Pattern Recognition in Procedural Knowledge

Breandan Considine

January 14, 2021

Contents

1 Introduction

Historically, most knowledge was stored as natural language. A growing portion is now *code* [1]. Code is procedural knowledge intended for execution by a machine. Though it shares many statistical properties in common with natural language [14], code is written in a formal language with a deterministic grammar and denotational semantics [25]. We can use this specification to precisely reason about operational or procedural correctness.

Prior work explored differentiable programming [8]. Differentiability plays a key role in learning, but does not provide the necessary vocabulary to describe human knowledge. In order to capture human knowledge and begin to reason as humans do, programs must be able to express the concept of *uncertainty*. In this work, we propose a set of tools and techniques for reasoning about uncertainty in the form of procedural knowledge.

To reason about procedural knowledge, we must first define what it means for two procedures to be equal. Although equality is known to be undecidable in most languages, various equivalence tests and semi-decision procedures have been developed. For example, we could rewrite said procedures [3], compare them in various contexts [11], and simulate or execute them on various input data [6] so as to ascertain their exact relationship.

In practice, exact equality is too rigid to operationalize. A more useful theory would allow us to compare two procedures in the presence of naturally-arising stochasticity. What is the probability of observing local variations? How are those observations related? And how do local variations affect global behavior? In order to correctly pose these questions and begin to answer them, we must be able to reason probabilistically.

Graphs are a natural representation for both procedural knowledge [2] and probabilistic reasoning [24]. The language of linear algebra provides a unifying framework for many graph algorithms and program analysis tasks [18]. Recent evidence suggests probabilistic inference is tractable for a large class of graphical models [7]. And sparse matrix representations enable efficient processing of large graphs on modern graphics processors [17].

In this work, we will define exact and approximate equality and cover some deterministic and probabilistic algorithms for deciding it. We will then describe a few graph representations for encoding approximate procedural knowledge. Finally, we will discuss some opportunities for applying these ideas to search-based software engineering, in particular code search, vulnerability detection, fault localization and program repair.

2 From exact to approximate equality

Reason is the source of all human knowledge. In order to understand reason, we need to understand how concepts are related. Next to identity, one of the simplest relations between concepts is equality.

For such an important concept, the notation for equality is recklessly overloaded in mathematics and computer science. For example, the expression $x = y$ may denote: (1) define x to be y, (2) x and y are the same, (3) are x and y the same? (4) x and y are exchangeable, (5) assign y to x, (6) assign x to y, among other peculiar programming idioms. If two expressions are equal, it is generally possible to treat them in the same manner.

But this convention does not always hold! Suppose we need to compute the derivative of a logical function with respect to its inputs. The trouble is, logical equality is not differentiable. Consider the Kronecker δ-function:

$$\delta_k(x, y) := \begin{cases} 1 \text{ if } x \stackrel{?}{=} y, \\ 0 \text{ otherwise} \end{cases}$$

When encountering δ_k, how should we represent its derivative? Since $\mathbb{B}$ is finite, $\delta_d^{-1}(B \subset \mathbb{B})$ is not open, thus δ_d is not continuous and $\nabla\delta_k$ is undefined. Now consider the Dirac δ-function, which is defined as follows:

$$\forall f \in \mathbb{R}^2 \to \mathbb{R}, \int_{\mathbb{R}^2} f(x, y)\delta(x - a, y - b)d(x, y) \stackrel{\Delta}{=} f(a, b)$$

Unlike $\nabla\delta_k$, it can be shown that $\nabla\delta_d$ is well-behaved everywhere on $\mathbb{R}^2$. However we encounter an important distinction between intensional and extensional equality. Unlike elementary functions, there exist many functions which can only be described indirectly, e.g. a probability distribution on a set of measure zero. Nevertheless, these constructions are convenient abstractions for modeling many physical and computational processes.

