RefiNym: Using Names to Refine Types*

Santanu Kumar Dash
University College London
London, UK
santanu.dash@ucl.ac.uk

Miltiadis Allamanis
Microsoft Research
Cambridge, UK
miallama@microsoft.com

Earl T. Barr
University College London
London, UK
e.barr@ucl.ac.uk

ABSTRACT

Source code is bimodal: it combines a formal, algorithmic channel and a natural language channel of identifiers and comments. In this work, we model the bimodality of code with *name flows*, an assignment flow graph augmented to track identifier names. Conceptual types are logically distinct types that do not always coincide with program types. Passwords and URLs are example conceptual types that can share the program type string. Our tool, RefiNym, is an unsupervised method that mines a lattice of conceptual types from name flows and reifies them into distinct nominal types. For `string`, RefiNym finds and splits conceptual types originally merged into a single type, reducing the number of same-type variables per scope from 8.7 to 2.2 while eliminating 21.9% of scopes that have more than one same-type variable in scope. This makes the code more self-documenting and frees the type system to prevent a developer from inadvertently assigning data across conceptual types.

CCS CONCEPTS

• **Software and its engineering → Data types and structures**;

KEYWORDS

Type Refinement, Information-theoretic Clustering

ACM Reference Format:
Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2018. RefiNym: Using Names to Refine Types. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236042

1 INTRODUCTION

During development, programmers tend to use imprecise types, especially under time pressure. For example, they might combine conceptually distinct types like passwords or last names into a single built-in type like `string`. Working with a simpler type hierarchy helps rapid prototyping, but comes at a cost. The members of this coarse-grained type lattice often do not naturally belong together and are a source of bugs at deployment. For example, if a program

*This research was funded by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/P005314/1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236042

```
1  var server = Config.Get("fxcm-server", ... );
2  var terminal = Config.Get("fxcm-terminal", ... );
3  var userName = Config.Get("fxcm-user-name", ... );
4  var password = Config.Get("fxcm-password", ...);
5
6  var downloader = new FxcmDataDownloader(server, terminal,
7                       userName, password);
8
9  foreach (var ticker in tickers) {
10   if(downloader.HasSymbol(ticker)){...}
11 }
```

Figure 1: Motivating example from the Lean C# Project; we lightly modified the code to fit.

uses only built-in types, then invalid or dangerous operations like `cost + distance` or `string.Concat(password, lastname)` are type-correct. This is a well-documented problem called *primitive obsession* that occurs whenever a developer avoids defining bespoke types [12, 20, 26].

Figure 1 shows a classic example of primitive obsession in Lean (commit hash `f574bfd7`), an open-source algorithmic trading engine written in C#. This snippet connects to a data provider for Foreign Exchange Capital Markets (FXCM) and searches for ticker symbols. Using it requires logging in through the `FxcmDataDowloader` class. This class takes in four arguments. All four arguments are annotated as `strings`, intermixing disparate concepts such as server addresses, terminals, usernames and passwords with each other and plain-text strings. Accidentally using the password in place of the user name would *not* trigger a type error and go unnoticed until the attack surface is exploited at runtime. Robin Milner famously observed "*Well-typed programs don't go wrong*." Here, we see an example of code that typechecks but is not well-typed.

Conceptual types are the types the developer had in mind while writing a program. Often they coincide with the program's types, but sometimes, as with primitive obsession, they are not explicitly defined and are latent to a program's explicit type hierarchy. In this work, we combat this problem, including primitive obsession, by assisting developers to identify refinements of a program's type lattice: we mine conceptual types, then suggest their reification into actual types. To discover latent conceptual types, we introduce RefiNym, a tool that automatically mines conceptual types, then presents them to developers to help them define a more specific type that surfaces distinct conceptual types, their values and behaviours, to the type checker. RefiNym uses data flow and name information to detect conceptual types. In Figure 1, RefiNym suggests separating all four variables — `server`, `terminal`, `userName`, and `password` — into distinct types, so that the type checker can prevent a developer from carelessly directly assigning between them.

We formulate the problem of refining a type in a C# program's type lattice as a search-based software engineering problem [14] and search for valid and coherent type lattices whose elements are a

mix of explicit and conceptual types. Information within identifiers in the code and assignment flows between them constrains the search. Our approach is *bimodal* because it intermixes semantic information (flows) and syntactic information (names). Our method exploits lexical similarities to build a conceptual type lattice using the information-theoretic principle that a candidate conceptual type is good if it minimizes the shared information distance between names and types: given a conceptual type, the names of the variables of that type and names of the methods that return that type should have low entropy (*i.e.* be predictable); given a variable or method name, its conceptual type should be easy to predict.

RefiNym constructs a *name flow graph* (Section 2.1) from a program that type checks. A name flow graph contains all the type constraints imposed by assignment flows in the program across variables and parameter bindings. RefiNym then uses variation of information, an information-theoretic objective, to cluster the nodes of the name flow graph subject to C#'s subtype rules (Section 2.2). These clusters are candidate conceptual types. RefiNym generates fresh names for these clusters and suggests them as type refinements. If a developer accepts one, RefiNym rewrites its input to use that type (Section 3).

RefiNym finds refinements for an existing program type; so, as a sanity check, we artificially create conceptual types by merging user-defined types, then ask RefiNym to reconstruct them. RefiNym exactly reconstructed 62% of these types (Section 4.1). We have equipped RefiNym with a rewriter that automates refactoring a codebase to use its suggested refinements for C#, a syntactically rich, industrial language (Section 3). Software engineers care about how code evolves in response to their needs. Some of these changes worsen the codebase, moving it toward a critical point where further changes can cascade into error. A scope in which two conceptual types share a primitive is such a critical point. In these *critical scopes*, the type system cannot prevent a developer from mistakenly assigning values from one conceptual type to the other. Over our corpus, RefiNym automatically eliminates 21.9% (on average) of these critical scopes (Section 4.2) while reducing the potential of inadvertently introducing cross-conceptual type flows within scopes by drastically reducing the number of same-type variables per scope from an average of 8.7 to 2.2, thereby enabling the type system to better protect the developer from her own mistakes.

Contributions. Our core contributions follow: a) we present an information-theoretic nominal type refinement method that uses identifier names and dataflow to mine conceptual types; b) we present RefiNym, a practical C# tool that implements our method; and c) we comprehensively evaluate RefiNym on real world open–source projects and show that it eliminates 21.9% (on average) of potential inadvertent flows. RefiNym and our evaluation artifacts are available at http://github.com/askdash/refinym.

2 INFERRING CONCEPTUAL TYPE LATTICES

The latent types in a program often differ from the type lattice in the program the developer explicitly defined with type annotations. We call these latent, precise types, *conceptual types*. These types group related data and operations. They are similar to Guo's *et al.*'s [11] abstract types, except that they are not always more abstract than the types in the program's type lattice. We are interested in using names to infer conceptual types, then look for refinements that bring a program's explicit type lattice closer to its implicit conceptual type lattice, by splitting the merged conceptual types.

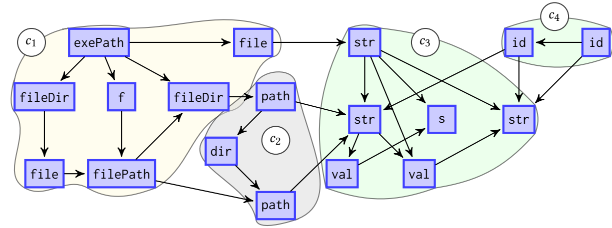


Figure 2: Conceptual clustering of a name flow graph for `string`: A node is a variable or a function, while edges are assignments or actual to parameter bindings (Section 2.1). RefiNym clusters nodes into conceptual types, such as the shadowed areas. Here, c_1 represents file path, c_2 general paths, c_4 identifiers; and c_3 arbitrary strings.

