  where
    filterPrime (p : xs) =
      p : filterPrime [x | x ← xs, x ‘mod‘ p /= 0]
```

To answer research question RQ3 collected identifiers are additionally classified. For example, *primes* is classified as top-level definition, *filterPrime* as a local definition, *p* and *xs* as parameters of a

| | Repo Name | Stars | Variables | Definitions | | Repo Name | Stars | Variables | Definitions |
|---|---|---|---|---|---|---|---|---|---|
| 1 | sharkdp/insect | 3 179 | 298 | 194 | 51 | purescript-react/purescript-lumi-components | 106 | 1 169 | 2 195 |
| 2 | nammayatri/nammayatri | 2 102 | 30 175 | 33 491 | 52 | purescript-contrib/purescript-routing | 105 | 127 | 62 |
| 3 | sharkdp/cube-composer | 2 007 | 168 | 177 | 53 | purescript/purescript-quickcheck | 104 | 134 | 93 |
| 4 | purescript-halogen/purescript-halogen | 1 543 | 437 | 716 | 54 | citizennet/purescript-ocelot | 103 | 1 116 | 1 596 |
| 5 | purescript/spago | 794 | 2 081 | 2 162 | 55 | hoodunit/purescript-payload | 99 | 608 | 635 |
| 6 | thomashoneyman/purescript-halogen-realworld | 792 | 316 | 348 | 56 | purescript/registry-dev | 97 | 2 406 | 1 943 |
| 7 | ad-si/Transity | 638 | 585 | 410 | 57 | MonoidMusician/dhall-purescript | 96 | 3 783 | 1 961 |
| 8 | JordanMartinez/purescript-jordans-reference | 592 | 687 | 941 | 58 | hendrikniemann/purescript-graphql | 94 | 313 | 357 |
| 9 | purescript-contrib/pulp | 445 | 655 | 466 | 59 | purescript/purescript-free | 94 | 242 | 69 |
| 10 | purescript-contrib/purescript-react | 402 | 181 | 1 011 | 60 | Plutonomicon/cardano-transaction-lib | 93 | 4 194 | 3 306 |
| 11 | paf31/purescript-thermite | 348 | 93 | 38 | 61 | natefaubion/purescript-tidy | 93 | 695 | 387 |
| 12 | adkelley/javascript-to-purescript | 304 | 533 | 437 | 62 | natefaubion/purescript-routing-duplex | 91 | 179 | 98 |
| 13 | easafe/purescript-flame | 296 | 187 | 1 061 | 63 | natefaubion/purescript-psa | 90 | 72 | 39 |
| 14 | purescript-contrib/purescript-aff | 286 | 56 | 50 | 64 | Kamirus/purescript-selda | 89 | 335 | 263 |
| 15 | sharkdp/purescript-flare | 286 | 131 | 70 | 65 | andys8/type-signature-com | 88 | 117 | 192 |
| 16 | purescript-react/purescript-react-basic | 283 | 11 | 17 | 66 | bodil/purescript-test-unit | 87 | 185 | 93 |
| 17 | purescript-hyper/hyper | 281 | 269 | 200 | 67 | natefaubion/purescript-checked-exceptions | 81 | 2 | 3 |
| 18 | purescript-concur/purescript-concur-react | 272 | 87 | 698 | 68 | f-o-a-m/chanterelle | 80 | 517 | 422 |
| 19 | bodil/purescript-signal | 260 | 39 | 53 | 69 | chriskiehl/home-theater-calculator | 79 | 299 | 532 |
| 20 | pure-c/purec | 237 | 993 | 437 | 70 | purescript/purescript-transformers | 70 | 691 | 175 |
| 21 | j-keck/zfs-snap-diff | 229 | 413 | 488 | 71 | purescript/purescript-record | 68 | 48 | 26 |
| 22 | juspay/purescript-presto | 228 | 271 | 183 | 72 | jonasbuntinx/next-purescript-example | 67 | 28 | 50 |
| 23 | aristanetworks/purescript-backend-optimizer | 202 | 3 632 | 2 042 | 73 | collegevine/purescript-elmish | 67 | 139 | 109 |
| 24 | purescript-react/purescript-react-basic-hooks | 201 | 125 | 93 | 74 | citizennet/purescript-halogen-select | 65 | 55 | 54 |
| 25 | JordanMartinez/purescript-cookbook | 199 | 0 | 0 | 75 | purescript/purescript-foreign | 64 | 35 | 29 |
| 26 | nwolverson/purescript-language-server | 191 | 1 588 | 1 167 | 76 | purescript/purescript-typelevel-prelude | 64 | 5 | 17 |
| 27 | nwolverson/vscode-ide-purescript | 190 | 77 | 57 | 77 | paf31/purescript-foreign-generic | 63 | 78 | 55 |
| 28 | citizennet/purescript-httpure | 184 | 106 | 271 | 78 | yukikurage/purescript-jelly | 62 | 158 | 271 |
| 29 | purescript-express/purescript-express | 180 | 344 | 225 | 79 | f-f/purescript-react-basic-native | 60 | 101 | 2 818 |
| 30 | purescript/purescript-prelude | 163 | 368 | 221 | 80 | purescript-contrib/purescript-matryoshka | 59 | 165 | 140 |
| 31 | natefaubion/purescript-run | 160 | 129 | 124 | 81 | natefaubion/example-functional-compiler | 59 | 732 | 354 |
| 32 | purescript-contrib/purescript-parsing | 152 | 582 | 378 | 82 | mikesol/purescript-ocarina | 59 | 1 472 | 1 226 |
| 33 | jonasbuntinx/purescript-react-realworld | 145 | 366 | 386 | 83 | hdgarrood/multipac | 58 | 675 | 600 |
| 34 | purescript-contrib/purescript-profunctor-lenses | 144 | 503 | 176 | 84 | purescript-halogen/purescript-halogen-vdom | 58 | 266 | 129 |
| 35 | mikesol/purescript-deku | 144 | 1 164 | 5 382 | 85 | sharkdp/purescript-isometric | 58 | 133 | 99 |
| 36 | KSF-Media/gitlab-dashboard | 139 | 59 | 86 | 86 | purescript/purescript-arrays | 57 | 288 | 267 |
| 37 | thomashoneyman/purescript-halogen-formless | 137 | 74 | 87 | 87 | purescript/purescript-lists | 57 | 923 | 340 |
| 38 | kritzcreek/pscid | 136 | 108 | 81 | 88 | purescript/purescript-control | 56 | 18 | 12 |
| 39 | purescript-spec/purescript-spec | 134 | 370 | 208 | 89 | natefaubion/purescript-heterogeneous | 56 | 48 | 20 |
| 40 | justinwoo/purescript-simple-json | 134 | 34 | 45 | 90 | cdparks/lambda-machine | 56 | 605 | 519 |
| 41 | restaumatic/purescript-specular | 134 | 607 | 277 | 91 | purescript-web/purescript-canvas | 55 | 36 | 64 |
| 42 | natefaubion/purescript-variant | 132 | 169 | 182 | 92 | rowtype-yoga/purescript-protobuf | 55 | 1 625 | 1 257 |
| 43 | purescript-halogen/purescript-halogen-template | 128 | 3 | 5 | 93 | f-f/purescript-react-basic-todomvc | 54 | 50 | 79 |
| 44 | f-o-a-m/purescript-web3 | 128 | 654 | 392 | 94 | purescript-grain/purescript-grain | 36 | 408 | 522 |
| 45 | purescript-contrib/purescript-affjax | 123 | 70 | 66 | 95 | afcondon/purescript-d3-tagless-II | 36 | 1 112 | 1 590 |
| 46 | purescript-python/purescript-python | 122 | 19 | 22 | 96 | joneshf/purescript-option | 36 | 196 | 330 |
| 47 | purescript/trypurescript | 120 | 158 | 137 | 97 | laurentpayot/purescript-for-elm-developers | 36 | 6 | 11 |
| 48 | thomashoneyman/purescript-halogen-hooks | 116 | 138 | 134 | 98 | purescript-web/purescript-web-dom | 34 | 17 | 144 |
| 49 | nwolverson/atom-ide-purescript | 115 | 235 | 277 | 99 | albertprz/puresheet | 25 | 1 476 | 1 174 |
| 50 | purescript-contrib/purescript-css | 107 | 439 | 578 | 100 | rowtype-yoga/purescript-fetch | 25 | 22 | 40 |

Fig. 4. Non-archived, non-disabled, and non-outdated PureScript repositories with most stars on GitHub

local function, and $x$ as do variable[5]. We do not distinguish between constants and functions because it would require type information. For example, while *primes* is syntactically a constant, it is a function, because *filterPrime* is partially applied. Although we could classify definitions with arguments as functions, the remaining definitions could be constants or functions. Type annotations could aid the classification of the remaining definitions, but local definitions often lack them. In Elm 91.2 %, in Haskell 94.8 %, and in PureScript 98.7 % of top-level definitions have type annotations. However, in Elm 13.2 %, in Haskell 11.9 %, and in PureScript 21.5 % of local definitions have type annotations. While in Elm, Haskell, and PureScript typically a type annotation is provided for the entire function, in OCaml argument types and a return type are provided. For OCaml, we count the number of definitions with return type annotations, as these annotations allow the identification of eta-reduced definitions. In OCaml, only 7.4 % of top-level definitions, 6.5 % of

[5] We classify $x$ as do variable because a list comprehension is basically a syntactic variant of a do block using the list monad.

module definitions, and 0.9 % of local definitions have return type annotations. Consequently, even when type annotations are utilized, a significant number of definitions remain ambiguous.

In addition to data collection, we implemented a visualization tool that prints source files to the console with highlighted and annotated collected identifiers. This allowed us to efficiently verify data collection results for randomly selected repositories. For each language, we have examined three randomly chosen repositories, replacing any that contained only a very small number of identifiers to ensure meaningful inspection.

Highlighting identifiers proved unexpectedly challenging. For example, the parser for the Elm did not yield sufficient source code positions in order to highlight all identifiers. Therefore, we have extended the abstract syntax tree and the parser with additional source code positions. In some cases, Haskell's C preprocessor makes it impossible to uniquely determine source code positions. For Haskell and OCaml, we rely on file information from the abstract syntax tree, as the preprocessor inserts line markers that reference the original file. However, some Haskell repositories generate type class instances via preprocessor macros, which lack valid source positions and cannot be highlighted. Finally, some abstract syntax trees still do not provide sufficient source code positions. For instance, in PureScript the source position of a foreign definition references the start of the line and not the start of the identifier. To address these corner cases, our tool lists unhighlighted identifiers, which we manually inspected for all repositories.

As previously noted, we collect only binding or defining identifier occurrences. In rule-based function definitions, the function name is collected only once. For example, consider the following function definition from the Haskell repository `PostgREST/postgrest`.

```
resolveLogicTree :: ResolverContext → LogicTree → CoercibleLogicTree
resolveLogicTree ctx (Stmnt flt) = CoercibleStmnt $ resolveFilter ctx flt
resolveLogicTree ctx (Expr b op lts) = CoercibleExpr b op (map (resolveLogicTree ctx) lts)
```

While the function name *resolveLogicTree* is collected only once, the variable name *ctx* is collected multiple times, as developers could, in theory, choose different names for the two occurrences of *ctx*. This phenomenon is not restricted to rule-based definitions but occurs in case expressions as well. For example, consider the following example from the Haskell repository `haskell-servant/servant`. Our tool collects the variable name *ex* four times as these identifier occurrences are all binding.

```
unfoldRequestArgument _ errReq errSt mex =
  case (sbool :: SBool (FoldRequired mods), mex, sbool :: SBool (FoldLenient mods)) of
    (STrue, Nothing, _)  → errReq
    (SFalse, Nothing, _) → return Nothing
    (STrue, Just ex, STrue)  → return ex
    (STrue, Just ex, SFalse) → either errSt return ex
    (SFalse, Just ex, STrue)  → return (Just ex)
    (SFalse, Just ex, SFalse) → either errSt (return.Just) ex
```

As another example for binding or defining identifier occurrences, we collect record selectors when they are defined and not when they are used. For instance, all analyzed languages support pattern matching on records by using field names. We do not collect the names of these variables becuase they are determined by the names of the record fields. For example, consider the following Elm code from the repository `wende/elchemy`.

```
parse : String → String → List Statement
parse fileName code =
  case Ast.parse (prepare code) of
```

```
    Ok (_, _, statements) →
      statements
    Err ((), { input, position }, [ msg ]) →
      let
        (line, column) =
          getLinePosition position code
      in
        Debug.crash <|
          "]ERR> Parsing error in:\n "
            ++ fileName
            ++ ":"
            ++ toString line
            ++ ":"
            ++ toString column
            ++ "\n"
            ++ msg
            ++ "\nat:\n "
            ++ (input
              |> String.lines
              |> List.take 30
              |> String.join "\n"
              )
            ++ "\n"
    err →
      Debug.crash (toString err)
```

The constructor *Err* takes a triple as argument and the second component of this triple is a record. This record has at least two fields named *input* and *position*. We do not collect the identifiers *input* and *position* because they are record selectors and their names are chosen when defining the corresponding record.

In languages like Elm and PureScript, which provide lightweight record types, our approach may introduce unexpected duplicates or produce fewer record selectors than expected. For instance, consider the following function from the Elm repository `rtfeldman/elm-css`.

```
combineLengths :
  (Float → Float → Float)
   → { r | numericValue : Float, unitLabel : String, value : String }
   → { r | numericValue : Float, unitLabel : String, value : String }
   → { r | numericValue : Float, unitLabel : String, value : String }
combineLengths operation firstLength secondLength =
  let
    numericValue =
      operation firstLength.numericValue secondLength.numericValue
    value =
      String.fromFloat numericValue ++ firstLength.unitLabel
  in
  { firstLength | value = value, numericValue = numericValue }
```

Type declarations and type annotations determine the defining identifiers of record selectors. As a result, *numericValue*, *unitLabel*, and *value* are collected three times as renaming one requires updating the corresponding record selectors but not necessarily affecting their names in other arguments of *combineLengths*. However, omitting the type signature from *combineLengths* prevents the collection of the names of the record selectors entirely.

Finally, we collect identifiers from type classes in Haskell and PureScript as we consider them to be a fundamental part of these functional programming languages. Since the type class declaration serves as the defining element, we collect the names of definitions within type class declarations but not within type class instances. In contrast, we collect identifiers of parameters of definitions in both type class declarations and instances.

In OCaml, we introduce an additional class of module definitions that contains module-top-level definitions. The scope of these definitions is neither file-top-level nor local. We do not collect definition names from signatures in OCaml. A potentially more appropriate approach would be to collect definition names from signatures while excluding those from modules that implement a signature. However, unlike type classes, modules may contain module-top-level definitions that are not part of the signature. In contrast to definitions, in the case of record selectors, we do not differentiate between module-level and top-level record selectors.

To have greater control over the collected code, we extract the list of source files of a repository from build configuration files rather than indiscriminately collecting all language-specific source files from a repository. This approach is particularly crucial for Haskell, as the language supports various language extensions that must be explicitly passed to the parser. Language extensions can be enabled in the header of a source file or in the build configuration file. Enabling all extensions leads to parse errors in valid Haskell code. In the case of Haskell and OCaml the build configuration files indicate whether preprocessing is required, as both languages provide C preprocessor directives, which have to be processed before parsing the source file.

We focus on production code and exclude code explicitly marked as test code during data collection. Aman et al. [2021] similarly exclude "test programs" but do not specify how test programs are identified. In Haskell and OCaml, we leverage information from the build configuration file to exclude source files from test targets, as well as benchmark targets in Haskell. For Elm and PureScript, we exclude folders that are conventionally used for test code. However, this approach does not eliminate all test-related code, as some repositories include additional build configurations and source files for testing or benchmarking their provided code.

Some repositories in figs. 1 to 4 provide zero or only very few identifiers. To reduce the likelihood of bugs in the data collection, we have manually inspected these repositories. Kalliamvakou et al. [2014] did a study about promises and perils of mining GitHub and observed that "A large portion of repositories are not for software development." Similarly, the following repositories do not provide software projects but some form of documentation regarding the corresponding programming language, for example, best practices or links in the form of markdown files.

- `cultureamp/react-elm-components` (Elm)
- `jah2488/elm-companies` (Elm)
- `sporto/elm-patterns` (Elm)
- `system-f/fp-course` (Haskell)
- `krispo/awesome-haskell` (Haskell)
- `haskell-beginners-2022/course-plan` (Haskell)
- `BartoszMilewski/Publications` (Haskell)
- `fullstack-development/developers-roadmap` (Haskell)
- `graninas/software-design-in-haskell` (Haskell)

Some of these projects only provide dummy source files without identifiers or they provide some source files with example code but no build configuration file.

Besides repositories that mainly provide information material and not code, the following repositories use a Makefile or a custom script to build the code.

- `B-Lang-org/bsc` (Haskell)
- `JordanMartinez/purescript-cookbook` (PureScript)
- `ocaml-multicore/ocaml-effects-tutorial` (OCaml)
- `mahsu/MariOCaml` (OCaml)
- `esumii/min-caml` (OCaml)
- `vult-dsp/vult` (OCaml)
- `jrh13/hol-light` (OCaml)
- `uber/NEAL` (OCaml)
- `ott-lang/ott` (OCaml)

We are not able to extract the identifiers from these repositories because our collection is based on build configuration files. Makefiles are particularly popular in the OCaml community.

Finally, the repository

- `mmmdbybyd/CLNC` (OCaml)

is incorrectly classified as OCaml code. The repository contains a single file with the extension `ml`, which appears to be some sort of configuration file.

The data collection process produces three types of CSV files. First, the data collection produces one CSV file per language where each row contains information about a single repository. For each repository the CSV file contains owner, name, number of stars, and date of the last commit to the default branch in UTC, as well as the URL of the repository. This data is obtained from the GitHub API. Second, after cloning all repositories, another CSV file per language captures the commit hashes of checked-out commits. Third, for each repository a CSV file contains one row for every identifier that was collected from the repository. For every identifier the CSV file lists the path of the source file that contains the identifier, the build configuration file that references the source file, the source position in the form of line and column of the identifier, and a classification of the identifier, for example, *Definition TopLevel*, *Definition Local*, *Do*, *Case*, *Lambda*, *Parameter* (*FunctionParameter TopLevel*), etc.