Neither $\nabla\delta_k$ nor δ_d are a satisfactory basis for equality. To allow a more flexible definition of the $=$ operator, we require a relation which approximates the logical properties of δ_k, but can be made differentiable like δ_d. A more general notion is the concept of an *equivalence relation*. An equivalence relation $\equiv$ is a binary relation with the following logical properties:

$$\frac{}{a \equiv a}\ \textit{Identity} \qquad \frac{a \equiv b}{b \equiv a}\ \textit{Symmetry} \qquad \frac{a \equiv b \quad b \equiv c}{a \equiv c}\ \textit{Transitivity} \qquad \frac{a \equiv b}{f(a) \equiv f(b)}\ \textit{Congruence}$$

2.1 Decidability

To determine whether two expressions are equal, we need a decision procedure. Various approaches for deciding exact and approximate equality in the deterministic and probabilistic setting are possible. We list a few below.

	Deterministic	**Probabilistic**
Exact	Type Checking Model Checking	Variable Elimination Probabilistic Circuits
Approximate	Software Testing Dynamic Analysis	Monte Carlo Methods Bayesian Networks

It is seldom the case that two semantically equal expressions are trivially equal: we must first perform some computation to establish their equality. In the exact setting, this procedure might be summarized as follows:

1. Rewrite: Either enumerate a set of equivalent expressions, or reduce the proposition into normal form if possible, then,
2. Compare: Perform a computationally trivial (e.g. $\mathcal{O}(n)$) comparison.

Unfortunately, exact equality is known to be undecidable in first [12] and higher order theories [13]. We know there can be no machine which accepts every equality and rejects every disequality in a universal language [29]. By extension, any nontrivial property of partial functions is undecidable [26].

Tractability may be related to, but is not contingent upon decidability. When decidable, equality may be intractable in practice, and languages where equality is undecidable may have decidable fragments. But even when exact equality is intractable, we may be able to construct a probabilistic decision procedure (PDP) or semidecision procedure (SDP) terminating for all practical purposes. The latter approaches fall into two broad categories:

- Execute: Evaluate the program by running it on a small set of inputs
- Sample: Build a probabilistic model and sample from its distribution

In the following section, we will introduce a few compatible theories corresponding to intensional and extensional equality, then build on those definitions to include recent approaches to exact and approximate equality in the determinsitic and probabilistic setting. In so doing, we will see there is a delicate tradeoff between complexity, sensitivity and specificity.

2.2 Intensional equivalence

Let $\Omega \subseteq \mathcal{F} \times \mathcal{F}$ be a relation on representable functions which are closed under composition. We say two representations $f, g \in \mathcal{X} \to \mathcal{Y}$ are intensionally equal under Ω if we can establish that $g \in \Omega^n(f)$ for some $n \in \mathbb{N}$.

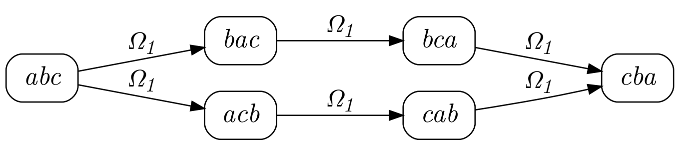
$$\frac{f, g : \mathcal{X} \to \mathcal{Y} \in \Gamma_g^0}{\Gamma_g^0 \vdash \{f\}, \{(f, f)\}}\textsc{Init} \qquad \frac{\Gamma_g^n \vdash E \subseteq \mathcal{F}, G \subseteq E \times E}{\Gamma_g^{n+1} \vdash \bigcup\limits_{\substack{e \in E\\ \sigma \in \Omega}} e' \leftarrow e[\sigma_1 \to \sigma_2], (e, e')}\textsc{Sub}$$

$$\frac{\Gamma_g^n \vdash E, G \qquad g \in E}{\Gamma_g^n \vdash f \equiv_\Omega g \text{ by } G^{-n}(g)}\textsc{Eq} \qquad \frac{\Gamma_g^n \vdash E \qquad \Gamma_g^{n+1} \vdash E \qquad g \notin E}{\Gamma_g^{n+1} \vdash f \not\equiv_\Omega g}\textsc{Neq}$$