Modern software engineering practice unanimously agrees on the need for well-named identifiers. Indeed, most developers choose names with great care [1, 4]. For example, names of variables and functions are selected in such a way that they reflect their semantic role and function. The correspondence between names and semantics is a crucial component when understanding code, since developers think semantically about code and representative names facilitate this process [17].

For these reasons, our core intuition is that variable and method names are often semantically rich and closely reflect their identifier's functionality and therefore provide strong information about an identifier's conceptual type. Specifically, we observe that names tend to flow into names that describe similar concepts following the general covariance and contravariance principles. For example, a numeric variable named `distance` sometimes flows into a numeric variable with a more abstract name like `value`, but rarely the other way around. Therefore, a program's data flow imposes "is-a" relationships among the names of the variables and methods and, consequently, to the concepts they describe. We capture these relationships in a *name flow graph*, described next. To recognize salient conceptual types, we cluster the graph, such that the inferred clusters, along with the original unrefined type as top and the standard bottom, form a lattice (Figure 2). In this work, we focus on using discrepancies between the program's implicit concept type lattice and its explicit type lattice to find and suggest type refinements. For example, when the `string` class holds both passwords and surnames, RefiNym will suggest distinct subtypes, like `PasswordString` and `SurnameString`. If a developer adopts these suggestions, the type checker can then prevent the developer from mistakenly sharing data between these types.

2.1 The Name Flow Graph

For each variable, we need to collect the names whose values flow into that variable. To this end, we augment a standard constraint-based type system to capture single-step name flow, then build a graph for the variable v whose traversal defines the inflow name closure of v.

For the type-correct program P, let V be its variables, L be its literals, and M be its method names. For the function $\Xi : V \to$

$$\frac{\Gamma \vdash v : \sigma_1 \qquad \Gamma \vdash k : \sigma_2 \qquad \sigma_2 <: \sigma_1}{\boxed{\Xi(v) \cup \{v, (k, \mathbf{l})\}} \qquad \Gamma \vdash v := k : unit} \quad (\text{AssignL})$$

$$\frac{\Gamma \vdash v : \sigma_1 \qquad \Gamma \vdash v' : \sigma_2 \qquad \sigma_2 <: \sigma_1}{\boxed{\Xi(v) \cup \{v, (v', \mathbf{v})\}} \qquad \Gamma \vdash v := v' : unit} \quad (\text{AssignV})$$

$$\frac{\Gamma \vdash v : \sigma_1 \qquad \Gamma \vdash m(\bar{e}) : \sigma_2 \qquad \sigma_2 <: \sigma_1}{\boxed{\Xi(v) \cup \{v, (m, \mathbf{m})\}} \qquad \Gamma \vdash v := m(\bar{e}) : unit} \quad (\text{AssignM})$$

$$\frac{e \notin L \cup V \cup M \qquad \Gamma \vdash v : \sigma_1 \qquad \Gamma \vdash e : \sigma_2 \qquad \sigma_2 <: \sigma_1}{\boxed{\forall n \in names(e), \Xi(v) \cup \{v, (n, label(n))\}} \qquad \Gamma \vdash v := e : unit} \quad (\text{AssignE})$$

$$\frac{\Gamma \vdash e_0 : \sigma_0 \qquad mtype(m(\bar{p}), \sigma_0) = \bar{\delta} \to \sigma \qquad \Gamma \vdash \bar{e} : \bar{\sigma} \qquad \bar{\sigma} <: \bar{\delta}}{\boxed{\vec{p} = \langle p_1 := a_1, \cdots, p_i := a_i, \cdots p_n := a_n \rangle} \qquad \Gamma \vdash e_0.m(\bar{e}) : \sigma} \quad (\text{Invoke})$$

Figure 3: A constraint-based system for typing variables, augmented to update $\Xi(v)$, the set of names whose values flow into v; our additions to standard rules are boxed; Invoke make formal to actual parameter bindings explicit for matching by the assignment rules.

$2^V \uplus 2^L \uplus 2^M$, $\Xi(v)$ is all the names that flow into v via an assignment $:=$ or a formal-to-actual binding. The names in $\Xi(v)$ are type compatible under the assumption that P type checks. Ξ returns an element from a disjoint union, where **v** tags variables, **l** tags literals, and **m** tags methods.

Figure 3 shows extensions to standard rules for typing variables that are relevant to RefiNym. RefiNym identifies refinement types for variables bound only to first-order terms. Therefore, we only discuss the subset of rules related to variables: Invoke and the four rules for collecting constraints from explicit assignments. AssignL, AssignV, AssignM, and AssignE handle assignments from constants, variables, method returns, and all other expressions, respectively. In our first order setting, we handle method types by projecting them onto their constituent parameter and return types.

As usual, the assertions in Figure 3 are of the form $\Gamma \vdash v : \tau$ where Γ is the type environment that maps types to terms in the calculus. For simplicity and without loss of generality, we restrict σ_i to scalars. We could have modified all expression judgments to collect names, but, for brevity, focused on assignment. All of the assertions in these rules are standard with the exception of the boxed assertion in each rule's conclusion and the $e \notin L \cup V \cup M$ in AssignE. The boxed assertions use disjoint union to extend Ξ. In Invoke, *mtype* returns the type of a method $m(\bar{p})$ in a class σ_0. The additional constraint on AssignE prevents its redex from overlapping those of the other assignment rules and is consequence of our decision to modify only assignment judgments. We augmented Invoke to convert m's parameter list $\vec{p}$ into an explicit sequence of assignments to which our four assignments rules apply and collect name flows from actuals into m's formals.

AssignE relies on two helper functions: *label* that distinguishes literal, variable, and method names and *names* that extracts names in the expression e that are type-compatible with v and adds them to the name-flow for v. Consider "`int x = strlen(str)+ y`". Because of the '+', AssignM does not match `strlen` and we fall through to AssignE, which applies *names*. This function collects all the assignment compatible names from an expression, here `strlen` and `y`. AssignE adds these names to `x` in the name-flow graph. The method *names* ignores `str` because it is a parameter of `strlen`; in other words, method invocations are sinks for the name-flows of their parameters.

After building Ξ, RefiNym uses it to construct the *name flow graph* $\mathbb{G} = (N, E)$. The nodes in N are fully scope-qualified variable, method, or formal names. An edge connects two names in two cases: 1) when the source name is on the RHS of an assignment and the target is on the LHS or 2) when the source names the actual bound to the formal name, *i.e.* the target. The *label* function labels each edge. Because we collect name flows alongside type constraints only for assignments and parameter bindings, the name flows from literals, variable, and method returns are type-compatible by construction; the name flows from the names used in other expressions may or may not be depending on the implementation of *names*().

In Figure 3, the boxed assertions that collect names into Ξ are independent of the underlying, standard constraint-based type checking. Thus, they can be adapted to other type systems, including type systems that support partially typed programs via an "any" type wildcard.

Figure 4 shows how Ξ evolves across function applications. At the top left of the figure, we have the `Calc` class which has a method `add` that takes two integers, `x` and `y`, as inputs. `Main` invokes `add` twice on lines 6 and 7. In each case, Invoke and AssignM type the assignment expression. For brevity, Figure 4 merges the updates to Ξ these two rules make. There are two dotted arrows leading to the type rule for the two assignment operations.

The solid arrows annotated with circled numbers indicate how Ξ changes as the two type rules are applied. Before line 6 in `Main`, `x` and `y` do not have any name flows coming into them. This is indicates by a $\cdot$ in their Ξ's. The execution of line 6 assigns the variable `fst` to `x` and `snd` to `y`. Additionally, through the assignment operator :=, line 6 also binds `sum` to the result of `c.add`. In the conclusion of type rule, we update Ξ's for `x`, `y` and `sum` with the information above. The assignment in line 7 is similar except that it binds `y` to a literal.