### 3.2 Data Analysis

Because we are interested in the length of identifiers, we might ask ourself, what the length of an identifier is. Most related work defines the length of an identifier as the number of characters. However, Feitelson et al. [2022], for example, present a histogram of identifier length and ignore underscores that separate words when calculating the length. In our opinion this approach is very reasonable. For example, in a language with underscore identifiers, the number of characters is also effected by the number of hard words if we count the underscores as well.

To assess the impact of non-alphabetic characters on identifier length, we have counted the number of identifiers that contain non-alphabetic characters. We distinguish between identifiers that contain only underscores and those where at least one non-alphabetic character is not an underscore. **??** presents the results for Elm, Haskell, OCaml, and PureScript.

Haskell has a considerable number, and OCaml a large number of identifiers in which all non-alphabetic characters are underscores. This is expected for OCaml, which favors underscore case, while the other languages prefer camel case In Haskell the considerable number is mainly due to `diku-dk/futhark`, which mixes camel and underscore case. In OCaml the proportion of identifiers with only underscores increases with length.

This observation aligns with findings of Aman et al. [2021] and Feitelson [2023], who observed that Java developers tend to avoid long words. If OCaml developers exhibit a similar tendency, longer identifiers are more likely to consist of multiple wors and, therefore, have a higher probability of containing at least one underscore. Figure 5 also shows that variable names containing at least one non-underscore non-alphabetic character are rare, with the exception of names of length two. Definition names show a similar distribution as variable names, while the percentage of identifiers containing non-alphabetic characters is slightly higher. In all languages we observe that for length two a comparatively high percentage are identifiers with non-alphabetic characters that are not all underscores. In Haskell, Elm, OCaml, and PureScript identifiers like *x1*, *x2*, and *x′* or *x_* are used to construct "another *x*". This phenomenon is also known in the literature, for example, Aman et al. [2021] observed that Java programs use "numbered" identifiers like x2, Peruma [2022] calls a name composed with a number at the end *Distinguisher*, while Gresta et al. [2023] call it *Kings*. As identifiers like *x*, *x1*, *x2*, *x′*, and *x_* almost convey the same semantic information, they may belong to the same equivalence class.

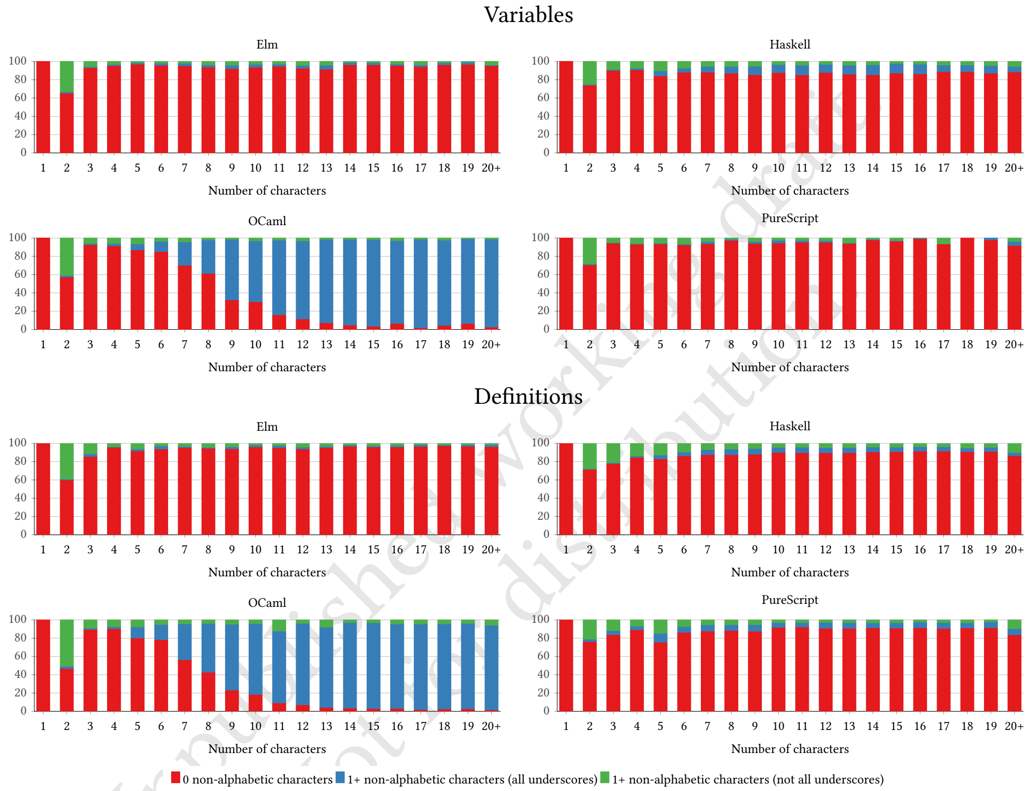


Fig. 5. Percentages of names with non-alphabetic characters

To account for variations in non-alphabetic characters across languages and identifier lengths, we adopt a refined definnition of identifier length. Besides identifiers that end with a number Peruma [2022] provides four other categories of identifiers that contain digits, namely, *Synonym*, *Version Number*, *Specification*, and *Domain/Technology*. In all these cases the digits are an integral part of the identifier and should contribute to the length. For instance, the 2 in the identifier *_offset2bag* from the Haskell repository hasktorch/hasktorch is classified as *Synonym* and is an integral part of the identifier. To account for numbered identifiers, we remove trailing digits before measuring the length. This approach is an approximation, for example, we assign length three to the identifier *ipv4* from the Haskell repository IntersectMBO/cardano-node although the digit is an integral part of the identifier. To account for identifiers like $x'$, $x\_$ and underscore case identifiers, we remove all non-alphanumeric characters. However, if we remove all non-alphanumeric characters from an identifier, there are identifiers that have length zero, for instance, names like *_1*. While the amount of these identifiers is very small, we decided to take the number of characters if all characters are non-alphabetic and remove all non-alphanumeric characters otherwise.

In summary the length of an identifier is the number of characters if all characters are non-alphabetic. Otherwise we remove trailing digits and count the number of alphanumeric characters. This refined definition prevents direct comparison with prior studies, which typically use the number of characters. However, we opted for this approach as character-based length measurement seems inadequate when comparing underscore with camel case identifiers.

The data analysis is performed by an additional Haskell program that processes the CSV file containing all repositories for a given language. From the name of the repository the name of the CSV file is derived that contains all identifiers in that repository. It derives the corresponding CSV file for each repository to extract identifier data. The program produces CSV files with aggregated statistics, such as identifier length distributions that were used to produce this document. Aggregated data is generated both per language and per repository. The per-repository data helps identify threats to validity, such as repositories with a high number of generated identifiers.

## 4 Threats to Validity

This section discusses threats to validity based on the guidelines of Petersen and Gencel [2013]. Following a pragmatist perspective, we address internal validity, external validity, construct validity, and reliability.

### 4.1 Internal Validity

Internal validity concerns factors that might affect the results but are not considered. We examine the relationship between projects in statically-typed functional programming languages and identifier length. While Elm, Haskell, OCaml, and PureScript are all statically-typed functional programming languages, Elm and PureScript are translated to JavaScript and mainly used for frontend web development. Therefore, the application area of the language might affect the length of identifiers. We reduce this threat by considering two frontend programming languages and two general-purpose programming languages.

Elm uses the model-view-update (MVU) architecture. In this architecture, there are functions called *init*, *view*, and *update* and parameters called *model* and *msg*. While developers are free to use different names for *init*, *view*, *update*, *model*, and *msg*, we expect many projects to follow this convention. To mitigate the thread that the length distribution in Elm is distorted by many occurrences of these names we put them into their own class to visualize their effect. Additionally, we analyze the most common identifiers for all languages to mitigate the threat that distributions are distorted by some other default naming pattern.

While Elm and PureScript target frontend developmen, Elm, Haskell, and PureScript are also pure functional programming languages. To mitigate the thread that purity affects the identifier length, we include OCaml, a non-pure functional programming language. OCaml differs from the other languages in many aspects, for example, it is a hybrid language as it also provides object-oriented programming features. Since we focus on functional programming, we exclude names and parameters of methods, classes and constructors in the our analysis. However, we include identifiers from case expressions, lambda expressions, and let definitions, even when used in object-oriented features.

Finally, other factors correlated with the programming language may influence identifier length. For instance, some languages may be used by more experienced developers, and experience might affect naming habbits. This threat remains open as we do not collect any additional metadata about the source code.

## 4.2 External Validity

Transferability or external validity is concerned with the aspect whether our results can be generalized. Since we examine the relationship between statically-typed functional programming languages and identifier length, analyzing a single language could limit generalizibility. To reduce this threat, we include four statically-typed functional programming languages.

Our results, based on non-archived, non-disabled, and non-outdated repositories with the most stars, may not generalize to active, mature projects. This threat remains open. Munaiah et al. [2016] found that using stars to identify engineered software projects has a high precision but a low recall. That is, almost all repositories with many stars are engineered software projects. However, many engineered software projects do not have many stars. Thus, selecting repositories by star count may capture only a specific subset of engineered software projects.

Finally, there is a threat that the collected data represents the naming style of a specific repository or a subset of developers rather than the naming style of the entire language. To reduce this threat, we manually inspected the data and excluded one repository with an unusually high number of identifiers. However, the threat that the collected data represents a subset of developers remains open. For instance, our PureScript repositories stem from only 51 owners. While an owner does not necessarily contribute most code to a repository or the owner might even be an organization, to reduce this threat we would have to collect additional metadata about the code, for example, which users contributed to a repository.

## 4.3 Construct Validity

Construct validity is concerned with the question whether we have actually measured the aspect that we intended to measure. Various technical factors pose threats to construct validity, such as collecting duplicate identifiers. For instance, Elm repositories often include demonstration projects that explicitly reference library directories. To address this, we remove duplicate source file directories. In general the tool for highlighting identifiers mitigates the thread that we collect the same code multiple times as the tool is not able to highlight an identifier twice. We applied this tool to all repositories and it reported only a few errors, primarily due to uses of the C preprocessor where it is impossible to highlight the identifier.

Another threat is collecting identical identifiers multiple times due to repositories containing code from other repositories. For instance, `ucsd-progsys/liquidhaskell` contains Haskell's `base`, `containers`, `text`, `vector` and `xmonad` library as benchmark. This threat remains open, as we did not implement measures to detect duplicate code across repositories.

There is a threat that the collected identifiers do not reflect the naming style of functional developers due to the inclusion of large amounts of generated code. Since we process build configuration files but do not actually build the project, no code is generated that is generated as part of the build process. However, some generaged code is put under version control. We reduce this threat by manually inspecting the length distributions of all 400 repositories (section A). We have excluded five repositories because they contain a significant amount of generated code.

Finally, the implementation of the data collection might contain bugs. To reduce this threat we have randomly sampled three repositories per language. For each language we have manually inspected the highlighted identifiers. However, the threat of potential bugs in the data analysis code remains open.

## 5 Data Exclusion

We have excluded five repositories from the data analysis due to significant amounts of generated code: dillonkearns/elm-graphql (Elm), github/semantic (Haskell), hasktorch/hasktorch (Haskell), rowtype-yoga/purescript-protobuf (PureScript), moonbitlang/moonbit-compiler (OCaml), and ocaml/merlin

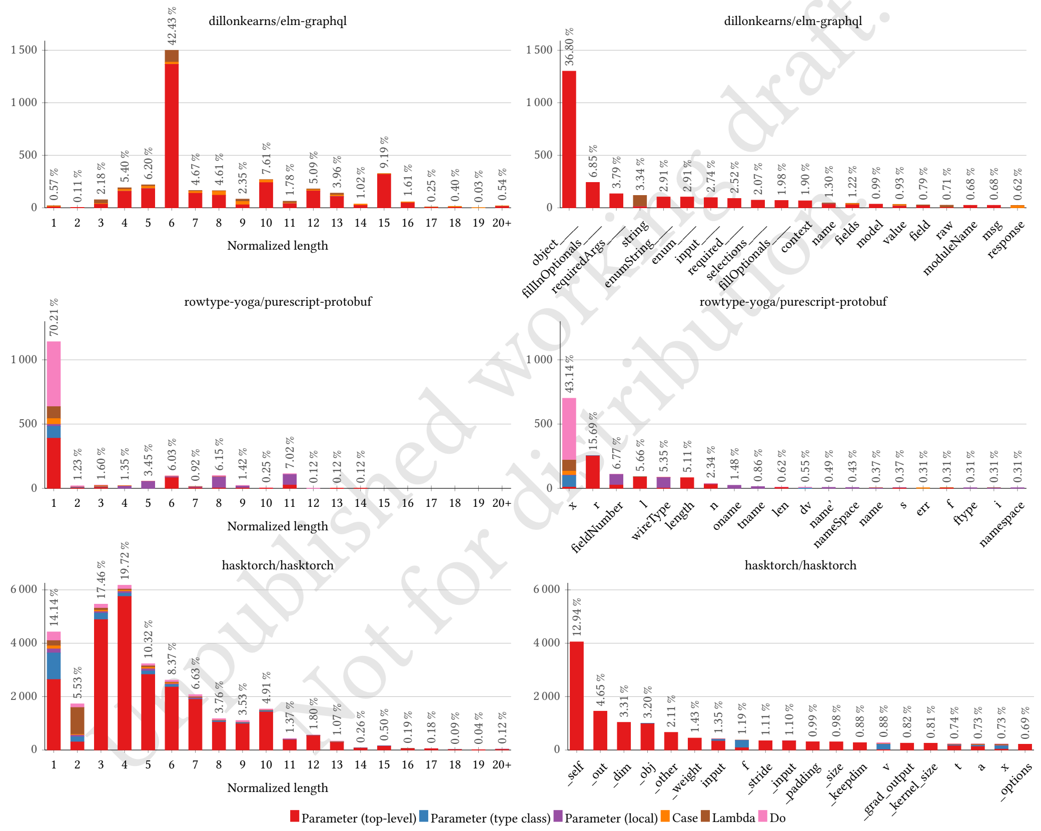


Fig. 6. Length of variable names and most common variable names

Figure 6 shows that in dillonkearns/elm-graphql more than 36 % of all variables are named *object____*. In rowtype-yoga/purescript-protobuf more than 44 % of all variables are named *x* and a large part of these variables are defined in do statements. The corresponding code shows quite repetitive patterns and these source files often begin with comments indicating they were generated. In hasktorch/hasktorch the variables prefixed with an underscore such as *_self* appear in files that begin with comments indicating they were generated.

Figure 7 shows that more than 50 % of definitions in github/semantic are local definitions of length two. An inspection of the collected identifiers revealed that these are almost exclusively of

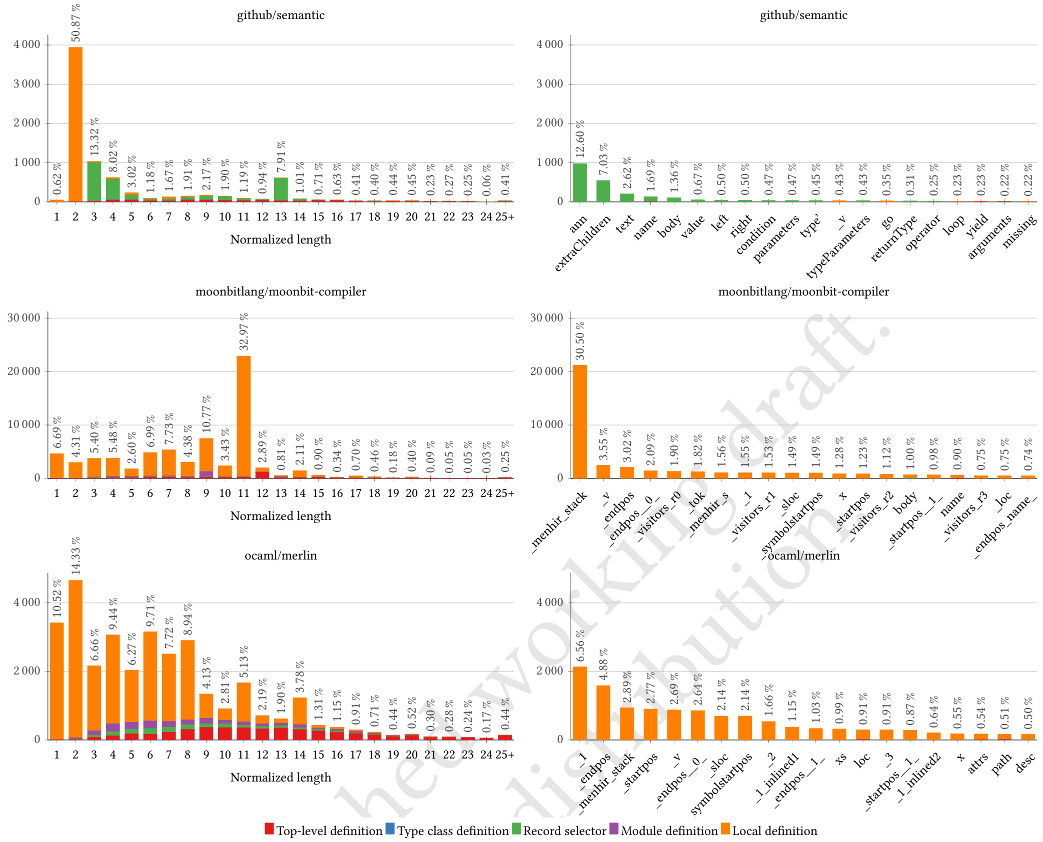


Fig. 7. Length of definition names and most common definition names

the form *c1_<number>*, *c2_<number>*, *r1_<number>*, and *r2_<number>*, where *<number>* is replaced by some arbitrary number. Our metric classifies these names as length two. In `moonbitlang/moonbit-compiler` almost 30 % of definitions are named *_menhir_stack*, indicating that the code is generaged by the Menhir parser generator. In `ocaml/merlin` only 4 % of the definitions are called *_menhir_stack*. However, we still exclude this repository as all identifiers that start with an underscore are from generated code and their amount approximately sums to 25 % of all definition identifiers.