For example, suppose we are given $f : \{a, b, c\} \mapsto abc, g : \{a, b, c\} \mapsto cba$ and $\Omega := \{(a, a), (ab, ba)\}$. Indeed, $\textsc{Eq}[f, g]$ can be established as follows:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{f := abc,\ g := cba \in \Gamma_g^0}{\Gamma_g^0 \vdash \{abc\}, \{(abc, abc)\}}\textsc{Init}}{\Gamma_g^1 \vdash \{\ldots, bac, acb\}, \{\ldots, (abc, bac), (abc, acb)\}}\textsc{Sub}}{\Gamma_g^2 \vdash \{\ldots, bca, cab\}, \{\ldots, (bac, bac), (acb, acb), (bac, bca), (acb, cab)\}}\textsc{Sub}}{\Gamma_g^3 \vdash \{\ldots, \mathbf{cba}\}, \{\ldots, (bca, bca), (cab, cab), (cab, \mathbf{cba})\}}\textsc{Sub}}{\Gamma_g^3 \vdash f \equiv_\Omega g \text{ by } G^{-3}(g := cba) = \{f := abc\}}\textsc{Eq}$$

We can visualize G as a directed graph, omitting all loops. Notice how each path converges to the same term, a property known as *strong confluence*.



Let us suppose $|\mathcal{X}|, |\Omega^*| \in \mathbb{N}$ and consider the complexity of establishing $\textsc{Eq}[f, g], \forall f \equiv_\Omega g \in \mathcal{X} \to \mathcal{Y}$. It can be shown the above procedure requires:

$$\mathcal{O}_{\textsc{Eq}} = \max_{i \leq n} \operatorname*{argmin}_n \{|G| \mid \Gamma_g^i \vdash G, \Gamma_g^n = \Gamma_g^{n+1}\}$$

Assuming termination, $\mathcal{O}_{\textsc{Eq}} = \Theta_{\textsc{Neq}}$ although $\mathbb{E}[\Theta_{\textsc{Eq}} | f \equiv g]$ is more tractable. However termination is not necessarily guaranteed, e.g. $\Omega' := \{(a, 1a)\}$. Equality and termination under arbitrary Ω are known to be undecidable [3].

2.3 Computational equivalence

Clearly, the procedure defined in § 2.2 is highly sensitive to $|\mathcal{X}|$ and Ω. While equality may be tractable, disequality is definitely an obstacle. In the computational setting, we will see the opposite holds, ceteris paribus.

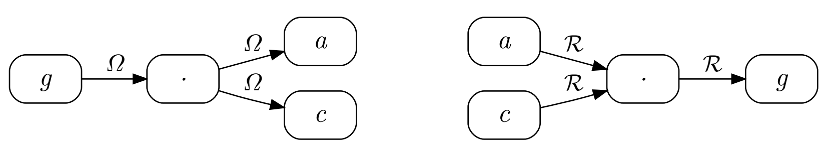
$$\frac{fg : \mathcal{X} \to \mathcal{Y}, \Omega : \{\mathcal{X} \to \mathcal{Y}\} \to \mathcal{Y} \in \Gamma}{\Gamma \vdash fg(\Omega) \Downarrow f(\Omega) \cdot g(\Omega)}\textsc{Inv} \qquad \frac{\Gamma \vdash f \qquad \Gamma \vdash \Omega}{\Gamma \vdash f(\Omega) \Downarrow \Omega[f]}\textsc{Sub}$$

$$\frac{\Gamma, \Gamma' \vdash f(\Omega) \Downarrow g(\Omega) \ \forall \Omega}{\Gamma, \Gamma' \vdash f \equiv g}\textsc{Eq} \qquad \frac{\Gamma, \Gamma' \vdash \exists \Omega \mid f(\Omega) \not\Downarrow g(\Omega)}{\Gamma, \Gamma' \vdash f \not\equiv g \text{ by } \Omega}\textsc{Neq}$$

$\textsc{Sub}$ loosely corresponds to η-reduction in the untyped λ-calculus, however $f \notin \Omega$ is disallowed and we assume all variables are bound by $\textsc{Inv}$. Let us consider $f : \{a, b, c\} \mapsto abc, g : \{a, b, c\} \mapsto ac$ under $\Omega := \{(a, 1), (b, 2), (c, 2)\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f \ := abc, \Omega := \{(a,1),(b,2),(c,2)\} \in \Gamma}{\Gamma \ \vdash \ a(\Omega) \cdot bc(\Omega)}\textsc{Inv}}{\Gamma \ \vdash \ 1 \cdot bc(\Omega)}\textsc{Sub}}{\Gamma \ \vdash \ 1 \cdot b(\Omega) \cdot c(\Omega)}\textsc{Inv}}{\Gamma \ \vdash \ 2 \cdot c(\Omega)}\textsc{Sub}}{\Gamma \ \vdash \ f(\Omega) \Downarrow 4}\textsc{Sub} \qquad \dfrac{\dfrac{\dfrac{g \ := ac, \Omega := \{(a,1),(b,2),(c,2)\} \in \Gamma'}{\Gamma' \ \vdash \ a(\Omega) \cdot c(\Omega)}\textsc{Inv}}{\Gamma' \ \vdash \ 1 \cdot c(\Omega)}\textsc{Sub}}{\Gamma' \ \vdash \ g(\Omega) \Downarrow 2}\textsc{Sub}}{\Gamma, \Gamma' \vdash f \not\equiv g \text{ by } \Omega := \{(a,1),(b,2),(c,2)\}}\textsc{Neq}$$

