Ayudante: Identifying Undesired Variable Interactions

Irfan Ul Haq Juan Caballero
IMDEA Software Institute, Spain
{irfanul.haq, juan.caballero}@imdea.org

Michael D. Ernst
University of Washington, USA
mernst@cs.washington.edu

Abstract

A common programming mistake is for incompatible variables to interact, e.g., storing euros in a variable that should hold dollars, or using an array index with the wrong array. This paper proposes a novel approach for identifying undesired interactions between program variables. Our approach uses two different mechanisms to identify related variables. Natural language processing (NLP) identifies variables with related names that may have related semantics. Abstract type inference (ATI) identifies variables that interact with each other. Any discrepancies between these two mechanisms may indicate a programming error.

We have implemented our approach in a tool called Ayudante. We evaluated Ayudante using two open-source programs: the Exim mail server and grep. Although these programs have been extensively tested and in deployment for years, Ayudante's first report for grep revealed a programming mistake.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords Undesired variable interactions, NLP, natural language processing, testing

1. Introduction

Sometimes programmers incorrectly use variables together that should not interact. For example, a programmer may store euros in a variable that should hold dollars, or add distance variables in miles and kilometers, or accidentally use a file descriptor as an index into an array.

Identifying such *undesired variable interactions* is the focus of this work. If two variables have different types, the compiler can catch undesired interactions between them statically through type checking. The compiler finds a bug by identifying a discrepancy between a variable specification (the programmer-written type) and how the variable is used. However, the compiler cannot detect all undesired interactions between variables. Sometimes a programmer uses the same type for variables with different semantics, e.g., integers holding costs in different currencies, or distances in different metrics, or regular expressions that apply to different string data.

Programmers often choose variable names that indicate the intended semantics of the variable. For example, a programmer may define integer variables `totalMiles`, `remainingKms`, and `dollars`; although these may have the same programming-language type (`int`), the programmer thinks of them as distinct subtypes of integer. The intuition behind our work is that we can use variable names as a type of specification; any mismatch between that specification and how variables are used may indicate a program defect.

This work proposes a novel approach for identifying undesired variable interactions. At a high level, our approach uses two different clusterings to identify related variables: (1) natural language processing (NLP) clusters variables with related names and therefore possibly related semantics, and (2) abstract type inference (ATI) [8, 16, 20] clusters variables that interact with each other during a program's execution. A discrepancy between the two clusterings may indicate incorrect source code, poorly-named variables, or both. For example, consider an ATI cluster of variables that interact with one another by means of `+`, `=`, `==`, and `<` operators. It is suspicious if such a cluster contains variable names with unrelated semantics.

A direct implementation of the above intuition would yield poor results; for example, it would not indicate which suspicious clusters are most worthy of a user's attention. Our tool, Ayudante, operates as follows. First, Ayudante clusters variables according to ATI; each cluster contains variables that interact during the program execution. Next, Ayudante measures the semantic cohesiveness of variable names within each cluster. It does so using a function we designed for measuring the natural-language similarity between two variable names. Finally, Ayudante outputs the clusters with the least-coherent variable names and also shows the location in the code where these semantically-dissimilar variable names interact.

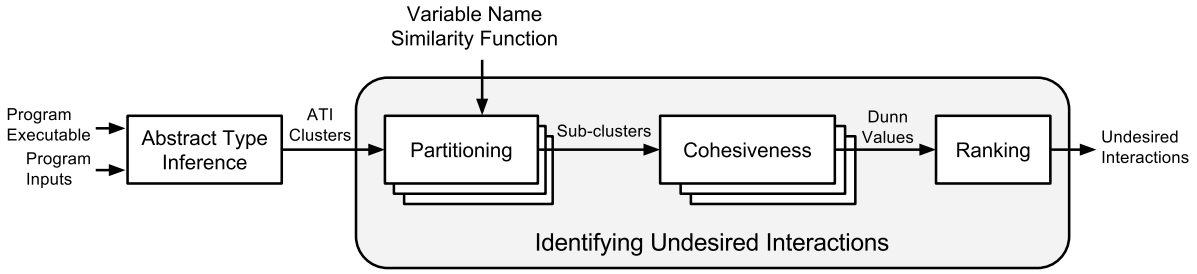

Figure 1: Ayudante architecture. Section 4 defines the variable name similarity function, and Section 5 explains the "Identifying Undesired Interactions" part of the figure.