2.2 Clustering the Graph

In this work, we are interested in partitioning (clustering) the nodes in a name-flow graph $\mathbb{G}$ into clusters that form a lattice over the graph. These clusters represent mined conceptual types. Conceptual types name program variables that share a common purpose and identify operations over them (Figure 2). Precisely because they share a purpose, we expect their names to reflect that purpose to remind developers about it and the constraints it imposes. Thus, we expect names and conceptual types to be mutually informative. To cluster $\mathbb{G}$, we therefore exploit the interplay between name flows in $\mathbb{G}$ and the names of the variables and methods by clustering. For example, two variables that are not closely related in $\mathbb{G}$ but share a name have a high chance of being related, while two variables that have very different names but with very similar name flows are also

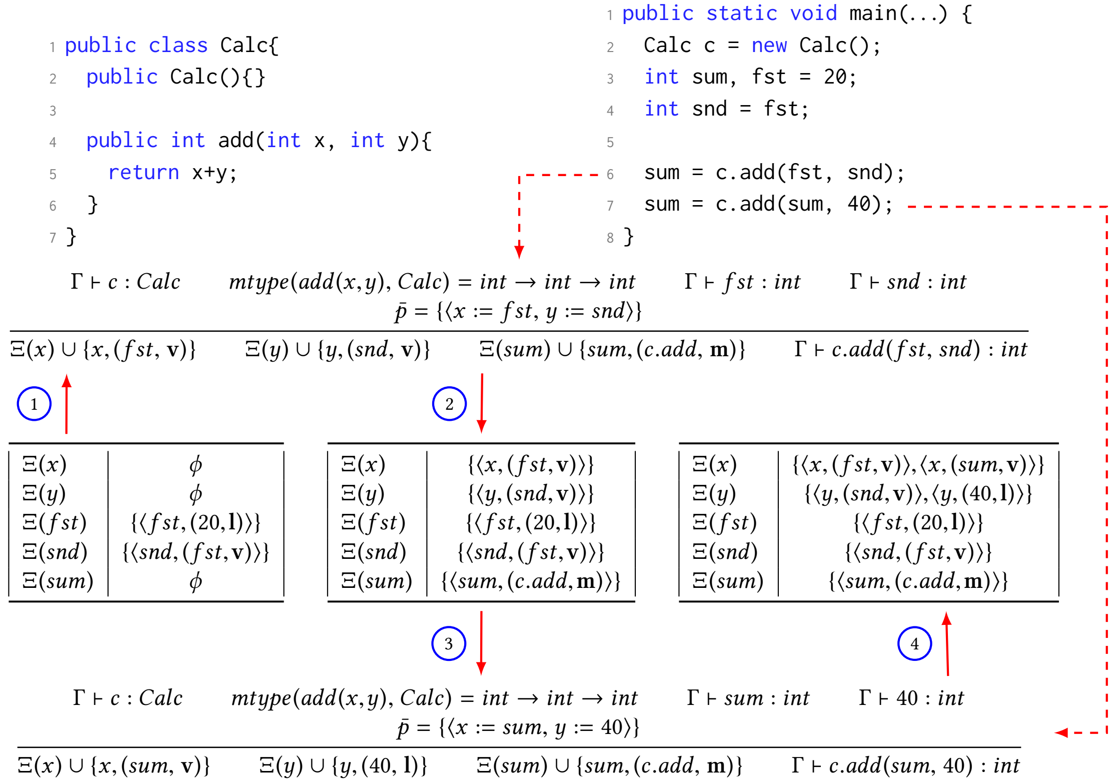


Figure 4: An example of how Ξ, which maps direct value exchanges between nodes, evolves over method invocations.

probably related. To extract maximal signal from names, we exploit their internal structure and extract their subtokens (Equation 3).

The key problem then is to exploit these names to automatically find groups of nodes in $\mathbb{G}$ that are conceptually similar. To achieve this, we design a novel non-parametric clustering method over graphs that is based on information-theoretic concepts; we describe it below. Figure 2 shows a sample depiction of the task.

In the clustering method we present here, $names(e) = \varnothing$ (AssignE in Figure 3). We made this choice because name flows across operators must be selective. Some of these flows would conflate conceptual types. Thus, handling operators would unnecessarily complicate our model, so here we treat operators as name flow breaks, just as AssignE treats functions. To illustrate the problem, imagine the concatenation of two strings of different conceptual types. The output might be yet another conceptual type. For example, concatenating a "username" string with a "delimiter" string could result into a "csv" string. Correctly addressing this problem would require teaching the clustering method to be aware of operators polymorphic over conceptual types. By excluding the cross-operator name flows that AssignE adds, our clustering method can infer the types of all involved terms by learning from their names and all other usages, essentially circumventing this problem with minimal loss of information. Further, RefiNym tolerates information loss from not adding cross-operator flows that did occur because it uses probabilistic methods that robustly handle noise and extrapolate from missing data.

Notation. Let $\mathbb{G} = (N, E)$ be a name-flow (directed) graph. Let C partition N and $c_i \in C$ be a part of C. We equip each partition C with the function $parent_C : C \to 2^C$. Its application, $parent_C(c_i)$, returns $\{c \mid (n_0, n_1) \in E \wedge n_0 \in c \wedge n_1 \in c_i\}$: all the clusters of C that contain a node directly connected to a node in c_i. We overload the notation for C to imply both the partition and the random variable of selecting some $c_i \in C$, since the intended meaning is clear from context. N contains fully scope-qualified names, but here we are only interested in the unqualified name because the namespace prefix adds noise for our task. We use the function *name* to extract the unqualified name from a node. As with C, we overload N to refer both the nodes in $\mathbb{G}$ and the random variable of selecting some name from $\{name(n) \mid n \in N\}$. Finally, we refer to c_i either as a part or as a cluster.

Objective. For the purpose of this explanation, first consider the case where we ignore the structure of $\mathbb{G}$ and simply partition N. Say we cluster N such that each part c_i contains only similar names. In the limit, this would yield uninformative partitions that contain nodes with identical names. Instead, we choose to minimize the variation of information[1] (VI) [6] between node names and parts. VI naturally represents the tension between creating too many clusters that do *not* differ significantly or too few that are not informative about their elements and is defined as

$$\mathrm{VI}(C, N) = H(C|N) + H(N|C) = H(N, C) - I(N, C), \quad (1)$$

[1] VI is also known as "shared information distance".

where C and N are the random variables of the nodes' cluster labels and names, $H(A|B)$ is the conditional entropy of A given B, $H(A,B)$ is the joint entropy of A and B, and $I(A,B)$ is the mutual information between A and B. VI is a true metric: it obeys the triangular inequality. Computing $H(C|N)$ and $H(N|C)$ requires computing $P(C|N)$ and $P(N|C)$. We discuss the modeling choices we made to formulate these probability distributions at the end of this section.

Intuitively, to minimize Equation 1, we simultaneously require positive answers to two questions: "Given the name of node n, can we easily predict the cluster (part) it came from (*i.e.* $P(C|N)$ has low entropy)?" and "Given a cluster, can we easily predict the name of each node (*i.e.* $P(N|C)$ should have low entropy)?". Thus, the goal is to pick a "sweet spot" among these two conflicting goals. This "sweet spot" correspons to the clustering that provides the conceptual types. VI selects a single trade-off, valuing $H(C|N)$ and $H(N|C)$ equally. If we minimized $H(C|N)$ and $H(N|C)$ separately, *i.e.* as a multiobjective optimization problem, we would retrieve a Pareto front of optimal solutions.

We choose VI as our metric for three reasons. First, it is non-parametric: it does not require us to specify the number of clusters upfront (in contrast to methods like k-means). Second, it relies on information-theoretic principles that capture our core intuition that conceptual types and names reflect each other. Finally, it is a true metric, so we know that the (theoretical) optimal solution yields a VI of 0.