We can view the above process as acting on a dataflow graph, where $\textsc{Inv}$ backpropagates Ω, and $\textsc{Sub}$ returns concrete values $\mathcal{Y}$, here depicted on g:



Assuming $f, g \sim P(\mathcal{F}), \Omega \overset{iid}{\sim} P_{\textsc{Test}}(\Omega \mid f \not\equiv g)$ yields a fixed but unknown distribution, $P_{\textsc{Neq}}(\Omega) = P(f(\Omega) \not\Downarrow g(\Omega) \mid f \not\equiv g)$. Let $\Theta_{\textsc{Inv}} = 1$ for all f, Ω. The complexity of certifying $\textsc{Neq}$ in n trials follows a geometric distribution:

$$\Theta_{\textsc{Neq}} \sim (1 - P_{\textsc{Neq}}(\Omega))^n P_{\textsc{Neq}}(\Omega) \text{ with } \mathbb{E}[\Theta_{\textsc{Neq}}] = (1 - P_{\textsc{Neq}}(\Omega)) P_{\textsc{Neq}}(\Omega)^{-1}$$

Although a single witness Ω s.t. $f(\Omega) \not\Downarrow g(\Omega)$ is sufficient for disequality, this procedure may be intractable depending on $|\mathcal{X}|, P_{\textsc{Neq}}(\Omega)$ and $\Theta_{\textsc{Inv}}$. Other fuzzing methods for selecting Ω based on the structure of f are also possible.

2.4 Observational Equivalence

As presented, both intensional (§ 2.2) and computational (§ 2.3) equivalence require an external definition of equality to satisfy. One solution to this problem known as *observational equivalence* [22] allows a language $\mathcal{L}$ to implement an internal mechanism to verify equality. Given $\mathcal{L}$, a term t, and one-hole context $C[\![\cdot]\!]$, our job is to check for termination: if $C[\![t]\!]$ is both well-defined and halts, we write $C[\![t]\!] \Downarrow$, otherwise $C[\![t]\!] \Uparrow$.

$$\frac{\Gamma \vdash C[\![t]\!] \Downarrow \iff C[\![t']\!] \Downarrow \forall\ C[\![\cdot]\!] \in \mathcal{L}}{\Gamma \vdash t \equiv_{\mathcal{L}} t'}\ \textsc{Eq}$$

$$\frac{\Gamma \vdash \exists C[\![\cdot]\!] \in \mathcal{L} \mid C[\![t]\!] \Uparrow \text{ and } C[\![t']\!] \Downarrow, \text{ or } C[\![t]\!] \Downarrow \text{ and } C[\![t']\!] \Uparrow}{\Gamma \vdash t \not\equiv_{\mathcal{L}} t' \text{ by } C[\![\cdot]\!]}\ \textsc{Neq}$$

We can think of this definition as dual to computational equivalence: instead of searching for inputs which distinguish functions, we search for contexts which distinguish terms, or a proof that no such context exists. Two terms t and t' are contextually equivalent with respect to $\mathcal{L}$ if we can prove that for all contexts $C[\![\cdot]\!]$ in $\mathcal{L}$, $C[\![t]\!]$ halts if and only if $C[\![t']\!]$ halts – if no such proof can be found, the test is inconclusive. While this definition does not admit a decision procedure, many promising SDPs exist.