Finally, we have excluded the repository `nammayatri/nammayatri` (PureScript) to prevent it from disproportionately influencing the results. This repository contributes 63 666 identifiers, while the remaining 99 PureScript repositories provide 98 739 identifiers (fig. 4), meaning it accounts for more than 38 % of all PureScript identifiers. Additionally, the variable length distribution of `nammayatri/nammayatri` differs substantially from the distribution of the rest of the PureScript repositories.

## 6 Results

This section discusses the results of our case study. As mentioned before we distinguish two categories of identifiers: variables and definitions. Variables include identifiers introduced through pattern matching in function parameters, case expressions, lambda parameters, and do notation.

Definitions include top-level, module-top-level, and local function and constant definitions, as well as record selectors.

We first analyze the distribution of syntactic classes (fig. 8). Haskell and PureScript provide type classes, do notation, and rule-based pattern matching, while Elm and OCaml lack these features. Consequently, Haskell and PureScript exhibit similar distributions for variables, as do Elm and OCaml. Elm has a similar number of top-level definitions as OCaml has top-level and module definitions combined. For definitions, Elm and PureScript have a higher proportion of record selectors and a lower proportion of local definitions. Similarly, the category of top-level definitions in other languages is divided into top-level and module definitions in OCaml.

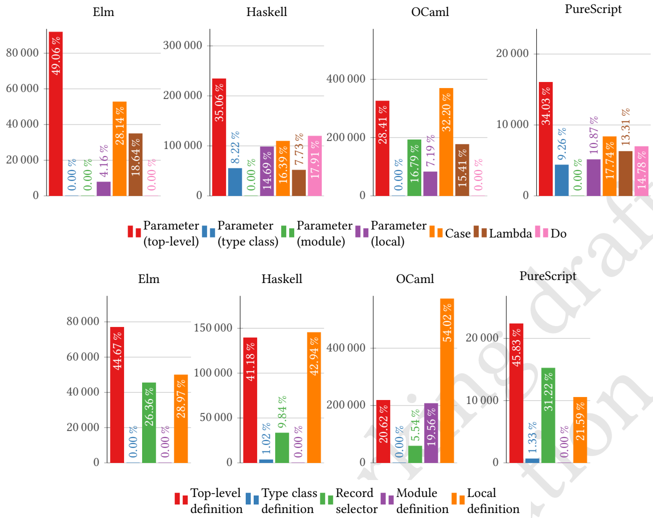


Fig. 8. Percentages of syntactic identifier classes for variables (top) and definitions (bottom)

Additionally, OCaml exhibits a notably high proportion of local definitions. We hypothesize that this results from the sequencing of side effects using local constants.

The number of type class definitions is relatively small, as we collect definition names only from type class declarations and not from instances. If we were to inlcude definition names from both type class declarations and instances, the proportion of names in type classes would be approximately 10 % in Haskell and PureScript. This suggests that a significant number of duplicate names would be collected.

Furthermore, we observe that the number of parameters in local functions is quite small relative to the number of local definitions. This indicates that either local functions have fewer parameters than top-level functions or that local constants outnumber top-level constants.

## 6.1 What length do identifiers have?

To address RQ1, we examine fig. 9, which presents histograms of the normalized length of variable names in Elm, Haskell, OCaml, and PureScript. Section 3.2 defines how the normalized length of an identifier is calculated. The stacked bars represent different syntactic classes, but for RQ1 we focus on the total number of identifiers per length and ignore syntactic classes for now. The percentage at the top of each stack is the percentage of identifiers with the corresponding length in relation to the total number of variables/definitions. To enhance comparability, all diagrams are scaled to 40 %.

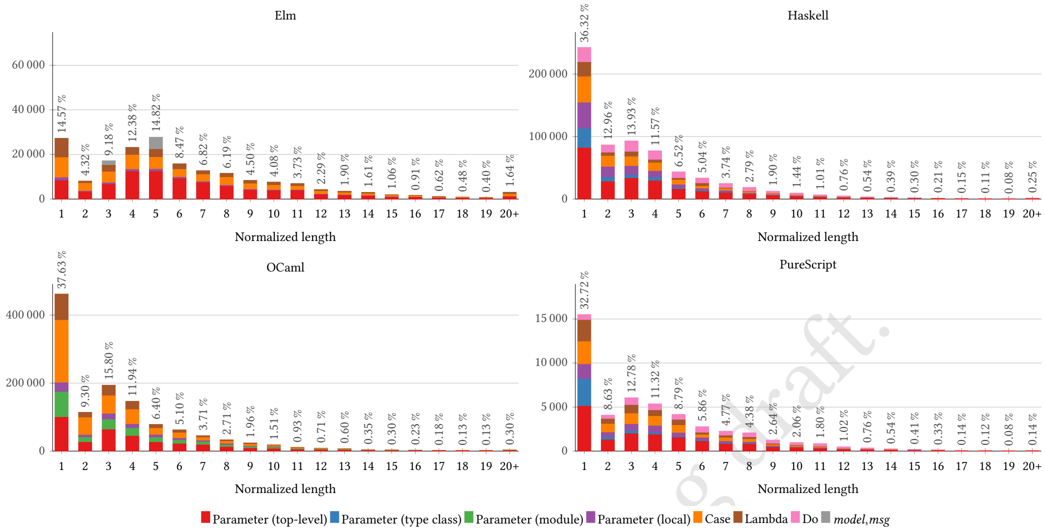


Fig. 9. Length of variable names in Elm, Haskell, OCaml, and PureScript

In Haskell, OCaml, and PureScript, more than 30 % of all variables use single-letter names. In contrast, in Elm fewer than 15 % of all variables use single-letter names. Feitelson [2023] observes that the length distribution of variable names in Java is bi-modal. The distributions of variable name length in fig. 9 are bi-modal as well. The first mode corresponds to single-letter variable names, while the second mode peaks at length five in Elm and at length three in Haskell, PureScript and OCaml. In Elm we have introduced a separate class for variables named *model* and *msg*. A substantial number of variable names fall into this class, for instance, the number of variables named *model* is comparable to the number of other variables of length five introduced in case expressions. However, the prevalence of *model* and *msg* alone does not fully account for the differences between Elm and the other languages.

Comparing results across programming languages requires caution. As noted in section 3.2 comparing numbers of characters might be misleading when comparing a language with camel case identifiers with a language with underscore identifiers. Additionally, identifier counts vary significantly: we have extracted 190 764 variable identifiers for Elm, 702 879 for Haskell, 1 220 356 for OCaml, and 78 891 for PureScript. A small number of total identifiers increases the risk that a single repository strongly influences the distribution. For example, as discused in section 4.3 we have excluded the PureScript repository `nammayatri/nammayatri`. It contributes 30 175 variable identifiers, while all other 99 PureScript repositories contribute 48 716 variable identifiers. Figure 10 presents the length distribution of variable names as well as the list of most common variable names in `nammayatri/nammayatri`. To improve comparability, these distributions are again scaled to 40 %. Unlike other PureScript repositories, where more than 30 % of variables have single-letter names, this repository has fewer than 15 %. Almost all single-letter variables are named *a* and are almost exclusively used in lambda expressions. The variable name *state* is very frequent and is almost exclusively used in parameters of top-level functions. We suspect that the PureScript application in `nammayatri/nammayatri` uses a model-view-update architecture. While Elm uses the names *model* and *msg*, in other languages the names *state* and *action* is preferred in the context of an

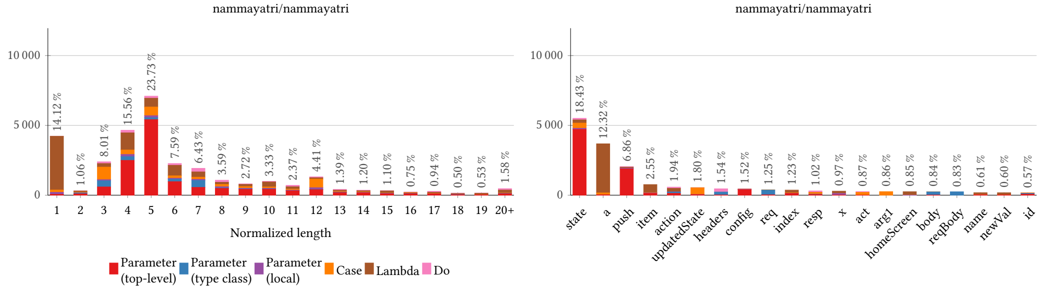


Fig. 10. Length of variable names and most common variable names in nammayatri/nammayatri

MVU architecture. However, the reasons for the other unusual naming patterns in this repository remain unclear.

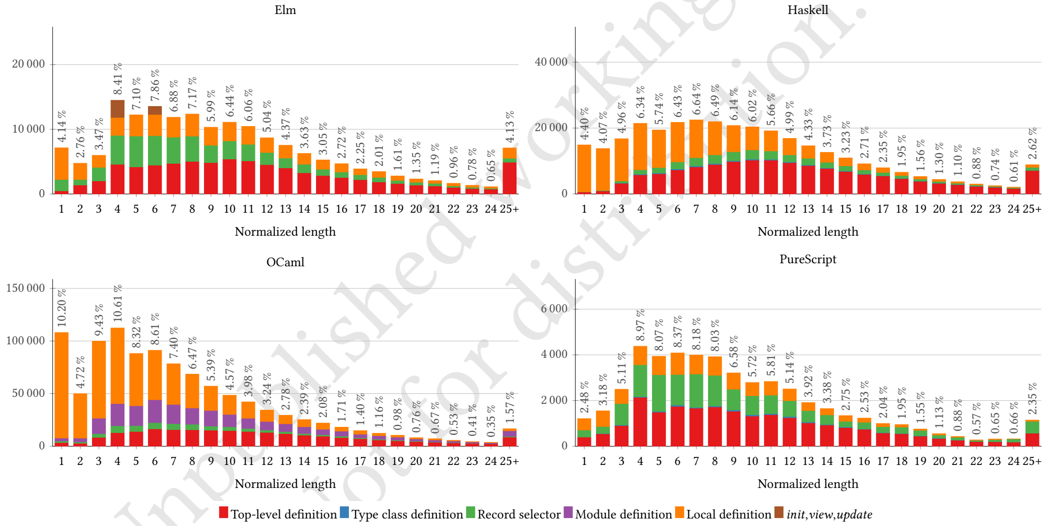


Fig. 11. Length of definition names in Elm, Haskell, OCaml, and PureScript

Figure 11 presents a histogram of identifier length for definitions. As before, we focus on the total number of identifiers per length and ignore syntactic classes for now. The distribution of definition names differs significantly from that of variable names. Notably, Elm's distribution does not stand out as it did for variables. Overall, definition names tend to be longer than variable names. The mode of the distribution is four for Elm, OCaml, and PureScript, while it is seven for Haskell. In Elm we highlight definitions named *init*, *view*, or *update*. Finally, in Elm, Haskell, and in particular in OCaml, there is a noticeable increase in single-letter definition names compared to two letter names.

## 6.2 Does the length of identifiers differ for different languages?

Figure 9 shows that Elm has fewer single-letter variables compared to Haskell, OCaml, and PureScript. Additionally, the mode of non-single-letter variable lengths in Elm is five, whereas it is three in Haskell, OCaml, and PureScript. Figure 12 further supports these observations with average length of identifier categories (variables and definitions) as well as of syntactic classes. In Elm

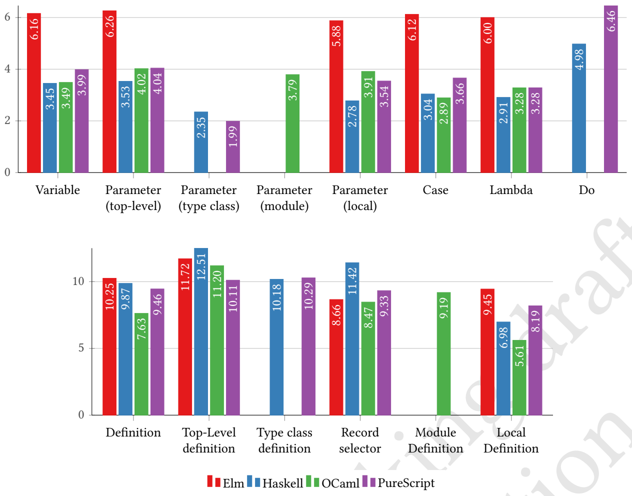


Fig. 12. Average normalized lengths of variable names (top) and definition names (bottom)

variable names have an average length of 6.16, while the average length of all other languages is below 4. The picture for specific classes, that is, parameters, case, and lambda expressions is similar.

In contrast the average length of definition names in Elm is not significantly larger than the average length in Haskell, OCaml, or PureScript. OCaml definitions are the shortest, as alos observed in fig. 11. As we can see here, this is primarily due to shorter local definitions. However, fig. 11 suggests that this effect is largely driven by the high number os single-letter names of local definitions in OCaml. We suspect that this results from the use of single-letter names in local constants, though it remains unclear why this effect is much stronger in OCaml than in Elm and Haskell and is absent in PureScript.

### 6.3 Does the length of identifiers differ for different classes of identifiers?

In all languages, varaible names are on average shorter than definition names. The distributions in fig. 9 and fig. 11 are also quite different. While variables contain more single-letter names than definitions, there are more longer definition names than there are variable names. Additionally, on average local definitions are shorter than top-level definitions as shown in fig. 11. In OCaml, the average length of module definitions is between the average length of top-level and local definitions. The distributions of local and top-level definitions also differ: very short names are rare among top-level definition names and record selectors but more common in local definition names, particularly in OCaml and Haskell. This effect is much smaller in Elm and PureScript.

The average length of parameters of type class functions in Haskell (2.35) and PureScript (1.97) is shorter than that of parameters of top-level functions in Haskell (3.53) and PureScript (4.05). Figure 9 shows that parameters of type class functions contain particularly many single-letter names compared to other identifier classes. In contrast, on average variables in do notation in Haskell (4.98) and PureScript (6.48) are longer than other variable classes. There is no significant difference between average length of other syntactic classes, that is, top-level parameters, local parameters, case, and lambda.

Finally, the mode of the distribution of parameters of top-level functions in Haskell, OCaml, and PureScript is one, whereas it is five in Elm (fig. 9). This suggests that Elm developers adhere to the official style guide's recommendation to avoid single-letter names in top-level function parameters.

### 6.4 Which identifiers are used most often?

Figure 13 presents the 20 most common variable names in Elm, Haskell, OCaml, and PureScript. These names are not normalized, meaning their frequencies reflect exact occurrences. The percentage at the top of the stack is the percentage of identifiers with the corresponding name in relation to the total number of variables/definitions. The number in parentheses is the number of repositories that contain this name.

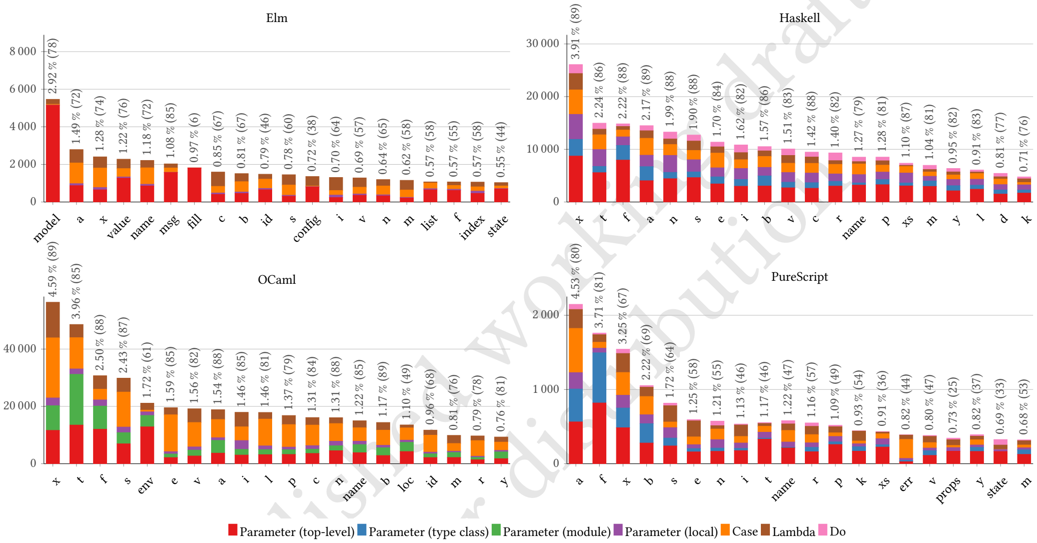


Fig. 13. Most common variables in Elm, Haskell, OCaml, and PureScript

In Elm the most common variable names are dominated by full words like *model*, *value*, and *name*, whereas in Haskell, OCaml, and PureScript they are dominated by single-letter names. The lengths of the most common names align with the modes observed in fig. 9. The full-word *name* is the only non-single-letter name that appears in the 20 most common variable names of all four languages.

The variable name *x* is prevalent in all languages. Although the frequency of variables named *x* is lower in Elm compared to other languages, it still ranks third in the most common variable names. Additionally, *x* is not only frequently used but also used in many repositories, for example, in Elm *x* appears in 74 distinct repositories. While we expected the name *xs* to be widely used for lists, it only appears among the 20 most common variable names in Haskell and PureScript. Additionally, the pattern of naming the head of a list *x* and its tail *xs* likely account for only a small portion of the occurrences of *x*, as 3.91 % of all variable names are *x*, while only 1.10 % of all variable names are *xs*.

The variable name *f* ranks among the top three most common variable names in Haskell, OCaml, and PureScript but appears only in position 18 in Elm. In all languages *f* is primarily used in parameters and rarely in case expressions. Similarly, the names *model*, *msg*, *state*, and *env* are more frequently used as parameters than in case expressions, lambda expressions or do notation.