We evaluated Ayudante using two popular open-source programs: the Exim mail server and grep. For both programs, we analyzed the top five ATI clusters reported by Ayudante. Although these programs have been heavily tested and have been in deployment for years, we found an undesired variable interaction in grep (see Section 7.2).

This paper makes the following contributions:

- We proposed a novel approach for automatically identifying undesired variable interactions by combining natural language processing (NLP) and abstract type inference (ATI) techniques.
- We designed a technique to identify incompatible variables based on the semantics embedded in their variable names. Applying this technique on clusters of variables that interact with each other enables detecting undesired variable interactions.
- We implemented Ayudante, a tool that implements our approach, and evaluated Ayudante on the Exim and grep programs, finding one undesired interaction in grep.

The remainder of the paper is structured as follows. Section 2 overviews our approach. Section 3 reviews abstract type inference. Section 4 defines our variable name similarity function. Section 5 details how our approach combines ATI and NLP. Section 6 provides implementation details, and Section 7 evaluates our approach. Sections 8 and 9 present related work and conclude.

2. Approach Overview

Figure 1 gives the architecture of Ayudante. Ayudante is comprised of the following steps:

1. The ATI module groups variables that interact with each other into ATI clusters. ATI can be computed statically [5, 16] or dynamically [8]. While our approach is independent of the ATI technique used, the Ayudante implementation uses dynamic ATI due to higher precision. Ayudante runs the program on user-supplied inputs, such as a test suite, and uses dynamic information flow techniques to compute ATI clusters.
2. Ayudante partitions each ATI cluster into 2 sub-clusters, using a novel distance metric based on variable name semantic similarity described in Section 4. The sub-clusters are chosen to maximize the following properties: within a sub-cluster, all the variable names are semantically similar, and between sub-clusters, the variable names are semantically different.
3. Ayudante computes the separation of the sub-clusters obtained in the partitioning step. It uses this separation as a measurement of the cohesiveness of each ATI cluster. ATI clusters with well-separated sub-clusters have low cohesiveness in terms of name semantics. They are more likely to contain an undesired interaction.
4. Ayudante sorts ATI clusters based on cohesiveness and selects those clusters with cohesiveness less than a threshold. For variables that directly interact, Ayudante also outputs the code location for the interaction. Recall that within an ATI cluster, every pair of variables (transitively) interacts during program execution.

A user runs Ayudante, reads its reports, investigates the reported suspicious variable interactions, and for each report decides whether to change the source code logic and/or the variable names.

3. Abstract Type Inference

A program's declared types only partially capture the similarities among variable values. For example, a programmer may use the `int` type to represent array indices, sensor measurements, the current time, file descriptors, etc. Abstract type inference (ATI) aims to infer these finer-grained groupings of variables. ATI can be performed statically [5, 16], but in order to achieve higher precision, Ayudante performs ATI dynamically by running the DynComp tool [8].

At run time, DynComp assigns each value a unique abstract type. If two values interact, then DynComp merges their abstract types. For example, if the program executes `x+y` or `x<y`, then the programmer considered `x` and `y` to have the same abstract type. Ayudante uses DynComp's default settings, in which all arithmetic and bitwise operators are considered interactions among the operands and the result.