2.5 Approximate Equivalence

Approximate equivalence requires a *distance metric*, $\delta : \mathcal{Z} \times \mathcal{Z} \to \mathbb{R}_{\geq 0}$. This is a generalized equivalence relation with the following logical properties:

$$\underset{\textit{Definiteness}}{\frac{\delta(a,b) = 0}{a \equiv_\delta b}} \qquad \underset{\textit{Symmetry}}{\frac{\delta(a,b)}{\delta(b,a)}} \qquad \underset{\textit{Triangularity}}{\frac{a \qquad b \qquad c}{\delta(a,c) \leq \delta(a,b) + \delta(b,c)}}$$

A *kernel function* can be defined as a metric on $\mathcal{Z}$ with some additional structure. In particular for every kernel function, there exists a feature map $\varphi : (\mathcal{X} \to \mathcal{Y}) \to \mathcal{Z}$ such that $\Delta : (f, g) \mapsto \langle \varphi(a), \varphi(b) \rangle$. Consider a space of valid kernel functions, $\blacktriangle \subset (\mathcal{Y} \to \mathcal{Z})^2 \to \mathbb{R}_{\geq 0}$. We may define it inductively:

$$\frac{\Delta \in \blacktriangle, k \in \mathbb{R}_{>0}}{k\Delta \in \blacktriangle} \qquad \frac{\Delta_{1,2} \in \blacktriangle}{\Delta_1 + \Delta_2 \in \blacktriangle} \qquad \frac{\Delta_{1,2} \in \blacktriangle}{\Delta_1 \Delta_2 \in \blacktriangle} \qquad \frac{\Delta \in \blacktriangle, f \in (^* \!\to \mathcal{Y})^2}{\Delta \circ f \in \blacktriangle}$$

Suppose we are given a feature map $\varphi : \mathbb{R}^n \to \mathbb{R}^m$ and consider the complexity of computing the inner product $\langle \varphi(f), \varphi(g) \rangle$. The computational benefit of a kernel emerges when $n \ll m$: instead of applying the map and computing the inner product, we can instead apply the kernel function directly in input space. This may be easier to visualize as a commutative diagram:

$$\begin{tikzcd}
\mathcal{X}_1 \arrow[r, "f"] & \mathcal{Y} \times \mathcal{Y} \arrow[r, "\varphi"] \arrow[dr, "\Delta"] & \mathcal{Z} \times \mathcal{Z} \arrow[d, "\langle \cdot, \cdot \rangle"] \\
\mathcal{X}_2 \arrow[ur, "g"] & & \mathbb{R}_{\geq 0}
\end{tikzcd}$$

Some common kernel functions are listed below:

Polynomial	$k(\mathbf{f}, \mathbf{g}) := (\mathbf{f}^T \mathbf{g} + r)^n$	$\mathbf{f}, \mathbf{g} \in \mathbb{R}^d, n \in \mathbb{N}, r \geq 0$
Laplacian	$k(\mathbf{f}, \mathbf{g}) := \exp\left(-\frac{\lVert \mathbf{f} - \mathbf{g} \rVert}{\sigma}\right)$	$\mathbf{f}, \mathbf{g} \in \mathbb{R}^d, \sigma > 0$
Gaussian RBF	$k(\mathbf{f}, \mathbf{g}) := \exp\left(-\frac{\lVert \mathbf{f} - \mathbf{g} \rVert^2}{2\sigma^2}\right)$	$\mathbf{f}, \mathbf{g} \in \mathbb{R}^d, \sigma > 0$
String Kernel	...	...
Tree Kernel	...	...
Graph Kernel	...	...

We want a want a kernel function $M_\theta : (\mathcal{X} \to \mathcal{Y}) \times (\mathcal{X} \to \mathcal{Y}) \to \mathbb{R}$ between two computable functions, which predicts their semantic similarity. In other words, the closer two functions are with respect to M_θ, the more likely they are to be equal...

2.6 Probabilistic Equivalence

Though precise, the prior definitions of equality are far too rigid in practice. The spaces involved either lack formal semantics or are intractable to exhaustive search. We now turn to a language for probabilistic reasoning, which admits among other things, a decision procedure for probabilistic equivalence.

Various measures have been proposed...

Hypothesis testing

Kolmogorov-Smirnov

Kantorovich-Rubinstein

Kullback-Leibler

Lévy-Prokhorov

Gromov-Hausdorff

Jensen-Shannon

Cauchy-Schwartz

EMD

TODO: Define probability distributions, integration, kernel functions and metrics.