Figure 13 confirms that the Haskell definition from haskell.org (section 1) is actually a well-chosen representative of naming conventions in Haskell. The variable name *x* is the most common variable name and the names *p* and *xs* are among the twenty most common variable names in Haskell. Even the PureScript definition from purescript.org (section 1) is properly chosen as *name* appears in the 20 most common variable names, although, a single-letter variable name would better reflect PureScript's naming conventions.

Figure 14 presents the 20 most common definition names in Elm, Haskell, OCaml, and PureScript. Notably, there is less consensus on the naming of definitions than on the naming of variables. In all languages the most common definition name accounts for fewer than 1 % of all definitions, whereas the most common variable name accounts for 3 % to 4 % of all variable names.

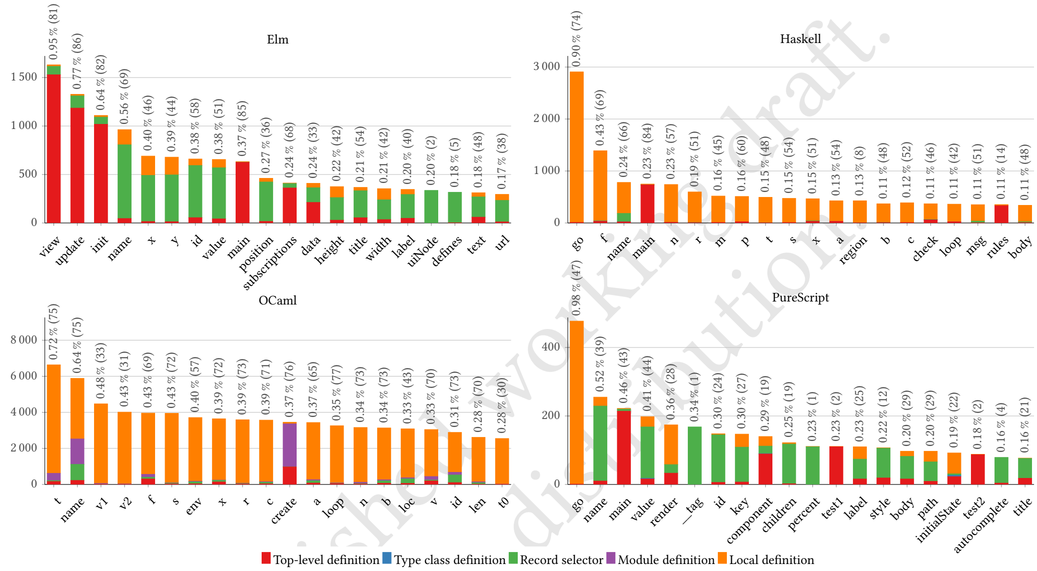


Fig. 14. Most common definitions in Elm, Haskell, OCaml, and PureScript

In Elm, only two of the most common definition names are single-letter names, while PureScript has none. The names *x* and *y* in Elm are primarily used for record selectors, with nearly identical proportions. This suggests that definition names *x* and *y* in Elm are primarily used to store coordinate components. It is unlikely, that observation extends to other languages, as *x* is significantly more frequently than *y* for both variable and definition names.

In contrast to Elm, in Haskell and OCaml, many of the most common definition names are single-letter names. For instance, in Haskell, the second most common definition name is *f*, which is mostly used in local definitions. To illustrate the use of this identifier consider the following code taken from the Haskell repository `fosskers/aura`.

```
fetchTarball :: NonEmpty PkgName → RIO Env ()
fetchTarball ps = do
  ss ← asks settings
  traverse_ (liftIO.g ss) ps
  where
    f :: PkgName → String
```

```
f (PkgName p) =
  "https://aur.archlinux.org/cgit/aur.git/snapshot/" <> T.unpack p <> ".tar.gz"
g :: Settings → PkgName → IO ()
g ss p@(PkgName pn) = urlContents (managerOf ss) (f p) ≫= λcase
  Nothing → warn ss $ missingPkg_5 p
  Just bs → writeFileBinary (T.unpack pn <> ".tar.gz") bs
```

Unlike variable names, the most common definition names are each dominated by a single syntactic class. This suggests a consensus in naming conventions, where specific names are used for specific classes of definition identifiers. Among the most common definition names, those primarily appearing as top-level definitions are associated with the model-view-update architecture (*view*, *update*, *init*, and *subscriptions*) or correspond to *main* – which is a mandatory name of the main entry of Elm, Haskell, and PureScript programs.

In Elm and PureScript, the most common definition names are primarily record selectors, while in Haskell and OCaml, they are primarily local function names. However, Elm and PureScript have more top-level definition names than record selectors overall (fig. 8) and the total number of local definition names is comparable to that of record selectors. This suggests a higher degree of naming consensus for record selectors in Elm and PureScript compared to top-level and local function definitions. In Haskell, the number of local definitions and top-level definitions are approximately equal, indicating a higher degree of consensus in naming local definitions compared to top-level definitions.

Finally, in Haskell and PureScript, local functions are often named *go*. This convention is nearly as prevalent as the use of *view*, *update*, and *init* in the model-view-update architecture. This is notable since the model-view-update architecture – and its associated naming conventions – are very prominent in Elm's official documentation, whereas we are not aware of a similar official guidance for the use of *go* in Haskell.

## 7 Conclusion

This exploratory case study analyzed naming patterns in statically-typed functional programming languages by examining 3 830 575 identifiers from 400 GitHub repositories that use Elm, Haskell, OCaml, or PureScript. Our analysis primarily focuses on identifier length and common naming conventions.

As a preliminary observation, we found that non-underscore, non-alphabetic characters are rarely used in identifier names, with two notable exceptions. First, some names contain only underscores as non-alphabetic characters, with OCaml in particular making use of underscore case. Second, between 21 % and 53 % of two-character names contain non-alphabetic characters that are not underscores. Across all languages, the proportion of names with non-alphabetic characters (excluding those composed entirely of underscores) is highest for names of length two. Based on these observations, we adopted a revised notion of identifier length that excludes trailing digits and non-alphanumeric characters.

As a second preliminary observation, we found that a majority of top-level definitions (91.2 % in Elm, 94.8 % in Haskell, and 98.7 % in PureScript) have type annotations. In contrast, type annotations are significantly less common in local definitions (13.2 % in Elm, 11.9 % in Haskell, and 21.5 % in PureScript). Notably, the use of type annotations is less common in OCaml, only 7.4 % of top-level definitions, 6.5 % of module definitions, and 0.9 % of local definitions have return type annotations.

Our analysis of variable naming conventions reveals significant differences across the examined languages. Specifically, variable names in Elm tend to be longer on average compared to variable names in Haskell, OCaml, and PureScript. The naming conventions among Haskell, OCaml, and

PureScript exhibit greater similarity in terms of length and the use of single-letter names. Single-letter variable names account for over 30 % of names in Haskell, OCaml, and PureScript, while in Elm, their usage is significantly lower (below 15 %). The less widespread use of single-letter names is also observed in the most common variable names: while single-letter names dominate in Haskell, OCaml, and PureScript, full-word names are more common in Elm. The reduced prevalence of single-letter variable names in Elm aligns with its official style guide, marking a notable difference from Haskell despite their syntactic similarities.

Unlike the clear trends in variable naming, definition naming conventions show no uniform pattern across Elm, Haskell, OCaml, and PureScript. In all cases, local definition names tend to be shorter than those of other definition categories. The distributions of local and top-level definitions also differ: very short names are rare among top-level definition names and record selectors but more common in local definition names, particularly in OCaml and Haskell. On average, local definition names in Elm are longer than those in Haskell, OCaml, and PureScript, while OCaml has the shortest local definition names and the highest proportion of single-letter names in local definitions. For other classes of definition names, no language demonstrates a particularly distinctive naming pattern.

The most recurring name in variable names across all languages is *x*. While *x* is the most common name in Haskell and OCaml, it has rank three in Elm and PureScript. In Elm the variable name *model* from the model-view-architecture naming pattern is most common, while in PureScript the name *a* is most common. The name *f* is very common in Haskell, OCaml, and PureScript but not in Elm. In Elm the most common definition names are dominated by names from the model-view-architecture naming conventions. Otherwise *name* is the only name that occurs in the most common definition names of all four languages. In Haskell and PureScript the most common definition name is *go*, which is used for local definitions while in OCaml the most common definition name is *t*.

Our data suggests that there is less consensus on naming definitions across languages than there is for variables. However, there seem to be naming conventions, where specific names are used for specific classes of definition identifiers. For example, the name *go* is used for local definitions and *name* is mainly used for record selectors. In the most common definition names in Elm and PureScript record selectors are overrepresented, which suggests, that there is more consensus in these languages in naming record selector than in naming top-level definitions, which are more common in these languages.

A limitation of our study is the lack of distinction between constants and functions in our dataset. We hypothesize that a significant proportion of short local definitions correspond to constants and that their length distribution is more similar to that of variables. However, without applying type inference, we are unable to make a precise distinction between local functions and constants.

## 8 Future Work

There are three main directions for future work. The first direction is the replication of existing studies. While numerous studies analyze naming practices in existing code, few replicate findings or unify results. Our inspection of supplementary materials revealed several issues, including missing source code references (such as positions or even file names), missing collection dates (which hinder exact replication), unspecified versions of external tools, and incomplete datasets (e.g., entirely missing identifier classes). Therefore, an initial step could be to evaluate the replicability of existing studies and address these issues systematically.

The second direction of future work involves collecting additional data on identifiers in functional programming languages. For instance, incorporating a language such as F# could help verify findings across a broader set of languages. Additionally, we suspect that types influence naming patterns. From our experience, functional programming languages exhibit specific naming conventions for functional types, optional types, error types, and list types. Additionally, while the variable name *x* is common in Haskell, OCaml, and PureScript, it is unclear whether it is used consistently across all types or follows specific conventions. Similarly, we suspect that in PureScript, the variable name *a* often corresponds to a value of a polymorphic type using the type variable *a*. Type information would also facilitate distinguishing between constants and functions, thereby improving our understanding of definition name distributions. In particular in the case of OCaml, this would allow us to reduce the effect of local constants do to sequencing of side effects.

However, analyzing identifier types poses challenges in languages with type inference, such as Elm, Haskell, OCaml, and PureScript. Relying solely on type annotations is insufficient, as, for instance, only 11.9 % of local definitions in Haskell and even only 7.4 % of top-level definitions in OCaml provide explicit type annotations. Thus, type inference would be required, which necessitates resolving project dependencies.

The third direction of future work involves adapting our tool for use in other empirical studies on statically-typed functional programming languages. Our tooling already supports processing abstract syntax trees from projects written in four functional programming languages. This infrastructure could be leveraged to investigate aspects beyond naming, such as the usage of common language features. Additionally, we could extend our tool to compute code metrics for functional languages. For example, prior work [Kamps et al. 2020] has analyzed metrics such as intra-modular complexity, inter-modular complexity, and module size, correlating them with post-release defects.

## References

Surafel Lemma Abebe, Sonia Haiduc, Andrian Marcus, Paolo Tonella, and Giuliano Antoniol. 2009. Analyzing the Evolution of the Source Code Vocabulary. In *2009 13th European Conference on Software Maintenance and Reengineering*. 189–198. doi:10.1109/CSMR.2009.61

Rachel Alpern, Ido Lazer, Issar Tzachor, Hanit Hakim, Sapir Weissbuch, and Dror G. Feitelson. 2024. Reproducing, Extending, and Analyzing Naming Experiments. doi:10.48550/arXiv.2402.10022 arXiv:2402.10022 [cs]

Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. 2021. A Large-Scale Investigation of Local Variable Names in Java Programs: Is Longer Name Better for Broader Scope Variable? In *Quality of Information and Communications Technology*, Ana C. R. Paiva, Ana Rosa Cavalli, Paula Ventura Martins, and Ricardo Pérez-Castillo (Eds.). Vol. 1439. Springer International Publishing, Cham, 489–500. doi:10.1007/978-3-030-85347-1_35

Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G. Feitelson. 2017. Meaningful Identifier Names: The Case of Single-Letter Variables. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, Buenos Aires, Argentina, 45–54. doi:10.1109/ICPC.2017.18

Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. 2009. To CamelCase or Under_score. (2009).

C. Caprile and P. Tonella. 1999. Nomen Est Omen: Analyzing the Language of Function Identifiers. In *Sixth Working Conference on Reverse Engineering (Cat. No.Pr00303)*. 112–122. doi:10.1109/WCRE.1999.806952

Florian Deissenboeck and Markus Pizka. 2006. Concise and Consistent Naming. *Software Quality Journal* 14, 3 (Sept. 2006), 261–282. doi:10.1007/s11219-006-9219-1

Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*, Forrest Shull, Janice Singer, and Dag I. K. Sjøberg (Eds.). Springer London, London, 285–311. doi:10.1007/978-1-84800-044-5_11

Asaf Etgar, Ram Friedman, Shaked Haiman, Dana Perez, and Dror G. Feitelson. 2022. The Effect of Information Content and Length on Name Recollection. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. ACM, Virtual Event, 141–151. doi:10.1145/3524610.3529159

Dror G. Feitelson. 2023. Reanalysis of Empirical Data on Java Local Variables with Narrow and Broad Scope. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, Melbourne, Australia, 227–236. doi:10.1109/ICPC58990.2023.00037

Dror G. Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer. 2022. How Developers Choose Names. *IEEE Transactions on Software Engineering* 48, 1 (Jan. 2022), 37–52. doi:10.1109/TSE.2020.2976920

Remo Gresta, Vinicius Durelli, and Elder Cirilo. 2023. Naming Practices in Object-oriented Programming: An Empirical Study. *Journal of Software Engineering Research and Development* (Feb. 2023). doi:10.5753/jserd.2023.2582

S. Haiduc and A. Marcus. 2008. On the Use of Domain Terms in Source Code. In *2008 16th IEEE International Conference on Program Comprehension*. IEEE, Amsterdam, 113–122. doi:10.1109/ICPC.2008.29

Johannes Hofmeister, Janet Siegmund, and Daniel V Holt. 2019. Shorter Identifier Names Take Longer to Comprehend. (2019), 11.

Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. ACM Press, Hyderabad, India, 92–101. doi:10.1145/2597073.2597074

Sander Kamps, Bastiaan Heeren, and Johan Jeuring. 2020. Assessing the Quality of Evolving Haskell Systems by Measuring Structural Inequality. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. ACM, Virtual Event USA, 67–79. doi:10.1145/3406088.3409014

Dawn Lawrie, Henry Feild, and David Binkley. 2007. Quantifying Identifier Quality: An Analysis of Trends. *Empirical Software Engineering* 12, 4 (July 2007), 359–388. doi:10.1007/s10664-006-9032-2

D. Lawrie, C. Morrell, H. Feild, and D. Binkley. 2006. Whats in a Name? A Study of Identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, Athens, Greece, 3–12. doi:10.1109/ICPC.2006.51

Ben Liblit, Andrew Begel, and Eve Sweetser. 2006. Cognitive Perspectives on the Role of Naming in Computer Programs. (2006), 15.

Neil Mitchell and Colin Runciman. 2007. Uniform Boilerplate and List Processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop - Haskell '07*. ACM Press, Freiburg, Germany, 49. doi:10.1145/1291201.1291208

Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2016. *Curating GitHub for Engineered Software Projects*. Preprint. PeerJ Preprints. doi:10.7287/peerj.preprints.2617v1

Anthony Peruma. 2022. Understanding Digits in Identifier Names: An Exploratory Study. (2022).

Kai Petersen and Cigdem Gencel. 2013. Worldviews, Research Methods, and Their Relationship to Validity in Empirical Software Engineering Research. In *2013 Joint Conference of the 23rd International Workshop on Software Measurement and the 8th International Conference on Software Process and Product Measurement*. IEEE, Ankara, Turkey, 81–89. doi:10.1109/IWSM-Mensura.2013.22

M F Porter. 1980. An Algorithm for Suffix Stripping. (1980).