4. Variable Name Similarity

This section describes our variable name similarity function $varsim(v_1, v_2, d_l, d_a, s_e) \rightarrow [0, 1]$ that outputs a value between 0 and 1 capturing the semantic similarity between variables names v_1 and v_2, where 1 represents completely similar and 0 represents completely dissimilar. In addition to the two variable names, it takes as input: a *language dictionary* (d_l), e.g., an English dictionary; an *abbreviation dictionary* (d_a) that contains common abbreviations used in programming, e.g., *acc* for accumulator or *arr* for array; and a *stoplist* (s_e) that contains frequent words that are unlikely to contribute to a variable name's semantics, e.g., prepositions and conjunctions such as *from* and *where*. We use WordNet [15] as our language dictionary because its definitions feature multiple senses (meanings) for most words. For brevity, we may refer to the variable similarity function as $varsim(v_1, v_2)$ removing the other parameters. The variable name similarity computation is comprised of 3 phases.

1. Tokenize each variable name into a sequence of *dictionary words*, i.e., words appearing in the language dictionary (Section 4.1).
2. Compute the word similarity $wordsim(w_i, w_j)$ of all pairs of dictionary words w_i, w_j where w_i belongs to v_1 and w_j belongs to v_2 (Section 4.2). The word similarity function uses WordNet [15] to extract the meanings of each dictionary word and computes the semantic similarity between those meanings.
3. Compute the variable name similarity $varsim(v_1, v_2)$ from the word similarity results (Section 4.3).

Throughout this section we use as running example the computation of the variable name similarity between 'in_authskey15' and 'maxDepth'.

4.1 Tokenization

Tokenization takes as input one variable name and outputs a list of dictionary words contained in the variable name. Variable names often consist of multiple (possibly abbreviated) dictionary words, concatenated using capitalization (e.g., CamelCase notation), special separator characters (e.g., underscore), and/or no clear demarcation (e.g., 'authskey').

Tokenization is comprised of 3 steps:

1. Tokenize the variable name based on lexical structure induced by capitalization and separator characters (Section 4.1.1). These initial tokens may or may not be dictionary words.
2. For any word that does not appear in the language dictionary, the abbreviation dictionary, or the stoplist, attempt to further tokenize it by partitioning it into multiple dictionary words and abbreviations (Section 4.1.2). This step handles tokens that correspond to multiple concatenated dictionary words or abbreviations, without capitalization marking their boundaries.
3. Expand abbreviations into dictionary words so that they can be looked up using WordNet (Section 4.1.3).

4.1.1 Lexical Tokenization

A variable name is first tokenized based on its lexical structure induced by capitalization and separator characters. These initial tokens are not necessarily dictionary words. We use the following two rules to identify token boundaries.

- A separator character ends the previous token and starts a new token after the separator character. For variable 'in_authskey15', this rule produces tokens ['in','authskey']. The separators '_', '1', and '5' are discarded. Ayudante considers the following separators: '_','-','+','*', [0-9].
- A capital letter marks the start of a new token, unless preceded and followed by another capital letter. This rule handles CamelCase notation and avoids breaking up acronyms and all-caps variable names. For example, variable 'useLLVMWorkarounds' in Valgrind 3.10.1 would be split into tokens ['use','LLVM','Workarounds'] rather than ['use','L','L','V','M', 'Workarounds']. For variable 'maxDepth' in our running example, this rule produces tokens ['max','Depth'].

In our running example, lexical tokenization yields tokens ['in','authskey'] for variable 'in_authskey15' and tokens ['max','Depth'] for variable 'maxDepth'.

4.1.2 Dictionary Tokenization

Lexical tokenization does not split variable names that are concatenations of words and abbreviations without distinguishable boundaries marked by capitalization or separators (such as 'authskey'). Dictionary tokenization further splits tokens that are not dictionary words or abbreviations into multiple dictionary words and abbreviations.