So far, we have ignored the structure within the graph and treated each node separately. However, we want to cluster (or partition or color) $\mathbb{G}$ so that the parts form a lattice, *i.e.* there is a well-defined ordering relation $R = (C, \sqsubseteq)$ defined by the transitive closure of $\forall c_j \in parent_C(c_i) : c_j \sqsubseteq c_i$. Thus, we formulate our clustering problem as

$$C^* = \arg\min_C \mathrm{VI}(C, N) \text{ s.t. } \exists R = (C, \sqsubseteq). \tag{2}$$

Although the evaluation of Equation 1 has not changed, our search space — all possible partitionings of $\mathbb{G}$ that form a lattice — imposes a very strong constraint on our method.

Optimization. To minimize Equation 2, we need to iterate over all partitions of $\mathbb{G}$ that maintain the lattice property on C and pick one that minimizes VI. However, there is no obvious way to enumerate all such partitions of $\mathbb{G}$ and, even if there were, the space is prohibitively large. Since we are unaware of existing methods for minimizing Equation 2, we resort to the greedy heuristic in Algorithm Algorithm 1. We start from $\mathbb{G}$ and assign all disconnected components to different clusters. Then, we uniformly sample a random permutation over pairs of clusters in C and try to merge each pair. The predicate LATTICE? checks $\exists R = (C, \sqsubseteq)$, *i.e.* a lattice ordering exists over C. The second loop uniformly at random enumerates clusters for splitting. After a candidate merge or split, LATTICE? filters out those partitions that violate the lattice constraint. Otherwise, we apply VI to determine whether the new partition improves on the current partition. The VI measurements in the inner loops consider two partitions that differ by only a single split or merge operation, so we need only compute VI on the changed part(s) (cluster(s)), significantly speeding up the computation.

Algorithm 1 Greedy Optimization of Equation 2.

```
▷ This helper function applies ⊕ (either MERGE or SPLIT) to the
current cluster C and each element of a list to form a new cluster,
then checks whether the new cluster is better under VI. In the
case of MERGE, the list elements are pairs of clusters; for SPLIT,
the elements contain a single cluster.
function ImproveCluster(list, C, N_b, ⊕)
    improved ← False
    while !improved ∧ !list.empty? do
        C′ ← ⊕(list.next(), C)
        continue if !LATTICE?(C′)
        if VI(C′, N_b) < VI(C, N_b) then
            C ← C′; improved ← True
    return C, improved

function OptimizeVI(𝔾 = (N,E))
    C ← DisconnectedComponents(𝔾)
    N_b ← BASENAMES(N)
    repeat
        improved ← False
        pairs ← SAMPLE a permutation over pairs of clusters in C
        C, improved ← ImproveCluster(pairs, C, N_b, MERGE)
        if !improved then
            parts ← SAMPLE a permutation over clusters in C
            C, improved ← ImproveCluster(parts, C, N_b, SPLIT)
    until !improved ∨ max number of iterations
    return C
```

Algorithm Algorithm 1 does *not* enumerate all sequences of split/merge operations: some valid clusterings that maintain the lattice ordering cannot be reached by combining single node split/merges. Nonetheless, we find that, for practical name flow graphs, we reach good solutions. Furthermore, the first step of Algorithm Algorithm 1 helps avoid local optima by first splitting $\mathbb{G}$ into its disconnected components.

As we discuss later in Section 4, we empirically find that our method usually converges to the same or highly similar solutions, across all random restarts. This suggests that the structure of our problem allows us to reach a similar solutions despite using a greedy approximation.

Modeling $P(N|C)$ and $P(C|N)$. To compute the conditional entropies in Equation 1, we need to compute the conditional probabilities $P(C|N)$ and $P(N|C)$. Computing $P(N|C)$ is intricate, so we explain it in detail here. Once we have $P(N|C)$, computing $P(C|N)$ is straightforward thanks to Bayes' rule. To compute the probability of a particular name, we could use the empirical distribution of names (*i.e.* their count) within a cluster c. However, we want to take advantage of the internal structure of the names — the subtokens present in source code identifiers. We use a simple regular expression for `pascal_case` or `camelCase` to split each name into subtokens. Let t tokenize a name using this regular expression, then $t(n)$ is the multiset of subtokens for n. Let $S_N = \{s \mid s \in t(name(n)), n \in N\}$. Then, we define $P_{S_N}(s)$ as a multinomial distribution over all the subtokens across the set N of all node names in a cluster. Formally, for each possible subtoken $s \in S_N$, we associate a probability w_s

such that $\sum_s w_s = 1$ and $w_s \geq 0$. Then

$$P_{S_N}(n) = \prod_{s \in t(n)} w_s. \quad (3)$$

Similarly, to compute P_{S_c} for a cluster, we could count the empirical frequency of the subtokens within the cluster (*i.e.* in S_c). However, computing the maximum likelihood estimate for each cluster is prone to overfitting. To partially overcome this issue, we resort to Bayesian statistics. We specialize Equation 3 for each cluster c by redefining w_s^c, as follows: We introduce a Dirichlet prior $\mathbf{w}^c \sim Dir(\alpha\mathbf{d})$ so, for a subtoken s at a cluster c, we have

$$w_s^c = \frac{\text{count}(s \in c) + \alpha d_s}{\sum_i \text{count}(s_i \in c) + \alpha} \quad (4)$$

where $\alpha \geq 0$ is the importance we assign to our prior and $\mathbf{d}$ is weights of the prior distribution. In this work, we set $\mathbf{d}$ to the empirical frequency of the subtokens within $\mathbb{G}$, reflecting our prior belief that the subtokens within a cluster are similar to the subtokens throughout the program. If $\alpha = 0$, we retrieve the empirical subtoken distribution within the cluster c. Adding this prior biases the subtoken distribution within a cluster toward the empirical global subtokens distribution and allows our objective to create new clusters only when a cluster's subtoken distribution significantly differs from the global subtoken distribution. It also "smooths" small differences between clusters allowing them to merge when their differences are small.

3 IMPLEMENTATION

RefiNym uses the .NET Compiler framework (Roslyn) [9], to collect name flows. Our source code is written in C# language version 7.1 and our runtime version is 4.6. The Roslyn compiler that we used for our experiments is version 2.6. RefiNym's clustering module learns conceptual types from name-flows harvested through Roslyn using the techniques described in Section 2.2. The rewriting module then modifies the program automatically to introduce the learned conceptual types into the original program.

RefiNym's clustering algorithm is implemented in C# using standard .NET libraries. To speed-up the implementation of Algorithm 1 we make extensive use of caching. At each iteration, only two clusters will be merged or only one will be split. Consider two clusterings that differ only in a single merge $m(c_0, c_1)$ or split $s(c_1)$ operation. Algebraically, the difference in the VI of these two clusterings depends only on the two merged clusters or the single split cluster, independent of the rest of the clusters in each clustering. Because computing VI is so expensive, we cache the VI differences of these operations, keyed by the operation and its operand(s) *i.e.* $m(c_0, c_1)$, then check the cache when evaluating $\text{VI}(C', N_b) < \text{VI}(C, N_b)$ in Algorithm 1. Furthermore, computing the VI improvement over potential split/merge operations is embarrassingly parallel.

RefiNym's rewriter translates the lattice formed by the clustering approach described in Section 2.2 into C# code. To do this, we define lattice points as user-defined types and subtyping relations between the points as type casts. Additionally, we need two other forms of implicit casts: from a primitive type to a conceptual type and back. For defining these casts, we use the `implicit operator` keywords in C#. An implementation of a conceptual type with conversions to and from the `string` type is shown in Figure 5.

```
1 class Word {
2     public Word(string s) { val = s; }
3     private readonly string val;
4     public static implicit operator string(Word w) {
5         return w.val; // Convert Word to a string
6     }
7     public static implicit operator Word(string s) {
8         return new Word(s); // Convert string to a Word
9     }
10 }
```

Figure 5: Auto-casting between a class `Word` and strings

Here, `val` stores the value bound to the original primitive type. The methods with `implicit operator` qualifiers cast a `string` to the learned conceptual type `Word` and back.