Giuseppe Scanniello and Michele Risi. 2013. Dealing with Faults in Source Code: Abbreviated vs. Full-Word Identifier Names. In *2013 IEEE International Conference on Software Maintenance*. IEEE, Eindhoven, Netherlands, 190–199. doi:10.1109/ICSM.2013.30

Andrea Schankin, Annika Berger, Daniel V. Holt, Johannes C. Hofmeister, Till Riedel, and Michael Beigl. 2018. Descriptive Compound Identifier Names Improve Source Code Comprehension. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, Gothenburg Sweden, 31–40. doi:10.1145/3196321.3196332

Forrest J. Shull, Jeffrey C. Carver, Sira Vegas, and Natalia Juristo. 2008. The Role of Replications in Empirical Software Engineering. *Empirical Software Engineering* 13, 2 (April 2008), 211–218. doi:10.1007/s10664-008-9060-1

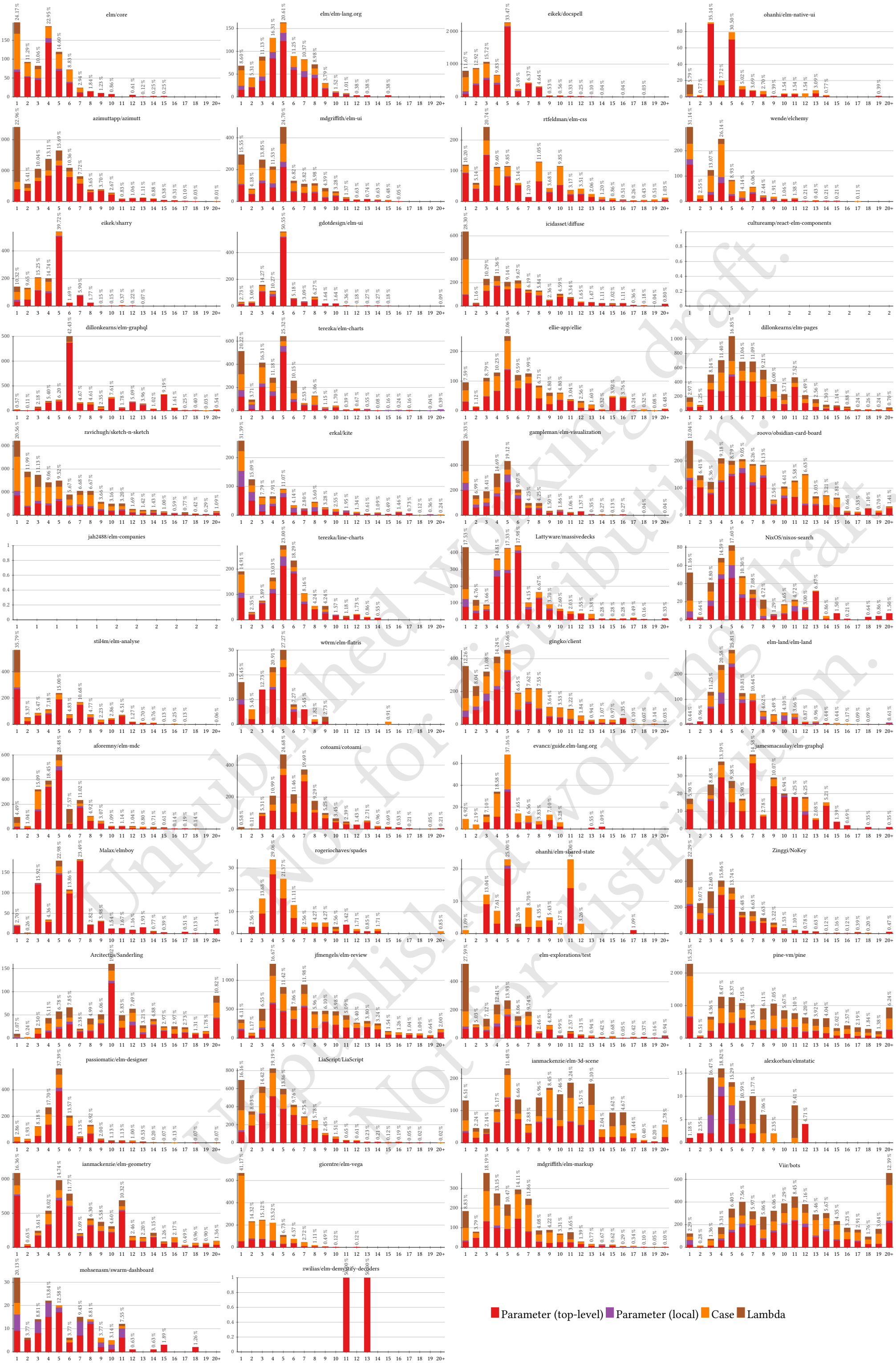
Janice Singer, Susan E. Sim, and Timothy C. Lethbridge. 2008. Software Engineering Data Collection for Field Studies. In *Guide to Advanced Empirical Software Engineering*, Forrest Shull, Janice Singer, and Dag I. K. Sjøberg (Eds.). Springer London, London, 9–34. doi:10.1007/978-1-84800-044-5_1

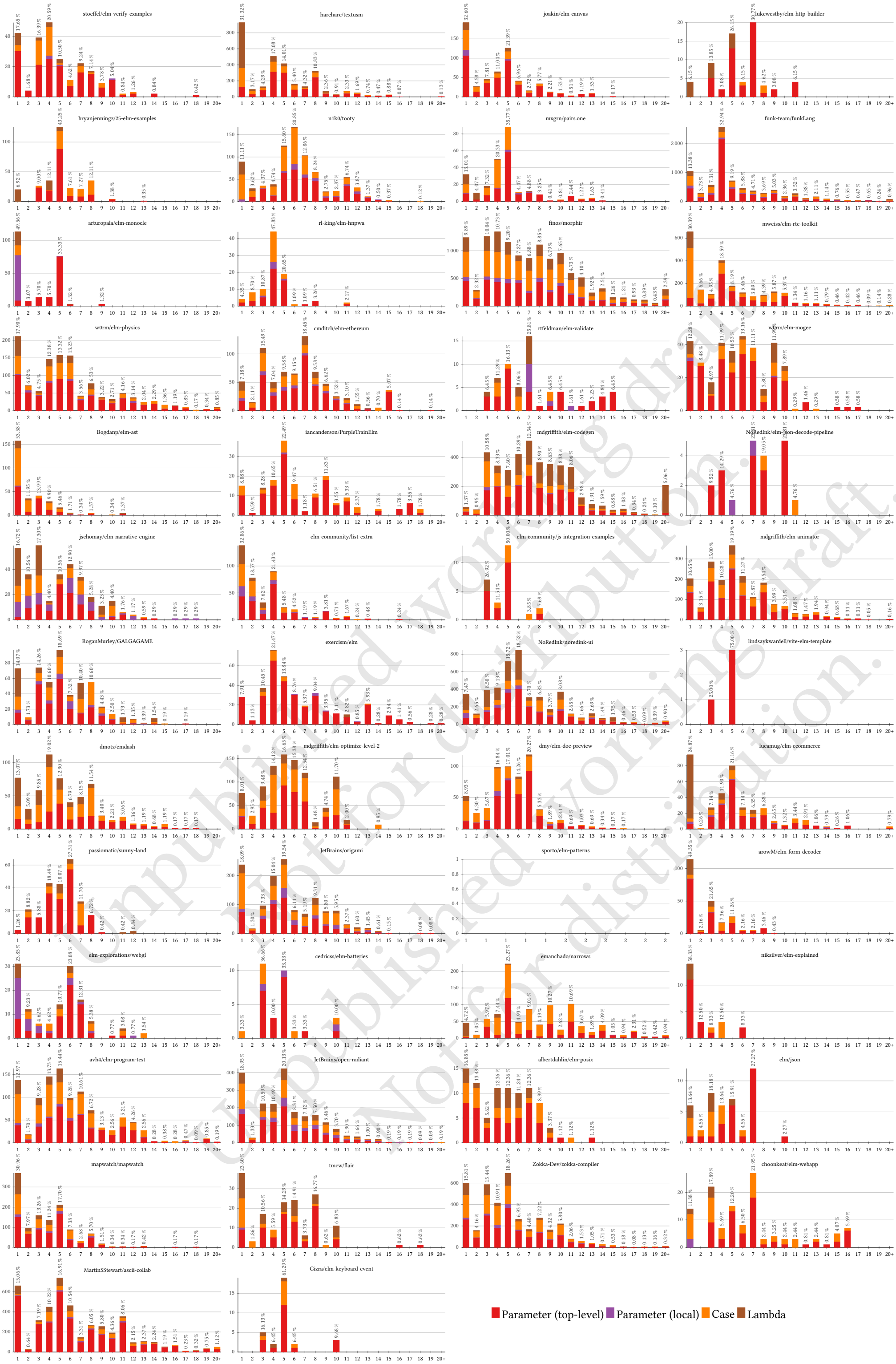
Alaaeddin Swidan, Alexander Serebrenik, and Felienne Hermans. 2017. How Do Scratch Programmers Name Variables and Procedures?. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, Shanghai, 51–60. doi:10.1109/SCAM.2017.12

## A Appendix

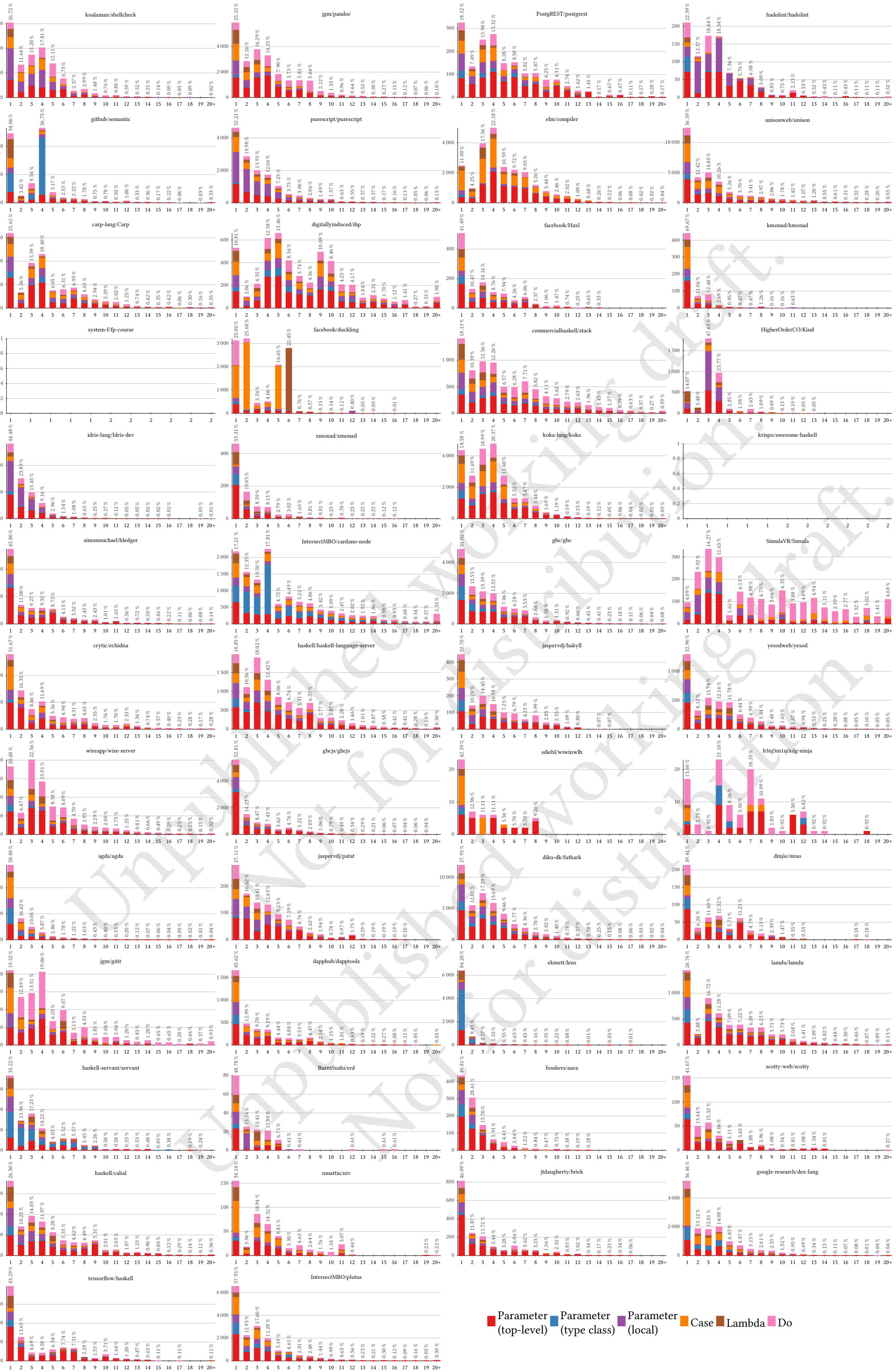
This appendix presents repository specific distributions of name length. In all cases the x-axis displays normalized length. In contrast to the diagrams in section 6, here they are not scaled to a common maximum.

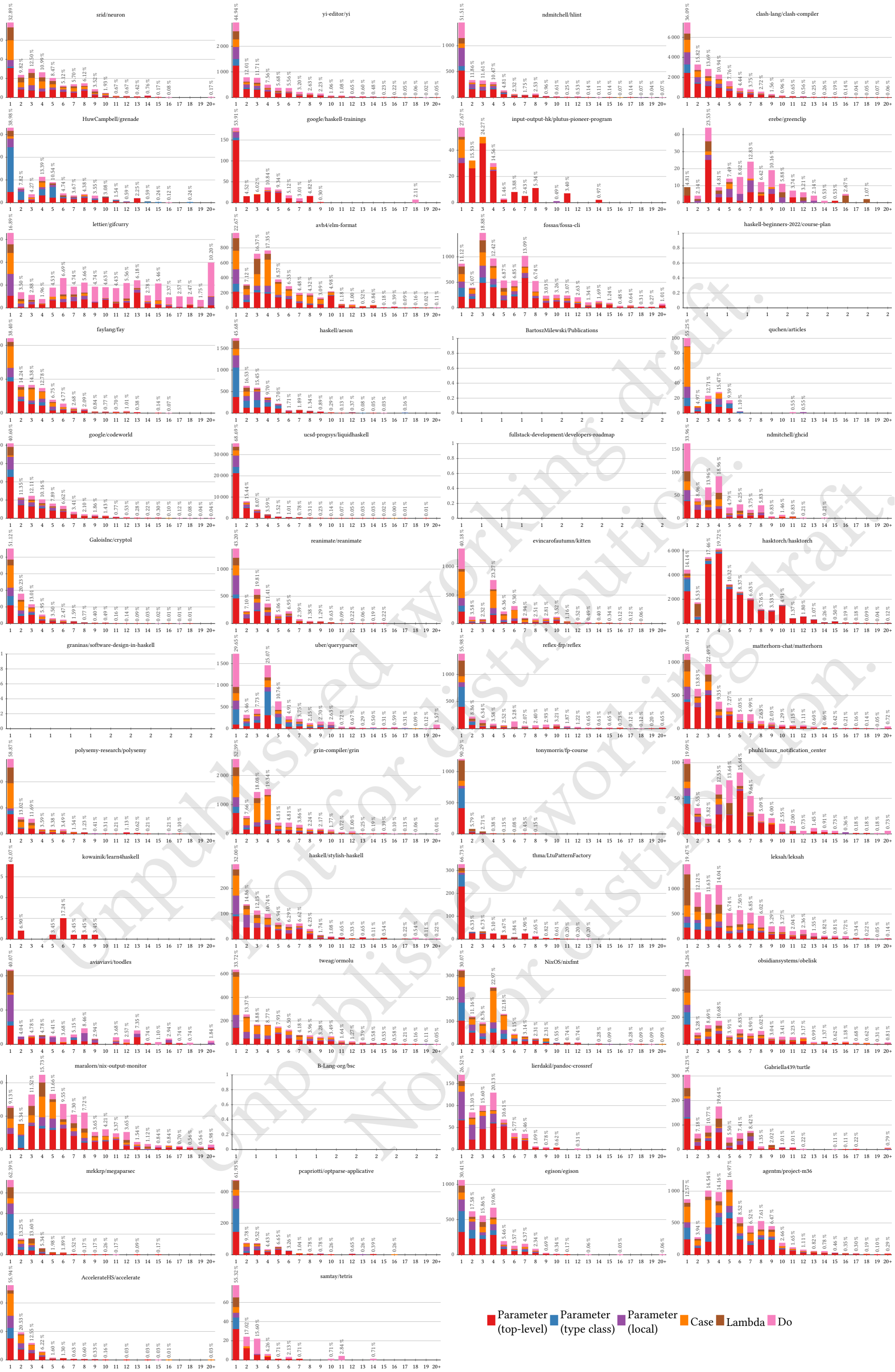
## A.1 Variable Name Length in Elm



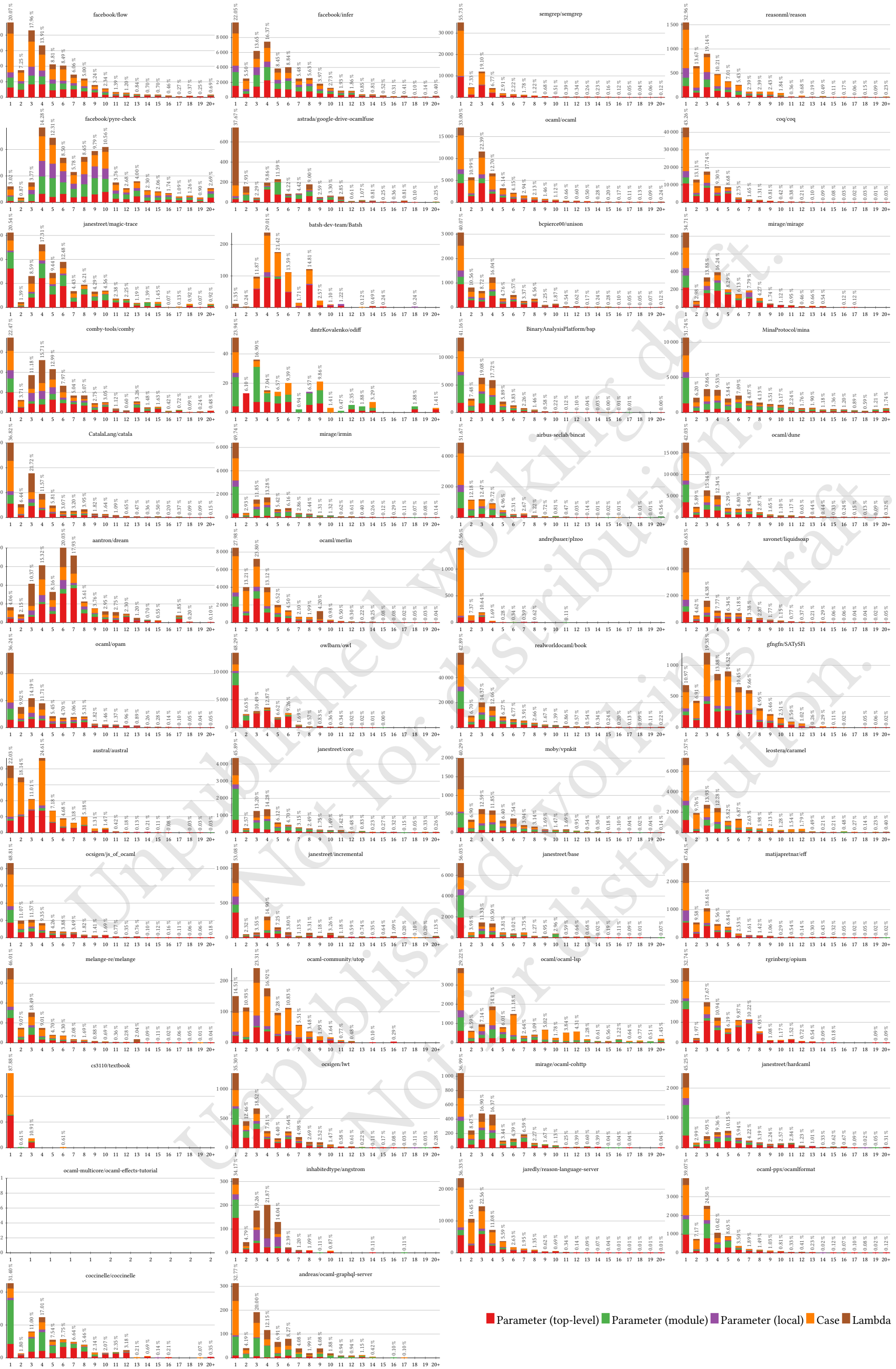

stoeffel/elm-verify-examples
harehare/textusm
joakin/elm-canvas
lukewestby/elm-http-builder
bryanjenningz/25-elm-examples
n1k0/tooty
mxgrn/pairs.one
funk-team/funkLang
arturopala/elm-monocle
rl-king/elm-hnpwa
finos/morphir
mweiss/elm-rte-toolkit
w0rm/elm-physics
cmditch/elm-ethereum
rtfeldman/elm-validate
w0rm/elm-mogee
Bogdanp/elm-ast
iancanderson/PurpleTrainElm
mdgriffith/elm-codegen
NoRedInk/elm-json-decode-pipeline
jschomay/elm-narrative-engine
elm-community/list-extra
elm-community/js-integration-examples
mdgriffith/elm-animator
RoganMurley/GALGAGAME
exercism/elm
NoRedInk/noredink-ui
lindsaykwardell/vite-elm-template
dmotz/emdash
mdgriffith/elm-optimize-level-2
dmy/elm-doc-preview
lucamug/elm-ecommerce
passiomatic/sunny-land
JetBrains/origami
sporto/elm-patterns
arowM/elm-form-decoder
elm-explorations/webgl
cedricss/elm-batteries
emanchado/narrows
niksilver/elm-explained
avh4/elm-program-test
JetBrains/open-radiant
albertdahlin/elm-posix
elm/json
mapwatch/mapwatch
tmcw/flair
Zokka-Dev/zokka-compiler
choonkeat/elm-webapp