3 From computation to knowledge graphs

Duality is an important concept in mathematical optimization. Many optimization problems can be seen as dual to each other: KKT and SVM duality. Duality occurs in automatic differentiation with dual number arithmetic.

Duality is also an important concept in computer science. One famous example is the duality between code and data: in *homoiconic* languages, we can treat code as data and data as code. cf. Kleene's recursion theorem.

One way to view automatic differentiation is that it allows us to compute the sensitivity of numerical values in a fixed computation graph. What we wanted to compute sensitivities with respect to changes in the computation graph itself? For that, we need to define a the graph as an algebraic object.

3.1 Algebraic graphs

Graphs can be modeled algebraically [30] using algebraic data types [21].

Graphs are algebraic structures...

Semirings arise in strange and marvelous places. $(min, +), (max, \times)$

1. `https://people.cs.kuleuven.be/~luc.deraedt/Francqui4ab.pdf#page=71`

2. `http://www.mit.edu/~kepner/GraphBLAS/GraphBLAS-Math-release.pdf#page=11`

3.2 Programs are graphs

computation graphs [5] in machine learning

e-Graphs [31] in reasoning systems

arithmetic circuits [20] in numerical computing

probabilistic circuits [7] in probabilistic modeling community

3.3 Probabilistic graphical models

Probabilistic graphical models (PGMs) are very expressive, but even approximate inference on belief networks (BNs) is NP-hard [10] We can faithfully represent a large class of PGMs and their corresponding distributions as probabilistic circuits (PCs) [7], which are capable of exact inference in polynomial time and empirically tractable to calibrate using SGD or EM. PCs share many algebraic properties with PGMs and can propagate statistical estimators like variance and higher moments using simple rules.

3.4 Knowledge graphs

Knowledge graphs [15] are multi-relational graphs whose nodes and edges possess a type. Two entities can be related by multiple types, and each type can relate many pairs of entities. We can index an entity based on its type for knowledge retrieval, and use types to reason about compound queries, e.g. “Which company has a direct flight from a port city to a capital city?”

4 From procedural knowledge to code

Experts typically encode their knowledge using pen and paper and leave developers to decipher it. It is somewhat tedious, but generally possible for skilled programmers to translate textual information into computer programs. Unfortunately, there are many equivalent ways to translate text into code – the same algorithm implemented in the same language by different authors is seldom written the same way. Usually we end up reinventing the wheel. So we need some mechanism to detect exact or approximate equality in procedural knowledge.

Instead of the Sisyphean task of forever translating these ideas from scratch, coders need to step back and think: Is it possible to just encode the axioms and enough knowledge to derive a family of algorithms, then let the compiler derive the most appropriate procedure for computing some desired quantity? A good compiler might be able to use those facts to optimize computation graphs, e.g. for latency or numerical stability. This has been the holy grail of declarative programming.

Short of that, can we have some kind of *bibliotheca universalis* containing many human-written code examples, which a human could select from manually (a la Hoogle [16]) – or even better – which could be linked to during compilation. We can think of big code as a kind of procedural knowledge system, describing common data transformations. We would like a way to extract common snippets and reason about those transformations, for example to detect similar procedures or optimize an existing procedure.

Knowledge systems or *ontologies* are a collection of related facts which have been established by human beings. For example, we can treat mathematics as a knowledge base of rewrite rules. This has been successfully operationalized in Theano [4], Kotlin∇ [9] and other DSLs. More broadly, we can also think of constructive mathematics libraries like Metamath [19], Rubi [27], Probonto [28] and KMath [23] as working towards this same goal.

In knowledge graphs, approximate equality is known as entity *alignment* or *matching*. With a probabilistic matching algorithm, we could accurately detect near duplicates in a codebase. We could retrieve code samples to assist developers writing unfamiliar code. And we could search for bugs in code or fixes from a knowledge base to repair them. Probabilistic reasoning can be gainfully employed on these and many related tasks.

Suppose we want to search through a software knowledge base for an error and stack trace, then use the information in the KB to repair our bug.