C# is an industrial language with many features. RefiNym's rewriter supports a large subset of all rules in C#'s grammar. RefiNym handles pass by value and by reference. For the latter, RefiNym passes a reference to the `val` inside a conceptual type bound to the primitive type. For example, in Figure 5, RefiNym replaces all occurrences of the qualified type `ref string x` with `ref Word x.val`. For library calls, where RefiNym cannot rewrite the source code, it upcasts arguments that are conceptual types into the original primitive type. Similarly, it downcasts return values from the library calls that are of primitive types to conceptual types. It applies the same technique when calling a method that is defined only for a primitive type and when indexing collections of primitive types such as arrays. RefiNym currently abstains from rewriting primitive types that have type qualifiers such as `const` to preserve program semantics. Details can be found in the source code at http://github.com/askdash/refinym.

A testament to the versatility and utility of RefiNym's rewriter is the fact that it only makes 4 errors per 10KLOC on our corpus (Table 1), which covers diverse application domains. All of these errors are due to the fact that RefiNym does not yet cover all terms in the C# grammar, like its higher-order functions.

Deployment. RefiNym is fully automatic. It can be easily merged into workflows with little integration overhead. Although we demonstrate the utility of our tool using C# as an example, our techniques and the core of our learning framework is programming language agnostic. We harvest name-flows though assignments and our tool can leverage any flow analysis framework for any language. For C#, we have implemented our tool as a command line tool that can be used at development time. The programmer can select a primitive or a user-defined type (UDT) and invoke RefiNym to automatically collect name-flows for variables of that type, identify potential conceptual type through the clustering, and rewrite the syntax tree to automatically introduce these conceptual types. After rewriting, the syntax tree is automatically recompiled and the tool reports the effect of the rewriting by echoing Roslyn's messages, including compilation errors, to the plugin's console.

While inferring the initial clustering is an overnight task, once a clustering is inferred, it could be efficiently updated, given incremental changes, since only the parts of the graph that change need to be checked for potential split or merge operations with other clusters. Furthermore, given a static clustering, classic type inference methods can be used to infer the conceptual types of newly introduced code elements.

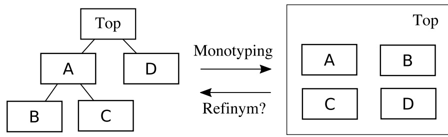


Figure 6: Validating RefiNym via type reconstruction.

RefiNym facilitates a developer's exploration of type lattices. Conceptual types and program types can diverge during development. Developers are focused on bug-fixing and rapid prototyping but not on type constraints. For a developer, merging changes upstream through branch promotion is natural time to address the divergence between program types and conceptual types. This is where RefiNym can be an excellent aid. Developers can install and use it locally on their machines to review their use of types without impacting other developers or needing to secure permission from management. We believe that self-vetting prior to branch promotion is a use case for which RefiNym is particularly well-suited.

4 EVALUATION

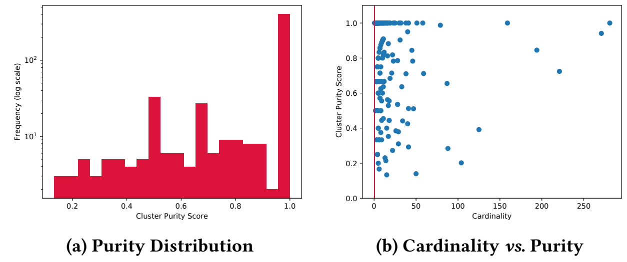
Here, we evaluate how well RefiNym identifies latent conceptual types by applying it to the task of reconstructing user-defined type after they have been artificially merged (Section 4.1) and how well it combats primitive obsession (Section 4.2). Section 4.3 closes with a case study of the clusters RefiNym produces.

Initially, we used the NancyFx project as our development set for constructing RefiNym. To evaluate RefiNym, we turned to real-world open-source projects, systematically selected as follows. First, we selected top C# GitHub projects and removed projects that we could not compile. We also selected a scientific or physics library called BEPUphysics — a 3D real-time physics simulator — to show how RefiNym reifies types for variables that represent physical quantities or units. Table 1 shows details of these 14 C# projects. Our experiments were performed on a 2.7 GHz Intel Core i7 machine with 16GB of memory running OSX version 10.13.3.

4.1 Reconstructing UDTs

To assess how well RefiNym infers conceptual types, we artificially create a mismatch between a program's conceptual types and its program type lattice. Essentially, we monotype user-defined types (UDTs) in the program. We restrict our attention to UDTs because they tend to be precise and not subject to primitive obsession. Restricted to a program's UDT sub-lattice, we rewrite all UDT type annotations to $\top$, *i.e.* `object` in C#. Figure 6 shows the process. The result is a program with a maximal divergence between its program lattice and its conceptual type lattice for UDTs; a program for which we know the ground truth, assuming the correctness of the original program type UDT sublattice. Recovering the original UDTs from this program is the *UDT reconstruction problem.* This problem is particularly challenging for RefiNym because it monotypes all UDTs, not a small subset of them. Then we ran RefiNym on the monotyped program and ask "How well does RefiNym recover the original program's UDTs?".

To assess the quality of RefiNym's reconstruction of UDTs, we compute *homogeneity* $\mathcal{H}$ and *completeness* [24]. Homogeneity measures the diversity of labels across all clusters (low diversity implies high homogeneity); completeness measures the extent to which



(a) Purity Distribution **(b) Cardinality *vs.* Purity**

Figure 7: RefiNym's performance at UDT reconstruction. Figure 7a shows the distribution of cluster purity over our corpus (notice log-scale in y axis): RefiNym *perfectly* (generating purity 1 clusters) reconstructs 77% of UDTs. Figure 7b shows the distribution of cluster purity *vs.* size.

clusters contain all elements of a label. Like the more familiar precision and recall from information retrieval, homogeneity and completeness are in tension: increasing homogeneity tends to decrease completeness, as you may lose elements of a target label when shedding elements with an undesirable label. For a set of elements for which C is a clustering and K is a labeling, homogeneity ($\mathcal{H}$) is

$$\mathcal{H}(C,K) = \begin{cases} 1, & \text{if } H(C,K) = 0 \\ 1 - \frac{H(K|C)}{H(C)} & \text{otherwise,} \end{cases} \quad (5)$$

where H is the Shannon entropy. Completeness is $\mathcal{H}(K,C)$. $\mathcal{H}$ ranges from 0 to 1. High homogeneity means that the label distribution of elements within a cluster is skewed toward one label *i.e.* the conditional entropy $H(K|C)$ is close to zero.

As Table 1 shows, RefiNym averages a homogeneity score of 0.80 and a completeness score of 0.91 on the UDT reconstruction task. The balance between these two averages shows that RefiNym is effectively extracting signal from names to group the majority of related names to achieve high completeness without simultaneously adding many unrelated names, thereby maintaining high homogeneity.

Homogeneity and completeness are global measures of a clustering. RefiNym equates clusters with conceptual types. Thus, to understand RefiNym's performance at conceptual type inference, we must assess its performance at the granularity of individual clusters and look at its purity. For RefiNym to perfectly solve the UDT reconstruction problem, each cluster must exactly coincide with conceptual types. The *purity* of a cluster is the proportion of its dominant label. The objective of the type reconstruction here is to segregate nodes in the name-flows into pure clusters, those that contain only names of the same type. This is not enough, however, because an original UDT's type could be fragmented into k different clusters, each of purity 1.