MartinSStewart/ascii-collab
Gizra/elm-keyboard-event
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20+
🟥 Parameter (top-level) 🟪 Parameter (local) 🟧 Case 🟫 Lambda

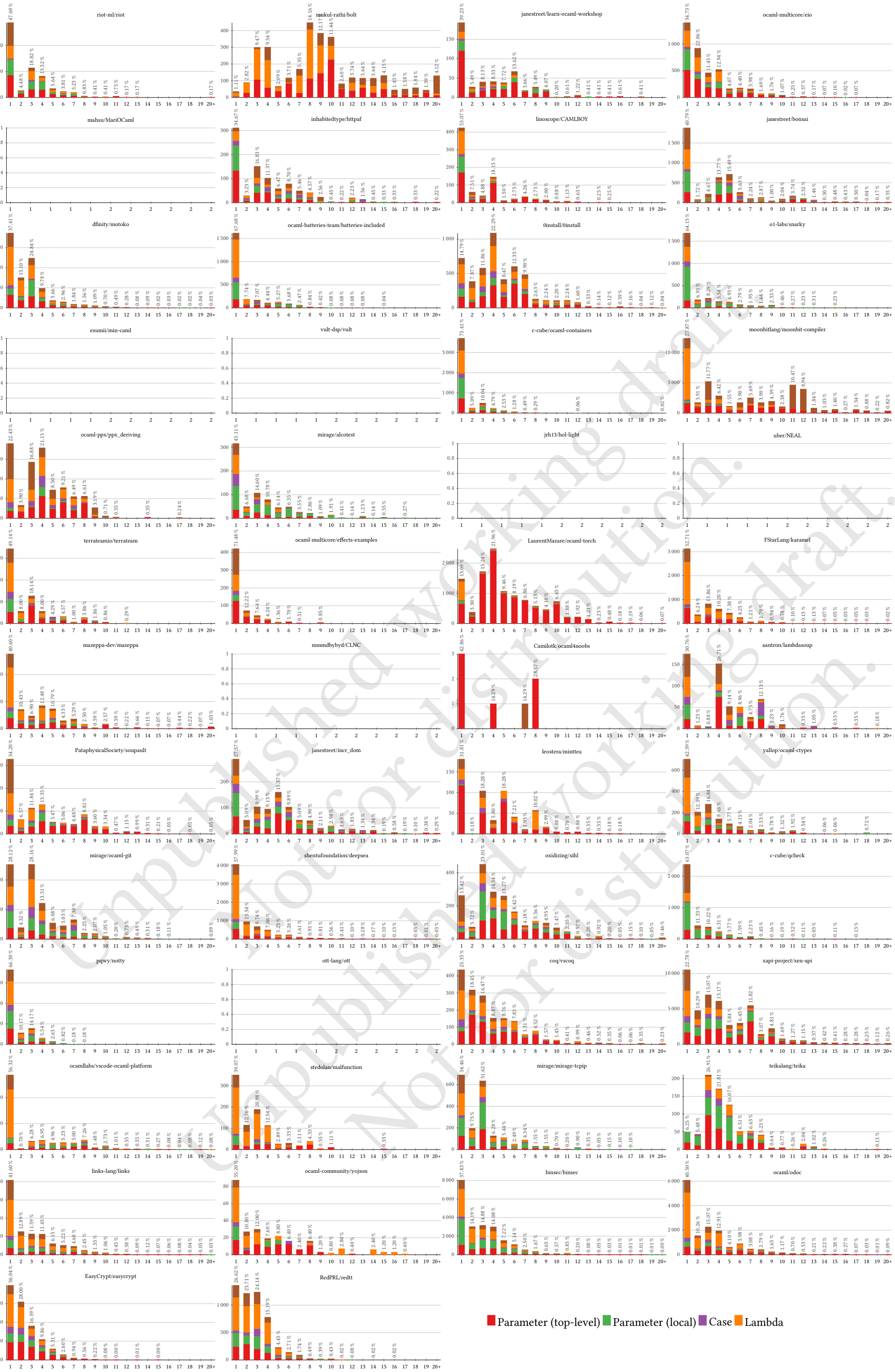
## A.2 Variable Name Length in Haskell



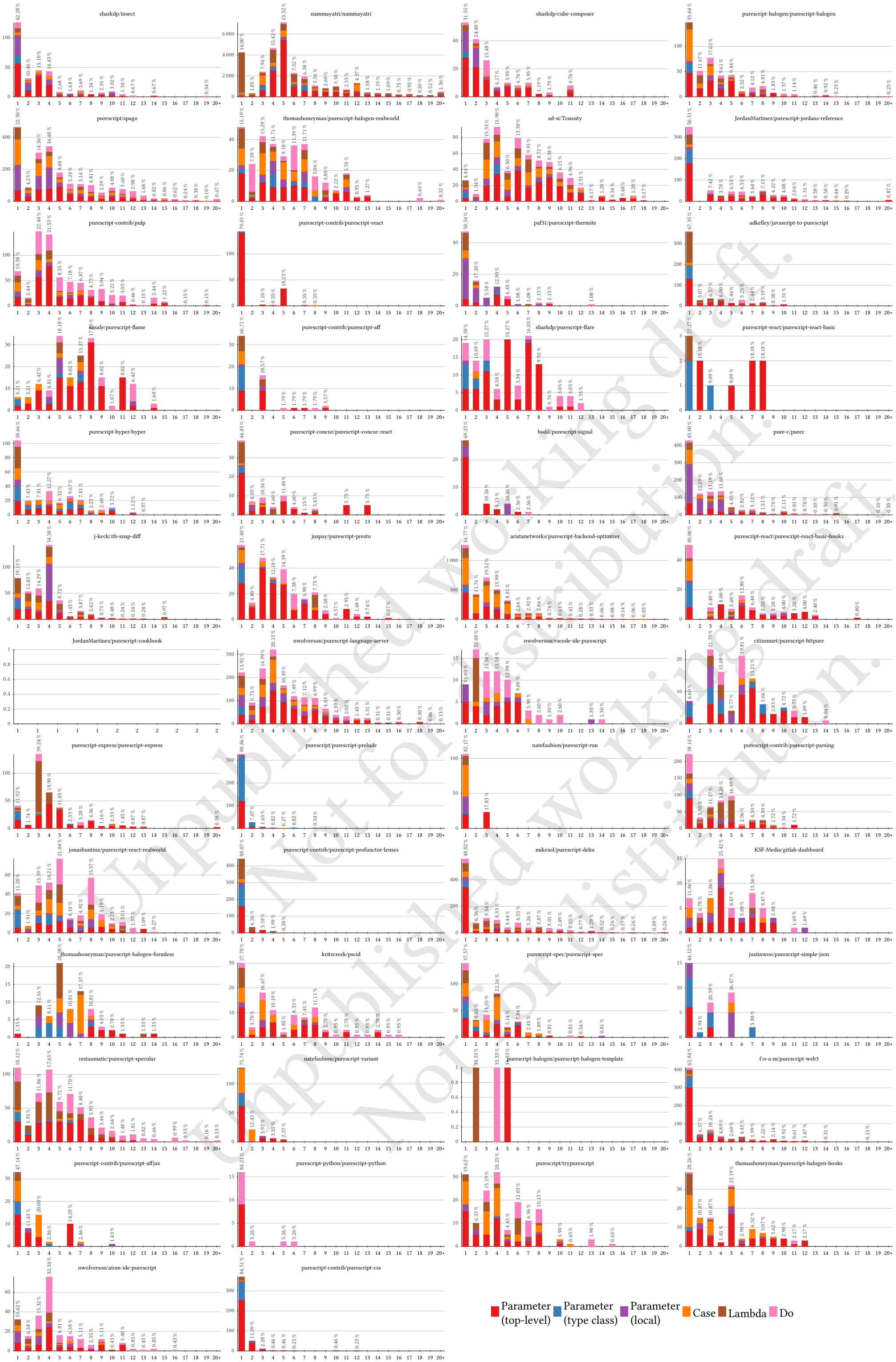

srid/neuron
yi-editor/yi
ndmitchell/hlint
clash-lang/clash-compiler
HuwCampbell/grenade
google/haskell-trainings
input-output-hk/plutus-pioneer-program
erebe/greenclip
lettier/gifcurry
avh4/elm-format
fossas/fossa-cli
haskell-beginners-2022/course-plan
faylang/fay
haskell/aeson
BartoszMilewski/Publications
quchen/articles
google/codeworld
ucsd-progsys/liquidhaskell
fullstack-development/developers-roadmap
ndmitchell/ghcid
GaloisInc/cryptol
reanimate/reanimate
evincarofautumn/kitten
hasktorch/hasktorch
graninas/software-design-in-haskell
uber/queryparser
reflex-frp/reflex
matterhorn-chat/matterhorn
polysemy-research/polysemy
grin-compiler/grin
tonymorris/fp-course
phuhl/linux_notification_center
kowainik/learn4haskell
haskell/stylish-haskell
thma/LtuPatternFactory
leksah/leksah
aviaviavi/toodles
tweag/ormolu
NixOS/nixfmt
obsidiansystems/obelisk
maralorn/nix-output-monitor
B-Lang-org/bsc
lierdakil/pandoc-crossref
Gabriella439/turtle
mrkkrp/megaparsec
pcapriotti/optparse-applicative
egison/egison
agentm/project-m36
AccelerateHS/accelerate
samtay/tetris
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20+
Parameter (top-level)
Parameter (type class)
Parameter (local)
Case
Lambda
Do

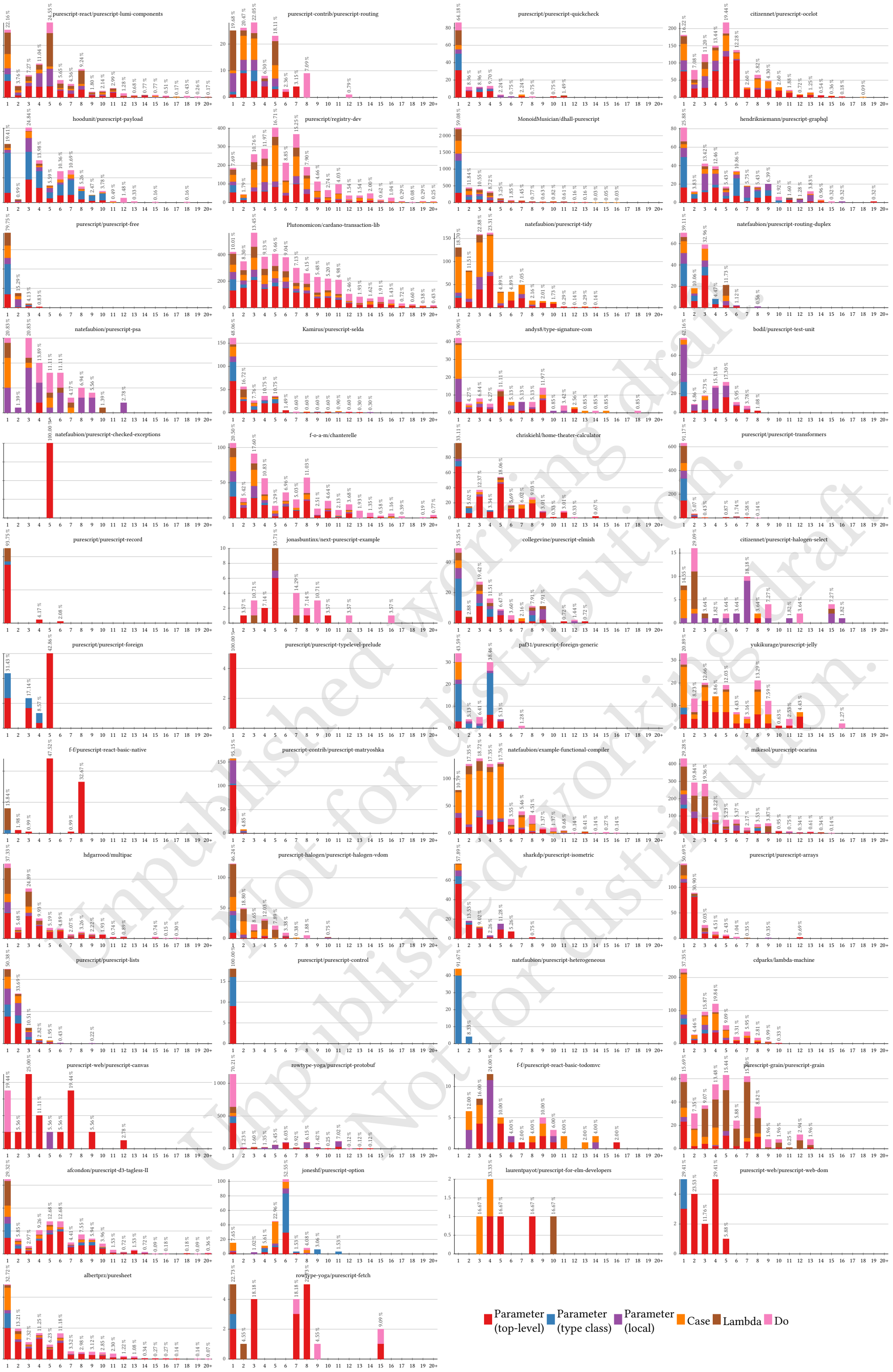
## A.3 Variable Name Length in OCaml



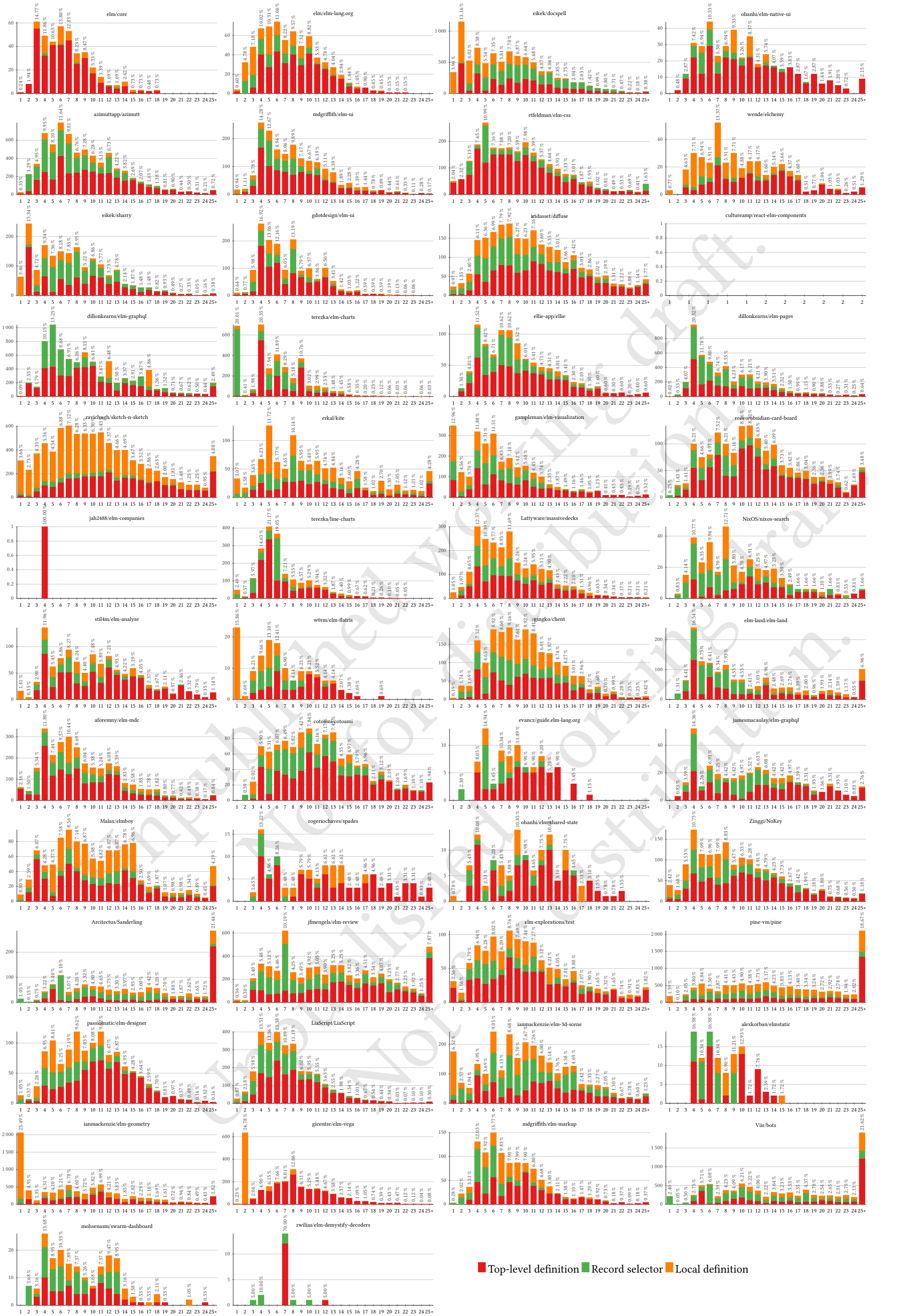

riot-ml/riot
mukul-rathi/bolt
janestreet/learn-ocaml-workshop
ocaml-multicore/eio
mahsu/MariOCaml
inhabitedtype/httpaf
linoscope/CAMLBOY
janestreet/bonsai
dfinity/motoko
ocaml-batteries-team/batteries-included
0install/0install
o1-labs/snarky
esumii/min-caml
vult-dsp/vult
c-cube/ocaml-containers
moonbitlang/moonbit-compiler
ocaml-ppx/ppx_deriving
mirage/alcotest
jrh13/hol-light
uber/NEAL
terrateamio/terrateam
ocaml-multicore/effects-examples
LaurentMazare/ocaml-torch
FStarLang/karamel
mazeppa-dev/mazeppa
mmmdbybyd/CLNC
Camilotk/ocaml4noobs
aantron/lambdasoup
PataphysicalSociety/soupault
janestreet/incr_dom
leostera/minttea
yallop/ocaml-ctypes
mirage/ocaml-git
shentufoundation/deepsea
oxidizing/sihl
c-cube/qcheck
pqwy/notty
ott-lang/ott
coq/vscoq
xapi-project/xen-api
ocamllabs/vscode-ocaml-platform
stedolan/malfunction
mirage/mirage-tcpip
teikalang/teika
links-lang/links
ocaml-community/yojson
binsec/binsec
ocaml/odoc
EasyCrypt/easycrypt
RedPRL/redtt
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20+
🟥 Parameter (top-level) 🟩 Parameter (local) 🟪 Case 🟧 Lambda

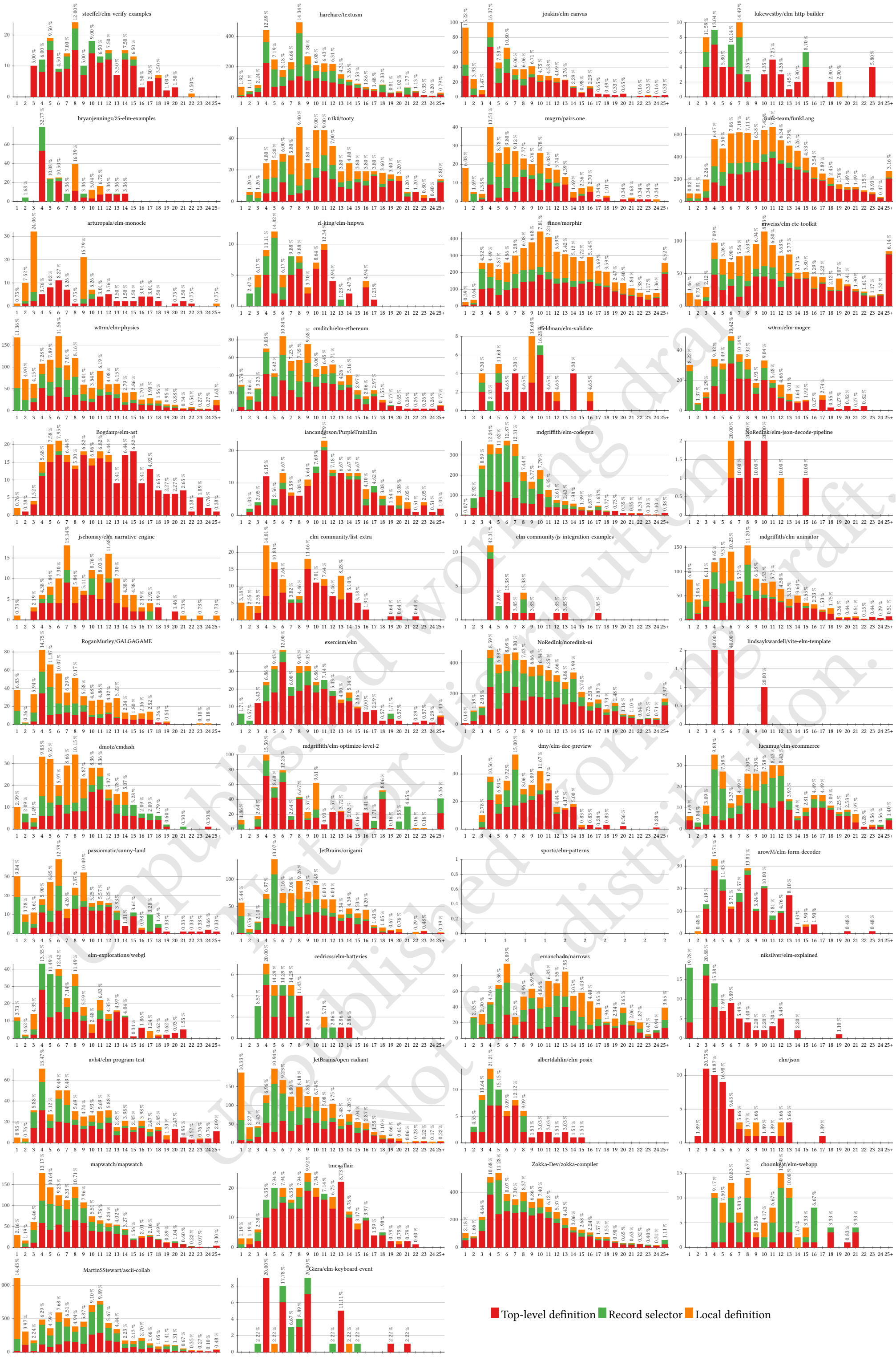
## A.4 Variable Name Length in PureScript



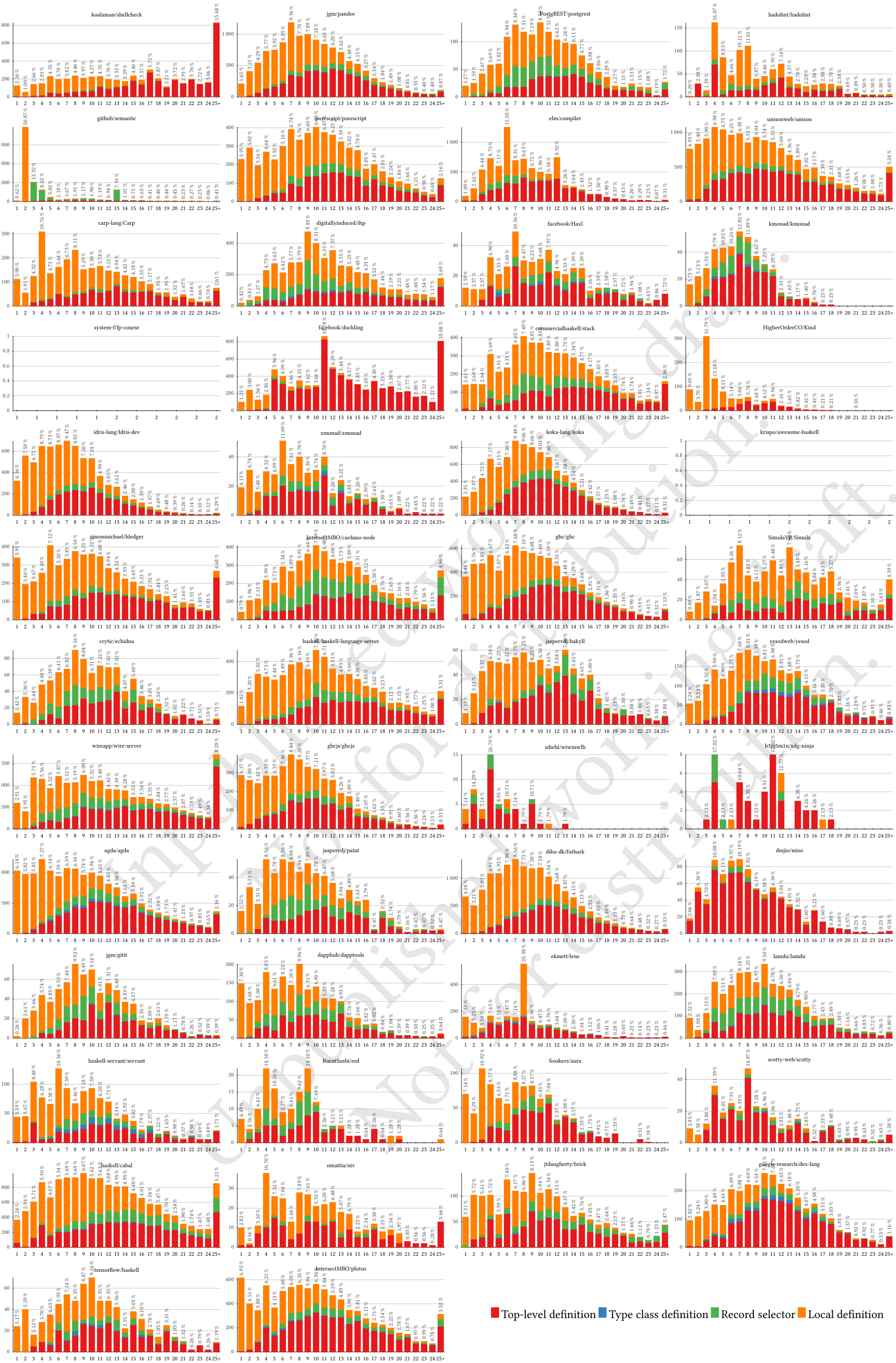