1. Partition the string into substrings, each one of which is a dictionary word. Our implementation does this by partitioning the token into all possible substrings, then removing partitionings in which any substring is not a dictionary word or an abbreviation.
   For 'authskey', the output is: [['au', 'ths', 'key'], ['auth', 's', 'key'], ['au, 't', 'hs', 'key'], ['auth', 's', 'k', 'e', 'y'], ..., ['a', 'u', 't', 'h', 's', 'k', 'e', 'y']]
2. Choose the partitioning with greatest average word length (awl). Break ties using average English word frequency [10]. For further ties, choose the first partitioning in lexical order. Two special-case rules are:
   - If the top-ranked partitioning does not contain a dictionary word (or an abbreviation) with at least 3 characters, skip the dictionary tokenization step. This rule is designed to avoid overly small tokens. The running example does not trigger this rule because both of the top-ranked partitionings have dictionary word 'key' with length = 3.

- Handle abbreviation plurals by concatenating an abbreviation followed by a single letter 's' to generate a plural abbreviation. In our running example, plural concatenation transforms ['auth', 's', 'key'] into ['auths', 'key'].

For 'authskey', the top 3 partitionings according to awl are:
awl(['auths', 'key']) = $\frac{5+3}{2} = 4$
awl(['au', 'ths', 'key']) = $\frac{2+3+3}{3} = 2.6$
awl(['auths', 'k', 'e', 'y']) = $\frac{5+1+1+1}{4} = 2$
Since there is not an awl tie, the selected partitioning is ['auths', 'key'].

To summarize, after this step 'in_authskey15' has been split into ['in', 'auths', 'key'] and 'maxDepth' has been split into ['max', 'Depth'].

4.1.3 Expand Abbreviations

The last step in tokenization is to expand abbreviations into dictionary words using the abbreviation dictionary. This enables the next phase to compute the similarity between the tokens using a dictionary such as WordNet.

In our running example, 'auths' is expanded to 'authentications' and 'max' to 'maximum'. The final output of the tokenization phase is that variable 'in_authskey15' has been split into dictionary words ['in', 'authentications', 'key'] and variable 'maxDepth' into ['maximum', 'Depth'].

4.2 Dictionary Word Similarity

This section explains how to calculate the dictionary word similarity $wordsim(w_i, w_j)$ between two dictionary words w_i and w_j. If w_i or w_j does not appear in WordNet, their similarity is defined instead as one minus their normalized edit distance[1] [13].

The first step is to use WordNet to extract the (possibly multiple) meanings of a dictionary word, with each meaning being called a *synset* in WordNet. Then, it uses Wu and Parmer's similarity metric [19] that computes the semantic similarity between two synsets (*synsetsim*). To calculate $wordsim$, it computes $synsetsim$ of all pairs of meanings between the two words and chooses the highest similarity among them. For example, there are 6 synsets for 'authentications' and 2 synsets for 'Depth', so to compute $wordsim$ ('authentications', 'Depth') it computes $synsetsim$ for the 12 combinations, and outputs the highest value. In our running example, the dictionary word similarities are:
$wordsim$('in', 'maximum') = 0.09
$wordsim$('in', 'Depth') = 0.11
$wordsim$('authentications', 'maximum') = 0.31
$wordsim$('authentications', 'Depth') = 0.36
$wordsim$('key', 'maximum') = 0.53
$wordsim$('key', 'Depth') = 0.62

[1] Normalized into $[0, 1]$ by dividing by the maximum length of w_i or w_j.

4.3 Variable Name Similarity

Finally, the variable name similarity $varsim(v_1, v_2)$ is computed from the word similarity results in Section 4.2. For each word output after dictionary tokenization, it first computes $maxwordsim$, the maximum $wordsim$ for that word. The variable name similarity is the average of $maxwordsim$ values. We have evaluated other combinations of $wordsim$ to compute $varsim$, in particular maximum of $maxwordsim$, and the average and maximum of $avgwordsim$ (i.e., the average $wordsim$ for a word). The average of $maxwordsim$ worked best.