Figure 7a shows the distribution of purity scores. The key finding here is that, at this very hard UDT reconstruction problem, 72% of the types/clusters RefiNym finds have purity 1. This means that RefiNym rewriter could reify them in the program and the rewritten program would still type check. Since RefiNym's clustering module is unaware of the types for which the flows have been harvested, this purity result also verifies how much signal identifiers carry about their type. We now consider the fragmentation

Table 1: C# Corpus from GitHub. Homogeneity and Completeness Scores for Reconstructed User-Defined Types (UDT). Right-hand columns report the reduction in the number of co-occurrences of the the type `string` in the same scope by running RefiNym with the exception of BEPUphysics that contains no string variables.

Name	SHA	Description	kSLOC	Homogeneity	Completeness	# same-type variables per scope Before Mean	Before Median	After Mean	After Median	% removed critical scopes
BEPUphysics	2f05f5e3	3D Physics Simulation Library	45.3	0.79	0.93	–	–	–	–	–
Commandline	451be765	Command Line Parser	9.6	0.76	0.86	4.2	4	2.3	1	9.3
CommonMark	e94800e6	Markdown Parser Library	13.9	0.79	0.80	1.5	1	0.1	0	35.2
Hangfire	ffc4912f	Background Job Processing Library	33.4	0.70	0.94	2.3	2	0.8	0	15.4
Humanizer	cc11a77e	String Manipulation Library	27.1	0.76	1.00	2.7	2	0.9	0	42.9
Lean	f574bfd7	Algorithmic Trading Engine	190.0	0.82	0.94	4.1	1	1.1	0	33.1
Nancy	e589dc9b	HTTP Service Framework	69.5	0.69	0.88	3.1	2	1.1	1	21.1
Newtonsoft.Json	372c396e	JSON Library	119.7	0.76	0.90	3.4	3	2.8	2	8.0
Ninject	1c84b358	Code Injection Library	12.8	1.00	1.00	1.4	1	0.4	0	4.9
NLog	39541571	Application Logging Library	67.4	0.84	0.89	3.5	3	1.5	1	30.2
Quartznet	b33e6f86	Scheduler	49.3	0.81	0.89	66.8	14	12.5	2	19.7
RavenDB	9d9ed290	Database	581.7	0.89	0.93	4.0	3	2.0	1	11.9
RestSharp	70de357b	REST and HTTP API Client Library	20.3	0.79	0.85	6.7	5	1.3	0	30.2
Wox	cdaf6272	Windows Application Launcher	13.2	0.80	0.94	4.6	4	1.3	0	22.7
		Average	86.4	0.80	0.91	8.7	3.4	2.2	0.6	21.9

of these pure clusters relative to the ground truth UDT labels. 62% of the UDTs map to exactly one fragment, meaning that RefiNym *perfectly* reconstructs the type of their UDT! 90% of the UDTs have 4 or fewer fragments and the maximum number of fragments for a UDT is 30. Data sparsity, which we explore below when analyzing the impure clusters, accounts for some fragmentation. Other fragmentation may arise because RefiNym is correctly separating merged conceptual types (defined next), despite our focus on UDTs.

Impure clusters appear to result from insufficient data: Figure 7b is scatterplot of purity *vs.* size and shows that smaller clusters, for which we have fewer data points, tend to be impure, while bigger clusters tend to be pure. Manual inspection of mid-sized clusters shows that misclassfication tends to occur when the name-flow graph is small and therefore RefiNym relies too much names alone. For example, `ConfigurationItemFactory` and `LogFactory` are instances of the factory design pattern in our data set. Their variables do not flow into other variables, so RefiNym clusters their variables into a single conceptual type because of their linguistic similarity. The second class of errors RefiNym makes is the merging of an uncommon type and its subtype (*e.g.* `LoggingRule` and `ConsoleRowHighlightingRule` in NLog) into one cluster. This is due to data sparsity again; RefiNym assumes that the diversity of these names does not warrant creating a new type.

4.2 Combating Primitive Obsession[2]

Merged conceptual types (MCT) are distinct conceptual types that share the same program type. Merged conceptual types may hold very different data. A program may store both passwords and usernames in `string`. When variables of these merged conceptual types share a scope, the type system cannot warn the developer if she mistakenly stores a password into a name. Examples of such scopes, which we call *critical*, in our corpus include Figure 1 in which server address, terminal, username and password conceptual types share `string`. A second example from the RavenDB project is shown in Figure 8 where the URL and the request JSON are conflated into the `string` type.

[2] Arguably, primitive obsession is a misnomer, especially when applied to strings in C# because strings are built-in types, not primitives. Throughout this paper, we use primitive obsession to refer to the overuse of either the primitive or built-in types of a language, not including the types in a language’s standard library (although one could argue for their inclusion).

```
1 var json = RavenJObject.FromObject(backupRequest).ToString();
2 var url = "/admin/backup";
3 var req = CreateRequest(url, "POST")
4 req.WriteAsync(json).Wait();
```

Figure 8: Primitive obsession in RavenDB.

RefiNym can identify such scopes, suggest new type annotations that separate their MCTs, and *automatically* rewrite the program to use these new types when the developer accepts a suggestion (Section 3). Here, we ask “How many critical scopes does RefiNym eliminate?”. To answer this question, we ran RefiNym on our corpus to refine `string` and find that 21.9% of critical scopes are eliminated *i.e.* scopes that contained MCT variables before running RefiNym but contains reified variables of a given (refined) type, after accepting RefiNym’s refinements. In all these scopes, RefiNym has freed the type system to do its job and warn the developer about potentially unwanted flows across conceptual types. For every critical scope, we also report the number of MCT variables which are potential sources of confusion that result in no type errors. Table 1 shows that on average RefiNym reduces the number of same-type variables in each scope from 8.7 to 2.2. This shows that RefiNym’s suggestions reduce the possibility of inadvertently introducing cross-conceptual type flows.

4.3 Case Study

We have just shown that RefiNym effectively identifies conceptual types through its performance at reconstructing UDTs and then showed how accepting its refinements can reduce the number of scopes in which a developer could introduce an unwanted flow. Here, we manually delve into and analyze some of its results.

RefiNym’s Primitive Refinements. We ran RefiNym over our corpus refining `string` for all projects except `BEPUphysics` where we refined `float`. Table 2 and Table 3 present a selection of the names RefiNym clustered into into conceptual types for `Nancy` and `BEPUphysics`. Natural language utterances or text can be diverse,

Table 2: Conceptual types from `string` refinements in NancyFx, a framework for building HTTP based services.

	Full name of nodes or constant values
1	`path`, `originalRequestPath`, `modifiedRequestPath`, `contentPath`, `"/emailConstraint/"`, `basePath`, `IViewEngineHost::ExpandPath`, `AspNetRootPathProvider::GetRootPath`, `"/"`, `owinRequestPath`, `DiagnosticsConfiguration::GetNormalizedPath`, `Path`, `NancyContext::ToFullPath`, `ModulePath`
2	`DefaultCulture`, `defaultCulture`, `cookieCulture`, `cultureLetters`, `name`
3	`earlyExitReason`, `"Requires Any Claim"`, `"Requires Claims"`, `"Requires Authentication"`, `"Accept"`
4	`IObjectSerializer::Serialize`, `DefaultObjectSerializer::Serialize`, `JsonObject::ToString`, `SimpleJson::SerializeObject`, `HttpQsCollection::ToString`
5	`method`, `Method`, `"PUT"`, `"POST"`, `"PATCH"`, `"OPTIONS"`, `"HEAD"`, `"GET"`, `"DELETE"`
6	`value`, `cookieValue`, `sourceString`, `"SomeValue"`, `cookieValueEncrypted`, `attemptedValue`, `decryptedValue`, `defaultValue`
7	`password`, `"password"`, `realPassword`, `plainText`, `Password`
8	`HttpUtility::UrlDecode`, `HttpUtility::UrlPathEncode`, `url`, `path`, `HttpUtility::UrlEncodeUnicode`, `redirectUrl`, `fallbackRedirectUrl`
9	`header`, `"Accept-Language"`, `"Accept-Encoding"`, `"Accept-Charset"`, `"X-Custom"`, `"Content-Disposition"`, `"Vary"`, `"Etag"`

exhibiting correct variation or noisy, exhibiting erroneous variation. Here, we discuss RefiNym's robustness to natural language's diversity and noise.