purescript-react/purescript-lumi-components
purescript-contrib/purescript-routing
purescript/purescript-quickcheck
citizennet/purescript-ocelot
hoodunit/purescript-payload
purescript/registry-dev
MonoidMusician/dhall-purescript
hendrikniemann/purescript-graphql
purescript/purescript-free
Plutonomicon/cardano-transaction-lib
natefaubion/purescript-tidy
natefaubion/purescript-routing-duplex
natefaubion/purescript-psa
Kamirus/purescript-selda
andys8/type-signature-com
bodil/purescript-test-unit
natefaubion/purescript-checked-exceptions
f-o-a-m/chanterelle
chriskiehl/home-theater-calculator
purescript/purescript-transformers
purescript/purescript-record
jonasbuntinx/next-purescript-example
collegevine/purescript-elmish
citizennet/purescript-halogen-select
purescript/purescript-foreign
purescript/purescript-typelevel-prelude
paf31/purescript-foreign-generic
yukikurage/purescript-jelly
f-f/purescript-react-basic-native
purescript-contrib/purescript-matryoshka
natefaubion/example-functional-compiler
mikesol/purescript-ocarina
hdgarrood/multipac
purescript-halogen/purescript-halogen-vdom
sharkdp/purescript-isometric
purescript/purescript-arrays
purescript/purescript-lists
purescript/purescript-control
natefaubion/purescript-heterogeneous
cdparks/lambda-machine
purescript-web/purescript-canvas
rowtype-yoga/purescript-protobuf
f-f/purescript-react-basic-todomvc
purescript-grain/purescript-grain
afcondon/purescript-d3-tagless-II
joneshf/purescript-option
laurentpayot/purescript-for-elm-developers
purescript-web/purescript-web-dom
albertprz/puresheet
rowtype-yoga/purescript-fetch
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20+
Parameter (top-level)
Parameter (type class)
Parameter (local)
Case
Lambda
Do

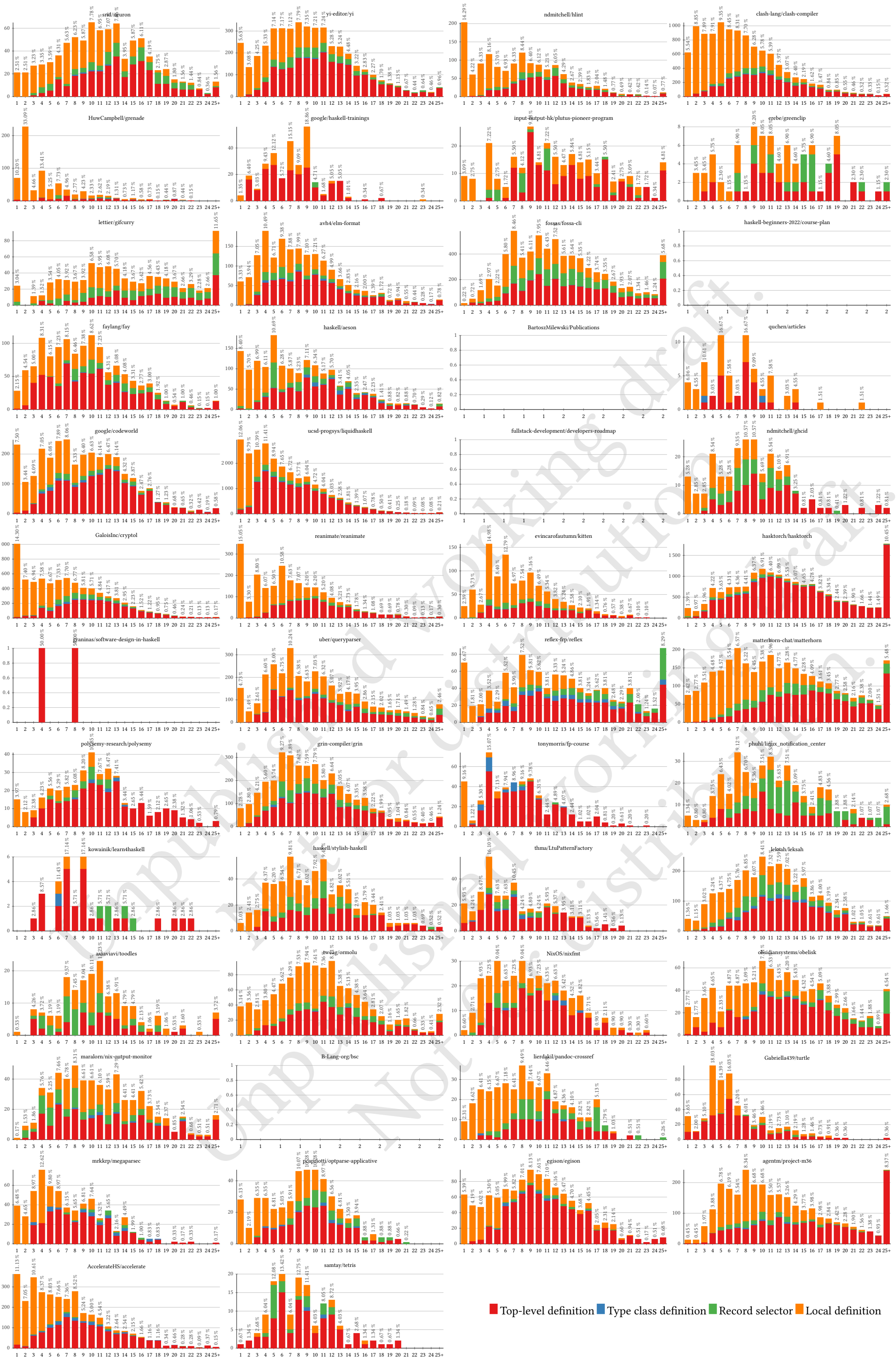
## A.5 Definition Name Length in Elm



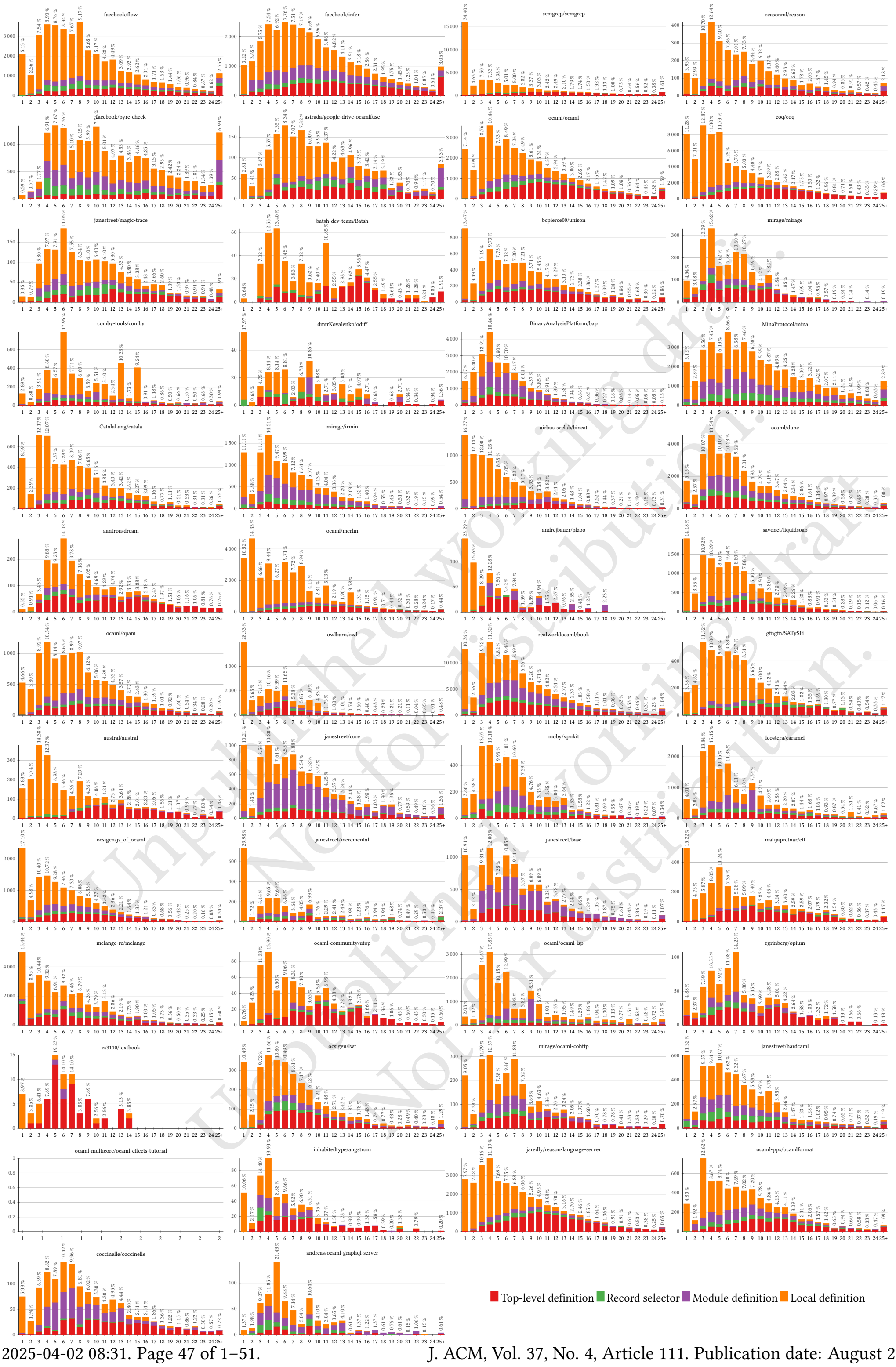

stoeffel/elm-verify-examples
harehare/textusm
joakin/elm-canvas
lukewestby/elm-http-builder
bryanjenningz/25-elm-examples
n1k0/tooty
mxgrn/pairs.one
funk-team/funkLang
arturopala/elm-monocle
rl-king/elm-hnpwa
finos/morphir
mweiss/elm-rte-toolkit
w0rm/elm-physics
cmditch/elm-ethereum
rtfeldman/elm-validate
w0rm/elm-mogee
Bogdanp/elm-ast
iancanderson/PurpleTrainElm
mdgriffith/elm-codegen
NoRedInk/elm-json-decode-pipeline
jschomay/elm-narrative-engine
elm-community/list-extra
elm-community/js-integration-examples
mdgriffith/elm-animator
RoganMurley/GALGAGAME
exercism/elm
NoRedInk/noredink-ui
lindsaykwardell/vite-elm-template
dmotz/emdash
mdgriffith/elm-optimize-level-2
dmy/elm-doc-preview
lucamug/elm-ecommerce
passiomatic/sunny-land
JetBrains/origami
sporto/elm-patterns
arowM/elm-form-decoder
elm-explorations/webgl
cedricss/elm-batteries
emanchado/narrows
niksilver/elm-explained
avh4/elm-program-test
JetBrains/open-radiant
albertdahlin/elm-posix
elm/json