In our running example, the maximum $wordsim$ are:
$maxwordsim$('in') = 0.11
$maxwordsim$('authentications') = 0.36
$maxwordsim$('key') = 0.62
$maxwordsim$('maximum') = 0.53
$maxwordsim$('Depth') = 0.62
and the variable similarity is:
$varsim$('in_authskey15', 'maxDepth')
$= \frac{0.11+0.36+0.62+0.53+0.62}{5} = 0.45$

5. Identifying Undesired Interactions

This phase takes as input a list of ATI clusters and a variable similarity function such as that defined in Section 4. It measures the cohesiveness of each ATI cluster. The less cohesive a cluster is according to the variable similarity metric, the more likely it contains multiple variable semantics, and thus an undesired variable interaction. Finally, it ranks the ATI clusters in increasing order of cohesiveness.

Partitioning. The first step is to partition each ATI cluster into two sub-clusters. To do so, it computes the variable similarity between all pairs of variables in the ATI cluster. This information is represented as a cluster similarity matrix: an $m \times m$ matrix where m is the number of variables in the cluster and each cell c_{ij} contains $varsim(v_i, v_j)$. It is a symmetric matrix since $varsim(v_i, v_j) = varsim(v_j, v_i)$.

The cluster similarity matrix is passed as input to the k-means clustering algorithm [14] with $k = 2$, which finds the best partitioning of the ATI cluster into two sub-clusters, according to the variable similarity.

Cohesiveness. To compute cohesiveness of an ATI cluster we measure the quality of the partitioning of the ATI cluster into sub-clusters. For this, we use the Dunn validity index [9], a numerical value that captures simultaneously how compact each sub-cluster is and how separated the sub-clusters are. A high Dunn value indicates a good partitioning of the ATI cluster into sub-clusters. This indicates that the ATI cluster is not very cohesive and may contain different variable semantics.

Ranking. ATI clusters are ranked according to decreasing Dunn index, which corresponds to increasing cohesiveness, i.e., from most likely to least likely to contain undesired variable interactions. All ATI clusters having Dunn value

higher than a fixed threshold (0.70 by default) are reported as likely candidates for undesired interactions.

6. Implementation

We have implemented Ayudante using 1,880 lines of Python. For the k-means partitioning and the Dunn index computation, Ayudante uses the clustering package of the R project for statistical computations [2]. We provide other implementation details below.

ATI. We use DynComp [8] as our ATI module. DynComp takes as input a program executable compiled with debugging symbols and input, runs the program on the input, and applies a dynamic unification-based analysis to infer ATI clusters. DynComp supports 4 operating modes that capture different types of interactions: dataflow, dataflow and comparisons, units, and arithmetic. We use DynComp's default arithmetic mode, which considers binary operations (e.g., addition, subtraction) and comparisons as interactions.

DynComp can output variable interactions at various program points during program execution, e.g., function entry and exit. We configure DynComp to output ATI clusters at function exit points. This enables identifying incompatible variable semantics for each program function.

Dictionaries. While our approach is language-agnostic, Ayudante currently only supports English. It uses WordNet as an English language dictionary, and the abbreviation dictionary and the stoplist are also specific to English. We manually populated the abbreviation dictionary, which contains 96 common and 28 application specific abbreviations (15 for Valgrind, 10 for Exim, 3 for Putty, and none for grep). To apply Ayudante to programs in other languages, e.g., Spanish, the user can supply the global WordNet version for that language [1], a stoplist (e.g., `http://www.ranks.nl/stopwords/spanish`), and optionally an abbreviation dictionary.

7. Evaluation

We have performed a preliminary evaluation of our approach. Section 7.1 measures the effectiveness of the tokenization algorithm of Section 4.1 against manually-established ground truth for 2,500 variable names. Section 7.2 reports end-to-end results of running Ayudante on two popular open-source programs, the Exim mail server and grep, and manually analyzing the 5 top-ranked ATI clusters for undesired variable interactions.