Unsurprisingly, most clusters contain lexically similar, even identical, names, because we expect similar or identical names to flow into each other. For example, all names in cluster 1 of Table 2 contain the subtoken `path`. In this cluster, `path` appears 73 times, although we show each name only once in the tables. Cluster 2 in Table 3 contains many names related to distance, suggesting that RefiNym not only finds similar, or identical names, but copes with the diversity of real-world names.

RefiNym also finds clusters that include lexically dissimilar elements because of noise. For example, cluster 7 in Table 2 predominantly contains the subtoken `password` but also the (badly named) variable `plainText`. We deem this name noisy because it has no textual similarity with the other elements. Indeed, manual inspection reveals that `plainText` is always used to represent passwords in the method `byte[] GenerateSaltedHash(string plainText, byte[] salt)` that generates a hash for storing passwords. Clustering `plainText` together with `password` would *not* have been possible if RefiNym only considered names and not the lattice constraint over the name flow graph. The interplay of the lattice constraint and names allows RefiNym to tolerate noise and to generalize conceptual types across synonyms, alternative names, even typos. For example, in Wox (not shown), we find a cluster that contains both `pythonDirectory` and `pythonDirecotry` (misspelled).

RefiNym also includes literals in $\mathbb{G}$. Their values are *not* used during clustering, since values instantiate types, *i.e.* are elements of types, and therefore we expect little (textual) similarity between literal values and variable or method names. However, given the lattice constraint, we can still cluster them together with other literals and variables. An example of a successful clustering of literals and variables is cluster 5 in Table 2. It shows a conceptual type that corresponds to HTTP methods. Refining this conceptual type is straightforward, requiring the introduction of an enumeration type which can prevent bugs triggered when a random string is passed as an HTTP method. RefiNym also mined a conceptual type for HTTP headers (cluster 9 in Table 2).

Table 3: Sample conceptual type (cluster) nodes for `float` type in BEPUphysics, a physics simulation C# project.

	Full name of nodes or constant values
1	`damping`, `SuspensionDamping`, `starchDamping`, `dampingConstant`, `angularDamping`, `LinearDamping`
2	`currentDistance`, `distance3`, `candidateDistance`, `pointDistance`, `distanceFromMaximum`, `grabDistance`, `VariableLinearSpeedCurve::GetDistance`, `tempDistance`
3	`goalVelocity`, `driveSpeed`, `GoalSpeed`
4	`minRadius`, `MinimumRadius`, `Radius`, `minimumRadiusA`, `WrappedShape::ComputeMinimumRadius`, `topRadius`, `MaximumRadius`, `graphicalRadius`, `TransformableShape::ComputeMaximumRadius`
5	`blendedCoefficient`, `KineticFriction`, `dynamicCoefficient`, `KineticBrakingFrictionCoefficient`
6	`angle`, `myMaximumAngle`, `MinimumAngle`, `currentAngle`, `MaximumAngle`, `steeringAngle`, `MathHelper::WrapAngle`
7	`targetHeight`, `Height`, `ProneHeight`, `crouchingHeight`, `standingHeight`
8	`mass`, `effectiveMass`, `newMassA`, `newMass`
9	`m22`, `m11`, `M44`, `resultM44`, `M43`, `intermediate`, `m31`, `X`, `Y`, `Z`

Inferring Unit Types. Units of measure and unit types are important in numerical computation. They can prevent a wide range of errors, such as performing inconsistent operations on physical quantities, *e.g.* adding velocity to an angle, or adding two variables of the same physical quantity that are nevertheless measured on a different unit, *e.g.* adding radians to degrees. The well-known NASA's Mars Climate bug [25] exemplifies their importance.

RefiNym's focus is proposing nominal type refinements, not unit types. Nevertheless, it distinguishes physical quantities and coefficients assigning them to different conceptual types. Table 3 shows a selection of clusters of `float` variables RefiNym finds in BEPUphysics, a physics simulation project. Cluster 3 represents speed-related nodes; cluster 8 contains mass-related nodes; and cluster 6 refers to angles. Thus, RefiNym can help developers identify unit types, alleviating the burden of manual annotation.

Since RefiNym infers a lattice of conceptual types, it detects unitless coefficients, such as damping (cluster 1) and friction (cluster 5) factors. Although all unit variables may flow into generic variables, RefiNym clusters such variables to appropriate supertype clusters. For example, cluster 9 represents a common supertype of many numerical quantities and includes generic (or conceptless) variables such as elements of a matrix (*e.g.* `m11`) or generic vector coordinates (`X`, `Y`, `Z`). Again, the reader may notice that cluster 9 (Table 3) contains highly diverse names with no textual similarity. These names are nevertheless clustered together thanks to the constraints imposed by the RefiNym's lattice constraint.

Qualitative Error Analysis. Like all machine learning, RefiNym does not yield perfect results. Here, we qualitatively discuss common errors we observed and speculate about their causes. RefiNym's most common error is to conflate nodes that are linguistically similar, but represent distinct conceptual types. Manual investigation reveals that most of these errors arise because these nodes belong to disconnected components, so RefiNym could *not* collect name-flow constraints among them. This commonly occurs in libraries which provide external APIs to their users without using them internally. These unused (public) methods (*i.e.* the library APIs) break the flow in the name flow graph resulting in disconnected or loosely connected components. In the future, aggregating information from external uses of a library may alleviate this issue.

RefiNym also generates "superclusters" that contain a large set of mostly unrelated nodes. RefiNym may be creating these superclusters because of the incompleteness of its split/merge operators and the greedy heuristic (Alg. 1). These superclusters may not be errors. They suggest that our information-theoretic objective does *not* consider further splitting to be statistically important or useful. So, even though these superclusters may seem erroneous to humans, it may be that RefiNym is finding base types in them that are *not* worth refining. This suggests an interesting new empirical research direction: "*Does* VI *correlate with the usefulness of a program's type lattice?*" Starting from a monotyped program, VI improves as the type lattice becomes more fine-grained. When the type lattice becomes too fine-grained, VI starts worsening again. This phenomenon may also be related to the increasing burden of gradually more complex type annotations in a program.

5 RELATED WORK

Types in programming languages are recognized for their ability to provide reasonable guarantees to a programmer, given a well-typed program. This works rests on the observation that the conceptual types developers have in mind and the types they actually reify in a program can diverge. An instance of this is the "primitive obsession" code smell [12, 20, 26, 28], which practitioners acknowledge. This smell is bad practice of using primitive or builtin types for elements of distinct conceptual types. Primitive obsession merges conceptual types, like the use of `int` for both IDs and counts that may lead to a type-correct addition of an ID and a counter. RefiNym uses code's bimodality to find type refinements that aligns conceptual and program types, surfacing latent conceptual types to the type system and alleviating primitive obsession.