mapwatch/mapwatch
tmcw/flair
Zokka-Dev/zokka-compiler
choonkeat/elm-webapp
MartinSStewart/ascii-collab
Gizra/elm-keyboard-event
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25+
🟥 Top-level definition
🟩 Record selector
🟧 Local definition

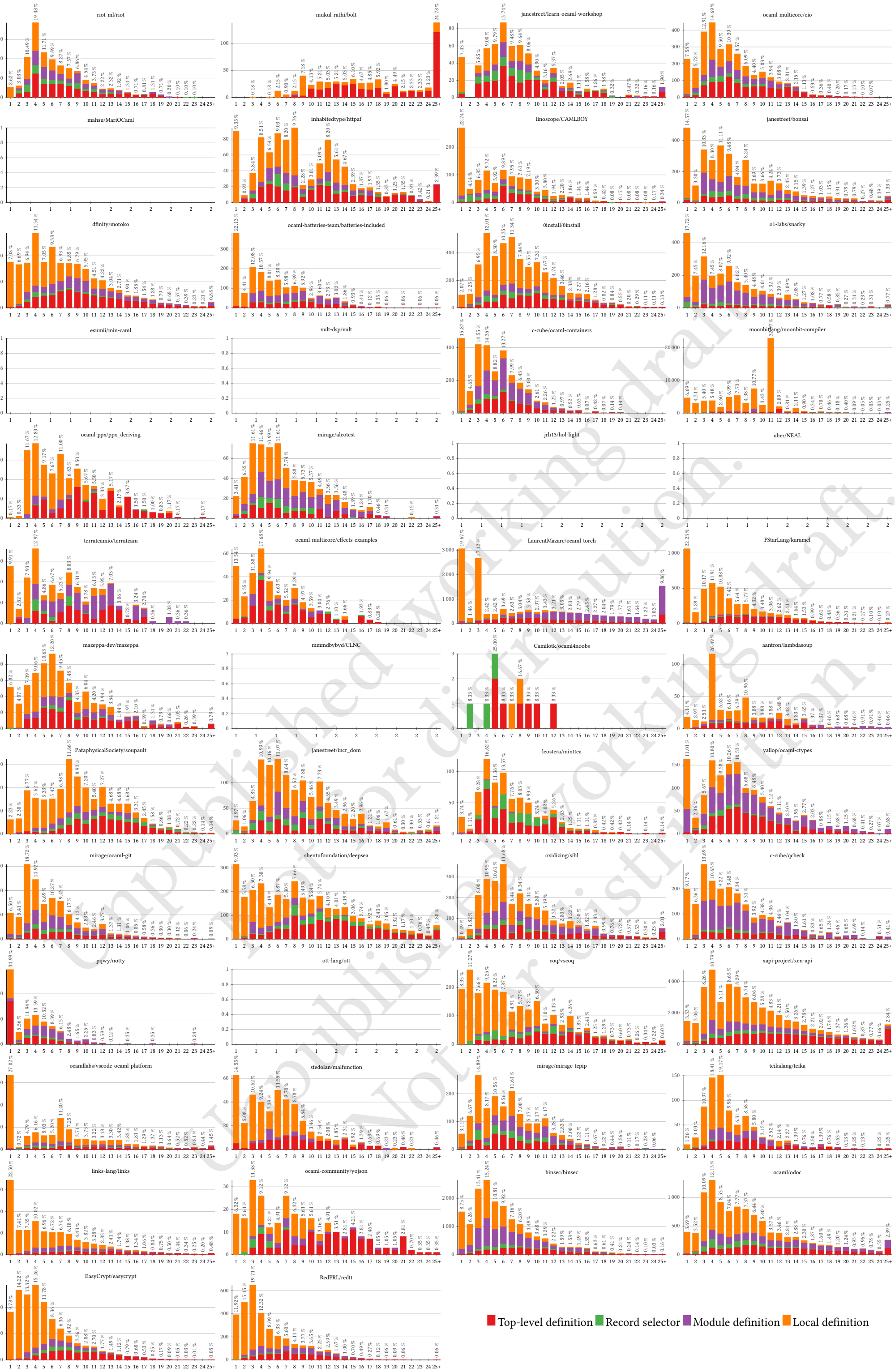
## A.6 Definition Name Length in Haskell



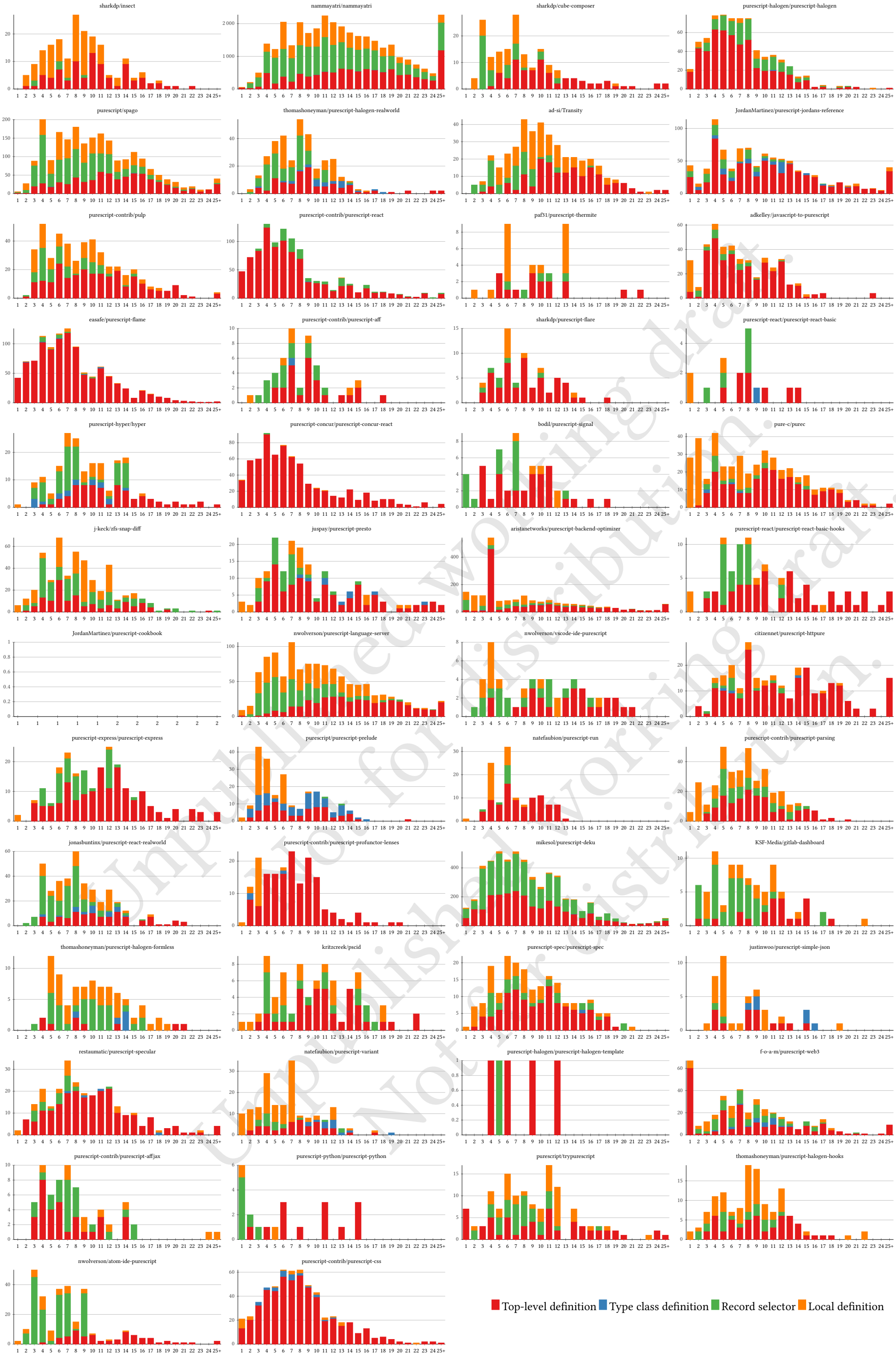

srid/neuron
yi-editor/yi
ndmitchell/hlint
clash-lang/clash-compiler
HuwCampbell/grenade
google/haskell-trainings
input-output-hk/plutus-pioneer-program
erebe/greenclip
lettier/gifcurry
avh4/elm-format
fossas/fossa-cli
haskell-beginners-2022/course-plan
faylang/fay
haskell/aeson
BartoszMilewski/Publications
quchen/articles
google/codeworld
ucsd-progsys/liquidhaskell
fullstack-development/developers-roadmap
ndmitchell/ghcid
GaloisInc/cryptol
reanimate/reanimate
evincarofautumn/kitten
hasktorch/hasktorch
graninas/software-design-in-haskell
uber/queryparser
reflex-frp/reflex
matterhorn-chat/matterhorn
polysemy-research/polysemy
grin-compiler/grin
tonymorris/fp-course
phuhl/linux_notification_center
kowainik/learn4haskell
haskell/stylish-haskell
thma/LtuPatternFactory
leksah/leksah
aviaviavi/toodles
tweag/ormolu
NixOS/nixfmt
obsidiansystems/obelisk
maralorn/nix-output-monitor
B-Lang-org/bsc
lierdakil/pandoc-crossref
Gabriella439/turtle
mrkkrp/megaparsec
pcapriotti/optparse-applicative
egison/egison
agentm/project-m36
AccelerateHS/accelerate
samtay/tetris
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25+
🟥 Top-level definition
🟦 Type class definition
🟩 Record selector
🟧 Local definition

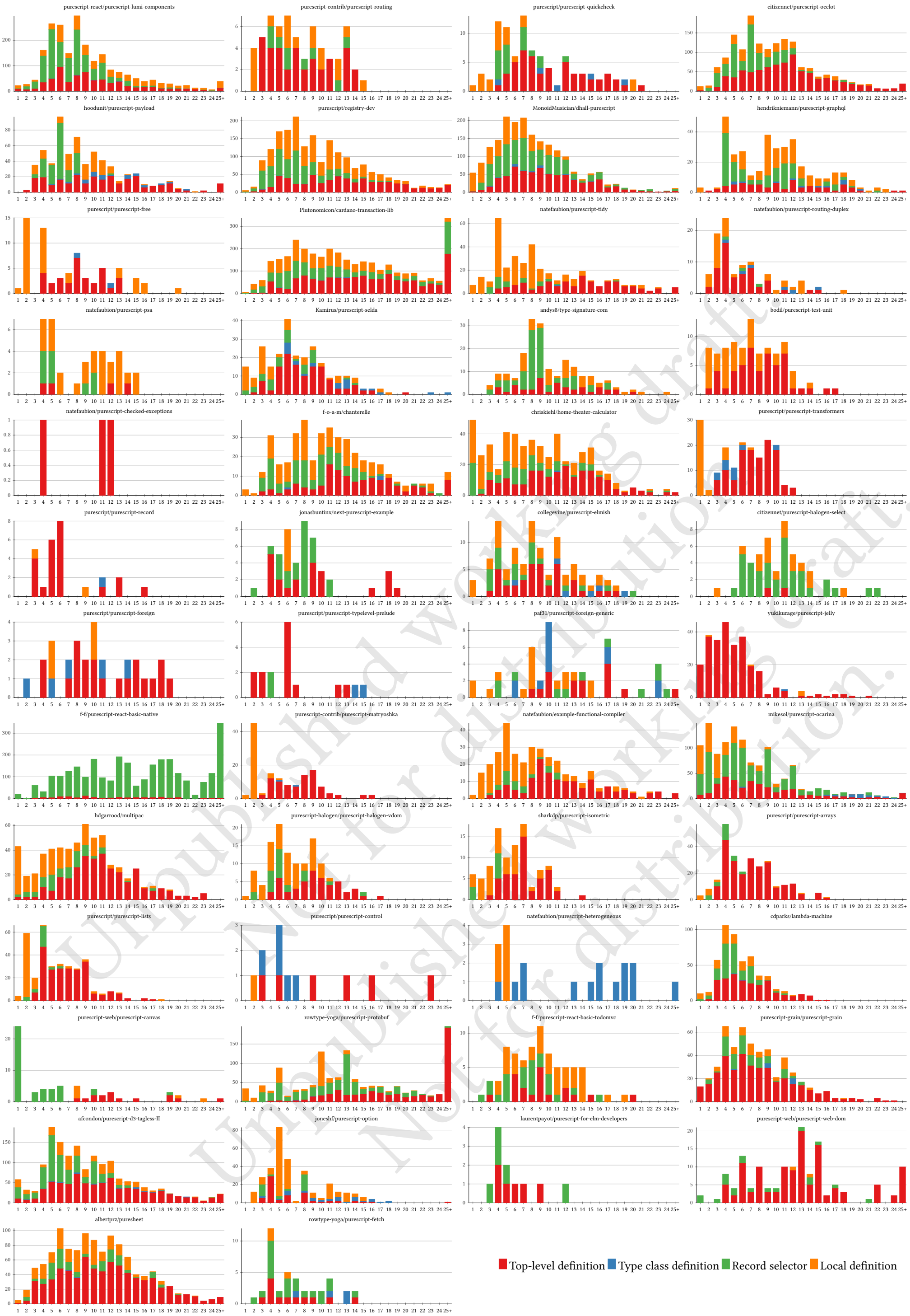
## A.7 Definition Name Length in OCaml




riot-ml/riot
mukul-rathi/bolt
janestreet/learn-ocaml-workshop
ocaml-multicore/eio
mahsu/MariOCaml
inhabitedtype/httpaf
linoscope/CAMLBOY
janestreet/bonsai
dfinity/motoko
ocaml-batteries-team/batteries-included
0install/0install
o1-labs/snarky
esumii/min-caml
vult-dsp/vult
c-cube/ocaml-containers
moonbitlang/moonbit-compiler
ocaml-ppx/ppx_deriving
mirage/alcotest
jrh13/hol-light
uber/NEAL
terrateamio/terrateam
ocaml-multicore/effects-examples
LaurentMazare/ocaml-torch
FStarLang/karamel
mazeppa-dev/mazeppa
mmmdbybyd/CLNC
Camilotk/ocaml4noobs
aantron/lambdasoup
PataphysicalSociety/soupault
janestreet/incr_dom
leostera/minttea
yallop/ocaml-ctypes
mirage/ocaml-git
shentufoundation/deepsea
oxidizing/sihl
c-cube/qcheck
pqwy/notty
ott-lang/ott
coq/vscoq
xapi-project/xen-api
ocamllabs/vscode-ocaml-platform
stedolan/malfunction
mirage/mirage-tcpip
teikalang/teika
links-lang/links
ocaml-community/yojson
binsec/binsec
ocaml/odoc
EasyCrypt/easycrypt
RedPRL/redtt
🟥 Top-level definition 🟩 Record selector 🟪 Module definition 🟧 Local definition

## A.8 Definition Name Length in PureScript





Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009