1. Efficiently searching corpus for a pattern

2. Identifying alignment and matching results

3. Incorporating information into user's context

4.1 Code search

model fragments of code and natural language as a graphs and learn a distance metric which captures the notion of similarity. Some graphs will be incomplete, or missing some features, others will have extra information that is unnecessary.

Given a piece of code and the surrounding context (e.g. in an IDE or compiler), search a database for the most similar graphs, then to recommend them to the user (e.g. fixes or repairs for compiler error messages), or suggest some relevant examples to help the user write some incomplete piece of code.It is similar to a string edit distance, but for graph structured objects. There are a few pieces to this:

1. Semantic segmentation (what granularity to slice?)

2. Graph matching (how to measure similarity?)

3. Graph search (how to search efficiently?)

4. Recommendation (how to integrate into user's code)

The rewriting mechanism is similar to a string edit distance, but for graphs. One way of measuring distance could be measuring the shortest number of steps for rewriting a graph A to graph B, i.e. the more "similar" these two graphs are the fewer rewriting steps it should take.

4.2 Fault localization

searching for stack trace on stackoverflow

4.3 Program repair

adapt some knowledge to match user's context

4.4 eDSL generation

Take a procedural knowledge base, generate a DSL from it.

References

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That*. Term Rewriting and All that. Cambridge University Press, 1999.

[4] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, pages 1–7. Austin, TX, 2010.

[5] Olivier Breuleux and Bart van Merriënboer. Automatic differentiation in Myia. 2017.

[6] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases, 2020.

[7] YooJung Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic Circuits: A unifying framework for tractable probabilistic models. 2020.

[8] Breandan Considine. Programming Tools for Intelligent Systems. Master's thesis, Université de Montréal, 2019.

[9] Breandan Considine, Michalis Famelis, and Liam Paull. Kotlin∇: A Shape-Safe eDSL for Differentiable Programming. `https://github.com/breandan/kotlingrad`, 2019.

[10] Paul Dagum and Michael Luby. Approximating probabilistic inference in bayesian belief networks is np-hard. *Artificial intelligence*, 60(1):141–153, 1993.

[11] Matthias Felleisen. On the expressive power of programming languages. In Neil Jones, editor, *ESOP '90*, pages 134–151, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[12] Kurt Gödel. *Über die vollständigkeit des logikkalküls.* na, 1929.

[13] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.

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

[15] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, José Emilio Labra Gayo, Sabrina Kirrane, Sebastian Neumaier, Axel Polleres, et al. Knowledge graphs. *arXiv preprint arXiv:2003.02320*, 2020.

[16] Michael B James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. Digging for fold: synthesis-aided API discovery for Haskell. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, 2020.

[17] Jeremy Kepner, Peter Aaltonen, David Bader, et al. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2016.

[18] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[19] Norman Megill. Metamath. In *The Seventeen Provers of the World*, pages 88–95. Springer, 2006.

[20] Gary L Miller, Vijaya Ramachandran, and Erich Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM Journal on Computing*, 17(4):687–695, 1988.

[21] Andrey Mokhov. Algebraic graphs with class (functional pearl). *ACM SIGPLAN Notices*, 52(10):2–13, 2017.

[22] James Hiram Morris Jr. *Lambda-calculus models of programming languages.* PhD thesis, Massachusetts Institute of Technology, 1969.

[23] Alexander Nozik. Kotlin language for science and kmath library. In *AIP Conference Proceedings*, volume 2163, page 040004. AIP Publishing LLC, 2019.

[24] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier, 2014.

[25] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software Foundations. `https://softwarefoundations.cis.upenn.edu`, 2020.

[26] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[27] Albert D Rich and David J Jeffrey. A knowledge repository for indefinite integration based on transformation rules. In *International Conference on Intelligent Computer Mathematics*, pages 480–485. Springer, 2009.

[28] Maciej J Swat, Pierre Grenon, and Sarala Wimalaratne. ProbOnto: ontology and knowledge base of probability distributions. *Bioinformatics*, 32(17):2719–2721, 2016.

[29] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[30] B Weisfeiler and A Leman. The reduction of a graph to canonical form and the algebgra which appears therein. *NTI, Series*, 2, 1968.

[31] Max Willsey, Yisu Remy Wang, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. egg: Easy, efficient, and extensible e-graphs. *arXiv preprint arXiv:2004.03082*, 2020.