7.1 Tokenization

This section measures the effectiveness of our tokenization algorithm (Sections 4.1.1 and 4.1.2). We first manually tokenized 2,500 randomly-chosen variable names from 4 open-source projects: 700 from the Exim mail server, 500 from Valgrind, 300 from Grep, and 1000 from Putty. We treated the manual tokenization as the ground truth. Then, we compared the output of the tokenizer with the ground truth. For example, for variable name 'totsum' (which stands for total sum), tokens 'tot' and 'sum' represent the ground truth.

Table 1: Tokenization accuracy results. It compares the results of the full tokenization (top row) with the tokenization without using the abbreviation dictionary (bottom row).

	Exim	**Grep**	**Valgrind**	**Putty**
With Abbreviations	95%	87%	81%	76%
Without Abbreviations	91%	75%	77%	66%

Table 1 shows the tokenizer accuracy with and without using the abbreviation dictionary. Good coding style and use of standard programming abbreviations are the reasons for high accuracy in Exim. On the other hand, Valgrind and Putty use many domain-specific and custom abbreviations and do not comply with CamelCase or underscore for naming conventions. When removing the abbreviation dictionary, the accuracy drops by 4% for both Exim and Valgrind, 10% for Putty, and 12% for Grep.

7.2 Undesired Variable Interactions

This section analyzes the end-to-end results of running Ayudante on Exim and grep. We run grep with $-w$ option on a single file and Exim is initiated with $-t$ and $-d$ options and a recipient (To) address. We did not test Valgrind and Putty because DynComp cannot run those programs. DynComp is based on Valgrind, and Valgrind cannot be run on Valgrind. Putty runs slowly under DynComp and we did not have time to finish that experiment.

The top-ranked ATI cluster in grep indicates a programming mistake. It contains the 4 variable names `delta`, `depth`, `tree`, and `eolbyte`. The relevant code for this cluster is in function `treedelta` of file `kwset.c` and looks like:

```
if (depth < delta[tree->label])
    delta[tree->label] = depth;
```

where `delta` is an unsigned char pointer and `depth` is an unsigned integer. The undesired interaction happens because the unsigned char variable `delta[tree->label]` is assigned the unsigned integer `depth` and both variables are semantically dissimilar. Since the unsigned char can only hold 8-bit values, the top 3 bytes of `depth` are lost.

This example shows an undesired interaction between variables of different type. We verified that GCC does not identify this interaction statically by compiling grep version 2.2.1 from source code using GCC version 4.8.4 on Ubuntu version 14.04. We used GCC flags `-Wall` and `-Wextra` to force a warning for any error. GCC issues 6 warnings for this module, but none of the warnings include this code. This likely happens because the compiler does not flag all incompatible type interactions, performing casts for some of them. A compiler for a programming language with more strict type checking like Java would flag this error.

On detailed analysis of the source code, we find that a guard instruction at a higher level may ensure this code is not exploitable. But we believe this detection is an encouraging sign of the potential of our approach.

None of the other clusters among the 5 top-ranked ATI clusters revealed an undesired variable interaction to us. These clusters contain variables with poor naming that interact transitively, often through global variables. Due to transitivity, no single location in the code may be responsible for an interaction, which makes it difficult to analyze the source code for potential bugs. We plan to modify DynComp to output all the code locations that cause an interaction transitively to be able to analyze these interactions.

8. Related Work

Prior work has shown that proper code identifiers including variable names can improve code quality and should be used consistently [6, 11]. It has also shown that using identical terms in different contexts may increase the risk of faults, and used this to find fault-prone methods [4]. Our work differs in that we combine NLP with ATI techniques to automatically identify undesired variable interactions.

Some programming languages like F# [3] have support for defining units for variable names, which can be type checked by the compiler. This is similar to forbidding assignments between Ada's derived types [18, §5.3.1]. Our approach operates on program executables and thus can also support programs written in programming languages like C and C++. Some identifier naming conventions, e.g., Hungarian notation [17], include a prefix in variable and function names indicating the functional type of the identifier. Our approach leverages variable name semantics even when not intentionally added.