Inference of abstract data types (ADT) [11, 22] is related to RefiNym since both methods can be used to refine a program's types by learning various forms of ADTs. Lackwit [22] statically infers ADTs by using the flow of data and assigning ADTs to a set of variables that can flow among each other. However, Lackwit solely uses data flow. In contrast to RefiNym, Lackwit cannot find conceptual types that share lexemes but whose variables have no flow and therefore suffers from low precision [11]. RefiNym avoids this issue by exploiting the rich naming information in source code to provide precise and consistent ADTs. Guo et al. [11] inferred ADTs from primitives by dynamically observing the interactions only among variable values, without learning or refining a type lattice. Their approach is dynamic, requiring a representative set of run-time data and broad code coverage. In contrast, our method is based on purely static information and aims to refine a program's types. Haq et al. [13] use dynamic ADT inference to find variables that interact. Then, they use variable names and a heuristic name-similarity metric to detect undesired variable interactions, by clustering the names within each ADT. In contrast, RefiNym refines types based on signal from names, rather than using names in a post-processing step. Furthermore, our unsupervised information-theoretic method circumvents specifying a priori the number of clusters.

RefiNym makes heavy use of code bimodality, *i.e.* the property that human-written code contains two modalities: one for communicating with the hardware and one for communicating among humans. Bimodality extends the observations of Hindle et al. [15] who found that code is "natural". With this intuition, a large corpus of work has followed [2]. Central to our work are the source code identifiers of variables and methods. Identifier names have been found to have a profound effect on code readability [5, 17] and developers greatly care about the names used within source code [1]. Bad variable names have been useful for detecting bugs [19] and are treated as anti-patterns in software development [3]. RefiNym is based on this intuition and that developers make great effort to pick good names and therefore can provide valuable information to both humans and machine learning methods. Of course, types and identifier names are intricately connected since they both aim to disambiguate entities, something that we explore in this work.

The notion of types appears in natural language processing through the concepts of hypo/hypernymy [7], the identification of named entities [18, 23] and semantic parsing [16]. Although conceptually this notion of type is similar to the one used in programming languages, a "softer" non-discrete approximation is used given the ambiguous nature of natural language.

Clustering is a common topic in unsupervised machine learning and data mining [30] that aims to infer the structure of some data in an unsupervised way. Common machine learning methods such as k-means clustering and Gaussian mixture models (GMM) [21] make strong parametric assumptions and are mostly used to cluster individual independent elements. Non-parametric methods, such as DPGMM [27] and information clustering [10, 29] do not require to define the number of clusters a priori. Information-theoretic clustering, which we use in this work, is an attractive set of methods since it makes no assumptions about the underlying data. Our core novelty over traditional non-parametric clustering methods, including information-theoretic methods, is that our method operates on graph structures and infers clusters that have a lattice structure.

Finally, the problem of grouping elements of a graph, resembles the idea of community detection in (social) network analysis [8]. There, the goal is to detect coherent communities in the network. In contrast to RefiNym, such methods mostly rely on detecting tightly connected regions within a graph based on its structure without considering additional constraints over the content of each node or the structure/relationship of each community/cluster.

6 CONCLUSION

In this work, we presented RefiNym, a method for mining conceptual types from existing source code. To achieve this, we cluster over source code elements, such as variables, methods and literals exploiting the interplay of their names and the structure of assignment flows. Through this process, we retrieve a fine-grained implicit conceptual type lattice that we use to suggest refinements. We presented quantitative and qualitative evaluation of our method that show the promises of this direction finding that the proposed refinements related strongly with conceptual types and can be useful for assisting developers to define more explicit type lattices.

In the future, we believe that the bimodality of source code, *i.e.* its natural language aspects and its rich structure can be exploited to offer improved software engineering tools and assist program analyses. This requires to develop new techniques that bridge programming language concepts with machine learning methods and preserve code semantics.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 281–293.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018 (to appear). A Survey of Machine Learning for Big Code and Naturalness. *Comput. Surveys* (2018 (to appear)).

[3] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gael Gueheneuc. 2013. A new family of software anti-patterns: Linguistic anti-patterns. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 187–196.

[4] Venera Arnaoudova, Laleh M Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gael Gueheneuc. 2014. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering* 40, 5 (2014), 502–532.

[5] Raymond PL Buse and Westley R Weimer. 2010. Learning a metric for code readability. *IEEE Transactions on Software Engineering* 36, 4 (2010), 546–558.

[6] Thomas M Cover and Joy A Thomas. 2012. *Elements of information theory*. John Wiley & Sons.

[7] Jia Deng, Jonathan Krause, Alexander C Berg, and Li Fei-Fei. 2012. Hedging your bets: Optimizing accuracy-specificity trade-offs in large scale visual recognition. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 3450–3457.

[8] David Easley and Jon Kleinberg. 2010. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press.

[9] .NET Foundation and Contributors. 2018. .NET Compiler Framework – Roslyn. https://github.com/dotnet/roslyn. (2018). [Online; accessed 2-March-2018].

[10] Erhan Gokcay and Jose C. Principe. 2002. Information theoretic clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24, 2 (2002), 158–171.

[11] Philip J Guo, Jeff H Perkins, Stephen McCamant, and Michael D Ernst. 2006. Dynamic inference of abstract types. In *Proceedings of the 2006 international symposium on Software testing and analysis*. ACM, 255–265.

[12] Refactoring Guru. 2017. Primitive Obsession. https://refactoring.guru/smells/primitive-obsession. (2017). [Online; accessed 15-Aug-2017].

[13] Irfan Ul Haq, Juan Caballero, and Michael D Ernst. 2015. Ayudante: Identifying undesired variable interactions. In *Proceedings of the 13th International Workshop on Dynamic Analysis*. ACM, 8–13.

[14] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001), 833–839.

[15] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 837–847.

[16] Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. 2017. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 1516–1526.

[17] Dawn Lawrie, Henry Feild, and David Binkley. 2007. An empirical study of rules for well-formed identifiers. *Journal of Software: Evolution and Process* 19, 4 (2007), 205–229.

[18] Xiao Ling, Sameer Singh, and Daniel S Weld. 2015. Design challenges for entity linking. *Transactions of the Association for Computational Linguistics* 3 (2015), 315–328.

[19] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. 2016. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 1063–1073.

[20] Mika Mantyla. 2003. Bad smells in software – a taxonomy and an empirical study. *Helsinki University of Technology* (2003).

[21] Kevin P. Murphy. 2012. *Machine learning: a probabilistic perspective*.

[22] Robert O'Callahan and Daniel Jackson. 1997. Lackwit: A program understanding tool based on type inference. In *In Proceedings of the 19th International Conference on Software Engineering*.

[23] Jonathan Raiman and Olivier Raiman. 2018. DeepType: Multilingual Entity Linking by Neural Type System Evolution. *arXiv preprint arXiv:1802.01021* (2018).

[24] Andrew Rosenberg and Julia Hirschberg. 2007. V-Measure: A Conditional Entropy-Based External Cluster Evaluation Measure. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning(EMNLP-CoNLL)*. 410–420.

[25] Brian J Sauser, Richard R Reilly, and Aaron J Shenhar. 2009. Why projects fail? How contingency theory can provide new insights–A comparative analysis of NASA's Mars Climate Orbiter loss. *International Journal of Project Management* 27, 7 (2009), 665–679.

[26] Mark Seemann. 2017. Primitive Obsession. http://blog.ploeh.dk/2011/05/25/DesignSmellPrimitiveObsession/. (2017). [Online; accessed 15-Aug-2017].

[27] Yee Whye Teh and Michael I Jordan. 2010. Hierarchical Bayesian nonparametric models with applications. *Bayesian nonparametrics* 1 (2010), 158–207.

[28] Beyond the Lines. 2018. Leveraging the type system to avoid mistakes. https://www.beyondthelines.net/programming/leveraging-the-type-system-to-avoid-mistakes/. (2018). [Online; accessed 13-May-2018].

[29] Greg Ver Steeg, Aram Galstyan, Fei Sha, and Simon DeDeo. 2014. Demystifying information-theoretic clustering. In *International Conference on Machine Learning*. 19–27.

[30] Rui Xu and Donald Wunsch. 2005. Survey of clustering algorithms. *IEEE Transactions on neural networks* 16, 3 (2005), 645–678.