Prior work has proposed tokenization algorithms for code identifiers including variable names. TRIS [7] and GenTest [12] are two such algorithms. The tokenization accuracy for GenTest is 82% and 86% for TRIS [7]. Our tokenization algorithm achieves 76%–95% depending on the program, but the ground truth used in both works is different. The source code for those algorithms is not available and the online service for GenTest was not working when we tried it. We plan to reimplement both algorithms and compare them to our tokenization on the same dataset.

9. Conclusion

A common programming mistake is for incompatible variables to interact. This paper proposed an automatic approach for identifying undesired interactions between program variables. Our approach uses natural language processing (NLP) to identify variables with related names and therefore possibly related semantics. It uses abstract type inference (ATI) to cluster variables that interact with each other. Then, it identifies undesired variable interactions by checking for discrepancies between the two clustering approaches. Our approach is general and can be used with any variable similarity metric, e.g., variable names, aliasing information, or types.

We have implemented our approach in a tool called Ayudante. Our preliminary evaluation of Ayudante on two popular open-source programs identified a programming mistake that shows the potential of the approach.

Acknowledgments

This material is based upon work supported by the United States Air Force under Contract No. FA8750-12-C-0174. This research was partially supported by the Regional Government of Madrid through the N-GREENS Software-CM project S2013/ICE-2731 and by the Spanish Government through the StrongSoft Grant TIN2012-39391-C04-01. All opinions, findings and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Wordnets in the World. `http://globalwordnet.org/wordnets-in-the-world/`.

[2] The R Project for Statistical Computing. `https://www.r-project.org/`.

[3] "Units of Measure (F#)". `https://msdn.microsoft.com/en-us/library/dd233243.aspx`.

[4] V. Arnaoudova, L. Eshkevari, R. Oliveto, Y.-G. Gueheneuc, and G. Antoniol. Physical and Conceptual Identifier Dispersion: Measures and Relation to Fault Proneness. In *ICSM*, 2010.

[5] H. Baker. Unify and conquer (garbage, updating, aliasing, ...). In *LFP*, pages 218–226, June 1990.

[6] F. Deissenboeck and M. Pizka. Concise and Consistent Naming. *Software Quality Journal*, 14(3):261–282, 2006.

[7] L. Guerrouj, P. Galinier, Y.-G. Guéhéneuc, G. Antoniol, and M. D. Penta. Tris: A fast and accurate identifiers splitting and expansion algorithm. In *WCRE*, 2012.

[8] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic Inference of Abstract Types. In *ISSTA*, 2006.

[9] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. On clustering validation techniques. *J. of Intelligent Information Systems*, 17(2):107–145, 2001.

[10] A. Kilgarriff. BNC database and word frequency lists. `http://www.kilgarriff.co.uk/bnc-readme.html`.

[11] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Effective Identifier Names for Comprehension and Memory. *Innovations in Systems and Software Engineering*, 3(4), 2007.

[12] D. Lawrie, D. Binkley, and C. Morrell. Normalizing Source Code Vocabulary. In *WCRE*, 2010.

[13] V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals, 1966.

[14] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the fifth Berkeley symposium on mathematical statistics and probability*, pages 281–297, 1967.

[15] G. A. Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[16] R. O'Callahan and D. Jackson. Lackwit: A Program Understanding Tool Based on Type Inference. In *ICSE*, 1997.

[17] C. Simonyi. Hungarian Notation. `https://msdn.microsoft.com/en-us/library/aa260976%28VS.60%29.aspx`.

[18] Software Productivity Consortium. Ada 95 quality and style: Guidelines for professional programmers. Technical Report SPC-94093-CMC, Department of Defense Ada Joint Program Office, Oct. 1995.

[19] Z. Wu and M. Palmer. Verb Semantics and Lexical Selection. In *ACL*, 1994.

[20] Q. Yan and S. McCamant. Conservative Signed/Unsigned Type Inference for Binaries Using Minimum Cut. Technical report, Dept. of Comp. Sci. & Eng., U. of Minnesota, Jan. 2014.